Computation in the Real World:

Foundations and Concepts of

Programming Languages

José Ulisses Ferreira Junior

Doctor in Philosophy

Trinity College, University of Dublin

2001

First Declaration

The work described in this thesis has not been submitted as an exercise for a degree at this or any other university.

Signed:

José Ulisses Ferreira Junior

October, 2001

Second Declaration

The work described in this thesis is, except where otherwise stated, entirely of the author.

Signed:

José Ulisses Ferreira Junior

October, 2001

Permission to Lend or Copy

I agree that Trinity College Library may lend or copy this thesis upon request.

Signed:

José Ulisses Ferreira Junior

October, 2001

Summary

Broadly speaking, the present PhD thesis supports that philosophy is in the foundations of computer science. Branches of artificial intelligence may also rest on the philosophy of computing science, which in its turn is in the referred to foundations of computer science.

Part of the work which is necessary for this thesis is a result from my identification of a number of programming languages concepts and constructs in accordance with my recent research and previous knowledge. Then, in accordance with these observations, I introduce a consistent number of programming languages constructs together with their semantics. Two of the core issues are code mobility and a constant for representing lack of information in the problem domain that I shall refer to as $\boldsymbol{uu}$, while both the idea and notation originally come from the Łukasiewicz and Kleene three-valued logics. Both notions are essential components of this work.

Code mobility is a relatively new issue in programming languages, distributed systems, artificial intelligence, software engineering and theoretical communities of computer science. I extend $\boldsymbol{uu}$ to other levels of abstractions, e.g. to programming languages and a notion of computation. My objectives are as follows:

1. *Turing machines do not necessarily correspond to functions.*

2. *Computation is not conceptually function application.*

3. *Some representation of the absence of information is essential in programming languages and the foundations of computer science.*

4. *A probably more general notion of computation with* **uu** *and four forms of mobility.*

5. *A general solution for mobile agents technologies on global computers.*

6. *Some programming languages concepts and constructs suitable for global environments. In addition to* **uu**, *the concepts and constructs are combined to form a hybrid programming paradigm which supports the following sub-paradigms: functional, imperative, object-oriented, logic, uncertainty, and mobile agency.*

7. *Philosophy is essential in the foundations of computer science.*

As part of the adopted methodology in the contained discussion, I use deductive logics, make comparisons, and present examples from the insights. I do research at libraries, as well as on the WWW where the document is reliable. During my research, in the rare cases where some somewhat similar work was found, I made comparisons between the present one and theirs. Finally, as usual in any philosophical piece of work, I make use of the following skills: belief, non-mathematical induction, intuition, opinions and analogies when appropriate, together with the more traditionally used ones in sciences, that are knowledge, observations, deductions and five-sense perception.

Acknowledgements

As the author of the present work, I would like to thank everyone who has worked towards my PhD.

The work of the present PhD thesis was dedicated to my family and the people from Edinburgh.

Contents

Computation in the Real World: Foundations, Concepts and Constructs of Programming Languages

Summary **5**

Acknowledgements **7**

The Current Table of Contents **9**

List of Tables **15**

List of Figures **16**

Glossary **17**

1 Introduction **19**

1.1 An Introduction 19

1.2 Some Motivations 21

1.3 The General Context 23

1.4 Methodology 27

1.5 Some Historical Comments 28

1.6 Connections Between Chapters 29

1.7 Contents of the Dissertation 35

1.7.1 Foundations of Computer Science 35

1.7.2 Concepts and Constructs of Programming Languages . 36

Part I - Foundations of Computer Science **39**

2 A Novel Space-Time Logic and The Deductive System **41**

2.1 Introduction 43

2.2 A five-valued propositional logic 49

2.2.1 Semantics, notions of space and time 50

2.2.2 The five values 54

2.2.3 Examples 61

2.2.4 Cycles: An Illustration 63

2.2.5 Analogy, Belief and Uncertainty 64

2.2.6 A Few More Examples 70

2.3 Sequents 71

2.4 Deduction 74

2.4.1 Axioms 77

2.4.2 Structural Rules 77

2.4.3 Logical Rules 77

2.5 The Space-Time Operational Semantics 86

2.5.1 The evaluation of Boolean expressions 87

2.5.2 The execution of commands 88

2.6 Representing Mobility 89

2.7 Conclusion 90

3 A Property of the Universal Turing Machine **93**

3.1 Introduction 93

3.2 Turing machines 95

3.3 Some interpretations 101

3.3.1 Unexpected effects 105

3.4 A Refuting Example 106

4 Mobility and Computation **119**

4.1 Introduction 121

4.2 Agents 126

4.3 Mobility and some related concepts . . . 127
4.4 Other Concepts . . . 130
4.5 An intuitive notion of computation . . . 134
4.6 A notion of computation . . . 141
4.6.1 A view of time, a representation . . . 141
4.6.2 States of the Real World . . . 142
4.6.3 The Present Semantics of Computation . . . 150
4.7 Computing in the real world . . . 165
4.8 Conclusion . . . 169

Chapter 5 A Complete Solution for Mobile Agents on Global Structures 171

5.1 Introduction . . . 172
5.1.1 Mobility and its Paradigms . . . 172
5.1.2 Contents of the Chapter . . . 174
5.2 Programming for a Global Computer . . . 175
5.3 Problems . . . 176
5.4 Communication Between Individuals . . . 187
5.5 Conclusion . . . 188

Part II - Concepts of Programming Languages 191

6 Programming Language Concepts and Constructs for Global Computers 193

6.1 Introduction . . . 194
6.2 Some Current Mobile Code Languages . . . 196
6.3 Plain . . . 200
6.4 *uu* in Global Computers . . . 202
6.5 Lazy Evaluation and Timeout . . . 207
6.6 Logic Programming . . . 208
6.7 Strong Mobility . . . 209

6.8 Other Features . . . 210
6.9 Conclusion . . . 212

7 *uu* for Programming Languages 213
7.1 Motivation . . . 213
7.2 Related Work . . . 215
7.3 *uu*: the Unknown Value . . . 216
7.3.1 Evaluators and Reactors . . . 217
7.3.2 Comparing Handers with Methods and Functions . . . 219
7.4 *uu* in Exception Handling . . . 221
7.5 Object-Oriented Programming with *uu* . . . 222
7.5.1 *uu* and Frames . . . 222
7.5.2 Classes with *uu* . . . 224
7.6 Imperative and Logic-Based Features . . . 228
7.7 *uu* in Lazy Evaluation (Call by Need) . . . 229
7.8 Implementation . . . 230
7.9 Conclusion . . . 231

8 *uu* and Uncertainty for Global Computing 233
8.1 Introduction . . . 235
8.2 *uu* and Uncertainty . . . 237
8.3 Uncertainty Handling . . . 242
8.4 Evaluation . . . 244
8.4.1 Forward Evaluation . . . 245
8.4.2 Operational Semantics - Forward Evaluation . . . 249
8.4.3 Backward Evaluation . . . 251
8.4.4 Inference Operator - Inferop . . . 259
8.5 Conclusion . . . 261

9 *uu* in Globallog 263
9.1 Introduction . . . 264
9.1.1 Conventions . . . 267

9.1.2 Contents of this Chapter 267
9.2 Syntactical and Semantic Definitions 268
9.2.1 Syntactical Definitions 270
9.2.2 Semantic Definitions 271
9.2.3 Intentions 281
9.3 An Operational Analysis 283
9.4 Examples 288
9.4.1 A Global Extension 289
9.5 Consistency of a Knowledge Base 291
9.5.1 Dealing with Inconsistency 293
9.6 Conclusion 294

10 Conclusion **297**

10.1 Another Approach 297
10.2 Foundations of Computer Science 298
10.3 Concepts for Programming Languages 300
10.4 Further Work 301
10.5 Sciences and Deductive Logics 303
10.6 The Conceptual Diagram 304
10.7 Synthesis in Programming 306
10.7.1 Concepts and Constructs of Kind π 307
10.7.2 Concepts and Constructs of Kind ω 307
10.7.3 Concepts and Constructs of Kind ψ 309
10.7.4 Concepts and Constructs of Kind ϕ 310
10.8 Synthesis in Knowledge Representation and Reasoning 313
10.9 Synthesis in Foundations of CS 314
10.9.1 A Paradox Example - Analogy 323
10.9.2 Another Paradox Example - Induction 324

The Appendices **328**

A The Space-Time Classical Logic and The Corresponding System **329**

A.1 Axioms 329

A.2 Logical Rules 330

A.2.1 Space and Time 330

B An Operational Semantics **333**

C Symbols and Conventions **339**

Bibliography **345**

List of Tables

The @-Logic Equivalence 54
The Other @-Logic Truth Tables 56
Belnap 4-Valued Logic Tables With *ii* 58
The @-Logic Implication 60
An Intuitionistic Implication for the @-Logic 72
The Five-Valued Implication for the @-Calculus 73

The **if-then** Statement Simulated in the **while** Language 154

BNF of a Very Simplified Subset of PLAIN 201

BNF of Hypothesis Declaration of PLAIN 237

Syntax for GLOBALLOG 270
The Classical Non-Flying Bird Example in GLOBALLOG 288

List of Figures

Two Views of the Time Flow 63
Two Formulae Referring to the Same Time (left) 75
The General Situation: Two Space-Time Formulae (right) 75
Possible Combinations of Wholeness of a Formula 76
Mobility of an Object 90

A Turing Machine With its Tape Squares 96
A Turing Machine Composition 100
Compositions 116

A Initial State for Logical Variables 242
A Logical Variable With Value *tt* 242
A Logical Variable With Value *ff* 242
A Logical Variable With Value *uu* 242
Unknown-Valued Variable and One Truth Threshold 243
A True Variable and One Truth Threshold 243
A False Variable and One Truth Threshold 243

Programming Languages Diagram 306
Foundations of Computer Science Diagram 316
Synthetic and Analytical CS Diagram 319
A Possible Hierarchy of Subjects 326

Glossary

AI	Artificial Intelligence
BNF	Backus-Naur Form
CE	Computational Environment
COD	Code On Demand
CPU	Central Processing Unit
CS	Computer Science or Computing Science
C-S	Client-Server
CWA	Closed-World Assumption
DS	Distributed System
ELP	Extended Logic Program
FTP	File Transfer Protocol
GC	Global Computer
GLP	General Logic Program
MA	Mobile Agent(s)
MAS	Mobile Agents System
NAF	Negation As Failure
OOP	Object-Oriented Programming
OWA	Open-World Assumption
PGC	Programming for a Global Computer
REV	Remote Evaluation
TM	Turing Machine
TTP	Trusted Third Party
WWW	World Wide Web
vpd	Value in the problem domain

Chapter 1

Introduction

As human beings do not fully know the universe, any current holistic view has the absence of information.

The absence of information (here, denoted by **uu***, the undefined notion) is of essential importance, not only in philosophical views and theories but also in practice for computation and programming languages. On the one hand, the obvious proof of this is* $0/0 =$ **uu***. On the other hand, the computation of* $0/0$ *programmed in a language without* **uu** *(up to date, unfortunately this applies to virtually all programming languages) has resulted in "run-time error" instead.*

1.1 An Introduction

In a sense, this dissertation defends the thesis that computer science does not need one theory of computation, but instead many theories of computation, each one based on its explicit philosophical view and a notion of computation. Those theories should be sound with respect to the reality in computer science. Because of this reason, the term *theory* itself should not be restricted to the sense of deductive logical theory, but instead have a more general sense, more appropriately described in natural languages. This PhD thesis yields the following major results:

1. The demonstration that *Philosophy is essential in the foundations of computer science.*

2. The demonstration that *the Turing machines model does not necessarily correspond to functional models.*

3. The demonstration that *the class of computations does not correspond to the class of function applications.*

4. The demonstration that *uu* is an essential notion in logics, programming languages, knowledge/belief representation and hence computer science. *uu* denotes the non-information.

5. The introduction of a physical and possibly more general notion of computation with *uu* and four forms of mobility.

6. The introduction of a general solution for mobile-agent technology on global computers with flexible and symmetric security policies.

7. The introduction of some programming languages concepts and constructs suitable for global computing. In addition to *uu*, the concepts are part of the hybrid-paradigm programming language which supports the following sub-paradigms: functional[39, 241], imperative, object-oriented[1, 206, 289], logic [283], uncertainty, and mobile agency. From the artificial intelligence (AI) [209, 255] standpoint, the same language can be seen as supporting objects and frames[206, 215][1] in addition to functions, logic rules[77] and production systems[255] with certainty factors.

The need for an integrated programming paradigm has been recognized[224]. PLAIN[103] is the programming language that supports almost all the features described in the present thesis dissertation, among other characteristics.

[1]In addition to chapter 7 of the present thesis dissertation, a discussion that explains differences between frames and objects, as well as some material on agents from the AI perspective, is in [251].

PLAIN design and implementation is a piece of individual work that has been carried out by the same author for many years. The current PLAIN implementation compiles a program into interpretable and dynamically portable byte-code. A PLAIN sublanguage, referred to as GLOBALLOG, aims at reconciling global environments with logic programming.

1.2 Some Motivations

I am used to separate the notion of language from its implementation. Part II of the present PhD thesis is not on implementation but instead on programming languages. The same applies to the logic of chapter 2. To give some brief ideas of the differences between GLOBALLOG and Prolog, supposing that there is a blank sheet containing only the classical-logic formula $A \Rightarrow B \vee C$. For GLOBALLOG, one may assume an open or closed world. Under an open-world assumption, if we know that B is true, we cannot infer that A is also true. In contrast, under the closed-world assumption, if we know that B is true, we can infer that A is also true, for $A \Rightarrow B \vee C$ is the only way we have to reason in. Therefore, there is an implicit equivalence in such implications, i.e. $A \Leftrightarrow B \vee C$, in an assumed closed world. The same observation applies to Prolog, where we divide this implication in two rules to fit the Prolog syntax: abstractly, $\{B \leftarrow A.\ ,\ C \leftarrow A.\}$. More generally, GLOBALLOG allows programmers to define each clause under either open- or closed-world assumption. All constructs for programming languages here rest upon some explicit philosophical view, as well as an explicit notion of computation.

On the one hand, although PLAIN has some influence from Java[21], Haskell [80, 291], Prolog[138, 227, 280], POP-11[26, 196] as well as knowledge representation and reasoning in general[293], my proposals on programming are different from the POPLOG system[2] which supports, among many other things, Prolog, Common LISP and POP-11. In my set of proposals, code is written in different paradigms but in the same programming language. In other words,

[2]POPLOG is a trademark of the University of Sussex.

the level of integration of the paradigms is large, and the syntax is more consistent, since the language is a single one. Another difference is that, after having chosen established programming languages, system designers should respect their original definitions. Therefore, in this way, we would not be as free as we are to make fundamental changes, such as including ***uu*** in Prolog, LISP and POP-11, for instance, or such changes would be more difficult to make. Finally, languages such as LISP are regarded as old.

On the other hand, the philosophy that supports hybrid programming paradigms is not the same as the one which could support the idea of multi-paradigms, for the former has a larger level of integration than the latter. Furthermore, PLAIN not only provides a hybrid paradigm but also aims at forming a unique well-balanced paradigm, and this differs from the approach of extending one paradigm or one language to permit programming in others, such as done in Prolog++[223] and LIFE[8].

Besides a broad notion, such as a hybrid paradigm, PLAIN supports *uu*. Briefly speaking, as I explain in the present thesis, *uu* is an epistemic[277] constant that is normally interpreted as "unknown" or "undefined". In a sense, *uu* is a kind of vac*uu*m. This is different from the undefined result in some partial function evaluation, which has been interpreted as "infinite computation" and is usually expressed as $\perp$. Yet, *uu* can represent $\perp$, not the opposite. Although *uu* is a very simple notion, *uu* could be regarded as a kind of zero for programming languages or computer science, metaphorically speaking. In particular, it is present in virtually all PLAIN constructs. Accordingly, except for chapter 3, which is on Turing machines, in turn a notion defined in the 1930's, *uu* is present throughout the present thesis dissertation.

Although this piece of work is essentially based on philosophy, this work can also be interesting for researchers from the AI community, e.g. from the area originally known as knowledge representation and reasoning. Branches of AI may rest on the philosophy of computer science.

1.3 The General Context

Since the beginning of the 20th century, mathematics and computer science have been very closely linked to each other, and one has helped the development of the other[79, 296].

Although there are what can be called "*philosophical views in mathematics*"[31], the idea of computation has been essentially perceived in the same form since the thirties, in particular, strongly influenced by constructivism, which is also one of the most recent views in mathematics. However, in the meanwhile, computer science has become an independent science as computers have become useful, personal, portable, even powerful, and so forth, while all philosophical views in computer science have remained essentially the same as in mathematics. In this way, I have observed the importance of making the existing views in computer science explicit, as well as proposing alternative views in the same science. As an example, Alan Turing established what is well known as the Turing test, a parameter for artificially intelligent programs. However, by studying psychology as well as philosophy, one becomes much more conscious of how difficult such a problem is and whether its solution is possible or even desired. By studying psychology and philosophy, one obtains answers to such questions more readily.

Semantics is a subject which is closely related to philosophy, and researchers such as Robin Milner[211] have studied such semantics of computation for years, and mobility has gradually played its due rôle in this subject. Moreover, I can observe that, from the moment that I conceive the idea of moving computation from a place to another in our real world, I should, to remain consistent, accept the idea that a number of physical factors are present in the meaning of computation. Because of such transcendental semantics, philosophical views absent from the philosophy of mathematics[261] become valid in the foundations of computer science.

In the present work, I introduce *"computing in the real world"*, which in turn lies on this holistic view. Given this, I do not neglect the importance of

some forms of reasoning[88] absent from deductive logics and present in our daily lives, e.g. analogy, induction and belief.

As an example of inductive reasoning in mathematics and computer science, Zobel states in [321],

It is common practice to use ellipsis to describe a sequence of integers; thus $m, \ldots, n$ *is all integers between* m *and* n *inclusive. An infinite sequence is usually represented by* $m_1, m_2, \ldots$*, where it is assumed that the reader can extrapolate from the initial values to the other members of the sequence. Thus "*$2, 4, 8, \ldots$*" would be assumed to be the sequence of positive powers of 2. Always state both the lower and the upper bound if the sequence is finite and ensure that the intended sequence is clear.*

As a general rule, this kind of synthetic reasoning depends on more subjective notions, such as *clarity*. The same kind of observation applies to analogies, for we humans assume that the reader will make the same kind of synthetic reasoning as we humans do. Although almost all subjects in mathematics are analytical, there are such exceptions as illustrated above. In the conclusion of the present thesis, I present a diagram placing induction and deduction in different classes of concepts.

As opposed to analogy, while an author is presenting a deductive proof[3] (if it were possible to do this using only deduction), he or she does not need to trust on the reader's ability to follow connections and be in tune with the author, with regard the involved ideas. This also illustrates why *ethics* is one of the branches of philosophy. In this case, ethics consists in both the author and the reader being honest. Ethics ought to be explicit in the foundations of computer science.

My work on computing in the real world is strongly based on two orthogonal polarities: *knowledge-induction* and *deduction-belief*. I regard knowledge and deduction as analytical notions, whereas I regard analogy, induction and belief as synthetic notions. Because to date most of work in computer science

[3]For a very introductory book on proof techniques which may be suitable for undergraduate students, I would suggest [306].

has been analytical, I aim at demonstrating a better balanced approach for computing. On the other hand, the four basic psychological functions according to Carl Jung[176], are thinking, feeling, sensation and intuition, which form a more general classification in some sense. Although the literature on these four functions is known, for those who are satisfied with a brief explanation, it is worth observing that feeling in his psychological theories is not in the sense of feelings or emotions, as *feeling* has been used instead, as a kind of inner skill which gives to the only person information on what he or she likes, as well as on what he or she dislikes. Moreover, typically, the concept of feeling is often related to human relationships, not to physical pain, which in turn comes by *sensation*, for instance. With regards intuition, it is often seen as an ability to guess. Clearly, this gives rise to philosophical belief. What is the most relevant issue here is that, apart from simulations of synthetic notions, it is reasonable to think that machines do not have feeling, they do not feel pain nor pleasure, they do not love nor hate, that certainly they do not guess, and that any philosophical view is typically based on belief.

Comparing deduction and induction, a reasonable form of deduction has been easier to implement on a digital computer than a reasonable form of induction on the same machine. However, this does not entail that deduction is a more primitive skill than induction. As an example of a primitive form of inductive reasoning, one can teach an animal by physically punishing that animal or feeding it in a consistent manner whenever it behaves against or in accordance with one's expectation. The animal then learns by induction[178, 180]. A common key word in both induction and the frame form of knowledge representation is *expectation*. Induction creates expectation, which is used in the deductive part of the reasoning. In this example, the animal not only builds a general rule, possibly involving some words from our vocabulary, but also deduces in a language that if it behaves in a particular way, it will be punished, for instance.

Comparing knowledge and belief, belief may be regarded as a kind of weak knowledge, but it is not very clear where the threshold between the two no-

tions rest, and this threshold may be personal. The synthetic nature of a large number of subjects is another characteristic of belief. As an example, if someone *believes* that he or she will succeed in a particular exam, this hypothesis is also a consequence of the general belief that people do not normally have *knowledge* about the future. Therefore, *future* is a notion that often leads to beliefs. There exist many other notions that lead to belief. In general, *fears* and *hopes* are closely linked with beliefs, and not necessarily with knowledge.

In deductive logics[83], belief has been confined to a modal operator in a doxastic logic[117]. The formula $\bigcirc A$ means "A is believed to be true".

Two of the fields that deserve a fruitful philosophical discussion are mobile agents and global computing since they both have a number of subtleties. Another field is programming languages, for this subject has always played an essential rôle in computer science.

I shall make novel connections in this dissertation, e.g. what *uu* and mobile agents have in common. Apparently, one does not have much to state but, for instance, mobile agents typically imply a global or wide-area network, agents have to be robust and *uu* is the solution that I have found to, among other benefits, permit agents to deal with faults, where, in many cases, lack of information should not be regarded as error, but instead as a normal condition. By propagating *uu* at the level of the semantics of language, agents neither stop running nor commit with any value in the problem domain.

After some years investigating *uu*, I have realized that we still need novel work. Because of this, and because philosophy is a very broad subject, the present dissertation is necessarily large. As completeness is a desirable property of logical theories, a philosophical view has to be discussed broadly.

Every real programming language design and implementation have two adjacent levels: the upper level can be the programmer's viewpoint, which includes paradigms. The lower level is the machine and computability. The upper level has been changed over the last few years as technologies have developed, and will possibly continue to do so. As regards the machine, although physics and electronic engineering have developed rapidly, in this thesis, I only

refer to virtual (software) machines, not physical ones.

Inconsistency is important because agents have to reason about the real world. For example, it is not rare to find inconsistent mathematical theories, in particular, when the piece of work is large. Similarly, testing is a normal phase in software development. Unless the mistake is critical, it is normally expected that such inconsistencies be tolerated or corrected without much effort. The present PhD thesis deals with inconsistency.

An interesting observation applies to the closed-world assumption with negation as failure, where the absence of a predicate in a program implies its negation as an answer to a query. That is, to answer questions about time-table of flights and similar applications, Prolog is appropriate. However, if a person answers a query by asserting that some proposition is false only because he or she knows nothing about the proposition, that may sound somehow arrogant in a real situation. I see philosophy as essential in the foundations of computer science, for such issues are fundamental.

With programming on the WWW, there has been a call for a new notion, namely *global computation*. This this term was coined by Luca Cardelli[57]. Code mobility also requires a new notion of what can be computed, and here, in this thesis, I provide a novel model of computing that includes mobility. Since semantics is a subject closely connected to philosophy, my work on semantics is also part of the broader and more informal proposal.

1.4 Methodology

Briefly speaking, in this thesis, I use insights, logical argument, formal proofs and also present examples, make comparisons with related work, and I draw syntheses. In other words, as well as interaction with other researchers, I use my own ideas, knowledge, beliefs, experience, and both deductive and inductive forms of reasoning.

As regards more philosophical texts, sample presentation is connected with induction and analogy, i.e. the ability to see similarities between things that

are apparently very different from each other, or whose similarities are difficult to be described. Accordingly, a formal proof is connected to deduction and the ability to observe differences where more than one object look very similar. I observe that these methods and skills are essential in this holistic view. As a consequence, the adopted methodology for the present PhD thesis necessarily includes both synthetic and analytical methods based on the corresponding skills.

With respect to programming languages paradigms, concepts and constructs [137, 269, 315], specifically, the author has personally programmed since 1981 and professionally worked on the subject since 1983 at a University. Such experiments provide induction, which is one of the important forms of inference. The author has also worked on language implementation since the same time, and has designed and implemented an experimental programming language for years, which has given empirical knowledge and eventually has brought insights to the present thesis. In this thesis dissertation, analogies are also adequate at some places, for analogy helps simplify informal explanations of complex subjects.

1.5 Some Historical Comments

The history of this work is started in 1989, at the Universidade Federal da Paraíba, in Campina Grande, Brazil, where the author did his Master degree. That Master dissertation was the design and implementation of a programming language for diagnoses[107] using AI techniques, in particular, knowledge representation and reasoning[44]. The name of the programming language was LIDIA (LInguagem para DIAgnose). The project was continued at the Universidade Federal da Bahia, in Brazil, until 1997. It was a tool for some expert system applications, such as classification. The compiler read the source file and then built a network which was not exactly Bayesian networks but had some properties in common[108]. Then the runtime system interpreted byte code. The main problem of implementing a Bayesian network[271] was speed.

Adding some fact to the knowledge base makes the system update too many probabilities and, because of this, I simplified the uncertainty model by *not* assuming that probabilities are related to each other, although such connections have been able to be done by programming at a lower abstraction level.

In 1992, probabilities in LIDIA were replaced by MYCIN confidence factors because the latter are much easier to program. Probabilities are very useful for machine learning and many other applications, but for knowledge representation[278], they are not a straightforward way of representing uncertainty.

LIDIA was gradually being improved and, in 1996, after the implementation of object-oriented programming, the name LIDIA was replaced by the name PLAIN.

Finally, my PLAIN programming language provides a constant called *uu* in the paradigms that it currently supports: functional, object-oriented, imperative, uncertainty and mobile agents. In this thesis, I explore a number of facets of a constant called *uu* in these programming language paradigms. Furthermore, GLOBALLOG, which is a programming sublanguage that is part of PLAIN, provides *uu*. The whole PLAIN provides a single form of negation.

1.6 Connections Between Chapters

Holism is a philosophical view where the whole or wholes are more focused and regarded as more important than the sum of their parts. Thus, each part is primarily important for their corresponding functions in the whole. Moreover, the connections between parts are regarded as specially important or even more important than the parts. Finally, each part can be recursively seen as a whole. Thus, this is also a systematic view of the universe.

Some of these chapters have already been published in the form of independent articles. Accordingly, because one of the most significant and general claim here is that philosophy is essential in the foundations of computing science, one of my intentions in the organization of this dissertation is to present

individual contributions in each chapter in such a way that in the end I have two levels of contributions: individual and from the synthesis. In this way, on the one hand, the body of the chapters might not seem to be so closely related to each other for I have to explore each of them deeply. On the other hand, I intend to make connections between their subjects, and such connections are expected to be not easy to be seen *before* reading this material. While one chapter is exploited the reader concentrates on that particular subject.

One of the means by which I conceived and developed the present thesis is to connect some subjects and issues that, to date, have not been connected in the computer science community, or, at least, have not been connected in the same way. However, to make these connections clearer, I give an overview in this section, make cross references in the body of the text and make the synthesis afterwards.

Broadly speaking, although it might seem that not all approved theses for the degree of Doctor of Philosophy in Computer Science have followed the meaning of the word *Philosophy*, it is reasonable to claim that philosophy should be considered more in the foundations of this science, perhaps not less than it has been considered in mathematics. Traditionally, it is one of the philosophers' rôles to observe natural laws and, as making an analogy, to set laws for humans. The recent global computational structure suggests this hypothesis. In this thesis, I apply some insights to the foundations of computing science, and apply one theory for programming[38] languages constructs.

Further, the task of designing a logic and the task of designing a programming language (part II) are similar in nature, and both are similar to the task of setting laws in some civilized community, for both logics and programming languages normally suggest and even state how other people can or have to think.

Thus, the present holistic view or theory can be summarized as follows: the environment is the universe in $\mathbb{E}^3$ with $\boldsymbol{uu}$, my notion of computation transcends pure mathematics, the adopted semantics must capture details such as mobility, space and time, and, finally, the programming paradigm is hybrid.

Because of this, I adopt the operational semantics to accommodate space and time. There have been other novel approaches to operational semantics, such as [303], which is based on labelled transition systems and an abstract machine.

Still broadly speaking, there is an obvious connection between parts I and II, for programming languages and the foundations of computer science have traditionally been closely related areas of studies.

To connect the part I to the II, I present a current technological scenario, including the Internet and the relatively recent idea of moving code during a computation. A motivation is to provide programming languages concepts and constructs for such a scenario. Then I present that, although such research work is very useful, the task of introducing language constructs and concepts raises some interesting issues as a PhD thesis. First, programming languages have many theoretical aspects in computer science, but it is very probable that we cannot prove that some constructs and concepts are more appropriate than others, in particular, using formal deductive logics. Such an assertion sheds new light on the foundations, and it makes interesting connections between the parts I and II of this dissertation. Here, as well as using a formal logic language in part II defined in chapter 2, I introduce the languages constructs together with a number of examples. I formalize the semantics only to be more precise in my explanation. Therefore, what underly this specific approach are not only deduction, discrimination and attention to small details, but also other abilities, namely *inductive reasoning*, *analogy* and *intuition*, and the latter three notions are subjects which are closely related to philosophy.

As another connection between the parts, there seems to be a tradition in the theoretical computer science community to view programs as functions and computation as function applications. The refutation of the former view, in chapter 3, leads me to my hybrid programming paradigm in part II, whereas the refutation of the latter view, also in chapter 3, leads me to exploit a more general notion of computation, which is the key subject in chapter 4.

As an example, the theoretical community likes functional programming

languages such as PCF[218, 231], which stands for *Programming Computable Functions*, ML[236], Haskell[159, 291], Miranda[267, 290], and other functional languages have traditionally been preferred among theoreticians. However, so far these traditional languages have not been strongly connected to the technologies which I shall consider. There are few exceptions and, in chapter 6, I briefly describe a few existing functional languages although they also provide imperative features, e.g. Tycoon[208] and Facile[292]. They support a weak form of code mobility. Therefore, my aim consists in concisely presenting both a holistic view over computing science and, accordingly, a hybrid programming paradigm with strong mobility, and this novel programming paradigm might be a class of preferred languages by both theoreticians and working programmers. This paradigm is part of the hybrid approach. However, as part of the present PhD research, I realize that *programs should not be limited* to the scope of pure functional programming, although pure languages such as Haskell and Miranda certainly deserve respect. Briefly, although the results from this study on Turing machines and mobility do not entail a hybrid programming paradigm, at least they discard a possible conservative opinion that programming should be functional.

As already mentioned, one of my arguments rests on code mobility. Another piece of argument rests on my analysis of the Turing machine model of computation. From the mobility side, I recognize two philosophical views in computer science, and demonstrate that depending on the view, the foundations of computer science are not exactly in mathematics as the notion of computation becomes physical. From the analysis on Turing machines, I demonstrate using a sample situation that the theoretical basis is not precisely functions. That is, I prove logically here that programs are not necessarily functions, even in Turing machines. As a consequence of this, one of my present theses is that *computation is a more general notion than function applications*. In this way, I also explore such a more general notion of computation.

Although these two different contributions, namely on Turing machines

and recent technological global structures with code mobility, are not very closely related, I use both kinds of arguments to provide a non-functional programming model in the part II. As a consequence of the theoretical results in computability theory, I observe that the foundations of computer science should be based on important physical and philosophical issues. Regarding the subtle rôle of chapter 2, logics has traditionally been part of philosophy, but the latter is very broad and contains many other subjects that require informal discussions, such as esthetics, ethics, religion and so forth because these are traditional branches in philosophy. Therefore, I generalize the argument on this underlying subject, starting from logics and mathematics, and arriving at philosophy via physics, also as modern physics raised philosophical issues regarding mobility in the last century. However, one should bear in mind that the individual arguments that I use, i.e. code mobility and the analysis on the Turing machine model, are individually important as results for this PhD thesis. As a consequence of providing a hybrid programming paradigm with the theoretical support in part I, as part of my conclusions, I intuitively generalize the result by drawing a diagram showing the kinds of languages constructs according to their features. Because I view knowledge representation and reasoning from the programming point of view, I draw another equivalent diagram from the AI perspective, containing the orthogonal polarities knowledge-induction, deduction-belief. Another diagram is drawn showing the kinds of research work in computer science that one ought to use deductive proofs, as well as other kinds of research work in computer science that, probably, one cannot make use of deductive proofs, or the proofs are more difficult. Because of this, one may want to consider other methods.

I use the space-time logic to represent the notion of computation in chapter 4. The space-time semantics could be explicitly used in part II. I prefer to think that they support the constructs in II: I only do not make explicit the details of the @-logic because almost all constructs do not produce mobility, and also because non-spatio-temporal formulae are in the @-logic syntax. An intuitionistic logic[156] or a classical logic formula can be viewed as a syntax

sugar of an equivalent @-logic formula. Therefore, the operational semantics presented in part II is in the @-logic, in this sense.

As an example of connection between chapters 5 and logic programming, while security mechanisms in current specific technologies is one of the major reasons why mobile agents have not made it yet into the mainstream of distributed systems[240], there is no concept of virus in my model, and the security rests on the lack of agents which provide local services. Thus, if some mobile agent makes some access to a local resource, it means that some local programmer should have written or obtained some piece of code that permitted that access. This leads to another representational problem, stating who is capable of doing what. Because there are many of these capabilities, a declarative form of knowledge representation fills a gap left in programming for a global environment.

At a more detailed level, as mentioned, the core issue in this dissertation is a value called *uu*. Similarly, *uu* is the core idea in the proposed hybrid paradigm, and this value links almost all chapters in this dissertation.

The notion of computation that I formalize, and the scenario that I describe, make use of ***uu***. This value is absent from the chapter 3, but one can still regard the blank symbol or, alternatively, a sequence formed by two blank symbols for instance, as a kind of ***uu***. The chapter 3 also motivates my hybrid programming paradigm with ***uu***. All other chapters of this thesis dissertation are very based on ***uu***, except for chapter 2.

Finally, as suggested before, the three notions that link the first to the second part of this dissertation are ***uu***, mobility and global computing. One tends to think of mobility as code mobility in the context of global computing, but one can conceive mobile computing without global computing and the inverse.

1.7 Contents of the Dissertation

This section briefly describes the contents of the present PhD thesis dissertation.

1.7.1 Foundations of Computer Science

In chapter 2, I briefly introduce a space-time logical language. As well as being a result *per se*, the logic helps other parts of my thesis, e.g. it can be used for writing the semantics of mobile-code languages. In chapter 3, I separate two essential notions: a program and a function in Turing machines. Accordingly, I separate the notions of computation and function application. Chapter 4 is philosophical and contains the present claim that philosophy is essential as part of the foundations for computing science. In this chapter, I introduce my own notion of computation that includes operations with mobility. I use the present logic to describe the semantics of computation with mobility. In particular, this chapter is essential for describing the semantics of any programming language that supports code mobility, including my adaptation of the **while** language, which in turn is one of the well-known abstract models of computation. Nonetheless, once the reader understands the model contained in this chapter, there is no need to represent explicitly space or time in all language constructs, instead, only in those that cause mobility. From chapter 4, I also obtain another conclusion that, due to mobility and global environments, programs are not necessarily functions. Together with the results from chapter 3, this is a result that demolishes the idea that programming languages have to be functional. Therefore, both chapters 3 and 4 support the second part where I try to combine different programming paradigms for a global environment. For the uniformity on the language constructs, my proposal is to provide the operational semantics only. Chapter 5 is a paper that has been published in [105]. When the Conference is annually held, a different set of subjects is discussed and, in 2000, one of the symposia of this conference was on mobile agents. In chapter 5, I discuss problems and solutions concerning

mobile-agent systems without going into any detail of programming language features, while I describe the current technological scenario, as well as the introduction of my solutions to the discussed problems.

1.7.2 Concepts and Constructs of Programming Languages

Chapter 6 introduces an overview of languages constructs for global computers, including some examples of *uu* for a global platform such as the Internet. Some form of lazy evaluation, in both parallel and sequential operations, when combined with timeouts can be useful for programming on such an environment. The idea is original in programming. Note that there are constructs that support global computers that do not cause code mobility. Some features are more directly related to remote operations than code mobility properly, although, depending on the level of abstraction, code mobility could also be seen as a form of remote operation. Chapter 7 [106] introduces a constant in the programming language community that has been called *uu*, since other pieces of work by Łukasiewicz and Kleene on three-valued logics. However, *uu* is not limited to logical variables, and the contribution is extended to the programming level. Here I regard general-purpose programming languages, although *uu* can also be useful for global computers and AI (knowledge representation and reasoning). The programming paradigms that are discussed in this chapter are functional, imperative, frame-based and object-oriented paradigms. This chapter and the corresponding article make references to the published appendix B, which presents the formal semantics of the discussed features of Plain, also referred to in chapter 6 and another article. The appendix B is a slight extension of the appendix of the published article and, because of this, I omit the published one. *uu* is the central component in the part II. One of the contained results is that *uu* is important. *uu* is useful for code mobility since most of mobile-agents systems are naturally implemented on global environments. Chapter 9 contains a logic programming language

based on Prolog that also makes use of *uu*. Chapter 10 contains a synthesis of this PhD thesis dissertation, as well as the demonstration that philosophy is in the foundations of computing science.

Appendix A introduces the classical version of the @-logic. Appendix B contains the semantics of the concepts discussed in chapters 6 and 7. Finally, appendix C contains a glossary of notation used in the present dissertation.

Part I - Foundations of Computer Science

Chapter 2

A Novel Space-Time Logic and The Deductive System

It is well known that space and time are two primary notions that have always been present in the human consciousness independently of contexts. In this chapter, I propose a space-time logic on five epistemic values, together with uncertainty. This logic has a somewhat expressive language and, because of this, I also introduce a deductive system for the present logic in the same syntax. Here, the present author's final aim is mainly to introduce a general framework for writing formal semantics of programming languages that support code mobility, in particular mobile agents, among other more traditional applications such as knowledge representation, while one eye is kept for general purposes.

Moreover, the present logic is powerful enough for representing more sophisticated forms of reasoning, such as to weigh up possibilities. For being able to make fair judgments one should consciously attach honesty factors (in this case, floating-point numbers in $[-1, +1]$*) to diversified implications between premises and some conclusion in such a way that that conclusion from weighing up possibilities can be based on those factors. As it is well known, this is a very common form of reasoning. Thus, the corresponding language also*

provides the expressiveness of uncertainty-based representation combined with deduction for such purposes.

2.1 Introduction

Classical logics, in both propositional and predicate forms[54], used to be the single one since the Greeks and George Boole's time[81], with many more recent contributions[121, 304, 281]. In the last century, a number of logics have been developed and established as alternatives to classical logic. In particular, intuitionistic logics, in both propositional and predicate forms, have increasingly attracted the attention of mathematicians and non-mathematicians [31]. In fact, computers have played important rôle in many logics with constructive proofs. As regards intuitionistic logic, since Brouwer and Heyting[31], there have been many important contributions in the field. In [74], for instance, the duality in Cartesian closed categories, λ-calculi, intuitionistic and classical logics from syntactic and semantic viewpoints are investigated, while, regarding philosophy of mathematics, in [33, 51], there are two kinds of defense of classical view in mathematics and logic. The range of subjects is broad. Regarding a more informal and philosophical (but informally logical) literature, logics and space-time together do not play lesser relevant rôles[284].

In the last decades, some other contributions to logics[126] have appeared. For example, Arthur Prior[71, 246, 247, 248] and others[128] are some of the important contributors to modal logics[62] and temporal logics. A reference on them is [98]. Here, I introduce a space-time logic that is called @-logic.

The proposed language of the present logic is based on five values, $\mathit{ff}, tt, uu, kk, ii$ representing, briefly speaking, $false$, $true$, $unknown$, *possibly known but consistent*, $inconsistent$, and that is an idea based on the Belnap four-valued logic[30] in the following sense: uu means "neither *true* nor *false* nor *inconsistent*" while ii means "inconsistent" which in its turn means both Boolean values at some level of reasoning. Łukasiewicz[116] introduced many-valued logics or infinitely-many-valued logics, both based on a set of values from *false* to *true*, e.g. $\{0/3, 1/3, 2/3, 3/3\}$. Stephen Kleene also introduced his many-valued system[186], with some modifications on Łukasiewicz's. Here, in this orthogonal work to the other components of the present logic, I do not adopt

degrees of veracity, although I deal with degrees of veracity in another context of the @-logic, which is uncertainty. On the other hand, the present calculus is somewhat similar to the Łukasiewicz three-valued logic or Kleene three-valued logic as $\neg ii$ also results in ii. The differences to those many-valued logics[203] will become explicit in sections 2.2 and 2.2.2.

Some many-valued logics, as well as modal and temporal logics, were introduced having as motivation the representation of forms of veracity referring to the future, i.e. propositions referring to the future are regarded as neither *true* nor *false*[302]. Thus, we can regard the truth value of the propositions as unknown. However, there are many other uses for the representation of lack of information. Here, I do not philosophically[179] discuss on whether it is possible for one to have knowledge about the future. In any case, for any event, unless we experience it somehow and in a particular situation, we humans do not normally know whether such an event will happen or not. In space[60] (a reference on spatial cognition is [122]), the need for the notion of lack of information is essentially the same. We often know what somehow reaches us by communication. Otherwise, even on the past events are normally unknown. Because of this, the present space-time calculus is based on the present five-valued logic, in particular, in one of the defined implications.

On the one hand, a number of temporal and spatial logics have been introduced[11] for a number of purposes[115] with success, whereas *spatial reasoning*[69] has been deeply studied in AI. On the other hand, there has been a relatively small number of spatial theories on predicate logics and other attempts have been made. For instance, interval temporal logics are suitable for planning systems and scheduling[10, 13]. For time, we propose a more general approach for representing time for actions, events and tasks, than that of James Allen's temporal logics, which is more at the AI or application level than here[12]. His logic sees time as intervals, which is more general than points and makes his calculus very suitable for planners. Thus, points can be represented as $[p_1, p_2]$ where $p_1 = p_2$. In the present work, both space and time are represented as sets, a more general form of representation. If we all want

to represent cyclical events, we are able to do so by considering unions between intervals, for instance. I mention other aspects, for instance, concerning models and derivations, studied in [65, 89, 226, 295]. However, the present piece of work is in the scope of the emerging *philosophy of computer science*, which is in (philosophical) foundations of computer science, and models of the @-logic are left for further work, in particular, since this chapter is long.

There are other pieces of work on applying logics in some areas in computer science[127, 257]. Briefly, in addition to the literature on different logics[92, 99, 100], Girard's linear logic[87, 142, 294] and labelled deductive systems[125] are two examples of work that can be applied to computer science. A very good paper on introduction to logics in computer science is in [264]. Here, my purpose is to represent knowledge-belief and reasoning upon representation. However, in terms of philosophy, issues over complexity in logics, whether **P** or **NP**, apply mainly for those who assume and prefer to see humans as machines, whereas here my philosophical work takes a contrasting direction. In short, complexity over logics can be seen as part of theory over implementation, while logic and language rest upon a higher level. Although this lack of interest on complexity seems to depart from computer science, the present work is much closer to the idea of logic and language for computer scientists and philosophy of computer science, not necessarily for running on computers. Therefore, although philosophy of computer science is an absolutely new field, the present work is in computer science.

Little work has been done on spatial logical theories. The Region Connection Calculus[250] is a predicate theory on space. On the other hand, the author of [114] concentrates on a more detailed level of abstraction. There are other approaches and spatio-temporal logics, such as [129], which is a logic for multi-agent problem domains. A different approach for agents is shown in [158]. My approach is to introduce a powerful and expressive language, while I abstract details addressed in specific applications, e.g. AI systems. It seems that, to date (that is 2000), there has not been space-time logic, at least as a universal one, i.e. for general purposes. Even [278] does not contain

relatively significant contribution combining both notions in one formal logic. Instead, in parallel to temporal logics, there has been more specific work on the broad subject, for instance, spatio-temporal databases[168], a model and language[305], predicate theories such as RCC-8, in particular, those useful for AI. In contrast with particular space-time logics, an updated bibliography for data mining research is [256]. In [69], the authors observe the similarities between temporal and spatial structures, but they did not collapse both into sets. Like in [40] which applies *rough sets*[243] to a spatio-temporal context, I collapse and generalize both notions, although I use *sets* (according to that article, rough set theory[237] provides a way of approximating subsets of a set when the set is equipped with a partition or equivalence relation. The same article contains another related issue. It notes that unfortunately the exact location of spatio-temporal objects is often *indeterminate*, which motivates the definition and interpretation of the *uu* value, as well as the notion of uncertainty, in the present space-time logic.), and because I accept sets (hence intervals) for both space and time in a Cartesian space representation, the present logic might also capture other notions such as geometrical or geographical relational operations and so forth.

However, perhaps because technologies such as mobile-code languages are relatively recent, computer science has not had symmetry and balance concerning attention to time and space. We have some temporal logics but, to date, space has had little attention from the academic community with rare exceptions such as [41]. Some proposals, in particular predicate theories, have appeared but no space-time logic has been established. It seems that, for various purposes, it would be desirable to arrange both approaches, with respect to space and time, in only one logic.

For this chapter, let $\mathbf{C}$ be the set of all formulae in the space-time logic i.e. the language of the @-logic. Thus, to start explaining the subject, a classical version of the space-time logic can be defined as follows:

Definition 1 *Let φ and ϕ denote two formulae and α be a variable (a quan-*

tifier). Thus, a classical @-logic language corresponds to φ (a non-terminal symbol, in formal languages terms) in the grammar as the following:

$$\varphi \longmapsto P \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid \varphi \Leftrightarrow \varphi \mid (\exists\alpha)\ \varphi \mid (\forall\alpha)\ \varphi$$
$$\varphi \longmapsto @s \cdot t[\varphi] \mid \phi \mid @s \cdot t[\ ]$$
$$\phi \longmapsto @s \cdot t[\![\varphi]\!]$$
$$\alpha \longmapsto x \mid y \mid ...$$

where φ is the starting symbol, P stands for a proposition or predicate, α denotes a quantified variable in $\mathbf{C}$, $s \in \mathbb{R}^3$ or $s \subseteq \mathbb{R}^3$ and $t \in \mathbb{R}$ or $t \subseteq \mathbb{R}$, depending on the focus of attention, points or sets, sometimes intervals. Let ϕ stand for semantics, a function symbol with domain in $\mathbf{C}$ and semantic range or image. Both my syntax and its intended semantics are simple and hence their formal definitions are not necessary here: if φ is a formula, then $@s \cdot t[\varphi]$ is a formula in the @-logic, where s indicates the place where φ holds, and t indicates the time when φ holds. A particular case is $@s \cdot t[\ ]$ which intuitively indicates that there is no assertion for space s and time t. This notation is capable of representing an empty data base or theory. Accordingly, by using a slightly different notation, I am able to express $@s \cdot t[\![\varphi]\!]$ as "the meaning of φ at place s and time t". This notation is used in semantics from this chapter on. Therefore, both this logic and this semantics provide the foundation for the present PhD thesis.

Note that this language implicitly introduces a conjunction between the represented space and time, for every space-time formula. In this PhD thesis or for the classical space-time @-logic, for any expression, everything happens intuitively in the same way as it would have happened in the classical logics, except that, here, there are variables of space and time, and that the expression, in question, of the classical logic is valid in the new context only.

There are two standard variables, namely *here* and *now* that can be used in space and time expressions, respectively. There is an alternative pair of variables with the same meanings in symbols, namely $\oplus$ and $\otimes$, respectively.

The notion of space-time orthogonality can be applied at more than one

level. Thus, in this chapter, I present a space-time deductive system that applies this pair of concepts at two or three levels:

- Derivation (deductive system)
- Logic (syntax)
- Object language

A calculus is embedded in this system, in such a way that I address the set of deductive rules as either. The present calculus is referred to as @-calculus. To design the system, I distinguish space and time at the logical level from space and time where and/or when the derivation carries out, and from the space and time referred to in nesting statements, and this means that the language permits references to other epistemic levels of space-time.

The time component is based on a flow that can be represented by reals or integer numbers, for instance, depending on one's purpose. As an example, if I set that $i \in \mathbb{N}$ represents the moment when I apply a logical rule, $i +_t 1$ corresponds to the next step, no matter when this step is performed in the real life. Thus, the structures for representing space and time can be two parameters of the @-logic. In this way, if $s \in \mathbb{N}$ is the current bus or train station, or the airport, $s +_s 1$ can be the next, for instance.

In this chapter, the space-time logic is based on a five-valued logic introduced here, without computational concerns such as decidability[300] and complexity[28], nor proof search in backwards[143]. This system combines natural deduction with sequent calculus although this yields redundancies: the exclusion rules of the deduction correspond to the left rules of that calculus as it is known. Briefly, my main concern is not the computerization of this system, but instead, to introduce a language that can be used in the formal semantics of mobile-code programming languages, as well as for philosophical purposes. That is why redundancy is preferred for human beings are free to reason and I have the obligation to define all possible logical paths for the sake

of generality. A book on automated theorem proving is [91]. In appendix A, I show a classical binary formalization of the space-time logic.

Two properties ought to be studied and addressed as further work, namely soundness and completeness. For the @-logic and the present deductive system are based on previous work, it is intuitive that they are both sound and complete, except in case of minor mistakes, and hence there is no need to prove soundness and completeness here in this introduction.

As part of my approach here, I observe that five values do not lead to a logic which is based on lattice or billatice, such as in [20]. As the title of the present PhD thesis indicates, I prefer to observe the *real world* before defining this logic ideally without any bias, and then should no mathematical tool is available for capturing the logic, the tool in question can be built. This is the way that I see mathematics in the foundations of computer science, although there are others.

Finally, this chapter is organized as follows: section 2.2 introduces the propositional logic. In the whole @-logic, the space and time models are parameters of the logic, i.e. one defines them for his or her specific purpose, as long as they are refinements of sets. Since the @-logic is used here e.g. in the system or calculus, two proper space and time models are defined here for the level of derivations. In subsection 2.2.2, I introduce the five truth values of the language, while in subsection 2.2.3 I present some motivating examples. Finally, in subsection 2.2.5 I consider uncertainty as well as analogy and belief. In terms of deductive system, section 2.3 introduces a pair of consequence relations, while section 2.4 directly deals with deduction. Section 2.6 briefly introduces one way to roughly represent matter on the move including resource. Finally, section 2.7 concludes the chapter.

2.2 A five-valued propositional logic

The whole logic which I am introducing here is based on a five-valued logic, with truth values represented in $C \stackrel{def}{=} \{uu, kk, ff, tt, ii\}$.

In the present piece of work, as well as the well known *ff* and *tt* Boolean values, *uu* stands for *unknown* or *undefined*, *kk* stands for (possibly) *known* (and *consistent*), while *ii* stands for *inconsistent*. I choose to work on inconsistency [27] for it often appears in contexts of the real world. In this way, mobile agents, for instance, ought to be able to decide and act even when it recognizes the presence of an inconsistent predicate. I consider that *ii* is stronger than *uu* and also stronger than *kk* in some kind of strict reasoning, but can be weaker than either in some forms of lazy computation. I shall explain the "known value", *kk*, in section 2.2.2 together with reasons for having five values. In advance here, it suffices to say that *kk* means "some other agent might know the truth value" to necessarily exclude the person who reasons. More generally and intuitively, the meanings of the values are the following:

- *ff*: (My partner and) I know that the value is $false$;
- *tt*: (My partner and) I know that the value is $true$;
- *kk*: I know that the value is either $true$ or $false$, but I do not know which of them. However, my partner might know which of them.
- *uu*: I do not know the Boolean value nor whether or not it is consistent. My partner neither;
- *ii*: (My partner and) I know that there is some inconsistency in the subject and, because of this, we two do not know whether the actual value is $true$ (nor whether the same actual value is $false$).

However, "my partner" here represents another agent, for example. Note that *kk* or *uu*, for instance, can abstractly represent uncertainty in some subset of domain $\mathbb{R}$.

2.2.1 Semantics, notions of space and time

In many articles on interval temporal logics, time is often represented by using real values where, as time goes by, *the present moment* corresponds to a value

which normally increases. For some applications, there can be branches along these lines to represent possible "futures". There are other approaches, such as in [257] that can also be useful for applications, including systems specification, and also to express natural sub-languages by using particular cases of modality. The underlying language here is both the natural language[131] and mathematics whenever it suits well. I do not adopt tense logics here. However, one can easily define some modal operators of tense logic.

As mentioned above, in this chapter I adopt a form of representing time by making use of a flow. More precisely, I shall define an algebra[96, 161, 210] that includes notions of time and space. Thus, I define $\mathbb{T} \doteq \mathbb{R}$ (or alternatively $\mathbb{T} \doteq \mathcal{P}(\mathbb{R})$, depending on the preference) as an infinite set for representing temporal moments. A temporal model is a structure of kind $M = \langle \mathbb{T}, <_t, \leq_t, =_t, \neq_t, \geq_t, >_t \rangle$ which is a flow of time with the present five-valued logical connectives, for each proposition p resulting in a value in C, the set of the five truth values. The semantics of the five-valued connectives are described in 2.2.2. There can be relational operators over time instants (the real numbers or so). Let $a, b, c, d \in \mathbb{T}$. Then,

- $a <_t b$ states *"a happens before b"*;
- $[a, b] <_t [c, d]$ states that *"the interval $[a, b]$ happens before the interval $[c, d]$"* (that is, $a \leq_t b <_t c \leq_t d$). Other 12 relations of Allen's interval temporal logic[11] can also be captured;
- $a =_t b$ states *"a and b happen at the same time"*;
- $a \leq_t b$ states $a <_t b \vee a =_t b$.

Let $Bool \stackrel{def}{=} \{tt, ff\}$. The operators defined in the algebra apply over $\mathbb{T}$. The signature for the above operators is $\mathbb{T} \times \mathbb{T} \longrightarrow Bool$.

Forms of representation for points in space are slightly more complex than in time. I normally consider $\mathbb{T} \doteq \mathbb{R}$ and $\mathbb{S} \doteq \mathbb{R}^3$ when I refer to these notions, and, as notation, $(\forall i)\ s_i \equiv \langle x_i, y_i, z_i \rangle$, with i an index. When more appropriate, I shall define $\mathbb{T} \doteq \mathcal{P}(\mathbb{R})$ and $\mathbb{S} \doteq \mathcal{P}(\mathbb{R}^3)$ instead. Letting $Bool \stackrel{def}{=} \{tt, ff\}$, the

relational operators in $\mathbb{S}$, namely $<_s\colon \mathbb{S}\times\mathbb{S}\longrightarrow Bool$, $=_s\colon \mathbb{S}\times\mathbb{S}\longrightarrow Bool$ and $\leq_s\colon \mathbb{S}\times\mathbb{S}\longrightarrow Bool$ can be defined as follows:

$$\begin{aligned} s_i <_s s_j \stackrel{def}{=}\ & \sqrt{x_i^2+y_i^2+z_i^2} < \sqrt{x_j^2+y_j^2+z_j^2} \vee \\ & \sqrt{x_i^2+y_i^2+z_i^2} = \sqrt{x_j^2+y_j^2+z_j^2} \wedge \\ & (x_i < x_j \vee x_i =_s x_j \wedge y_i < y_j \vee x_i = x_j \wedge y_i = y_j \wedge z_i < z_j) \end{aligned}$$

$$s_i =_s s_j \stackrel{def}{=} x_i = x_j \wedge y_i = y_j \wedge z_i = z_j$$

$$s_i \leq_s s_j \stackrel{def}{=} s_i <_s s_j \vee s_i =_s s_j$$

The precedence between coordinates can be, of course, different. Likewise, definition for $<_s$ depends on the application and can have many different ways.

Definition 2 *Let* $\mathbf{C}$ *be the set of all wffs of @-logic (its language) and* C *be the set of the five truth values* uu, kk, ff, tt, ii*. The corresponding space-time model* m *is a mapping of the type*

$$m:\ \mathbf{C}\longrightarrow C$$

together with an algebra (Notice that $\mathbf{C}$ and C are two different symbols). For instance:

$$\begin{aligned} M \stackrel{def}{=} \langle & \mathbb{S}, \mathbb{T}, \mathbf{C}, \\ & <_s, =_s, +_s, -_s, <_t, =_t, +_t, -_t, \\ & \cup, \cap, \setminus, ..., \\ & \neg, \ominus, \wedge, \&, \vee, \text{⅋}, ...\rangle \end{aligned}$$

with, informally, the following semantics: $M \models @s\cdot t[\varphi]$ for $s\in\mathbb{S}$ and $t\in\mathbb{T}$, meaning the value of $\varphi\in\mathbf{C}$ at place s and time t, where $\mathbf{C}$ corresponds to all formulae in the @-logic. I also interpret $@ >_s s\cdot t[\varphi]$ and $@s\cdot >_t t[\varphi]$, or alternatively $@s <_s \cdot t[\varphi]$ and $@s\cdot t <_t [\varphi]$ respectively, as shorthands for $(\exists s'\in\mathbb{S}, s' >_t s)\ @s'\cdot t[\varphi]$ and $(\exists t'\in\mathbb{T}, t' >_t t)\ @s\cdot t'[\varphi]$, respectively, including the binding of the respective variable, and the constraints that the

variables, namely s' and t', are chosen in such a way that they are not used in φ, or alternatively one renames variables in φ in a similar way. Accordingly, I interpret $@ <_s s \cdot t[\varphi]$ and $@s\cdot <_t t[\varphi]$, or alternatively $@s >_s \cdot t[\varphi]$ and $@s \cdot t >_t [\varphi]$ respectively, as shorthands for $(\exists s' \in \mathbb{S}, s' <_s s)\ @s' \cdot t[\varphi]$ and $(\exists t' \in \mathbb{T}, t' <_t t)\ @s \cdot t'[\varphi]$, respectively, including the same observation and constraints. That is, as well as the corresponding bindings, this relational form of a formula in the @-logic is not interpreted as universally quantified formula but instead as existentially quantified one.

Considering the hypothesis that the world was created at time B (the Big Bang, for instance) and that the future will always exist, I can express this consideration by the formulae $(\exists B \in \mathbb{T})\ (\forall t \in \mathbb{T})\ B \leq_t t$ and $(\forall t_1 \in \mathbb{T})\ (\exists t_2 \in \mathbb{T})\ t_1 <_t t_2$, respectively. Because I am adopting $\mathbb{T} \doteq \mathbb{R}$, $\mathbb{S} \doteq \mathbb{R}^3$ here, I should consider the continuum property:

$$(\forall t_0, t_1 \in \mathbb{T})\ t_0 <_t t_1 \leftrightarrow (\exists t \in \mathbb{T})\ t_0 <_t t \wedge t <_t t_1$$

$$(\forall s_0, s_1 \in \mathbb{S})\ s_0 <_s s_1 \leftrightarrow (\exists s \in \mathbb{S})\ s_0 <_s s \wedge s <_s s_1$$

And for the same reason, for applications where space and time are linear, both are also linear in the above algebra M. The following properties are also in the present algebra:

$$(\forall x, y \in \mathbb{S})\ x <_s y \ \underline{\mathrm{v}}\ x =_s y \ \underline{\mathrm{v}}\ x >_s y$$

$$(\forall x, y \in \mathbb{T})\ x <_t y \ \underline{\mathrm{v}}\ x =_t y \ \underline{\mathrm{v}}\ x >_t y$$

If I want to capture the idea of alternative pasts, x and y, in some theory atop the @-logic, I can write the following:

$$(\exists x, y \in \mathbb{T})\ \neg(y <_t x) \wedge \neg(y =_t x) \wedge \neg(x <_t y)$$

and/or the following, in some theory atop the @-logic, for representing alternative futures, x and y:

$$(\exists x, y \in \mathbb{T})\ \neg(y <_t x) \wedge \neg(y =_t x) \wedge \neg(x <_t y)$$

Or, alternatively and more succinctly, if one wants only one past

$$x <_t \otimes \wedge y <_t \otimes \rightarrow x <_t y \vee x =_t y \vee x >_t y$$

and/or no alternative futures

$$x >_t \otimes \wedge y >_t \otimes \rightarrow x <_t y \vee x =_t y \vee x >_t y$$

Clearly, this kind of choice depends not only on the application but also on the speaker's intention, and this does not necessarily include the philosophical issue of fate versus free will. For here, in this chapter, like the way in which one may represent space, it suffices to adopt only one straight line for representing time, i.e. $(\forall x, y \in \mathbb{T})\ x <_t y \vee x =_t y \vee x >_t y$ as a parameter whereas I keep the present logic general.

As regards the operators $+_s$ and $-_s$, they can be defined precisely for each theory, whereas $+_t$ and $-_t$ are normally interpreted as follows: $t_1 +_t t_2$ represents the sum of a duration t_2 to a time moment t_1 and the expression results in another time moment; and $t_1 -_t t_2$ represents the interval duration *equivalent* to the duration from a time moment t_2 to another moment t_1. The operators $+_t$ and $-_t$ can be defined in a different way elsewhere, but here these definitions suffice. For the deductive system that I define in section 2.4, for example, the temporal expression $t +_t 1$ simply indicates *the next step, if the inference is in the forward direction, otherwise the previous step.*

2.2.2 The five values

This section opens with the definition of the ontic and strongest five-valued equivalence in the following way:

$\doteq$	u	k	f	t	i
u	t	f	f	f	f
k	f	t	f	f	f
f	f	f	t	f	f
t	f	f	f	t	f
i	f	f	f	f	t

and similarly for discrimination: $A \neq B = \neg(A = B)$. One of the key subjects of part II of the present PhD dissertation is a programming language constant, denoted by uu. Here, in the @-logic, we are using kk and uu separately. For programming languages, uu suffices to deal with both situations with only one *unknown* value, whereas, at the application level, this distinction can be made by combining several language constructs. The same holds for the *inconsistent* value, which is another detailed form of *unknown*. In the @-logic, $=$ is the same as $\doteq$ and we use both interchangeable in this text.

In this section, I explain a hierarchy of veracity. There may be at least two kinds of *unknown*: "unknown because one does not know the value in the problem domain" (uu) or, alternatively, "unknown because the value is inconsistent" (ii). Thus, in comparison with other logics such as Belnap's, while ii may be interpreted as "the inconsistent value", the present uu and ii are not actually opposite values as uu is the opposite of kk. In fact, there are two views and sets of the connectives, ontic and epistemic. The present work is epistemic and the logic also deals with the concepts of true and false as usual. While $\{\neg, \wedge, \vee\}$ are more ontic operators, $\{\ominus, \&, \text{⅋}\}$ are more epistemic ones. To simplify the language during the presentation, I shall refer to them as "ontic" and "epistemic" connectives or operators, although this classification is relative, as well as I am using "connectives" and "operators" with the same meaning, for any reasoning. Thus, $kk \doteq \ominus kk$ and $uu \doteq \ominus uu$, i.e., both formulae are evaluated as *true* whereas $\neg kk \doteq uu$ and $kk \doteq \neg uu$ are valid and make use of the @-logic ontic negation.

For propagating inconsistency, I state $\ominus ii \leftrightarrow ii$, which means that, using sense of humor, "if a formula is inconsistent let alone its negation". The ontic negation of ii would be the value "consistent", which is absent from the logic, for I do not regard this consistency value as interesting for my purpose.

I introduce a few implications, e.g. $\looparrowright$ used above, the five-valued intuitionist logic implication, in section 2.3. For such logics, there exist many truth tables that can be interpreted as an implication, some stronger than others, and I also introduce the implication $\vdash$, which is a more general and weaker

one with all the necessary properties and, therefore, capable of supporting the axioms and rules entirely. Thus, in the above example, in the presence of $-$ symbol, at the outer level of this formula, I am only concerned about time while, at the inner level, I am only concerned about places.

I now introduce the connectives of the first set as follows:

a	¬a
u	k
k	u
f	t
t	f
i	i

∧	u	k	f	t	i
u	u	u	f	u	u
k	u	k	f	k	i
f	f	f	f	f	f
t	u	k	f	t	i
i	u	i	f	i	i

∨	u	k	f	t	i
u	u	k	u	t	i
k	k	k	k	t	k
f	u	k	f	t	i
t	t	t	t	t	t
i	i	k	i	t	i

a	(a)Ł
u	u
k	i
f	t
t	f
i	k

→	u	k	f	t	i
u	k	k	k	t	k
k	u	k	u	t	i
f	t	t	t	t	t
t	u	k	f	t	i
i	i	k	i	t	i

↔	u	k	f	t	i
u	u	u	k	u	i
k	u	k	u	k	i
f	k	u	f	f	i
t	u	k	f	t	i
i	i	i	i	i	i

The Ł negation (notation Ł after Łukasiewicz) is left for further work, for those who want to exploit a different feature of the @-logic, because, here, this logic is mainly defined for supporting the other chapters. Because of this, in the present piece of work, I do not make references to the latter negation, Ł, despite its relevance. $A \leftrightarrow B \stackrel{def}{=} (A \rightarrow B) \wedge (B \rightarrow A)$ is not usual, and this operator is not used in this chapter either. $\rightarrow$ is placed here only for a better comparison with the following set. In the present PhD thesis, $\Rightarrow$ and $\Leftrightarrow$ are used as classical connectives. For the purpose of any conflict of notation, all connectives have the same meaning for *Boolean* operands, I am only extending results in accordance with the five values. The connective $\wedge$ is commutative, associative and has a neutral element, tt. The $\vee$ is commutative, associative and has a neutral element, ff. For the equality connective that I shall define, both De Morgan's laws, $\neg(A \vee B) = \neg A \wedge \neg B$ and $\neg(A \wedge B) = \neg A \vee \neg B$, as well as both absorption laws, $A \vee (A \wedge B) = A$ and $A \wedge (A \vee B) = A$, hold in accordance with my automatic verifications. Furthermore, distributive laws: $A \vee B \wedge C = (A \vee B) \wedge (A \vee C)$ and $A \wedge (B \vee C) = A \wedge B \vee A \wedge C$ are valid.

Note that the present logic binds conjunctions tighter than disjunctions. The more epistemic connectives are the following:

a	⊖a
u	u
k	k
f	t
t	f
i	i

&	u	k	f	t	i
u	u	u	f	u	i
k	u	k	f	k	i
f	f	f	f	f	i
t	u	k	f	t	i
i	i	i	i	i	i

⅋	u	k	f	t	i
u	u	k	u	t	i
k	k	k	k	t	i
f	u	k	f	t	i
t	t	t	t	t	i
i	i	i	i	i	i

↠	u	k	f	t	i
u	u	k	u	t	i
k	k	k	k	t	i
f	t	t	t	t	i
t	u	k	f	t	i
i	i	i	i	i	i

↞↠	u	k	f	t	i
u	u	u	u	u	i
k	u	k	k	k	i
f	u	k	t	f	i
t	u	k	f	t	i
i	i	i	i	i	i

As the results of $A \wedge B$ and $A \vee B$ are the same as $A\&B$ and $A⅋B$, respectively, when $A \neq ii \ \wedge \ B \neq ii$, one can collapse both conjunctions and both disjunctions above in another four-valued logic by dropping ii and redefine a four-valued implication and equivalence, if we are sure that there is no inconsistency.

In the logic shown above, a possible interpretation for the operators is with respect to the knowledge on the operands of an arbitrary operation, typically in a programming language context. If one or more values are ff or tt, the connective gives the corresponding intuitive negation, as above. As an example, two of the tables above can be interpreted as permitting strict and lazy evaluations, if we are a little careful in order to avoid confusion. For instance, $kk \,\&\, uu$ can mean that, in a strict evaluation, the first operand is known and that the second one (or, alternatively, the same one) is completely unknown, whereas $kk \vee uu$ can mean the knowledge on the first operand value or no knowledge on the value of the second (or, alternatively, the same) operand in a lazy evaluation. Thus, in accordance with the tables, the first evaluation yields an unknown result whereas the second (lazy) evaluation yields a known result. In chapter 7, for example, I suggest that the specification of lazy and strict evalu-

ations should be present in parameters and functions definitions, instead of being a stable programming language feature. This idea motivates the existence of both approaches in a single context with uu. Here I consider that conjunction and disjunction are commutative connectives. There are other interpretations using these tables. The connective $\&$ is commutative, associative and has a neutral element, tt. The $⅋$ is commutative, associative and has a neutral element, ff. De Morgan's laws hold with the negations: $\neg(A ⅋ B) = \neg A \& \neg B$, $\neg(A \& B) = \neg A ⅋ \neg B$, $\ominus(A ⅋ B) = \ominus A \& \ominus B$, $\ominus(A \& B) = \ominus A ⅋ \ominus B$. Furthermore, $A ⅋ B \& C = (A ⅋ B) \& (A ⅋ C)$ and $A ⅋ (B \& C) = (A ⅋ B) \& (A ⅋ C)$ is one more important property. However, because my purpose is to propagate ii here, in contrast with the first scheme, $A ⅋ (A \& B) = A$ and $A \& (A ⅋ B) = A$ are not tautologies. While $\rightarrow$ and $\leftrightarrow$ are more ontic, whereas symbols such as $\twoheadrightarrow$ and $\twoheadleftarrow\!\!\!\!\twoheadrightarrow$ are more epistemic. While the ontic connectives can be seen as lazy, the epistemic connectives can be seen as strict. $A, B \in \{ff, tt, uu, kk, ii\}$, i.e. for two logical formulae or operands, the first implication can be defined as $A \rightarrow B = \neg A \vee B$ whereas $A \twoheadrightarrow B = \neg A ⅋ B$. Furthermore, $A \leftrightarrow B = ((A \rightarrow B) \wedge (B \rightarrow A))$ whereas $A \twoheadleftarrow\!\!\!\!\twoheadrightarrow B$, a very epistemic equivalence, cannot be defined in this brief way.

For comparison, I present the tables in Belnap four-valued logic. I present the tables below without implication and equivalence, for Belnap did not show them[30], and because of his work on entailment. His n value (none) corresponds to this uu value (u in my truth tables here), the b value (both) roughly corresponds to this kk value (k in my truth tables here). On the other hand, for helping comparisons, I add the i value to Belnap logic, and the usual properties are still valid between the three connectives, with some exceptions, e.g. $A \wedge ff = ff$ and $A \vee tt = tt$ no longer hold. I shall refer to the resulting five-valued scheme as Belnap-based five-valued logic. The tables become as follows:

a	¬a
u	k
k	u
f	t
t	f
i	i

∧	u	k	f	t	i
u	u	f	f	u	i
k	f	k	f	k	i
f	f	f	f	f	i
t	u	k	f	t	i
i	i	i	i	i	i

∨	u	k	f	t	i
u	u	t	u	t	i
k	t	k	k	t	i
f	u	k	f	t	i
t	t	t	t	t	i
i	i	i	i	i	i

Belnap Four-Valued Logic joined with *ii*

In [152], in chapter 2, Gupta and Belnap illustrate with schemes for two, three and four values. For the scheme with four values, they present the above conjunction but with the same negation as $\ominus$, except that I have one additional value, *ii*. Therefore, both the present $\neg$ and $\ominus$ are in fact relatively old connectives and exist since seventies, in the last century. Briefly, the key difference between my truth tables and Belnap's is $uu \wedge kk = ff$ in his tables, i.e. one difference between the @-logic and Belnap four-valued logic is that, while his $A \wedge B$ results in *ff* for A having value *uu* and B having value *kk*, this operation with these values results in *uu* in the @-logic. The other table results are exactly the same.

In the present five-valued logic, a formula is a *tautology* if and only if it results in *tt* for all models. Similarly, a formula is a *contradiction* if and only if it results in *ff* for all models. A formula is a *contingency* if and only if the following holds: there exists some model from which the formula produces value *tt* and there exists some model from which the formula produces value *ff*. The present classification is not mutually exclusive. Obviously, since I assume that the world is naturally consistent, a formula of the @-logic is said to be *consistent-valued* if and only if it does not result in *ii* in any model, and *unknown-valued* if and only if it results in *uu* in a model. A formula is *(possibly) known-valued* if and only if one of the following two holds: either it is a contingency and results in *kk* in the set of all models, or results in *kk* in all models. In this chapter, the number of previous occurrences of k in that complex and sequential truth values can be implicitly represented by nesting

space-time references, allowed by the logic grammar.

Even for an established logic, I consider that, if I initially define an equivalence connective independent from the implication, and define the implication as e.g. $A \Rightarrow B \stackrel{def}{=} \neg A \vee B \vee (A \Leftrightarrow B)$, a more general form is obtained. For a version of five-valued implication that has the properties of a classical logic, including the third-middle law, the truth table is the following:

$\multimap$	u	k	f	t	i
u	t	k	k	t	k
k	u	t	u	t	u
f	t	t	t	t	t
t	u	k	f	t	i
i	t	t	t	t	t

However, the formulae $A \wedge B \multimap A$, $A \& B \multimap A$, $A \multimap A \vee B$ and $A \multimap A \parr B$, like the implications introduced above, are not tautologies for $\multimap$. On the other hand, a deontic logic can be informally conceived in the following fashion: let φ be a formula of the @-logic and $\ominus\varphi$ denote obligation on φ, $\vartriangle\ \varphi$ denote permissibility on φ. Accordingly, $\neg \ominus \varphi \doteq \vartriangle\ \neg\varphi$ and $\ominus\neg\varphi \doteq\ \neg\ \vartriangle\ \varphi$. I than combine such modalities with the epistemic values, e.g. "one does not know φ if and only if he or she does not know whether φ is obligatory (or whether φ is permissible)." etc. Finally, the definitions in this paragraph suffices for modality, while other authors can extend the present set of rules with other more specific rules. Such modal operators are welcome to @-logic. While $\Diamond$ represents possibility, $\Box$ does not represent necessity in the real world, but instead sureness. The rules correspond to the implications $A \vdash \Diamond A$ and $\Box A \vdash A$ in Gentzen's style.

I shall introduce in a due course yet another implication symbol, $\looparrowright$, which has the properties of the intuitionistic logic, according to a well-known scheme that I reproduce below, with some adaptation. The @-logic conjunctions and disjunctions lead us to the second part of this dissertation, where *uu* and lazy evaluation are two of the subjects. Furthermore, the notion of *ii* might be interesting in other contexts, including when one speaks regarding space or

time, for instance, one can choose a hotel or a restaurant: one thinks "this one is suitable, that one is not so". $A \wedge \neg A$ is a particular case of $A \wedge B$ and, hence, not inconsistent for us as a principle, but inconsistency, and not necessarily falsity or contradiction, might temporarily appear in the inference, e.g. if someone refers to a larger place (or time) with the same set of hotels or restaurants. I shall identify inconsistency in the @-logic as follows: $B \Rightarrow \neg A$ and $B \Rightarrow A$. I represent that A is inconsistent in the @-logic as $A \doteq ii$.

2.2.3 Examples

In this section I present examples in the proposed logic language. I have also defined the semantics for the quantifiers, and I make use of them here, although I do not define them in the present PhD thesis. I use the @-logic for defining unexpected effect in chapter 3 and, in chapter 9, I introduce a logic programming language that supports reasoning under lack of information and inconsistency. The whole of part II is based on the *uu* value, which is part of this logic. Here, I attempt to demonstrate the suitability of the @-logic as a language for representing knowledge and belief.Thus, as an example, "day" and "night" are taken as vague variables, where ignorance and inconsistency may arise in some possible form of knowledge representation. Note that words whose first letter is lower-case can be variables or predicates, except space or time variables which can be in either case. Words whose first letter is upper-case can be constants or time or space variables.

The sun always shines everywhere:

$$@\forall \cdot \forall[shines(Sun)]$$

John is working now:

$$@\exists \cdot \otimes[isworking(John)]$$

Marry is now traveling from London to York:

$$(\exists p, f \in \mathbb{T})\ @London \cdot p <_t \otimes[is(Mary)] \wedge @\exists \cdot \otimes[travels(Mary)] \\ \wedge @York \cdot f >_t \otimes[\Diamond is(Mary)]$$

All women were girls at some time in the past:

$$@\exists \cdot \otimes[woman(x)] \rightarrow (\exists t <_t \otimes)\ @\exists \cdot t[girl(x)]$$

In addition to the pair of modal $\Box$ and $\Diamond$, one can define others[100]. Operators concerning space, such as:

$$[s]p \ = \ @\forall \cdot t[p] \qquad for\ some\ moment\ of\ time\ t.$$

$$\langle s\rangle p \ = \ \neg\ [s]\neg p$$

. . .and for time, such as:

$$[t]p \ = \ @s\cdot \geq_t \otimes[p] \qquad restricted\ to\ some\ particular\ place\ s.$$

$$\langle t\rangle p \ = \ \neg[t]\ \neg p$$

John is standing up:

$$@s\cdot <_t \otimes[issitdown(John)] \wedge \langle t\rangle isstandup(John)$$

It is day in Brazil iff it is night in Japan:

$$(\forall t)\ @Brazil \cdot t[day] \doteq @Japan \cdot t[night]$$

If someone orders a book, within five days he or she will receive it:

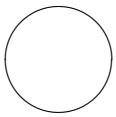
$$(\forall b, s)\ @s \cdot t[orders(b)] \rightarrow @s\cdot \leq_t t +_t 5d[receives(b)]$$

If someone loses his or her passport, he or she will not receive it within 15 days:

$$(\forall p, s, t)\ @\exists \cdot t[looses(s, p)] \rightarrow @\exists\cdot <_t t +_t 15d[\neg receives(s, p)]$$

2.2.4 Cycles: An Illustration

One can represent time in a somewhat subjective and flexible way using three dimensions. In particular, since time is seen as sets, angles over time are capable of helping represent (infinite) cyclical events. Hence, if one chooses a spiral to represent a view of time, one of the orthogonal projections of the spiral on the 2D plane is a circle. Thus time goes by cyclically and here time is represented using an angle. On the other hand, if one views a time line as a wave, one can perceive the infiniteness of it. If one wants to get a straight line one projects the wave uniformly in one of the dimensions.



Two Views of the Time Flow

The subjectivity and flexibility concerning this circle is that one can associate angles to the daily life. Each angle is a fraction of an hour, for example. In another form of representation, the same angle can be the same fraction of a year, for example. Thus one obtains different spirals, each of which represents the current focus of attention. In this way, the observer is included in the notion of time. The time variable represented by a single real number can be easily converted from/to this tuple.

As an idea that is orthogonal to alternative pasts and alternative futures, the above picture informally illustrates two different views for a common notion of time.

There can also be an alternative form of representation as follows: a pair of real numbers where the second element is the angle while the first element is the number of complete circles before that angle (the correspondence between one circle and one unit of temporal notion that belongs to the real world is implicit, e.g. one hour or week or year or other). Let π be, as usual, the ratio

of the circumference of any circle to its diameter. For $\langle \alpha, \beta \rangle$ where $\alpha \in \mathbb{Z}$ and $\beta \in [0, 2\pi)$, the straight line representation for the time is

$$t = 2\pi \times_t \alpha +_t \beta$$

and thus one feels free to choose when to use α and when to use β, or both (t).

2.2.5 Analogy, Belief and Uncertainty

Although computer scientists do not normally think on certainty factors while writing the semantics of a programming language or writing some other academic pieces of work, sometimes scientists want to express uncertainty over propositions. Thus, because the expressiveness of the @-logic is one of the present approaches, (as languages form one of the main subject matters in this PhD dissertation,) in this section, I introduce a notation that can capture analogy, induction, belief as well as some models of uncertainty. This notation is orthogonal to the rest of the logic and analogy and belief can be seen as orthogonal notions with respect to uncertainty, that is, these notions can help each other in the expressiveness of the language.

If φ is a formula in @-logic, then $\bigcirc\varphi$ is a formula in @-logic that means "φ is believed to be true". Properties: $\neg\bigcirc\varphi \doteq \bigcirc\neg\varphi$, $\ominus\bigcirc\varphi \doteq \bigcirc\ominus\varphi$. Knowledge is not dual to belief, and that pseudo duality does not hold among the known modal logics. Both $\neg \bigcirc \varphi \wedge \varphi$ and $\neg \bigcirc \varphi \wedge \neg\varphi$, as well as $\neg \bigcirc \varphi \wedge \bigcirc\neg\varphi$, are often acceptable in natural languages. Moreover, $\bigcirc \bigcirc \varphi$ seems to be more uncertain belief than $\bigcirc\varphi$, but that is subjective.

'Analogy' is both a feature and a process of reasoning based on similar features, when two objects are compared. Here I use the second meaning. Intuitively, I also understand that analogy is an instantaneous form of synthetic reasoning based on intuition or a kind of personal perception, and, because of this, I prefer not to define the semantics in a universal way. Instead, I only standardize its symbol and syntax in the @-logic.

Thus, let φ_1 and φ_2 be two formulae. Then, to express that φ_1 is analogous to φ_2 I write $\varphi_1 \bowtie \varphi_2$, or, alternatively for analogy is commutative, one may also write $\varphi_2 \bowtie \varphi_1$. $\bowtie$ is probably commutative, associative and distributive even over other operators. Although analogy is clearly commutative, I do not do this, for I consider that analogy is personal. I have the same attitude towards the following properties, usual for other operations:

Reflexive:	$\varphi_1 \bowtie \varphi_1$
Symmetric:	$\varphi_1 \bowtie \varphi_2 \rightarrow \varphi_2 \bowtie \varphi_1$
Transitive:	$\varphi_1 \bowtie \varphi_2 \wedge \varphi_2 \bowtie \varphi_3 \rightarrow \varphi_1 \bowtie \varphi_3$

It is reasonable to state that $\bowtie$ is reflexive and symmetric, but the transitive property does not universally hold. Because of this, I leave the issue of analogy half open, although analogy can be used together with uncertainty.

For induction, a sequence of formulae separated by comma with the "..." symbol as its suffix indicates induction over the formulae. This induction is not mathematical induction, but instead a non-valid form of reasoning that humans make use in their lives. If a student submits his or her PhD thesis to a panel, the panel will make the final decision based on this kind of induction as follows:

$$E_1, E_2, ... \rightarrow Result$$

where $E_i,\ i \in \mathbb{N}$, indicates any list of examiners and *Result* indicates the result that the student obtains. To stress the importance of induction, any practice of democracy is based on this form of reasoning, although mathematically invalid.

For uncertainty, I initially have to define the truth values as a subset of $\mathbb{R}$: in this chapter, $\mathbb{P} = \{x \in \mathbb{R} :\ -1.0 \leq x \leq +1.0\}$ plays this rôle. I use the Greek letter ψ to denote an uncertainty formula, i.e. a ψ-formula, e.g. $\psi(n\ \varphi)$ for some formula φ, where n is a pair $\langle x, y\rangle : \mathbb{P} \times \mathbb{P}$ where $x, y \in \mathbb{P}$ are the certainty thresholds, *false* and *true* respectively, for $\psi(n\ \varphi)$. The variables of

the pair $n = \langle x, y \rangle$ are individually denoted as $n.x$ and $n.y$ respectively. In any case, n and φ allow the valuation system to know whether $\psi(n\ \varphi)$ is *true* or *false*, for example, where $n.x \leq n.y$. In the present language, this result is in $\{ff, tt, uu, kk, ii\}$ in accordance with n, φ and a few rules. In advance, as a simplified and informal example, the formula $\psi(\langle 0, 0.5 \rangle\ \varphi)$ indicates that if φ has certainty degree greater than or equal to 0.5, the formula is interpreted as tt. On the other hand, if φ has certainty degree less than 0, the formula is interpreted as ff. Otherwise, its resulting value is uu. This certainty degree will be defined later in this section.

It is well known that, for more complex systems, the notation of n as well as the interval can be different, e.g. $[0, 1]$ is normally used for representing probability[230].

With respect to the nature of veracity, it is easy to observe, for example, that a car is German, and write the proposition "the car is German". Then another person can look at the car and easily state "that is true". This may happen because the nature of the information that can be represented is in a sense objective (all that one needs to do is to recognize the German company). Or, alternatively, the simplification that one makes on the information from the real world converts a naturally subjective piece of information, given the natural complexity of the world, to another objective piece of information in a corresponding manner, e.g. one can still ask "yes, but how much in that car is German?" even if the answer happens to be 100%, as the world can be seen as fuzzy[189]. In this piece of work, I do not state the uncertainty explicitly in any rule of the deductive system presented below, because that is a matter of vocabulary in that purely deductive context. As long as there is a mapping without loss of generality, the more simplified the language the clearer its concepts and constructs. Different values and different certainty thresholds can be assigned to different views of the same object, person etc in the real world, and that is because the truth value of φ results from the subjective nature of some object. The use of the ψ letter in the present chapter is for suggesting a more subjective nature of human factors. A solely fuzzy

view of the universe is in [189].

With respect to the present sub-model of uncertainty, a truth value is a pair of values, $\langle v, n\rangle$, where v is a value in $\{ff, tt, uu, kk, ii\}$ and n is another pair $\langle \alpha, \omega\rangle$ of values in $\mathbb{P}$, where α means minimum and ω means maximum, i.e. $\alpha \leq \omega$ (otherwise, the formula is said to be inconsistent). Thus, if V is some logical value on uncertainty, I simplify this notation by making use of $V.\alpha$ and $V.\omega$ to denote this pair of values. For a formula $\psi(m\ \varphi)$, if φ is evaluated and results in $\langle v, n\rangle$ here, or simply $\varphi = \langle v, n\rangle$ as a value of a model, the resulting value of the whole evaluation of $\psi(m\ \varphi)$ is one of the resulting values as follows:

- $\langle uu, n\rangle$ iff $n.\alpha \geq m.x \wedge n.\omega < m.y \wedge m.x < 0 \leq m.y \wedge n.\alpha \leq n.\omega$
- $\langle kk, n\rangle$ iff $(n.\alpha < m.x \vee n.\omega \geq m.y) \wedge m.x < 0 \leq m.y \wedge n.\alpha \leq n.\omega$
- $\langle ff, n\rangle$ iff $n.\omega < m.x \wedge m.x < 0 \leq m.y \wedge n.\alpha \leq n.\omega$
- $\langle tt, n\rangle$ iff $n.\alpha \geq m.y \wedge m.x < 0 \leq m.y \wedge n.\alpha \leq n.\omega$
- $\langle ii, \langle 1, -1\rangle\rangle$, otherwise.

For example, two main interpretations for uu, and two main interpretations for kk, can be made: in the first and second items above, the evaluation system regards the certainty degrees as public. For an interpretation with private certainty degrees, I simply consider the following cases:

- $\langle uu, \langle 0, 0\rangle\rangle$ iff $n.\alpha \geq m.x \wedge n.\omega < m.y \wedge m.x < 0 \leq m.y \wedge n.\alpha \leq n.\omega$
- $\langle kk, \langle 0, 0\rangle\rangle$ iff $(n.\alpha < m.x \vee n.\omega \geq m.y) \wedge m.x < 0 \leq m.y \wedge n.\alpha \leq n.\omega$

To allow the evaluation of φ as a pair I provide a notation for an uncertain formula. Thus, if φ is a (possible ψ-) formula, the evaluation of an expression $\varphi?\beta$ is in accordance with the following:

$$\langle v, \langle \beta \times \gamma.\alpha, \beta \times \gamma.\omega\rangle\rangle \text{ if } \varphi \text{ results in } \langle v, \gamma\rangle$$

Here I define some helpful constructs, bearing in mind that they are optional, by making use of the meta-level predicate of the form $@\forall \cdot t[\![\varphi]\!]$ to stand for "the meaning of the formula φ at time t", in this case in the type $C \times (\mathbb{P} \times \mathbb{P})$. I first formalize the construct $\varphi \ ? \ \beta$ (β is its certainty factor), as explained before:

$$\frac{@\forall \cdot t_1[\![\varphi]\!] = \langle v, \gamma \rangle}{@\forall \cdot t_2[\![\varphi \ ? \ \beta]\!] = \langle v, \langle \beta \times \gamma.\alpha, \beta \times \gamma.\omega \rangle \rangle}$$

where $t_1 <_t t_2$, and this condition also holds throughout this section.

I still need a few more words on implication. I support the idea that deductive logics are not capable of capturing a really relevance implication, despite Belnap's fantastic and historical work on relevance logic among others. I observe that words such as "because" have a conjunctive component, for example, if one says "Ann is using an umbrella because it is raining" is different from *if it rains Ann uses an umbrella*. In the former, the person who states indicates four important and conjunctive ideas, in addition to the context: that Ann is using umbrella, that it is raining, her *awareness* that it is raining, and her intention, which depends on cause-effect relationships. There are some natural-language conditionals that, to be regarded as valid, indicate that both the antecedent and consequent are $false$[201]. Moreover, in the real world and using the natural language, we normally have causes before their consequences, often with a particular time interval as a constraint between actions and/or events, and it may happen that the relationship is not clear. On the one hand, logics is a study of reasoning in one of its broadest senses. On the other hand, temporal relations, which might seem to be information outside logics, may be in the core of the meaning of conditional, implication and/or entailment in the same sense of logics. Therefore, what is the truth and logical meaning for such ordinary implications? I regard that probably exist some stronger implications in comparison to others. Uncertainty is a more general tool for defining connectives and the importance of certainty factors in natural-language inferences seems to be clear. Although I do not define implication as $\neg A \vee B$, the rules for implication between two uncertainty formulae can be as

follows:

As well as *modus ponens*, entailment is useful for applications such as expert systems. For being sufficiently general, it ought to have a certainty factor attached. As an example of interpretation,

$$\frac{\begin{array}{c} v_1, v_2 \in \{uu, kk, tt\} \quad @\forall \cdot t_1[\![A]\!] = \langle v_1, \beta_1 \rangle \\ @\forall \cdot t_1[\![A \xrightarrow{\beta_2} B]\!] = \langle tt, \langle +1.0, +1.0 \rangle \rangle \quad @\forall \cdot t_1[\![B]\!] = \langle v_2, \beta_3 \rangle \end{array}}{@\forall \cdot t_2[\![B]\!] = \langle v_1 \vee v_2, \langle max(\beta_1.\alpha \times \beta_2, \beta_3.\alpha), max(\beta_1.\omega \times \beta_2, \beta_3.\omega) \rangle \rangle}$$

and symmetrically for $\{uu, kk, ff\}$ (although with some redundancy) and, finally, one rule that propagates *ii*.

Another known operation may be called *composition*, which simulates some form of inductive reasoning. The operations above make use of the *min* and *max* functions, but there are contexts in which more than one formula together should increase certainty. The more the pieces of evidence, the greater the confidence should be. MYCIN[272] was the first expert system that used a similar idea. Let the 2-ary ϕ be the following auxiliary function:

$$\phi(x_1, x_2) = \begin{cases} x_1 + (1 - x_1) \times x_2 & \text{if } 0 \leq x_1, x_2 \leq +1, \\ x_1 + x_2 & \text{if } x_1 < 0 \wedge x_2 \geq 0 \vee x_1 \geq 0 \wedge x_2 < 0, \\ x_1 + (1 + x_1) \times x_2 & \text{if } -1 \leq x_1, x_2 < 0; \end{cases}$$

The definition and syntax are: φ_1 and φ_2 are two formulae if and only if $\{\psi\colon \varphi_1, \varphi_2\}$ is an uncertainty formula called composition. More generally, if φ_1 is a formula, then $\{\psi\colon \varphi_1, \Gamma\}$ is an uncertainty formula called composition, where Γ is a non-terminal symbol which denotes a sequence of formulae in the object (final) language. The semantics for composition is in accordance with the following rules using ϕ:

Given the A and B formulae,

$$\frac{@\forall \cdot t_1[\![A]\!] = \langle v_1, \beta_1 \rangle \quad @\forall \cdot t_1[\![B]\!] = \langle v_2, \beta_2 \rangle}{@\forall \cdot t_2[\![\ \{\psi\colon A, B\}\ ]\!] = \langle tt,\ \langle \phi(\beta_1.\alpha, \beta_2.\alpha), \phi(\beta_1.\omega, \beta_2.\omega) \rangle \rangle}$$

For more than two formulae,

$$\frac{@\forall \cdot t_1[\![A]\!] = \langle v_1, \beta_1 \rangle \quad @\forall \cdot t_1[\![\{\psi\colon \Gamma\}]\!] = \langle v_2, \beta_2 \rangle}{@\forall \cdot t_2[\![\ \{\psi\colon A, \Gamma\}\ ]\!] = \langle tt,\ \langle \phi(\beta_1.\alpha, \beta_2.\alpha), \phi(\beta_1.\omega, \beta_2.\omega) \rangle \rangle}$$

For all truth values in form $\langle v, \beta \rangle$, the certainty degree is simply obtained as follows:

$$?\langle v, \beta \rangle = f(\beta) = \frac{\beta.\alpha + \beta.\omega}{2}$$

where f is a locally defined symbol.

From now on I shall not make explicit use of uncertainty, except to introduce a number of examples in the next paragraphs. Instead, I assume that uncertainty can be implicit, as stated above, for all formulae. The axioms and rules of the deductive system or the @-calculus deal with the final value, i.e. I simply use the value in $\{uu, kk, \not{f}\not{f}, tt, ii\}$. That is, if $\psi(a\ A)$ results in $\langle v, \alpha \rangle$ where $v \in \{\not{f}\not{f}, tt, uu, kk, ii\}$, then v is simply used instead.

2.2.6 A Few More Examples

I present some examples that make use of knowledge representation with uncertainty.

If one tosses d (a one-dollar coin) on a table T, one obtains 50% of probability of getting head after 10 seconds:

$$@T \cdot t[toss(d)] \rightarrow @T \cdot t +_t 10s[get(head)?0]$$

If the patient had x but now the test of x presents a degree of certainty less than 0.1, the patient does not have x.

$$@s \cdot <_t \otimes[has(patient, x)] \wedge @s \cdot \otimes[diagnose(patient, x) = y \wedge y < 0.1] \\ \rightarrow \neg @s \cdot \geq_t \otimes[has(patient, x)]$$

As another example, for a person who knows the Boolean value of A but does not know the Boolean value of B, the formula $A \rightarrow B$ might result in tt or uu, say, in equal probabilities. Thus, we can write the following propositions:

- $val(A)$ indicates the value of A.
- $val(B)$ denotes the value of B.

- $imp(A, B, r)$ represents that the result from $A \rightarrow B$ is r.
- $prob(x, y)$ denotes formula x with probability y.

In the @-logic, we can write as follows:

$$val(A) = kk \wedge val(B) = uu \rightarrow$$
$$prob(imp(A, B, tt), 0.5) \wedge prob(imp(A, B, uu), 0.5)$$

The above example does not require uncertainty. However, observe the following: there is one diagnosis d and three symptoms s_1, s_2, s_3 with certainty factors 0.3, 0.6 and *unknown*, respectively, together with one rule with more details:

$$\langle tt, 0.3\rangle \rightarrow s_1 \qquad \langle tt, 0.6\rangle \rightarrow s_2 \qquad \langle uu, 0\rangle \rightarrow s_3$$
$$\psi(\langle -0.5, +0.9\rangle \ \ \{\psi\text{: } s_1?0.8,\ s_2?0.5,\ s_3?0.3\}\ ) \rightarrow d$$

2.3 Sequents

In [125], Gabbay states a scheme for a *linear logic* in Hilbert style and using the classical implication symbol:

Identity:	$A \Rightarrow A$
Commutativity:	$(A \Rightarrow (B \Rightarrow C)) \Rightarrow (B \Rightarrow (A \Rightarrow C))$
Prefixing:	$(C \Rightarrow A) \Rightarrow ((B \Rightarrow C) \Rightarrow (B \Rightarrow A))$
Suffixing:	$(C \Rightarrow A) \Rightarrow ((A \Rightarrow B) \Rightarrow (C \Rightarrow B))$

The *relevance logic*[17, 252] is based on the schema above plus

$(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$

The *intuitionistic logic* is based on the relevance logic scheme plus

$A \Rightarrow (B \Rightarrow A)$

and, finally, by adding the following schema

$((A \Rightarrow B) \Rightarrow A) \Rightarrow A.$

to the previous one, we obtain the schema for *classical logic*. The original approach on the @-logic was to choose the calculus for one of the above logics, and then correct problems. There are rules of inference that are specific for the values other than *true* and *false*.

In this way, if I let A, B be formulae, the axioms in the classical logic $(A \Rightarrow B) \Rightarrow ((A \Rightarrow (B \Rightarrow \mathit{ff})) \Rightarrow (A \Rightarrow \mathit{ff}))$ and $A \Rightarrow ((A \Rightarrow \mathit{ff}) \Rightarrow B)$ had not been tautologies if I would want to propose a paraconsistent and relevance logic[95], together with some extra rules in the calculus. The latter axiom is also the sixth axiom above, that one which complements the scheme for intuitionistic logic, also called axiom K, from combinatory logic. However, from the holistic view presented here as example, I prefer to accept it as a normal tautology.

In the present logic, there is not a single notion of contradiction as a primitive because there are five values (including *ff* and *ii*) and two different consequence relations: weak and strong. The proposal of a pair of two consequence relations is probably a novelty.

An implication with the properties of the above scheme for intuitionistic logic is the following:

$\not\hookrightarrow$	u	k	f	t	i
u	t	k	f	t	i
k	u	t	f	t	i
f	t	t	t	t	t
t	u	k	f	t	i
i	u	k	f	t	t

In advance, the $\not\hookrightarrow$ implication, as well as most implications, do not support entirely the @-calculus, only a few rules. However, I introduce a weaker implication for the present calculus that has the properties of the classical scheme as well as makes the rules tautologies with the first tables (i.e. the connectives $\{\neg, \wedge, \vee\}$. $\vdash$ is also tautological for the truth tables of the second

scheme if there is no inconsistency in the calculus presented later), for the principle of contraposition does not need to hold in the above schemes.

$\vdash$	u	k	f	t	i
u	t	t	t	t	t
k	t	t	t	t	t
f	t	t	t	t	t
t	u	k	f	t	i
i	t	t	t	t	t

and now principles such as *simplification*, $A \wedge B \vdash A$ or $A\&B \vdash A$ for example, hold with the schemes presented above. However, because of ii, $A \vdash A \parr B$ is not a tautology where $A = tt$ and $B = ii$, although both $(A \vdash B) \vdash (A \vdash B \parr C)$ and $(A \vdash C) \vdash (A \vdash B \parr C)$ are tautologies. Furthermore, $A \vdash A \vee B$, for example, is a tautology. Thus, for the present deductive system together with the @-calculus, I propose the above definition of $\vdash$.

Space can be seen as an abstract notion whereas a sequent in the calculus is also an expression of one of the following alternative forms

- $\Delta \vdash C$ iff both time and space can be represented implicitly, or
- $@s \cdot t_0[\Delta] \vdash @s \cdot t_1[C]$ or, equivalently, $@\Delta \cdot [t_0, t_1][C]$, as long as s is not used in $\Delta,\ C,\ t_0$ or t_1.

In the second form, s and $[t_0, t_1]$ explicitly state the space and time where and when the expression $\Delta \vdash C$ takes place, respectively. Thus, here, there is no mobile derivation. Further, I can simplify my notation here by writing $@s \cdot t[\Delta \vdash C]$ or $@\Delta \cdot t[C]$ that indicates that a derivation makes use of the assumption Δ; starts at some time in which I am not interested, and finishes at time t.

The @-logic has another consequence relation, $\Vdash$. While $\vdash$, also called weak sequent, yields weak proof, $\Vdash$ (the strong sequent) yields strong proof. $\vdash$ and $\Vdash$ yield derivations. Weak and strong proofs may form a pair of novelties. Thus, $(\Delta \Vdash A) \stackrel{def}{=} (\Delta \vdash A)\&\neg(\Delta \vdash \neg A)$.

2.4 Deduction

In this section I initially concentrate on derivations. Let A be a formula in the present language. As usual, a proof for A here is a tree of steps from a set of valid assumptions (the leaves) that leads us to conclude that the logical formula A is *true* (the root) for all values in any model. On the other hand, a *derivation* is a more general notion. It does not imply that the assumed formulae and the final formula are valid.

The @-calculus works as follows: there is a set of assumed formulae and one final formula, where each variable or formula can have one of the five values presented here: $\{ff, tt, uu, kk, ii\}$.

Deductions are based on axioms and rules of inference. A Rule is a meta-level implication and here I assume the @-logic $\vdash$ implication to follow their semantics. As usual, I also represent rules of inference by using fractional notation, where

$$\frac{@\Delta_1 \cdot t_1[C_1] \quad @\Delta_2 \cdot t_2[C_2] \quad ... \quad @\Delta_n \cdot t_n[C_n]}{@\Delta_1, \Delta_2, ..., \Delta_n \cdot t[C]}$$

corresponds to, at a higher level,

$$@\Delta_1 \cdot t_1[C_1] \wedge @\Delta_2 \cdot t_2[C_2] \wedge ... \wedge @\Delta_n \cdot t_n[C_n] \vdash @\Delta_1 \cup \Delta_2 \cup ..., \Delta_n \cdot t[C]$$

I use comma instead of the $\cup$ set operation as I use multi-sets.

Here, I introduce the properties of the present calculus.

Reflexivity: $@s{\cdot}t[\Delta, \{C\} \vdash C]$ which captures inclusion: $C \in \Delta \rightarrow (\forall s \in \mathbb{S}, t \in \mathbb{T})$ $@s \cdot t[\Delta \vdash C]$, another property.

Monotonicity:

$$\frac{@\Delta \cdot t[C]}{@\Delta, \Gamma \cdot t +_t 1[C]}$$

Premise Commutativity:

For any rule, since time is relevant here, I include an axiom for exchanging premises considering the evaluation time as follows:

$$\frac{@\Delta \cdot t[A] \quad @\Gamma \cdot t +_t 1/2[B]}{@\Delta, \Gamma \cdot t +_t 1[C]} \doteq \frac{@\Gamma \cdot t[B] \quad @\Delta \cdot t +_t 1/2[A]}{@\Delta, \Gamma \cdot t +_t 1[C]}$$

As usual, the premises are also associative.

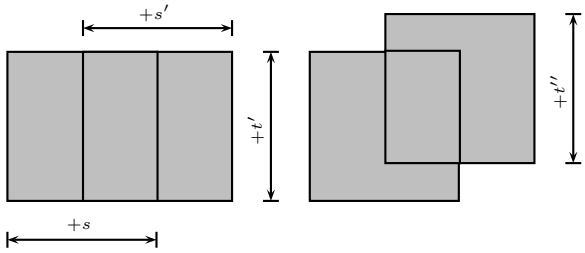
The cut rule is computationally redundant, as demonstrated in a theorem by Gentzen[6].

For the @-logic, I regard that derivations have place and also work as time goes by.

In both structural and logical rules, the time is placed explicit, every rule should be stated only as implication.

Negative and Positive Wholeness

In the following picture on the left, the rectangles will represent the scopes of two formulae, $@s \cdot t[A]$ and $@s' \cdot t[A]$ with the same time (ordinate) and a common place (abscissa). On the right, a more general situation: two formulae with a little common space and a little common time.



In natural language, space and time can be stated with or without wholeness. That is, both notions, space and time, can individually be interpreted as either universal (positive wholeness) or existential (negative wholeness), respectively. To indicate that the place (or time) indicates the negative wholeness, one places the symbol $-$ before the space (or time) expression. Accordingly, one places $+$ before the space (or time) expression to indicate that the notion is positive. Although $@s \cdot t[\varphi]$ without $-$ or $+$ is a valid expression in the @-logic, it does not allow deductions.

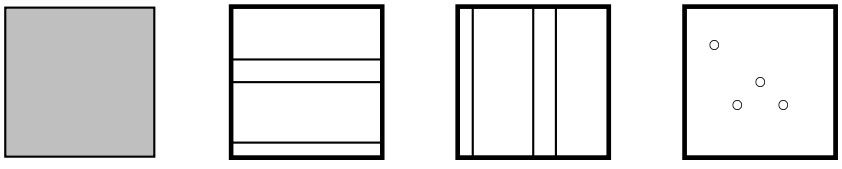
As an example of the positive wholeness, on the one hand, if one knows that in the *state* of São Paulo (in Brazil) citizens have the habits h_1, e.g. they like sports, formally $@+SP \cdot \exists[habits(p, h_1)]$, one is allowed to deduce that, in Santos city, which in turn is in the state of São Paulo, citizens have habits h_1,

that is, $@ + Santos \cdot \exists[habits(p, h_1)]$. On the other hand, if one understands that in the *state* of São Paulo citizens have the habits h_2, possibly the same ones but with *negative* wholeness, say $@-SP\cdot\exists[habits(p, h_2)]$, one is allowed to deduce that, in Brazil, citizens have habits h_2, say $@-Brazil\cdot\exists[habits(p, h_2)]$. The expression $@ - SP \cdot \exists[habits(p, h_2)]$ formally states that it is allowed to deduce that, in South America for instance, or even in America in some sense, there are citizens who have habits h_2. However, by using the last formula, one cannot deduce that, in São Paulo *city*, which is located inside São Paulo state, there are citizens who have habits h_2.

One may argue that, in a formula, the syntactical positions for space and for time are rigid and, therefore, if one wants to combine the positive and negative wholeness, one loses flexibility. That opinion is not a good standpoint, for space and time expressions themselves do not bind variables, and where both indicate that the inner formula is the scope. In this way, the following expression $(\forall s \in \mathbb{S})\ (\exists t \in \mathbb{T})\ @s \cdot t[\varphi]$ and the expression $(\exists t \in \mathbb{T})\ (\forall s \in \mathbb{S})\ @s \cdot t[\varphi]$ are not equivalent, while the expression $@s \cdot \exists[@\exists \cdot t[\varphi]] \doteq @\exists \cdot t[@s \cdot \exists[\varphi]]$ holds. In nesting formulae, the positive wholeness of space or time does not have priority over the negative wholeness and vice-versa. For example, depending on the interpretation, the formula $@ - s \cdot +t[@ + s \cdot -t[\varphi]]$ might not imply $@ + s \cdot +t[\varphi]$. Alternatively, one may want to write $@ - s \cdot +t[\varphi] \wedge @ + s \cdot -t[\varphi]$ instead or, better, $@ + s \cdot +t[\varphi] \wedge @ - s \cdot -t[\varphi]$.

The following picture demonstrates possible combinations of space and time wholeness, respectively,

$@ + s \cdot +t[A]$ $@ + s \cdot -t[A]$ $@ - s \cdot +t[A]$ $@ - s \cdot -t[A]$



2.4.1 Axioms

Identity:

$$\frac{}{@\{C\} \cdot t[C]}$$

The other axioms are defined in specific contexts.

2.4.2 Structural Rules

In this section, I present the structural rules of the @-calculus. Other space-time logics can be obtained by removing some of structural rules[254]. The structural rules almost in Gentzen's style are the following:

Hypothesis:

$$\frac{}{@s \cdot t[\Delta, \{C\} \vdash C]}\mathcal{Y}$$

Here, I use comma instead of the $\cup$ set operation as I use multi-sets. Therefore, this notation does not impose an order between two finite multi-sets of formulae, in such a way that there is no need for the so called *exchange* rule. The contraction rule is the following:

Contraction:

$$\frac{@\Delta, \{A, A\} \cdot t[C]}{@\Delta, \{A\} \cdot t +_t 1[C]}C\mathcal{L}$$

An essay on contraction is [124]. For proof theory without contraction, references, for example, are [49, 50, 143].

Weakening:

$$\frac{@\Delta \cdot t[C]}{@\Delta, \{A\} \cdot t +_t 1[C]}\mathcal{W}$$

Weakening explicitly expresses the monotonicity property.

2.4.3 Logical Rules

In this section, the logical rules are presented. The rules for $\ominus$, $\mathbin{\&}$, $\parr$ and $\twoheadrightarrow$ are not presented since the structures of the rules are equivalent to the rules for $\neg$, $\wedge$, $\vee$ and $\rightarrow$, respectively. More than this, rules with $\multimap$ are not

presented for the same reason with respect to $\rightarrow$. Therefore, I am going to present rules for the fragment $\{\neg,\ \wedge,\ \vee,\ \rightarrow\}$.

Deduction:

$$\frac{@\Delta \cdot t[A \rightarrow C]}{@\Delta, \{A\} \cdot t +_t 1[C]}\mathcal{D}\uparrow \qquad \frac{@\Delta, \{A\} \cdot t[C]}{@\Delta \cdot t +_t 1[A \rightarrow C]}\mathcal{D}\downarrow$$

Excluded 6th:

$$\frac{\neg@\Delta \cdot t[A \doteq kk] \quad \neg@\Delta \cdot t +_t 1/4[A \doteq ff] \quad \neg@\Delta \cdot t +_t 1/2[A \doteq tt] \quad \neg@\Delta \cdot t +_t 3/4[A \doteq ii]}{@\Delta \cdot t +_t 1[A \doteq uu]}$$

but all sequences of formulae in the premise can appear in any order.

Introduction:

The introduction rules are part of the deduction as well as the calculus.

Conjunction:

$$\frac{@\Delta, \{A\} \cdot t[C]}{@\Delta, \{A \wedge B\} \cdot t +_t 1[C]} \wedge\mathcal{IL}_1 \qquad \frac{@\Delta, \{B\} \cdot t[C]}{@\Delta, \{A \wedge B\} \cdot t +_t 1[C]} \wedge\mathcal{IL}_2$$

$$\frac{@\Delta \cdot t[A] \quad @\Gamma \cdot t +_t 1/2[B]}{@\Delta, \Gamma \cdot t +_t 1[A \wedge B]} \wedge\mathcal{IR}$$

Similarly, for inconsistent deduction:

$$\frac{@\Delta, \{A\} \cdot t[C]}{@\Delta, \{A \doteq ii\} \cdot t +_t 1[C]}\mathcal{I}ii\mathcal{L}_1 \qquad \frac{@\Delta, \{\neg A\} \cdot t[C]}{@\Delta, \{A \doteq ii\} \cdot t +_t 1[C]}\mathcal{I}ii\mathcal{L}_2$$

$$\frac{@\Delta \cdot t[A] \quad @\Gamma \cdot t +_t 1/2[\neg A]}{@\Delta, \Gamma \cdot t +_t 1[A \doteq ii]}\mathcal{I}ii\mathcal{R}$$

Disjunction:

$$\frac{@\Delta, \{A\} \cdot t[C] \qquad @\Gamma, \{B\} \cdot t +_t 1/2[C]}{@\Delta, \Gamma, \{A \vee B\} \cdot t +_t 1[C]} \vee\mathcal{IL}$$

$$\frac{@\Delta \cdot t[A]}{@\Delta \cdot t +_t 1[A \vee B]} \vee\mathcal{IR}_1 \qquad \frac{@\Delta \cdot t[B]}{@\Delta \cdot t +_t 1[A \vee B]} \vee\mathcal{IR}_2$$

Negation:

$$\frac{@\Delta \cdot t[A]}{@\Delta, \{\neg A\} \cdot t +_t 1[ii]} \neg\mathcal{IL} \quad \frac{@\Delta, \{A\} \cdot t[ff]}{@\Delta \cdot t +_t 1[\neg A]} \neg\mathcal{IR}$$

Implication:

$$\frac{@\Delta \cdot t[A] \quad @\Gamma, \{B\} \cdot t +_t 1/2[C]}{@\Delta, \Gamma, \{A \rightarrow B\} \cdot t +_t 1[C]} \rightarrow\mathcal{IL} \qquad \frac{@\Delta \cdot t[B]}{@\Delta \cdot t +_t 1[A \rightarrow B]} \rightarrow\mathcal{IR}$$

Elimination:

The elimination rules are part of the deductive system but not part of the calculus.

Conjunction:

$$\frac{@\Delta, \{A \wedge B\} \cdot t[C]}{@\Delta, \{A, B\} \cdot t +_t 1[C]} \wedge \mathcal{EL}$$

$$\frac{@\Delta \cdot t[A \wedge B]}{@\Delta \cdot t +_t 1[A]} \wedge \mathcal{ER}_1 \qquad \frac{@\Delta \cdot t[A \wedge B]}{@\Delta \cdot t +_t 1[B]} \wedge \mathcal{ER}_2$$

Similarly,

$$\frac{@\Delta, \{A \doteq ii\} \cdot t[C]}{@\Delta, \{A, \neg A\} \cdot t +_t 1[C]} ii\mathcal{EL}$$

$$\frac{@\Delta \cdot t[A \doteq ii]}{@\Delta \cdot t +_t 1[A]} ii\mathcal{ER}_1 \qquad \frac{@\Delta \cdot t[A \doteq ii]}{@\Delta \cdot t +_t 1[\neg A]} ii\mathcal{ER}_2$$

Disjunction:

$$\frac{@\Delta, \{A \vee B\} \cdot t[C]}{@\Delta, \{A\} \cdot t +_t 1[C]} \vee \mathcal{EL}_1 \qquad \frac{@\Delta, \{A \vee B\} \cdot t[C]}{@\Delta, \{B\} \cdot t +_t 1[C]} \vee \mathcal{EL}_2$$

$$\frac{@\Delta \cdot t[A \vee B] \quad @\Delta_1 \cdot t +_t 1/3[A \to C] \quad @\Delta_2 \cdot t +_t 2/3[B \to C]}{@\Delta, \Delta_1, \Delta_2 \cdot t +_t 1[C]} \vee \mathcal{ER}$$

and there also exist the following two rules:

$$\frac{@\Delta \cdot t[A \vee B]}{@\Delta \cdot t[A] \vee @\Delta \cdot t[B]} \vee \mathcal{E} \vee \mathcal{R} \qquad \frac{@\Delta \cdot t[A \vee B]}{@\Delta \cdot t[A] \parr @\Delta \cdot t[B]} \vee \mathcal{E} \parr \mathcal{R}$$

Negation:

$$\frac{@\Delta, \{\neg A\} \cdot t[ff]}{@\Delta \cdot t +_t 1[A]} \neg\mathcal{EL} \qquad \frac{@\Delta, \{A\} \cdot t[ff]}{@\Delta \cdot t +_t 1[\neg A]} \neg\mathcal{ER}$$

Implication:

$$\frac{@\Delta, \{A \to B\} \cdot t[C]}{@\Delta, \{B\} \cdot t +_t 1[C]} \to \mathcal{EL} \qquad \frac{@\Delta \cdot t[A] \quad @\Gamma \cdot t +_t 1/2[A \to C]}{@\Delta, \Gamma \cdot t +_t 1[C]} \to \mathcal{ER}$$

The left rule, above, is not part of the linear logic or relevance logic. The above right rule is what is often called *modus ponens*.

Space and Time:

From now on, for the following axioms and deductive rules, $s, s' \subset \mathbb{S}; t', t'' \subset \mathbb{T}$, while t ought to remain as before: $t \in \mathbb{T}$.

$\neg\mathcal{A}1 \quad \neg@ + s \cdot +t'[A] \doteq @ - s \cdot -t'[\neg A]$

$\neg\mathcal{A}2 \quad \neg@ + s \cdot -t'[A] \doteq @ - s \cdot +t'[\neg A]$

$\neg\mathcal{A}3 \quad \neg@ - s \cdot +t'[A] \doteq @ + s \cdot -t'[\neg A]$

$\neg\mathcal{A}4 \quad \neg@ - s \cdot -t'[A] \doteq @ + s \cdot +t'[\neg A]$

$\wedge\mathcal{AP} \quad @P \cdot Q[A] \wedge @P \cdot Q[B] \doteq @P \cdot Q[A \wedge B]$

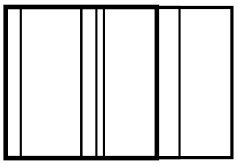
$\vee\mathcal{AP} \quad @P \cdot Q[A] \vee @P \cdot Q[B] \doteq @P \cdot Q[A \vee B]$

The above axioms can be used as two-way rules. I present the set of rules with an implicit correspondence between uu and empty space:

Space $\cup$ introduction

$$\frac{@\Delta, \{@ + s \cdot t'[A], @ + s' \cdot t'[A]\} \cdot t[C]}{@\Delta, \{@ + s \cup s' \cdot t'[A]\} \cdot t +_t 1[C]} + s \cup \mathcal{IL}$$

$$\frac{@\Delta \cdot t[@ + s \cdot +t'[A]] \quad @\Gamma \cdot t +_t 1/2[@ + s' \cdot +t'[A]]}{@\Delta, \Gamma \cdot t +_t 1[@ + s \cup s' \cdot +t'[A]]} + s \cup \mathcal{IR}$$



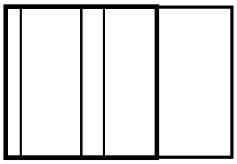
or

$$\frac{@\Delta, \{@ - s \cdot t'[A]\} \cdot t[C] \quad @\Gamma, \{@ - s' \cdot t'[A]\} \cdot t +_t 1/2[C]}{@\Delta, \Gamma, \{@ - s \cup s' \cdot t'[A]\} \cdot t +_t 1[C]} - s \cup \mathcal{IL}$$

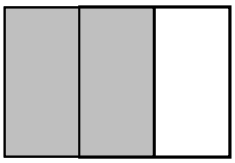
A rule for expansion:

$$\frac{@\Delta \cdot t[@ - s \cdot t'[A]]}{@\Delta \cdot t +_t 1[@ - s \cup s' \cdot t'[A]]} - s \cup \mathcal{IR}$$

and, for both $-s$ and $+s$, an alternative and interesting rule for



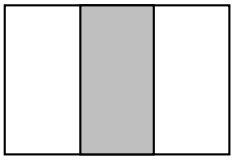
or

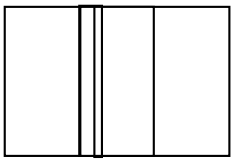


follows:

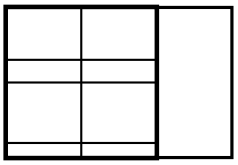
$$\frac{@\Delta \cdot t[@s \cdot + t'[A]]}{@\Delta \cdot t +_t 1[@ - s \cup s' \cdot + t'[A]]} s + \cup \mathcal{IR}$$

Space $\cap$ introduction

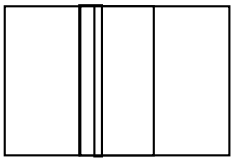




$$\frac{@\Delta, \{@ - s \cdot t'[A], @ - s' \cdot t'[A]\} \cdot t[C]}{@\Delta, \{@s \cap s' \cdot t'[A]\} \cdot t +_t 1[C]} s \cap \mathcal{IL}$$



$$\frac{@\Delta \cdot t[@ + s \cdot t'[A]]}{@\Delta, \Gamma \cdot t +_t 1[@ + s \cap s' \cdot t'[A]]} + s \cap \mathcal{IR}$$

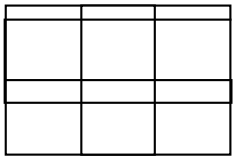


$$\frac{@\Delta, \{@ - s \cdot t'[A], @ - s' \cdot t'[A]\} \cdot t[C]}{@\Delta, \{@ - s \cap s' \cdot t'[A]\} \cdot t +_t 1[C]} - s \cap \mathcal{IL}$$

It can be somewhat interesting to observe that the formula $@ - s \cap s' \cdot t'[A]$ does not follow from $@ - s \cdot t'[A] \wedge @ - s' \cdot t'[A]$, but instead from $+ - sw\mathcal{R}$, defined later.

Space $\cup$ elimination

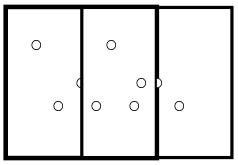
$$\frac{@\Delta, \{@ + s \cup s' \cdot +t'[A]\} \cdot t[C]}{@\Delta, \{@ + s \cdot +t'[A], @ + s' \cdot +t'[A]\} \cdot t +_t 1[C]} + s \cup \mathcal{EL}$$



$$\frac{@\Delta \cdot t[@ + s \cup s' \cdot t'[A]]}{@\Delta \cdot t +_t 1[@ + s \cdot t'[A]]} + s \cup \mathcal{ER}$$

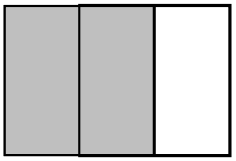
$$\frac{@\Delta, \{@ - s \cup s' \cdot t'[A]\} \cdot t[C]}{@\Delta, \{@ - s \cdot t'[A]\} \cdot t +_t 1[C]} - s \cup \mathcal{EL}$$

$$\frac{@\Delta, \{@ - s \cup s' \cdot +t'[A]\} \cdot t[C]}{@\Delta, \{@s \cdot +t'[A]\} \cdot t +_t 1[C]} s + \cup \mathcal{EL}$$

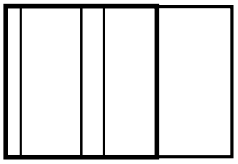


$$\frac{@\Delta \cdot t[@ - s \cup s' \cdot t'[A]]}{@\Delta \cdot t[@ - s \cdot t'[A]] \vee @\Delta \cdot t[@ - s' \cdot t'[A]]} - s \cup \mathcal{ER}$$

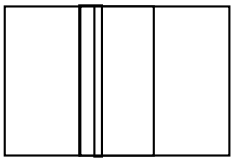
Space $\cap$ elimination:



$$\frac{@\Delta, \{@ + s \cap s' \cdot t'[A]\} \cdot t[C]}{@\Delta, \{@ + s \cdot t'[A]\} \cdot t +_t 1[C]} + s \cap \mathcal{EL}$$



$$\frac{@\Delta \cdot t[@ + s \cap s' \cdot t'[A]]}{@\Delta \cdot t +_t 1[@ - s \cdot t'[A]]} + -s \cap \mathcal{ER}$$



$$\frac{@\Delta \cdot t[@ - s \cap s' \cdot t'[A]]}{@\Delta \cdot t +_t 1[@ - s \cdot t'[A]]} - s \cap \mathcal{ER}$$

Space weakening - At the same time:

$$\frac{@\Delta, \{@ - s \cup s' \cdot t'[A]\} \cdot t[C]}{@\Delta, \{@ - s \cap s' \cdot t'[A]\} \cdot t +_t 1[C]} - sw\mathcal{L} \quad \frac{@\Delta \cdot t[@ - s \cap s' \cdot t'[A]]}{@\Delta \cdot t +_t 1[@ - s \cup s' \cdot t'[A]]} - sw\mathcal{R}$$

$$\frac{@\Delta, \{@ + s \cap s' \cdot t'[A]\} \cdot t[C]}{@\Delta, \{@ + s \cup s' \cdot t'[A]\} \cdot t +_t 1[C]} + sw\mathcal{L} \quad \frac{@\Delta \cdot t[@ + s \cup s' \cdot t'[A]]}{@\Delta \cdot t +_t 1[@ + s \cap s' \cdot t'[A]]} + sw\mathcal{R}$$

Different Spaces and Times

For the following rules, $s \neq s' \wedge t' \neq t''$.

$$\frac{@\Delta, \{@ + s \cdot -t'[A]\} \cdot t[C] \quad @\Gamma, \{@ + s \cdot -t'[A]\} \cdot t +_t 1/2[C]}{@\Delta, \Gamma, \{@ + s \cup s' \cdot -t' \cup t''[A]\} \cdot t +_t 1[C]} + s - t \cup \mathcal{IL}$$

$$\frac{@\Delta \cdot t[@ + s \cup s' \cdot -t' \cup t''[A]]}{@\Delta \cdot t +_t 1[@ + s \cdot -t'[A] \vee @ + s \cdot -t'[A]]} + s - t \cup \mathcal{ER}$$

$$\frac{@\Delta, \{@ - s \cdot +t'[A]\} \cdot t[C] \quad @\Gamma, \{@ - s \cdot +t'[A]\} \cdot t +_t 1/2[C]}{@\Delta, \Gamma, \{@ - s \cup s' \cdot +t' \cup t''[A]\} \cdot t +_t 1[C]} - s + t \cup \mathcal{IL}$$

$$\frac{@\Delta \cdot t[@ - s \cup s' \cdot +t' \cup t''[A]]}{@\Delta \cdot t +_t 1[@ - s \cdot +t'[A] \vee @ - s \cdot +t'[A]]} - s + t \cup \mathcal{ER}$$

$$\frac{@\Delta, \{@ + s \cdot +t'[A], @ + s' \cdot +t''[A]\} \cdot t[C]}{@\Delta, \{@ + s \cup s' \cdot +t' \cup t''[A]\} \cdot t +_t 1[C]} + st\mathcal{IL} \ \divideontimes$$

$$\frac{@\Delta \cdot t[@ + s \cdot +t'[A]] \quad @\Gamma \cdot t +_t 1/2[@ + s' \cdot +t''[A]]}{@\Delta, \Gamma \cdot t +_t 1[@ + s \cap s' \cdot +t' \cap t''[A]]} + st\mathcal{IR} \ \divideontimes$$

$$\frac{@\Delta, \{@ + s \cap s' \cdot +t' \cap t''[A]\} \cdot t[C]}{@\Delta, \{@ + s \cdot +t'[A], @ + s' \cdot +t''[A]\} \cdot t +_t 1[C]} + st\mathcal{EL} \divideontimes$$

$$\frac{@\Delta \cdot t[@ + s \cup s' \cdot +t' \cup t''[A]]}{@\Delta \cdot t +_t 1[@ + s \cdot +t'[A]]} + st\mathcal{ER} \ \divideontimes$$

although $@ + s \cup s' \cdot -t' \cup t''[A]$ does not imply $@ + s \cdot -t'[A]$ or $@ + s' \cdot -t''[A]$. Accordingly, $@ - s \cup s' \cdot +t' \cup t''[A]$ does not imply $@ - s \cdot +t'[A]$ or $@ - s \cdot +t'[A]$.

Time $\cup$ introduction

Since time is symmetric to space, it turns out that to repeat similar diagrams here are unnecessary. Instead, I place the rules without comments.

$$\frac{@\Delta, \{@s \cdot +t'[A], @s \cdot +t''[A]\} \cdot t[C]}{@\Delta, \{@s \cdot +t' \cup t''[A]\} \cdot t +_t 1[C]} + t \cup \mathcal{IL}$$

$$\frac{@\Delta \cdot t[@ + s \cdot +t'[A]] \quad @\Gamma \cdot t +_t 1/2[@ + s \cdot +t''[A]]}{@\Delta, \Gamma \cdot t +_t 1[@ + s \cdot +t' \cup t''[A]]} + t \cup \mathcal{IR}$$

$$\frac{@\Delta, \{@s \cdot -t'[A]\} \cdot t[C] \quad @\Gamma, \{@s \cdot -t''[A]\} \cdot t +_t 1/2[C]}{@\Delta, \Gamma, \{@s \cdot -t' \cup t''[A]\} \cdot t +_t 1[C]} - t \cup \mathcal{IL}$$

$$\frac{@\Delta \cdot t[@s \cdot -t'[A]]}{@\Delta \cdot t +_t 1[@s \cdot -t' \cup t''[A]]} - t \cup \mathcal{IR}$$

$$\frac{@\Delta \cdot t[@ + s \cdot t'[A]]}{@\Delta \cdot t +_t 1[@ + s \cdot -t' \cup t''[A]]} t + \cup \mathcal{IR}$$

Time $\cap$ introduction

$$\frac{@\Delta, \{@s \cdot -t'[A], @s \cdot -t''[A]\} \cdot t[C]}{@\Delta, \{@s \cdot t' \cap t''[A]\} \cdot t +_t 1[C]} t \cap \mathcal{IL}$$

$$\frac{@\Delta \cdot t[@s \cdot +t'[A]]}{@\Delta, \Gamma \cdot t +_t 1[@s \cdot +t' \cap t''[A]]} + t \cap \mathcal{IR}$$

$$\frac{@\Delta, \{@s \cdot -t'[A], @s \cdot -t''[A]\} \cdot t[C]}{@\Delta, \{@s \cdot -t' \cap t''[A]\} \cdot t +_t 1[C]} - t \cap \mathcal{IL}$$

Time $\cup$ elimination

$$\frac{@\Delta, \{@ + s \cdot +t' \cup t''[A]\} \cdot t[C]}{@\Delta, \{@ + s \cdot +t'[A], @ + s \cdot +t''[A]\} \cdot t +_t 1[C]} + t \cup \mathcal{EL}$$

$$\frac{@\Delta \cdot t[@s \cdot +t' \cup t''[A]]}{@\Delta \cdot t +_t 1[@s \cdot +t'[A]]} + t \cup \mathcal{ER}$$

$$\frac{@\Delta, \{@s \cdot -t' \cup t''[A]\} \cdot t[C]}{@\Delta, \{@s \cdot -t'[A]\} \cdot t +_t 1[C]} - t \cup \mathcal{EL}$$

$$\frac{@\Delta, \{@ + s \cdot -t' \cup t''[A]\} \cdot t[C]}{@\Delta, \{@ + s \cdot t'[A]\} \cdot t +_t 1[C]} t + \cup \mathcal{EL}$$

$$\frac{@\Delta \cdot t[@s \cdot -t' \cup t''[A]]}{@\Delta \cdot t[@s \cdot -t'[A]] \vee @\Delta \cdot t[@s \cdot -t''[A]]} - t \cup \mathcal{ER}$$

Time $\cap$ elimination

$$\frac{@\Delta, \{@s \cdot +t' \cap t''[A]\} \cdot t[C]}{@\Delta, \{@s \cdot +t'[A]\} \cdot t +_t 1[C]} + t \cap \mathcal{EL}$$

$$\frac{@\Delta \cdot t[@s \cdot +t' \cap t''[A]]}{@\Delta \cdot t +_t 1[@s \cdot -t'[A]]} + -t \cap \mathcal{ER}$$

$$\frac{@\Delta \cdot t[@s \cdot -t' \cap t''[A]]}{@\Delta \cdot t +_t 1[@s \cdot -t'[A]]} - t \cap \mathcal{ER}$$

Time Weakening - At the same place:

$$\frac{@\Delta, \{@s \cdot -t' \cup t''[A]\} \cdot t[C]}{@\Delta, \{@s \cdot -t' \cap t''[A]\} \cdot t +_t 1[C]} - tw\mathcal{L} \quad \frac{@\Delta \cdot t[@s \cdot -t' \cap t''[A]]}{@\Delta \cdot t +_t 1[@s \cdot -t' \cup t''[A]]} - tw\mathcal{R}$$

$$\frac{@\Delta, \{@s \cdot +t' \cap t''[A]\} \cdot t[C]}{@\Delta, \{@s \cdot +t' \cup t''[A]\} \cdot t +_t 1[C]} + tw\mathcal{L} \quad \frac{@\Delta \cdot t[@s \cdot +t' \cup t''[A]]}{@\Delta \cdot t +_t 1[@s \cdot +t' \cap t''[A]]} + tw\mathcal{R}$$

Space-Time $\forall$ and $\exists$:

$$\frac{@\Delta, \{@s \cdot t'[A]\} \cdot t[C]}{@\Delta, \{@\forall \cdot t'[A]\} \cdot t +_t 1[C]} \forall s\mathcal{IL} \quad \frac{@\Delta \cdot t[@\forall \cdot t'[A]]}{@\Delta \cdot t +_t 1[@s \cdot t'[A]]} \forall s\mathcal{ER}$$

$$\frac{@\Delta, \{@s \cdot t'[A]\} \cdot t[C]}{@\Delta, \{@s \cdot \forall[A]\} \cdot t +_t 1[C]} \forall t\mathcal{IL} \quad \frac{@\Delta \cdot t[@s \cdot \forall[A]]}{@\Delta \cdot t +_t 1[@s \cdot t'[A]]} \forall t\mathcal{ER}$$

$$\frac{@\Delta, \{@\exists \cdot t'[A]\} \cdot t[C]}{@\Delta, \{@s \cdot t'[A]\} \cdot t +_t 1[C]} \exists s\mathcal{EL} \quad \frac{@\Delta \cdot t[@s \cdot t'[A]]}{@\Delta \cdot t +_t 1[@\exists \cdot t'[A]]} \exists s\mathcal{IR}$$

$$\frac{@\Delta, \{@s \cdot \exists[A]\} \cdot t[C]}{@\Delta, \{@s \cdot t'[A]\} \cdot t +_t 1[C]} \exists t\mathcal{EL} \quad \frac{@\Delta \cdot t[@s \cdot t'[A]]}{@\Delta \cdot t +_t 1[@s \cdot \exists[A]]} \exists t\mathcal{IR}$$

Wholeness and Nesting Formulae:

$$\frac{@\Delta, \{@ - s \cdot t'[A]\} \cdot t[C]}{@\Delta, \{@ + s \cdot t'[A]\} \cdot t +_t 1[C]} - +sw\mathcal{L} \quad \frac{@\Delta \cdot t[@ + s \cdot t'[A]]}{@\Delta \cdot t +_t 1[@ - s \cdot t'[A]]} + -sw\mathcal{R}$$

$$\frac{@\Delta, \{@s \cdot -t'[A]\} \cdot t[C]}{@\Delta, \{@s \cdot +t'[A]\} \cdot t +_t 1[C]} - +tw\mathcal{L} \quad \frac{@\Delta \cdot t[@s \cdot +t'[A]]}{@\Delta \cdot t +_t 1[@s \cdot -t'[A]]} + -tw\mathcal{R}$$

Some theories on the @-logic might have rules for nesting formulae. I present some axioms only as an example:

$$\begin{aligned}
&++s\mathcal{N}: \quad @ + s \cdot t'[@ + s \cdot t'[A]] \doteq @ + s \cdot t'[A] \\
&--s\mathcal{N}: \quad @ - s \cdot t'[@ - s \cdot t'[A]] \doteq @ - s \cdot t'[A] \\
&++t\mathcal{N}: \quad @s \cdot +t'[@s \cdot +t'[A]] \doteq @s \cdot +t'[A] \\
&--t\mathcal{N}: \quad @s \cdot -t'[@s \cdot -t'[A]] \doteq @s \cdot -t'[A]
\end{aligned}$$

2.5 The Space-Time Operational Semantics

This section introduces an operational semantics useful for chapter 4, where I introduce a possibly more general notion of computation. This operational semantics with space and time is also helpful in a few places in part II of the present PhD thesis dissertation, more precisely, where I shall formalize programming language constructs in mobile agent systems. In this section, I illustrate an application of the present logic to operational semantics of programming languages. Informally, I adopt the following conventions:

- σ: a state of the computation, seen as a set.
- $\sigma(m/X)$: σ, in particular, $X = m \in \sigma$.
- $\langle \mathbf{true}, \sigma \rangle$: tt in state σ.
- $\langle \mathbf{false}, \sigma \rangle$: ff in state σ.

- $@s \cdot t[\![A]\!]$: the meaning of A (it requires that $A \doteq tt$) at place s and time t.
- n, m: two real numbers.
- ϵ: time spent to execute the referred to operation.
- $\rightsquigarrow$: evaluation of some operation and its meaning is obtained.

Traditionally, the formal semantics of programming languages do not require one to state the space-time components. For a semantic rule, it is assumed that the antecedents refer to executions before the execution of the statement that appears in the consequent in the rule. However, for mobile code languages, it becomes important to make it explicit that such statements do not change the locality while some other statements do change locality. Moreover, time becomes a major issue in global environments such as the Internet.

The @-logic can be used as a Space-Time semantics for more general purpose programming languages, or simply for those languages that support code mobility.

Here, I present an operational semantics of the well known *while* language, extracted from [318] with slight changes in addition to the present author's notation, to make it explicit that their constructs do not change locality.

2.5.1 The evaluation of Boolean expressions

$$@s \cdot t[\![\langle \mathbf{true}, \sigma \rangle]\!] \rightsquigarrow @s \cdot t +_t \epsilon[\![\mathbf{true}]\!] \qquad @s \cdot t[\![\langle \mathbf{false}, \sigma \rangle]\!] \rightsquigarrow @s \cdot t +_t \epsilon[\![\mathbf{false}]\!]$$

where ϵ is the time for executing the operation.

$$\frac{@s \cdot t_0[\![\langle a_0, \sigma \rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![n]\!] \quad @s \cdot t_0 +_t \epsilon_0[\![\langle a_1, \sigma \rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![m]\!] \quad n = m}{@s \cdot t_0[\![\langle a_0 = a_1, \sigma \rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\mathbf{true}]\!]}$$

Notice that the present author's notation allows the semantics to make explicit that a_0 is performed before a_1. For a parallel version, the rule would be slightly different from the one above.

$$\frac{@s \cdot t_0[\![\langle a_0, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![n]\!] \quad @s \cdot t_0 +_t \epsilon_0[\![\langle a_1, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![m]\!] \quad n \neq m}{@s \cdot t_0[\![\langle a_0 = a_1, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\mathbf{false}]\!]}$$

For the less than or equal to operator, there exist two extra rules such as:

$$\frac{@s \cdot t_0[\![\langle a_0, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![n]\!] \quad @s \cdot t_0 +_t \epsilon_0[\![\langle a_1, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![m]\!] \quad n \leq m}{@s \cdot t_0[\![\langle a_0 \leq a_1, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\mathbf{true}]\!]}$$

$$\frac{@s \cdot t_0[\![\langle a_0, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![n]\!] \quad @s \cdot t_0 +_t \epsilon_0[\![\langle a_1, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![m]\!] \quad n > m}{@s \cdot t_0[\![\langle a_0 \leq a_1, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\mathbf{false}]\!]}$$

More two rules for the negation:

$$\frac{@s \cdot t_0[\![\langle b, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![\mathbf{true}]\!]}{@s \cdot t_0[\![\langle \neg b, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\mathbf{false}]\!]}$$

$$\frac{@s \cdot t_0[\![\langle b, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![\mathbf{false}]\!]}{@s \cdot t_0[\![\langle \neg b, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\mathbf{true}]\!]}$$

Conjunction:

$$\frac{@s \cdot t_0[\![\langle b_0, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![\alpha]\!] \quad @s \cdot t_0 +_t \epsilon_0[\![\langle b_1, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\beta]\!]}{@s \cdot t_0[\![\langle b_0 \ \&\ b_1, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1 +_t \epsilon_2[\![\alpha \ \&\ \beta]\!]}$$

Disjunction:

$$\frac{@s \cdot t_0[\![\langle b_0, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![\alpha]\!] \quad @s \cdot t_0 +_t \epsilon_0[\![\langle b_1, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\beta]\!]}{@s \cdot t_0[\![\langle b_0 \vee b_1, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1 +_t \epsilon_2[\![\alpha \vee \beta]\!]}$$

2.5.2 The execution of commands

In this section, I present an operational semantics of the commands in the `while` language.

Atomic commands:

$$@s \cdot t[\![\langle \mathbf{skip}, \sigma\rangle]\!] \rightsquigarrow @s \cdot t +_t \epsilon[\![\sigma]\!]$$

$$\frac{@s \cdot t_0[\![\langle a, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![m]\!]}{@s \cdot t_0[\![\langle X := a, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\sigma(m/X)]\!]}$$

Sequencing:

$$\frac{@s \cdot t_0[\![\langle c_0, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![\sigma'']\!] \quad @s \cdot t_0 +_t \epsilon_0[\![\langle c_1, \sigma''\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\sigma']\!]}{@s \cdot t_0[\![\langle c_0; c_1, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\sigma']\!]}$$

Conditionals[73]:

$$\frac{@s \cdot t_0[\![\langle b, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![\mathbf{true}]\!] \quad @s \cdot t_0 +_t \epsilon_0[\![\langle c_1, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\sigma']\!]}{@s \cdot t_0[\![\langle \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\sigma']\!]}$$

$$\frac{@s \cdot t_0[\![\langle b, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![\mathbf{false}]\!] \quad @s \cdot t_0 +_t \epsilon_0[\![\langle c_2, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\sigma']\!]}{@s \cdot t_0[\![\langle \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\sigma']\!]}$$

While-loops:

$$\frac{@s \cdot t_0[\![\langle b, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon[\![\mathbf{false}]\!]}{@s \cdot t_0[\![\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon +_t \Delta t[\![\sigma]\!]}$$

$$\frac{@s \cdot t_0[\![\langle b, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0[\![\mathbf{true}]\!] \quad @s \cdot t_0 +_t \epsilon_0[\![\langle c, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\sigma'']\!] \quad @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1[\![\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma''\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1 +_t \epsilon_2[\![\sigma']\!]}{@s \cdot t_0[\![\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma\rangle]\!] \rightsquigarrow @s \cdot t_0 +_t \epsilon_0 +_t \epsilon_1 +_t \epsilon_2[\![\sigma']\!]}$$

2.6 Representing Mobility

Space, time and mobility are three orthogonal notions. Matter is the forth notion in order to complete the whole picture. I do not go deep into matter here whereas I leave material exaustive issues for others.

In this section, I demonstrate how the @-logic can capture material resources and mobility of objects up to some extent. The issue of matter is discussed in other contributions such as [40]. As already mentioned, mobility is one of the key concepts in the present thesis.

To achieve this purpose, let Obj denote the set of all physical objects in the real world. Therefore, the property of uniqueness of its elements can be written as:

$$\forall(o_1, o_2 \in Obj,\ t \in \mathbb{T},\ s_1, s_2 \in \mathbb{S}).\ (s_1 \not\subset s_2 \wedge s_1 \not\supset s_2) \wedge$$
$$(@ - s_1 \cdot +t[o_1] \wedge @ - s_2 \cdot +t[o_1] \rightarrow s_1 = s_2) \wedge$$
$$(@ + s_1 \cdot +t[o_1] \wedge @ + s_1 \cdot +t[o_2] \rightarrow o_1 = o_2)$$

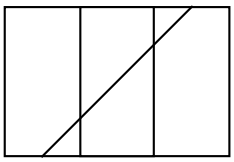
Thus, in the world described above, I can also represent mobility of an object slower than the speed of light in the following abstract way:

$$\nearrow (x, \langle p_0, t_0\rangle, \langle p_1, t_1\rangle) \stackrel{def}{=} @p_0 \cdot t_0[x] \wedge @p_1 \cdot t_1[x]$$

where $p_0, p_1 \in \mathbb{S}$, and x asserts the existence of some particular object, or, more abstractly,

$$\nearrow (x, p_0, p_1) \stackrel{def}{=} \exists t_0, t_1 \in \mathbb{T}.\ (t_0 <_t t_1) \wedge @p_0 \cdot t_0[x] \wedge @p_1 \cdot t_1[x]$$

or, given the above property, one can represent mobility graphically, from p_0 to p_1 during $t_1 -_t t_0$ interval, that is $@[p_0, p_1] \cdot [t_0, t_1][x]$, in accordance with the following space-time diagram:



which is in the same style of interpretation as the other space-time diagrams, presented above.

2.7 Conclusion

The combination of physical and psychological notions of mobility, space, time and knowledge is so common in the daily life that it is not difficult to find good examples of it: while an observer is sitting down in a café, she can see two men starting to shake hands, then a bus stops between her and the scene blocking her view. Then, she no longer knows when the men stop shaking hands, but can guess that it is not for so long, depending on the place of that scene and

her cultural background. The present calculus and deduction can be somewhat useful.

In this chapter, I presented a five-valued logic. For both sets of connectives, $\{\neg, \wedge, \vee\}$ and $\{\ominus, \&, \parr\}$, the proposal with *ii* does not mean that I presume that inconsistency is a natural notion, but instead that it appears in contexts with natural languages, for instance. After this chapter with *ii*, it might be a surprise to see that the underlying philosophical view is in accordance with a naturally consistent world. Another application for *ii* rests on the observation that agents ought to deal with conflicts.

When weighing up possibilities in any situation, one thinks carefully in order to make use of importance factors from premises to some associated conclusion. For performing such an inference, uncertainty is appropriate and provided in the @-logic. In this way, negative factors *tend* to refute hypotheses while positive factors *tend* to prove them. Thus, the result from reasoning means the average of the negative and positive results, and this is a significant skill for mobile agents.

The notions of space and time can be viewed as either abstract or physical. For example, the Pascal assignment instruction $a := a + b$ can be easily expressed as

$$@a \cdot t[value = a'] \& @b \cdot t +_t 1[value = b'] \& @a \cdot t +_t 2[value = a' + b']$$

where $t \in \mathbb{Z}$.

The grammar is expressive enough to represent sequents using sets of formulae as notion of space as exemplified here, such as $@\Delta \cdot t[C]$, although, for the @-calculus, one can make use of the form $@s \cdot t[\Delta \vdash C]$. I believe that the present calculus is sufficiently general and expressive to provide both notions, space and time, in a standard way. The calculus can be used by researchers who work with mobility and semantics of programming languages, for instance. Briefly, if one wants to extend the `while` language with the **flyto** command, equivalent to the **moveto** command as in [75], we can express this

in the present author's space-time operational semantics by using a rule similar to the following one:

$$\frac{t_0 <_t t_1}{@s_0 \cdot t_0[\![\langle \textbf{flyto } s_1, \sigma\rangle]\!] \rightsquigarrow @s_1 \cdot t_1[\![\langle done, \sigma\rangle]\!]}$$

where *done* indicates the next instruction. Or, in some cases, even

$$\frac{}{@s_0 \cdot t_0[\![\langle \textbf{flyto } s_1, \sigma\rangle]\!] \rightsquigarrow @s_1 \cdot (s_1 -_s s_0)/\omega[\![\langle done, \sigma\rangle]\!]}$$

where $-_s$ is linear and ω is the speed of light.

The @-logic will be useful in chapter 3, where I define a significant notion of unexpected effect, and in chapter 4, where I define another notion of computation. The @-logic can also be used in formalisms that are connected to mobility, including other chapters in the present PhD thesis dissertation. I mention uncertainty as part of the model in chapter 4 and, in chapter 8, I return to deal with uncertainty in the proposed hybrid programming paradigm.

Chapter 3

A Property of the Universal Turing Machine

Since Turing's work, there have been attempts to refute Church-Turing thesis by trying to discover some effectively calculable partial recursive function that does not correspond to any Turing-calculable partial recursive function. Here, I do not aim at refuting Church-Turing thesis but to mend it. One of the reasons for this is due to the fact that the statement of their thesis compares a formal notion, i.e. Turing-calculable partial recursive function, with an informal one, i.e. effectively calculable partial recursive function, but also because that thesis literally compares functions with functions.
However, in 2000, March, the present author discovered a significant example. Thus, in this chapter, I introduce this example which will lead to the following statement: The class of Turing machines is not equivalent (or, more in accordance with the discovered example, it is not isomorphic) to the class of Turing-calculable functions.

3.1 Introduction

Since 1930's, there has been very significant work into foundations of computer science, in theory of computation[198], category theory[193, 241] as well as in

recursive function[1] theory, functional programming[291] and other theoretical subjects. Since Alan Turing, we can make references to many good researchers, but, to date, nobody seems to have observed unexpected effects in computations of compositions of Turing machines on the tape. Perhaps those good researchers observed what I have called unexpected effects whereas I observed not only the unexpected effects but also the possible rôle of unexpected effects in one of the semantics of computation. Briefly and informally, let F and G be two Turing machines, and X be some input. I shall show that, if a programmer wants to form a composition such as $F(G(X))$ with both machines on the tape, we shall have to observe the dynamic possibility of G changing F for another Turing machine, say F'. Comparing F and G in this context, while G in this context does not prove anything other than $G(X)$, a Turing machine must prove something more than $F(G(X))$, i.e. a Turing machine must prove that the G calculation does not affect the F calculation. In this chapter, we shall see such details in a careful and more precise way. Variations on Universal Turing machines have been proposed[204] but not much work has been carried out in the only and traditional computability theory, which may cause the impression that the original computability theory is established. The fact is that, because of the parallel pioneering pieces of work by Church and Turing, we still have what is called Church-Turing thesis[299], or even called Church thesis[298].

In contrast, one of the main conclusions reached in this chapter is that programs do not necessarily correspond to functions. As a consequence of this,

[1]The standard use of the term *recursive function* includes the notion of function that does not explicitly contain the operation called recursion, as presented and used in the recursive function theory. I use both terms, i.e. function and recursive function, as referring to the same notion. Accordingly, the common use for the term *partial function* includes the concept and semantics of *total function*, but since not all functions in the present chapter are necessarily total, I simplify the language using both terms, function and partial function, as referring to the same notion. Thus, in this chapter, *functions* and *partial recursive functions* have the same meaning.

computations do not necessarily correspond to functions applications.

In section 3.2 I present Turing machines from the operational standpoint[2]. In section 3.3 I give interpretation of other notions used in the current chapter and, in section 3.4, I present my claims.

3.2 Turing machines

Perhaps the best definition that I have seen regarding non-deterministic Turing machine is in [207], and I reproduce it here with that notation, although I do not further use that notation:

Definition 3 *A Turing machine (TM) is an ordered system* $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ *where* Q *is a finite set of states,* Σ *is the input alphabet,* Γ *is the tape alphabet,* $\Gamma \cap Q = \emptyset$ *and* $\Sigma \subset \Gamma$*,* $q_0 \in Q$ *is the initial state,* $B \in \Gamma - \Sigma$ *is the blank symbol,* $F \subseteq Q$ *is the set of final states and* δ *is the transition function,*

$$\delta : Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

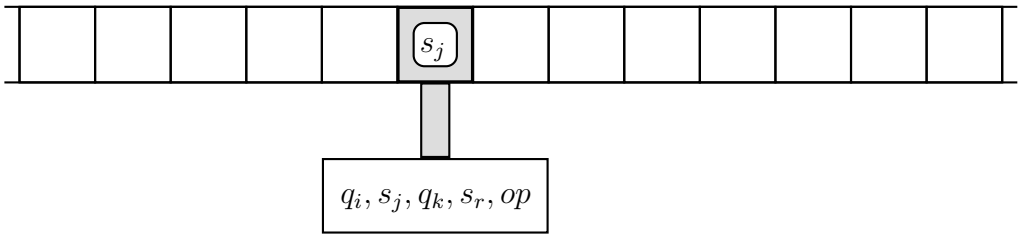
□

By defining the codomain of δ as above, that is $\mathcal{P}(Q \times \Gamma \times \{L, R\})$, the machine TM may be non-deterministic. Furthermore, it can be shown that non-deterministic Turing machines are equivalent to deterministic Turing machines. For a deterministic version, which I use in this thesis, δ can be redefined as $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$. In this section, however, as a matter of convenience for the present purpose, I briefly define Turing machines in a slightly different way, and this definition has also been used by other authors such as [42, 76, 171, 172, 198, 214, 234, 239, 274, 320] as well as Turing. The two

[2]In the present chapter, I prefer to use the term *operational* instead of the term *intentional*, for the former suggests that the corresponding scope is the artificial computation. Intention is a psychological notion more complex than operation. Although such words are well established in the computer science community since philosophy of mathematics, those from the community can also gradually reserve the latter for future use.

differences are that, here, a Turing machine program is a set of tuples (or a set of γ-transitions, as one may prefer to refer to) and there is one tape. As many of these simple transitions produce some effect on the tape, and this tape is shared by other programs, assuming Church-Turing thesis, what necessarily corresponds to a function is ultimately the whole sequence of Turing machines, including all machines that are interpreted, with the involved compositions if the Universal Turing machine is not outside the tape in order to guarantee that the computation is free from unexpected effects. Indeed, this is one of the results in this chapter: some assumptions have to be added to this assertion.

Without changing the notion, a Turing machine (TM) has also been defined as a 6-tuple, $M = (Q, \Sigma, T, P, q_0, F)$ where Q is a finite set of states $\{q_0, q_1, ..., q_n\}$, Σ is the alphabet which is a finite set of symbols $\{s_0, s_1, ..., s_m\}$, where s_0 is the blank symbol that I represent with $\circ$, $T \subseteq \Sigma \setminus \{\circ\}$ is the set of input symbols, P is the Turing machine program, $q_0 \in Q$ is what has been called initial state, and $F \subseteq Q$ is a set of final states of this Turing machine. According to Alan Turing's analogy, a Turing machine is supplied with an infinite tape divided into squares and one write/read head. Each tape square contains one symbol in Σ.



The diagram for Turing machine, with its tape squares

Given $\mathcal{O} = \{L, R, S, H\}$, the program P is a set of nn transitions or γ-transition functions, for $0 \leq ii \leq nn, \gamma_{ii} : Q \times S \longrightarrow Q \times S \times \mathcal{O}$ to which has been referred as a set of 5-tuples of form (q_i, s_j, q_k, s_r, op), $op \in \mathcal{O}$, such that each 5-tuple in P produces an effect that is described in the literature in the

following way:

As usual and in accordance with Alan Turing's original analogy using a tape, the write/read head is initially on the leftmost non-blank symbol, which means the leftmost symbol of the input for the Turing machine. The write/read head writes and reads symbols on the tape as it moves along the tape, one square to the right or to the left, depending on the current state of M and the current symbol, i.e. on which the write/read head is. For the purposes of this discussion, without loss of generality, I can set that, in the initial state q_0 of M, the write/read head is on the leftmost non-blank symbol of the tape. Thus, the machine works in the following way: for every simple[3] step of calculation, for some 5-tuple in P, if the machine is in state q_i and the write/read head is on the square which contains some symbol s_j, the machine substitutes s_r for s_j in the same square, substitute q_k for the current state, and perform one of the following actions:

- if $op = L$, the machine moves the write/read head one square to the left.
- if $op = R$, the machine moves the write/read head one square to the right.
- if $op = S$, the machine does not move the write/read head.
- if $op = H$, the machine halts.

Just a short note on notation used in the present chapter, from now on: given some Turing machine M as the first operand and some p which is represented here as a letter in the set of symbols $\{F, P, Q, T, \Sigma\}$ as the second operand, I define the meta-language infix operator $\star$ to denote a set from the

[3]In this chapter, I use the term *simple* step for both Turing machines and effective computation, although we only need to go a little more in detail here, as I am describing Turing machines. In a modern sense of computation, a simple step can migrate an agent, for instance, from one country to another because such an atomic operation is well defined in some way. Therefore, nowadays that simplicity is subjective and not relevant, and, because of this, I sometimes simply use the term *step* instead.

Turing machine, i.e. $M \star p$ denotes the corresponding set in M. For instance, $M \star P$ refers to the program of M and $M \star Q$ refers to the states of M.

As in the definition 3, a Turing machine does not have to have a tape or a write/read head. Here, we can define the alphabet $\Sigma_2 \stackrel{def}{=} \Sigma \cup \{\odot\}$ and the language $\mathcal{T}$ in Σ_2^* where $\odot \notin \Sigma$ is the symbol that indicates that the next symbol on the right of $\odot$ is under the write/read head. Thus, the tape is merely a string in Σ_2^*. To keep the comparison, the symbol $\odot$ occurs only once in the string. Those strings are infinite but only one finite part of them can contain non-blank symbols. Thus, so far the transition functions become: $0 \leq ij \leq nn,\ \gamma_{ij} : \Sigma_2^* \times Q \longrightarrow \Sigma_2^* \times Q$. Furthermore, because the state $q_i \in Q$ together with the symbol on the right of $\odot$ determine the transition function γ_{ij}, the function γ_{ij} can be defined as: for $\Sigma_3 = \Sigma_2 \cup Q, \Sigma_2 \cap Q = \emptyset :\ \gamma_{ij} : \Sigma_3^* \longrightarrow \Sigma_3^*$. A non-encoded Turing machine can be seen as a grammar.

I shall explain that, on a shared tape, the computation by the γ-transitions of one encoded Turing machine may affect the γ-transitions of other encoded Turing machines. Once Turing machines are encoded and placed on a tape, they are still separate entities while now they share the same tape. Therefore, in this chapter, I am not going to view Turing machines as a form of rewriting systems over strings, but, instead, from the operational standpoint, regarding compositions between Turing machines and so forth. Furthermore, regarding representation, I am going to use one tape and one write/read head, as explained in many books on computability theory since Alan Turing's papers, among others. I understand that this analogy with a physical machine is necessarily sound and helps the explanation.

In this chapter, I introduce an example and observe one feature that is present in one notion and absent from another. I use the notation $M[X]$ to stand for the computation of a Turing machine M that inputs x, where X is the representation of x. To describe the computation of a Universal[4] Tur-

[4]In this chapter, I use both "a Universal Turing machine" and "the Universal Turing machine" as meaning the same. I use the former when I want to refer to a class of Universal Turing machines that universally interpret Turing machines, and use the latter when I want

ing machine when simulating $M[X]$, I denote $U(M[X])$ instead of $U[M[X]]$. Accordingly, the functional composition $m(n(x))$ from two Turing machines, M and N, are denoted by $M(N[X])$. If there exists a Universal Turing machine interpreting this composition, that is $u(m(n(x)))$, I denote this situation by $U(M[N[X]])$. In this way, I use parentheses at the outermost level and brackets internally to make it clear that the former applying Turing machine is not represented on the tape, but instead outside the tape, while the latter is represented on the tape.

As notation for the computation of some composition, I make use of the up arrow symbol as a prefix. For instance, $\uparrow M(N[X])$ refers to the computation of $M(N[X])$ in the present piece of work.

Below, I define the Universal Turing machine with the characteristics that will be subsequently helpful in this chapter.

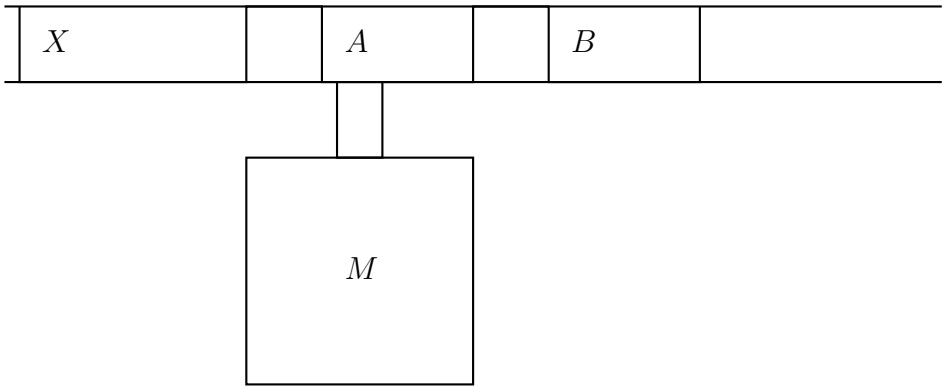
Definition 4 (Universal Turing machine) *Let U be a Turing machine. U is a Universal Turing machine if and only if, given any Turing machine $M : \mathbb{N} \longrightarrow \mathbb{N}$ encoded and placed on the tape, M corresponding to the Turing-computable function $m : \mathbb{N} \longrightarrow \mathbb{N}$, and given any input $X : \mathbb{N}$, which is some representation of the natural number $x \in \mathbb{N}$ on the tape, U calculates the value $m(x)$.*

□

To help the reader to understand this chapter, notice that the notion of function exists if and only if the notion of composition exists as a property (amongst the others). In this case, one cannot talk about functions without considering compositions as one of the fundamental properties of any related theory, in particular, theory of computation. Furthermore, let $f : D_1 \longrightarrow D_2$ and $g : D_2 \longrightarrow D_3$ be two total functions. There are at least the same number of Turing machines that assign to the values in D_1 the values in D_3 by implementing the $g(f(x))$ result (by concatenating corresponding Turing machines or by any other means), than the number of Turing-computable functions with

to stress the result, i.e. the interpretation itself. In both cases, my view is operational.

the same results by concatenating corresponding Turing machines, whether or not there is one such a Turing-computable function. Thus, this chapter shows that, more than that, depending on one hypothesis that I shall observe, there can be those compositions of the referred to Turing machines that yield values outside D_3. As an initial example of Turing machine composition, following a convention, the composition $M(B[A[X]])$ can be as in the following diagram:



As a usual example of arrangement (among others), one would previously establish that the input of a Turing machine is always on the left of its encoding and that they are separated by precisely one square. Furthermore, after having finished interpreting a Turing machine, M clears the interpreted Turing machine as well as its input, and then places the output of this interpretation in the correct place before starting interpreting another Turing machine, and so forth. Finally, in this example, one can establish that every Turing machine in the composition has its reserved space on the tape on the left of its input, but this is only *a priori* operational convention and, as such, does not prevent unexpected effects. The important point is that M has to prevent unexpected effects between Turing machines.

In category theory, for example, there are objects, arrows, functors, natural transformations, monads and many other concepts. Concepts are used to define more sophisticated concepts, and the essence or basis is sets and functions, as well as properties. Although the concept of monad[242] can be used to justify mathematically input/output in functional programming, that idea is far from refuting the present work on unexpected effects on Turing

machines. One of the reasons is that all those computable functions have signature $\mathbb{N} \longrightarrow \mathbb{N}$, sometimes represented with encoding the signature $\mathbb{N}^k \longrightarrow \mathbb{N}$ for some $k \in \mathbb{N}$. The Turing machine theory is relatively simple, and has been supported by programming. Likewise, the present refutation should be as such, using the same signature $\mathbb{N} \longrightarrow \mathbb{N}$ instead of more complex functional notions. Moreover, the meanings of the term "unexpected effect" are definitely not the same, with respect to functional programming. An essential difference between the concept of side-effects in input/output operations in functional programming and the term unexpected effect here is that, in the former case, the operation is programmed. In other words, there is control and intention. Here, unexpected effect during a computation is unpredictable from the point of view of the programs. Concisely, some unexpected effect appears during the computation (and possibly in books of computability theory) depending on the following:

1. The absolute positions of Turing machines on the tape;

2. Whether or not one Turing machine represented on the tape, by chance, affects another one represented on the tape;

3. Whether or not the Turing machine outside the tape avoids unexpected effects. Both alternatives indeed exist.

3.3 Some interpretations

In this section I present the interpretations of the notions used in my theorems. In a philosophical and insightful chapter [130] by Prof Galton, there is a discussion on different interpretations of Church-Turing thesis, including different assertions made for the thesis.

- **Model of Computation -** In this chapter, because I review a well-established model of computation only, it suffices to see computation as simply a sequence of (possibly infinite) simple steps that belong to

some well-established model of computation. Additionally, computation is carried out at some place and takes time, and this will be discussed in chapter 4. I do not redefine, intuitively or formally, *model of computation* here. Instead, for comparisons, I assume that λ-calculus is a model of effectively calculable functions. As suggested, a computation may be empty.

- **Effective Calculation -** For any natural number k, a function $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ is *effectively* calculable if and only if there exists some finite procedure p represented in h, that is, a unique function $ph(p) = h$, and a unique number-theoretic partial function $g\colon \mathbb{N}^k \longrightarrow \mathbb{N}$, such that given $x \in \mathbb{N}^k$, h calculates the value $g(x)$ according to p. By the way, in this chapter, up to date as usual, by *effective*, I do not mean that the model is finite.

  A more precise definition is the following: Let MC be the set of all models of effective computation, and P be the set of all simple steps regardless of the model. Let P^* denote the infinite set of all sequences of such simple steps, many of which are meaningless for they are steps from different models, and many of these sequences of steps are infinite, and let $p \in P^*$ denote a finite sequence of steps that form an effective procedure of some model $M \in MC$. Further, let $p[x]$ denote the computation of some procedure p given some value x, and $S \in P^*$ denote a (possibly infinite) sequence of simple steps carried out by the same computation. Let $S = p[x]$. Therefore, given the above notation, for any natural number $k > 0$, $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ is said to be an effectively calculable function if and only if there exists a number-theoretic function $g\colon \mathbb{N}^k \longrightarrow \mathbb{N}$, a function $f : MC \times P^* \times \mathbb{N}^k \longrightarrow \mathbb{N}$, and a unique function $\mu : MC \times P^* \times \mathbb{N}^k \longrightarrow \mathbb{N} \times MC \times P^*$ and also $\mu(f, M, S) = h$ (possibly many to one) for each (f, M, S, h), that denotes h, and for every $M \in MC$ and any $x \in \mathbb{N}^k$, there exists a sequence of simple steps $S \in P^*$, $S = p[x]$, and finally, the calculation of $h(x)$ or $f(M, S, x)$ is in accordance with

one of the following alternative cases:

- If the value of x is defined in g, the calculation follows a finite sequence of simple steps S (halts), and both $h(x)$ and $f(M, S, x)$ must result in the value of $g(x)$.
- If the value of x is not defined in g, a situation commonly and formally represented as $g(x) = \perp$, the calculation follows an infinite sequence of simple steps, i.e. $|S| = \infty$ and the application $h(x)$ or $f(M, S, x)$ never halts.

Roughly, $f(M, S, x) = g(x) = h(x)$, where $x \in \mathbb{N}^k$, $M \in MC$ and $S \in P^*$. More formally and using first-order predicate logic, and a predicate ec that states whether a function is effectively calculable,

$$\forall k \in \mathbb{N}.\ \forall f\colon MC \times P^* \times \mathbb{N}^k \longrightarrow \mathbb{N}.\ ec(f) \equiv$$
$$\exists!(g : \mathbb{N}^k \longrightarrow \mathbb{N}).\ \forall(M \in MC).\ \exists(S \in P^*).\ \forall(x \in \mathbb{N}^k).\ f(M, S, x) = g(x)$$

where g is unique since the quantifier indicates.

To allow composition of functions I can repeat the parameter M for the same model and sequences of steps S_0 and S_1, $f(M, S_1, f(M, S_0, x)) = g(g(x))$ holds here because $k = 1$. However, for $k > 1$, effectively calculable functions must accept and result in one encoding number, e.g. a Gödel number, and, by temporarily reducing k to 1, I do not loose generality. That is, every effectively calculable function must decode x before its calculation and, before resulting its final value, it must encode its result into a natural number. In the present chapter, after these definitions I shall use only functions with $k = 1$. The next interpretation is a particular case of this one.

- **Turing Computability and Calculable Functions -**

  For some partial and number-theoretic function $g : \mathbb{N}^k \longrightarrow \mathbb{N}$, some numbering interpretation (some codification previously established) $\mathcal{N}$

from a symbolic representation, for example including Gödel numbers, a partial function $t : MC \times P^* \times \mathbb{N}^k \longrightarrow \mathbb{N}$ is a Turing-calculable (or simply computable, or calculable) function if and only if there exists a Turing machine T such that, given the representation $r(x)$ (according to $\mathcal{N}$) of the value $x \in \mathbb{N}^k$ and once $r(x)$ is placed at some established place in the tape for input by that machine, and given a sequence of simple steps $S \in P^*$, which is the program of T, the calculation of $g(x)$ by t is in accordance with one of the following cases:

- if x is defined in g, the calculation of t halts in a finite number of simple steps S in a final state of T leaving the appropriate representation of $g(x)$, i.e. according to $\mathcal{N}$, at the established place of the tape for the result.
- if x is undefined in g, either the calculation of t does not halt or it halts in a state $s \notin M \star F$.

Intuitively and in a somewhat informal way, a predicate that states whether f is a Turing-calculable function (asserted using the predicate formula $tcf(t)$) is defined as follows: for all $\mathcal{N} \in I$,

$$\forall(k \in \mathbb{N}).\ \forall(t\colon MC \times P^* \times \mathbb{N}^k \longrightarrow \mathbb{N}).\ tcf(t) \equiv \exists!(g\colon \mathbb{N}^k \longrightarrow \mathbb{N}).$$
$$\exists(S \in P^*).\ \forall(x \in \mathbb{N}^k).\ \mathcal{N} \models t(TM, S, x) = g(x)$$

where TM is constant which is a value in the domain MC, I denotes the set of all possible codifications, and g is naturally unique.

From now on, I shall not write $\mathcal{N}$ in the formulae for the purpose of simplification, as I have already stated that an interpretation always exists and I simply assume that it is constant.

Thus, I shall demonstrate in proposition 2 that Turing machines do not necessarily entail functions. I demonstrate that unexpected effects introduce a problem that has a logical aspect and a conceptual one.

3.3.1 Unexpected effects

We can view *unexpected effect* as being the effect of some operation that a Turing machine (or, more generally, a program) can perform that can possibly change the representation of data or Turing machine on the tape (or of data or program in a common memory device) and therefore its result, in such a way that the effective effect depends on where the representations of the Turing machines are placed on the tape, as well as secondary conditions. If they are placed in different blocks in different runs, the results from these occurrences of computations are possibly different. The notion of unexpected effect is particularly important in any interpretation of Turing machine by a Universal Turing machine, for the latter has to guarantee the absence of unexpected effects, as we shall see. One can formally define the notion of unexpected effect in the @-logic as follows:

More abstractly, there exist two instants, $t_0 \neq t_1$, of time such that

$$\frac{TM,\ M \vdash @' \cdot t_0[\uparrow M[X] = m]\ \wedge\ TM,\ M \vdash @' \cdot t_1[\uparrow M[X] = n]\ \wedge\ m \neq n}{TM,\ M \vdash se(\uparrow M[X])}$$

The above definition is in terms of proof (if the results from the same Turing machine are different, I prove that there exists some unexpected effect). Or alternatively as follows:

$$se(M[X]) \equiv @TM \cdot' [@M \cdot t_0[M(X)] \neq @M \cdot t_1[M(X)] \wedge t_0 \neq_t t_1]$$

where M is a Turing machine, $M(X)$ here denotes the same as $M[X]$ and both denote M running for some input x, TM is the Turing machine model of computation, $@' \cdot t[M[X]]$ is the result from computation $M[X]$ at time t and under some set interpretation of encoding results, and $se(M[X])$ is the predicate that states the existence of unexpected effects in a Turing machine application $M[X]$.

Notice that unexpected effect is not a pure-mathematical notion regardless of its importance in computer science. Furthermore, M might produce or

receive unexpected effects. For the analysis in the next section, one can interpret that an unexpected effect indeed replaces one Turing machine by another one, while they are interpreted by a Turing machine that neither detects nor treats unexpected effects. From this perspective for unexpected effects, one alternatively interprets one unexpected effect as follows:

$$se(M[X]) \equiv @TM \cdot' [(\exists M, M')\ @s \cdot t_0[thereis(M)] \wedge @s \cdot t_1[thereis(M')] \\ \wedge M \neq M' \wedge M(x) \neq M'(x) \wedge t_0 <_t t_1]$$

where s denotes a place on the tape, $thereis(M)$ is a predicate that denotes the existence of the Turing machine M. Notice that the above formula makes use of the closed-world assumption.

3.4 A Refuting Example

In the following theorems of this chapter, I prefer to use both terms *computable* and *calculable* as meaning the same.

In my analysis, there exist two connections, namely, between Turing machines and Turing-computable functions, and between Turing-computable functions and effectively computable functions. I am at showing that the former one-to-one correspondence is broken, as well as their structural properties are not preserved because of the necessary notion of composition.

Example 1

Let U be a Universal Turing machine, and let G and H be two Turing machines that are placed on the tape. For my proofs, an example will suffice. Thus, as an example, H calculates double its input, which is encoded in binary and placed on the left of H, separated by, say, fifty blank symbols, initially. Thus, $H \star \Sigma = \{\circ, 0, 1\}$ and $H \star T = \{0, 1\}$. In this example, let $H \star P$ be

$(q_0, 0, q_2, \circ, R)$, $(q_0, 1, q_1, \circ, R)$,
$(q_2, 0, q_2, 0, R)$, $(q_2, 1, q_2, 1, R)$
$(q_2, \circ, q_4, 0, L)$, $(q_4, 0, q_4, 0, L)$, $(q_4, 1, q_4, 1, L)$,

$(q_4, \circ, q_5, 0, H)$,

$(q_1, 0, q_1, 0, R)$, $(q_1, 1, q_1, 1, R)$

$(q_1, \circ, q_3, 0, L)$, $(q_3, 0, q_3, 0, L)$, $(q_3, 1, q_3, 1, L)$,

$(q_3, \circ, q_5, 1, H)$.

Thus, $H \star Q = \{q_0, ..., q_5\}$, $n = 5$, $m = 2$, and $H \star F = \{q_5\}$.

Let $G = (H \star Q \cup \{q_{n+1}, ..., q_{n+3+w}\}, H \star \Sigma, H \star T, \Lambda \cup G \star P, q_0, H \star F \setminus \{q_0\})$ be defined as follows:

G moves the write/read head an arbitrary number w of squares to either left or right of $r(G)$, and, for some $s \in G \star \Sigma$, writes s on the tape. In this way, the Turing machine G is similar to H, except that G attempts to produce some unexpected effect on the tape. Without loss of generality, this can be done in the following way, assuming that I choose to move the write/read head to the right and that the write/read head is positioned at the leftmost square of $r(G)$ at q_0:

$\forall \gamma \in H \star P,\ \gamma \equiv (q_i, a, q_j, c, d) : i \neq 0 \wedge j \neq 0 \Rightarrow \gamma \in \Lambda.$

$\forall \gamma \in H \star P,\ \gamma \equiv (q_0, a, q_i, c, d) : \gamma \notin \Lambda \wedge (q_{n+1}, a, q_i, c, d) \in \Lambda.$

$\forall \gamma \in H \star P,\ \gamma \equiv (q_i, a, q_0, c, d) : \gamma \notin \Lambda \wedge (q_i, a, q_{n+1}, c, d) \in \Lambda.$

$q_0 \in H \star F \Rightarrow q_{n+1} \in G \star F.$

$(q_0, s_0, q_{n+2}, s_0, S) \in G \star P.$

$\forall s \in \Sigma \setminus \{s_0\} : (q_0, s, q_0, s, R) \in G \star P.$

For some arbitrary $w \in \mathbb{N}$:

$\forall i \in \mathbb{N}\ (i < w) : (q_{n+2+i}, s_0, q_{n+3+i}, s_0, R) \in G \star P.$

$(q_{n+2+w}, s_0, q_{n+3+w}, s_1, L) \in G \star P.$

$\forall i \in \mathbb{N}\ (i < w) : \forall j \in \mathbb{N}\ (1 \leq j \leq m) : (q_{n+3+i}, s_j, q_{n+3+w}, s_0, L) \in G \star P.$

$(q_{n+3+w}, s_0, q_{n+3+w}, s_0, L) \in G \star P.$

$\forall s \in \Sigma \setminus \{s_0\} : (q_{n+3+w}, s, q_{n+3+w}, s, L) \in G \star P.$

$(q_{n+3+w}, s_0, q_{n+1}, s_0, R) \in G \star P.$

Notice that, like H, G finally halts in s_5. That is, both $(q_4, \circ, q_5, 0, H)$ and $(q_3, \circ, q_5, 1, H)$ are in $G \star F$. Therefore, G is an algorithm.

Now, given $X : \mathbb{N}$, some Turing machine $F : P^* \times \mathbb{N} \longrightarrow \mathbb{N}$, and sequences of simple steps S and S_2, let $U(F[G[X]])$ be calculated: Suppose for the present example that F calculates the integer division modulus four of a number represented in binary digits (that is, F results in the two least significant digits). I define F as

$$F \star Q = \{q_0, q_{10}, q_{11}, q_{12}, q_{13}, q_{100}, q_{101}, q_{102}, q_{103}, q_{1000}, q_{1001}\}$$

and $F \star F = \{q_{1000}, q_{1001}\}$. Thus, $F \star P$ can be defined as follows:

$$(q_0, 0, q_{10}, \circ, R),\ (q_0, 1, q_{11}, \circ, R),$$
$$(q_{10}, 0, q_{10}, 0, R),\ (q_{10}, 1, q_{11}, 1, R),\ (q_{10}, \circ, q_{100}, \circ, L),$$
$$(q_{11}, 0, q_{12}, 0, R),\ (q_{11}, 1, q_{13}, 1, R),\ (q_{11}, \circ, q_{101}, \circ, L),$$
$$(q_{12}, 0, q_{10}, 0, R),\ (q_{12}, 1, q_{11}, 1, R),\ (q_{12}, \circ, q_{102}, \circ, L),$$
$$(q_{13}, 0, q_{12}, 0, R),\ (q_{13}, 1, q_{13}, 1, R),\ (q_{13}, \circ, q_{103}, \circ, L),$$
$$(q_{100}, 0, q_{100}, \circ, L),\ (q_{100}, 1, q_{100}, \circ, L),\ (q_{100}, \circ, q_{1000}, 0, R),$$
$$(q_{101}, 0, q_{101}, \circ, L),\ (q_{101}, 1, q_{101}, \circ, L),\ (q_{101}, \circ, q_{1001}, 0, R),$$
$$(q_{102}, 0, q_{102}, \circ, L),\ (q_{102}, 1, q_{102}, \circ, L),\ (q_{102}, \circ, q_{1000}, 1, R),$$
$$(q_{103}, 0, q_{103}, \circ, L),\ (q_{103}, 1, q_{103}, \circ, L),\ (q_{103}, \circ, q_{1001}, 1, R),$$
$$(q_{1000}, \circ, q_{1000}, 0, H),\ (q_{1001}, \circ, q_{1001}, 1, H).$$

and then, supposing $x = 93$, we obtain the following situation in q_0:

$$\text{tape starts here} \longrightarrow |1011101 \circ ...r(G) \circ ...r(F) \circ ...$$
$$\diamond$$

where $\diamond$ is the write/read head.

Because some simple steps of calculation of G might modify the representation of any Turing machine placed on the tape, including of F, we could obtain $U(F[G[X]]) \neq U(F[H[X]])$ from the calculation. The programmer who writes F does not have prior knowledge on G nor H. That is, G might change the representation of F if U allowed this. From the alternative view for unexpected effects, a computation could start as $U(F[G[X]])$ and finished resulting in $U(F'[G[X]])$ since G might change the Turing machine F in such a way that it would become F', if U allowed G to do so.

□

Definition 5 *For this chapter, let $k \in \mathbb{N}$, $k \geq 0$, $X : \mathbb{N}$ be some input, and $k+1$ Turing machines $F_k : \mathbb{N} \longrightarrow \mathbb{N}$. For any $k > 0$, a ($k-$level) Turing-machine composition is a composition of $k+1$ Turing machines $F_k[F_{k-1}[...[F_0[X]]...]]$. For any $0 < i \leq k$, F_i does not read or manipulate any Turing machine other than F_{i-1}.*

Lemma 1 (Universal Interpretation) *For any $k \in \mathbb{N}$, for any representation $X : \mathbb{N}$ on the tape, and for any Turing machines $F_0, F_1, ..., F_k$, let $F_k[F_{k-1}[...[F_0[X]]...]]$ be a k-level Turing-machine composition. Then, the Universal Turing machine is capable of reading the Turing machines $F_0, F_1, ..., F_k$.*

[Proof:] To calculate any $U(F_k[F_{k-1}[...[F_0[X]]...]])$, U interprets the operations of some of the involved Turing machines, i.e. some of $F_0, F_1, ..., F_k$, by following either lazy or strict evaluation.

□

Lemma 2 *Let $X : \mathbb{N}$, U be the Universal Turing machine, $M_0, ..., M_k : \mathbb{N} \longrightarrow \mathbb{N}$ be $k+1$ Turing machines, and $U(M_k[M_{k-1}[...[M_0[X]]...]])$ be a k-level Turing-machine composition where $k > 0$. There exists a non-empty set of transition functions in the Universal Turing machine that guarantees absence of any unexpected effect at any level $i \leq k$ in Turing-machine compositions.*

By example 1, a Universal Turing machine has to get round the problem of unexpected effects. In this chapter, the way is not important, but it may

be done by manipulating the tape configuration whenever the calculation of a Turing machine tries to modify another machine on the tape. That is, for all sequences of steps, U must always guarantee $\forall F, G, H : \mathbb{N} \longrightarrow \mathbb{N}, \forall X : \mathbb{N}, U(F[G[X]]) = U(F[H[X]])$. Therefore, since the programmable part of a Turing machine is in its set of transitions, there exists a non-empty set of transitions $\mathcal{S} \subset U \star P$ that can solve this problem of unexpected effects.

□

Theorem 1 *The class of Turing machines is not isomorphic to the class of effectively computable partial recursive functions. Furthermore, neither the former is necessarily equivalent to the latter, e.g. two Turing machines can correspond to the same function, nor all structural properties of the class of Turing machines correspond to the structural properties of the class of effectively computable functions with respect to the notion of composition.*

[Proof:] By lemma 2, there exists a non-empty set of transition functions $\mathcal{S} \subset U \star P$ that can solve the problem of unexpected effects. Now, let $U(U[G[X]])$ be calculated, from which the reader obtains the following situation in $U \star q_0$ and in $G \star q_0$:

tape starts here $\longrightarrow |1011101 \circ ...r(G) \circ ...r(U) \circ ...$

$\diamond$

and the final situation in $G \star F$ containing the double value, 186, is

tape starts here $\longrightarrow |10111010 \circ ...r(G) \circ ...r(U) \circ ...$

$\diamond$

although solution $\mathcal{S}$ might move the absolute positions of $r(G)$ and $r(U)$, and hence changing the tape configuration. Thus, the computation of $G(X)$ is represented as follows:

$$
\begin{array}{lllllll}
q_0 1011101\circ & \longrightarrow & \circ q_1 011101\circ & \longrightarrow & \circ 0 q_1 11101\circ & \longrightarrow \\
\circ 01 q_1 1101\circ & \longrightarrow & \circ 011 q_1 101\circ & \longrightarrow & \circ 0111 q_1 01\circ & \longrightarrow \\
\circ 01110 q_1 1\circ & \longrightarrow & \circ 011101 q_1\circ & \longrightarrow & \circ 01110 q_3 10 & \longrightarrow \\
\circ 0111 q_3 010\circ & \longrightarrow & \circ 011 q_3 1010\circ & \longrightarrow & \circ 01 q_3 11010\circ & \longrightarrow \\
\circ 0 q_3 111010\circ & \longrightarrow & \circ q_3 0111010\circ & \longrightarrow & q_3 \circ 0111010\circ & \longrightarrow \\
q_5 10111010 \circ . & & & & & &
\end{array}
$$

Assuming that there is no unexpected effects in the above computation. Then let two Turing machines, U and V, exist such that, except for the possibility of unexpected effects, U and V produce the same output: The only difference is that U contains $\mathcal{S}$ and calculates $U(U[G[X]])$, and V does not contain $\mathcal{S}$ and might calculate $V(V[G[X]])$ or $V(U[G[X]])$. As a possible example, V may sometimes calculate $U(U[G[X]])$ and sometimes not, depending on the physical places where V and G rest on the tape. Assuming that the class of Turing machines necessarily corresponds to the class of effectively computable functions, for later contradiction (although my Example 1 above clearly applies to any model based on functions), I can choose λ-calculus, defined by Church himself, as a functional model of effective calculability, denoted here by $\lambda{-}calculus \in MC$. Clearly, a simple case by case analysis demonstrates that parameters in λ-calculi cannot modify the operations of other functions (nor are able to replace a function application by another one). That is, no λ-calculi operations, namely $\{\beta$-reduction, α-conversion, η-conversion$\}$ and higher-order function application, are capable of doing this at all, as λ-expressions are always well formed. The same is valid for any functional model. Thus, let $sef_u, sef_v, sef_g \colon MC \times P^* \times \mathbb{N} \longrightarrow \mathbb{N}$ be the effectively computable functions which are supposed to correspond to U, V and G, respectively, and their corresponding sequences of steps S_u, S_v and S_g. The three sequences of steps depend on their respective effectively computable functions. Finally, while the applications $U(U[G[X]])$ and $V(V[G[X]])$ do not always produce the same value for all G, the correspond-

ing applications $sef_u(\lambda\text{–}calculus, S_u, sef_g(\lambda\text{–}calculus, S_g, x)) = u(g(x))$ and $sef_v(\lambda\text{–}calculus, S_v, sef_g(\lambda\text{–}calculus, S_g, x)) = v(g(x))$ always result in the same values for all g, regardless of whether $u(g(x)) = v(g(x))$ or $u(g(x)) \neq v(g(x))$ or not, since sef_u and sef_v are functions. By assumption, the absence of one corresponding function for $V(V[G[X]])$ is a contradiction.

□

Briefly, from the presented alternative view of unexpected effects, while in the computation of the Turing-machine composition $V(V[G[X]])$ the computation of the Turing machine G might replace the inner occurrence of the Turing machine V by another Turing machine V' and, therefore, the outermost occurrence of V might compute $V(V'[G[X]])$ instead, neither the steps S_g nor the function sef_g can replace any function, in particular, neither sef_u nor sef_v.

Theorem 2 *If each Turing machine implies one partial function, then the class of Turing-computable functions is not isomorphic to the class of effectively computable functions.*

This theorem is another way of seeing the theorem 1. □

Because the models of effectively computable functions provide ways of defining functions that result in any number as we wish, then for all $k \in \mathbb{N}$, for all $(x, y) \in \mathbb{N}^k \times \mathbb{N}$, intuitively, there must exist an effectively computable function which calculates y from x in a few steps. However, as I have shown, Turing machines are not necessarily functions. As already mentioned, unexpected effects introduce a problem with two aspects: logical and semantic. I solve the logical aspect of the problem by proposition 2, and I solve the semantic aspect of the problem by regarding Turing machines that produce different answers under different physical conditions as non-functional machines. Thus V is a Turing machine which does not have any corresponding function. Furthermore, it is easy to see that Turing-computable functions are still linked to Turing machines where unexpected effects are forbidden. Notice that this corollary holds for both intensional and extensional standpoints, as we can also

view unexpected effect as an action or effect of replacing one Turing machine by another with different results.

Theorem 3 *Computation is not necessarily function application.*

[Proof:] Given that computation may be mobile, e.g. by using mobile agents nowadays, given some insight of the present author in Edinburgh (1999), and deeply discussed in chapter 4, computation is conceptually a physical process.

On the other hand, by theorem 1, the class of Turing machines is not isomorphic to the class of Turing-computable functions. Following this, programs do not correspond to functions. Therefore, computation is not function application.

□

Proposition 1 *Let $k \in \mathbb{N}$. For every $k > 1$, there exists a k-level Turing-machine composition if and only if some representation of the Universal Turing machine is not in the composition.*

[Proof:] Let U be a Universal Turing machine, $M, N : \mathbb{N} \longrightarrow \mathbb{N}$ be two Turing machines with corresponding Turing-computable functions $m, n : \mathbb{N} \longrightarrow \mathbb{N}$, and $x \in \mathbb{N}$, and $X : \mathbb{N}$ be the representation of x on the tape.

The Universal Turing machine, by lemma 2, guarantees the absence of unexpected effects at all levels of its parameters. I can consider the composition $M[N[X]]$. It follows that U must have *direct* control over the operations of N in such a way that, if N tries to modify the operations in the M representation, U detects this unexpected effect and intervenes, for instance, by moving physically the representation of M or N to another place on the tape, to continue the computation of the composition keeping the isomorphism between Turing machines and computable functions. Therefore, because U must have dynamic knowledge about the computation carried out by N, $U(M[N[X]])$ is not really a function application, and therefore some representation of U is not in the composition.

With respect to the converse, setting $M \neq U \wedge N \neq U$ and $X \neq U$, there exists a Turing-machine composition, e.g. respectively $M(N[X])$ above, from which U is absent.

□

Remark: We can capture an intuitive and precise notion of Turing machine model as follows: Let $\mathcal{M}$ be the set of all Turing machines, T be the set of Turing-computable functions, U be the Universal Turing machine and u be the Universal Turing-computable function. Let $X : \mathbb{N}$ be the null-computation Turing machine that corresponds to the 0-ary function (i.e. without any input) that always results in the same value $x \in \mathbb{N}$. Therefore, $u : \mathcal{P}(T) \longrightarrow \mathbb{N}$ (where $\mathcal{P}$ is the ordered power set of a given ordered set), in such a way that the application $m(n(x))$ is equal to $u(m(n(x)))$ and abstractly represented as $U(\{M, N, X\})$ or, more precisely, as $U(s)$ where s denotes the string that encodes M, N and X, together with the write/read head and blank symbols, with the constraint that s does neither start nor finish with the blank symbol. Furthermore, composition is part of the notion of function, and such a representation does not capture the composition of Turing machines $U(M[N[X]])$. However, there may be applications as well as compositions involving U where there exist such representations with U, both on the tape and outside the tape. Therefore, there exist two different levels of functional abstraction in the Turing machine model of computation.

In other words, on the one hand, we separate what is encoded on the tape from what is outside the tape, by stating that only what is outside the tape is free from unexpected effects, and hence, can be functions. On the other hand, functions do not manipulate the operations of any function. In this way, there are two different levels of abstraction: at one level, only the Universal Turing machine is function and the encoded Turing machines on the tape form a one-level parameter. At another level, there exists a Turing machine composition on the tape, and the encoded Turing machines correspond to the computable functions because the Universal Turing machine does not correspond to any function in the same space, in the sense that U is capable of managing the

tape and guaranteeing absence of unexpected effects. Therefore, there exist two separate levels of function abstraction in the Turing machine model of computation.

In the next proposition, as usual, I do not regard time as a factor in the computation.

Proposition 2 *There exists a Turing machine that can correspond to more than one Turing-computable function.*

[Proof:] Let M be some Turing machine and x be its input. Let U be a Universal Turing machine, and V be another Turing machine, which, except for the existence of unexpected effects, produces the same output as U: the only difference is that U calculates $U(U[M[X]])$, and V calculates $V(V[M[X]])$ and sometimes calculates $U(U[M[X]])$, but sometimes not, depending on the physical places where V and M rest on the tape. Because the Turing machine composition $V(V[M[X]])$ can be placed at different places on the tape at different instants and the computations receive different kinds of unexpected effects, the same running Turing machine M can produce different results for the same input x. Each particular result from x corresponds to one Turing-computable function.

□

Corollary 1 *There exists a Turing machine that can avoid receiving unexpected effects from its parameter.*

[Proof:] Universal Turing machines, as discussed in the theorem 1, must ensure absence of unexpected effects.

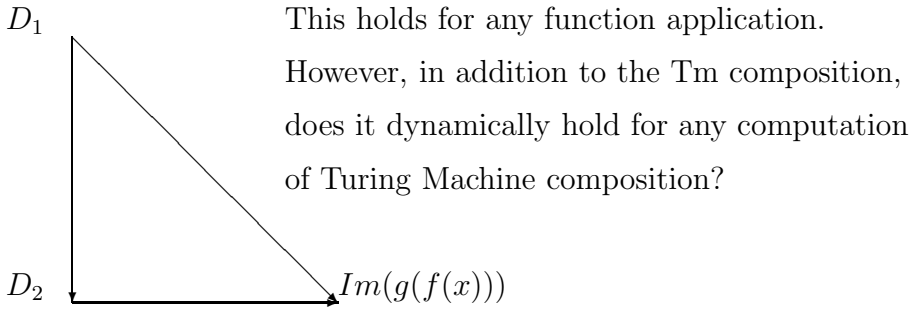
□

In this chapter, I have distinguished two levels of notions of computation. The first level is the purely mathematical or functional one, while the second one, which contains Turing machines, is slightly different as there is the notion of unexpected effect. In other words, for the latter level be isomorphic to the former one, it is assumed, perhaps implicitly, the absence of unexpected effects.

However, another level of notions of computation can be distinguished from applications of functions.

I observe that real-world machines have finite memory capacity, and this forms a third level of notions of serial computation. For instance, at this level, the halting problem is clearly decidable by the following algorithm, briefly described. I write a machine that simulates the serial computation of another machine M given its input data X. If the simulation halts, my machine halts giving the proper answer, but the simulation does not halt if and only if some state of the mentioned computation repeats. If it repeats, my machine detects the condition and halts giving the proper answer. By 'state of computation' I mean the current head position together with the content of the tape in a Turing machine model with a finite tape. However, this algorithm works on any finite-resource machine for any serial computing.

I can draw functions $f: \ D_1 \longrightarrow D_2$ and $g: \ D_2 \longrightarrow D_3$ as well as the corresponding composition $h: \ D_1 \longrightarrow D_3$, under the law $h = g(f(x))$ as in the following picture:



(where $Im(g(f(x))) \subseteq D_3$). For answering the question, I individually consider the correspondence between the functions f, g and h, and the Turing machines F, G and H, respectively. The answer for the question, i.e. whether the law of composition $F(G(X)) = H(X)$ holds for every computation of composition of Turing machines, depends on the absolute positions of the involved Turing machines, F and G, on the tape, as well as on whether the only Turing machine outside the tape avoids unexpected effects on the tape.

However, both factors are *external* to the machines that are on the tape and play rôles in the composition. As a consequence, the global view is a property of the Universal Turing machines, in particular, it avoids what I discovered and refer to as unexpected effects.

A final question: what is now the meaning of *algorithm*? In this chapter, the term *algorithm* is used to mean a program which always halts, and not necessarily a total function. A decision had to be made.

The conclusion that programs are not functions can directly lead to the part II of the present PhD thesis dissertation. After this chapter, one is free to introduce a hybrid programming paradigm where the functional programming paradigm is only one of them.

The other main result from this chapter, i.e. computation is not necessarily a function application, in a sense, is similar to a result from chapter 4 of this thesis dissertation, where I present a notion of computation which transcends pure mathematics.

Chapter 4

Mobility and Computation

In chapter on Turing machines, 3, I describe three levels of computation. The notions can be as follows: functional, can include Turing machines and, very briefly, can include Turing machines with finite tape. The third level briefly corresponds to real-world serial machines. In this chapter, I introduce a fourth level of computation, which can be called computation in the real world.

Mobile agents and the Internet have brought new ideas to theoretical foundations of computer science in the last few years.

As an example, during Christmas 1999, I had an interesting conceptual insight over computation: "...at the moment that I conceive the idea of moving computation from one place to another, I also observe that a general notion of computation transcends pure mathematics and meets the physical world". This itself requires new, informal and philosophical discussions in the theoretical foundations of computer science. During Christmas 2001, "and because the universe is on the move, computation is essentially mobile."

The present chapter discusses some meaning of computation, provides a different semantics and present a formalized, physical and abstract model after my simplification. The present model makes use of four forms of mobility,

namely strong mobility, intentional unity mobility, non-intentional unity mobility, and broadcast mobility. The formalization makes use of the @-logic.

It is part of the present PhD thesis that the Internet almost entails that philosophy becomes an essential subject in theory, foundations and practice in computing science. Together with the content of chapter 3, the above observation on computation suggests the replacement of a number of key terms such as parallel computation, distributed computation, global computation, mobile computing and mobile computation, by a more unified notion, physical computation, or simply computation. In this way, the notions of computation and computing get closer to each other, and I can also explicitly refer to as natural and artificial computing.

In the present chapter, I present other arguments for the most general and unified notion of computation, although it is only one among other good proposals. Mobility and global computing form two different classes of argument.

4.1 Introduction

It is not easy to perceive the characteristics of the era in which we all are, because an era transcends the life time of human beings as we miss comparisons which are more realistic than what the literature can teach. The Internet has grown very quickly. This global infrastructure, along with other recent technologies, such as satellite television, mobile phones and portable computers, have not only changed humans' behavior but have also made this planet psychologically smaller than ever. On the one hand, this new apparatus has led to new terminology regarding mobility[59], computation[198, 274, 320], programming languages[219], distributed systems[24, 47] and mobile agents[154], in such a way that this terminology deserves care, regarding the appropriateness of its use. On the other hand, I observe that one notion of computation cannot be captured using mathematics. As an example, as well as the work on unexpected effects on Turing machines computations, of chapter 3, one may regard parallel and concurrent computing, some forms of mobility, side-effects, unreliability of the physical media and other factors as non-mathematical, or mathematical via physics, although they can be abstractly captured by algebra and categories up to some extent. At a more general level, on these days, there is what has been been called *mathematical physics*. In the context of computer science, although we are neither mathematicians nor physicists, one may dare observe that, any attempt to model physics is based on insights, observations and common knowledge on the physical science up to *some time*, as new laws of physics may be discovered in the future that modify the way that we regard the real world, while theories in mathematics, for example, tend to be monotonic. Therefore, for us, it is more convenient and safe to adopt that view that physics and mathematics are always independent, for they deal with different natures of objects.

To exploit this view in this chapter, I consider the semantics of computation as based on its physical nature, with some rough simplifications in the present model as I do not deeply investigate psychological issues. It is also part

of the present view to regard the traditional theory of computation, which is based on recursive functions, as extremely and historically important although, perhaps from the present standpoint, that theory does not capture what I am calling computation here. In [221], the authors describe a method for proving termination of recursively defined functions based on ordinal measure, and such contributions are very relevant, even adopting a non-mathematical perspective, or a different philosophical view.

A philosophical central issue in computer science is whether or not humans are machines. A few years ago, Deep Blue beat Kasparov in chess, that is, regarding the time when the match happened, the best machine player beat the best human player in chess. On the one hand, the machine was not designed to generally simulate human beings behavior, for Deep Blue is not able to solve problems other than in chess. On the other hand, although chess allows a huge number of different positions, taking into consideration that any repeated position means draw, the number of chess games are finite. Thus, the central issue in the match was *efficiency*. Furthermore, chess is played under well-defined rules, which contrasts with most situations in the real life. Briefly, as we know, in many situations, machines are more efficient than humans, while in many others, humans are blessed with intuition and feeling, among many other skills and talents. While machines have been analytical, humans combine analysis and synthesis well, and have much more complex nature.

In this chapter, I use interchangeably the terms "computer science" and "computing science" as meaning the same science. In the thirties of the last century, "computer" was the person who used to calculate, normally some woman[82]. Although I have my own personal view, in the present PhD thesis, I do not discuss the philosophical issue of whether humans are machines, nor the issue on whether God exists. However, such questions are probably what distinguish what I refer to as *the fourth level of computation* from a possible fifth level, and also what distinguish this possible fifth level from higher levels (until the level of God). This may be seen as a kind of different levels of the Church-Turing thesis stating that there is a unique level of computation, or,

alternatively, as different levels of negations of the latter philosophical idea.

Some questions arise that might conceptually interest the theory of computer science. For example, let us imagine two mobile agents that travel in space at the same speed. The first one is traveling indefinitely. The second one is running a simple algorithm that, when it has met the first, halts. Should one regard the second agent as running an infinite computation? If we adopt the point of view that computation is a physical[190] concept and that space is flat, the answer is *yes*.

Another issue is whether mobility introduces new elements in the theory of computing science or not. A number of academic and theoretical work, to some of which I make references in the present chapter, forms strong evidence that the computing science community agrees it does. Under certain philosophical perspective, we can also observe that mobility is a primitive in computation and, since this, I provide a formal and physical semantics of computation (one of the proofs that the present meaning is more general is by introducing one different form of code mobility which can be via broadcasting media, such as radio or satellite television) and, finally, I mention other factors of the real world, such as faults and delays in network connections, as relevant to the probably more general and physical notion of computation. For the formal semantics, I write the rules using my space-time logic that I briefly introduce in chapter 2. In this chapter I argue that both physical and philosophical factors are part of the broader notion. Indeed, theory of computation has strong connection with philosophy, and this claim extends the connections between logic[67, 179] and philosophy[71]. Here, I simply discuss the matter while I present a number of examples that have been selected during these years of studies.

Regarding the physical nature of computation, I introduce an operational semantics of computation that includes space and time, as well as captures the present four forms of mobility. For the present PhD thesis, these factors are enough to demonstrate that, although pure mathematics and logics are still going to be used as computation and to simulate reasoning[200], they do not

capture more general notions of computation. In [240], there is an interesting introduction on mobile agents, where the author shows evidence of benefits which they achieve. For more advanced literature on this subject, [258], for instance, among some others.

As regards philosophical factors, one enters a subjective, informal and possibly psychological world, the real world, transcending the mathematical and logical language as well as traditional computer science texts. The notion of computation should not be confined to a unique and universal concept, but instead, there should be diversity, e.g. the concept of computation depends on the defined underlying machine, although these machines share common properties. Further, for each different notion of computation, there can be different theories of computation, including different theories of computability. Further, the referred to term *theory*, for instance, starts having a broader meaning, which not only includes the traditional one, from deductive logics, but also philosophical theories. Machines are becoming gradually more complex. A global computer, for instance, is an abstract and general machine atop some internet services and includes the notion of code mobility. There are other approaches to global computation, such as [70]. On the one hand, such a computer provides a more general notion in comparison to those notions defined since Alan Turing and others, who did not consider mobility as a physical primitive. On the other hand, the geographical distribution introduces other factors to the definition of the global computer that cannot be neglected. Because every computer has its repertoire of operations which not only defines but also constraints the capability of the machine, any global computer has to provide a repertoire of operations that depends on ethics, common sense and the laws of the civilized world, e.g. some issues are discussed in chapters of [195]. Those factors belong to the real world, not to an idealized world such as that of mathematics. As an example, a state or country can establish some law to prevent unsolicited e-mail, say, messages of advertisement have to contain the string "Advert" in the beginning of their subject fields. One of the subjective parts of this is that the laws for e-mail

often apply to the receiver and not to the sender, or to the sender and not to the receiver, and this makes it more controversial. At a different level, the same holds for mobile agent systems, which are typically spread out among different cultures. On the one hand, global computers have to establish what can be computed. On the other hand, laws of behavior depend on time and space, among other factors. Because of this, the work on the chapter 2 is also based on these factors, regarding human knowledge and awareness. As another hypothetical example, suppose that a country has a law that adopts the policy to use only software open to its public administration. However, should a global computer guarantee privacy of mobile agents? Can the country impose such a constraint on incoming code? Such a discussion might be controversial.

Therefore, not only in the present chapter, both formal work and informal discussions are significant. Informal discussions precede formalizations.

Section 4.2 is somewhat ontological. I discuss terminologies of notions related to agents[66] or mobility. Section 4.3 is very conceptual as I introduce a view of mobility detached from other concepts and I discuss its relationships with other related concepts. Section 4.4 discusses other complementary notions such as distribution and centralization. Section 4.5 introduces an intuitive notion of computation. In section 4.6, I provide one notion of computation that is somehow broader than the well-established notions[285], such as Turing machines, $\lambda-$ and $\pi-$ calculi. At a higher level, the proposed notion is based on physical, mental and philosophical factors, besides mathematics. I see computing science as a table whose legs are these four studies, and this table is in the synthesis, a diagram, in chapter 10. In this way, I extend the conventional operational semantics by adding space, time and mobility, as well as defining states in a more sophisticated way, in comparison to the basic literature. In section 4.7 I discuss more practical issues with respect to global environments borrowing some key words from philosophy. Finally, section 4.8 concludes the chapter.

4.2 Agents

In this chapter, I consider four forms of mobility. I dedicate this section to agents.

The term *agent* has been used by both the AI[160] and distributed systems (DS) communities with different meanings and at different levels[123]. In addition, the term *agent* in English has the same spelling as in French, and has almost the same spelling and meaning as the term *agente* in Italian, Spanish and Portuguese. In these languages, *agent* or *agente* can normally mean "a person who acts on the behalf of another person or other people" or "a person who does something or causes something to happen". However, the word *agente*, comes from the verb *ago* in Latin, which means *to act*.

The term *mobile agent* is somewhat ambiguous. For instance, robots are agents that act physically on the environment, some of them are mobile and they have been referred to (by some) as mobile agents, but they are objects very different from mobile agents with which some researchers in the programming languages and distributed systems communities deal. Researchers from AI have commonly used the term *software agent* to differ from the other forms of agency. In this chapter, I use the term mobile agent in the context of code mobility, rather than robotics. As well as code mobility, one of the four forms of mobility will be roughly related to the latter meaning of agent, from robotics.

In both fields of computer science, DS and AI, it has been noted that the term *agent* still lacks a clear and standard definition[123]. An interesting question now is whether it is really necessary to have a clear and standard definition of the term agent by a few particular computer scientists, or whether to let the problem of these notions and terminology disappear naturally in future work. That is, if different communities have used the same word in English with different meanings, it might also be the case that both technologies and fields have things in common[162]. From this perspective, we ought to explore and investigate this combination.

Since old times, societies have developed and the term agent has become more sophisticated. Nowadays, travel agents represent passengers in transactions with airlines, for instance. Agents are able to *act* on users' behalf and, because of this, must have autonomy and authorization to do so. In such a more complex context, intelligence is one of the desirable requirements for human, hardware or software agents. This is one of the points where AI has much to contribute. More than this, there is a subtle emerging area of research related to agent technology: users not only want agents to be as intelligent as themselves, but also want agents to behave in accordance with their corresponding psychological profiles. What does psychological profile mean? how to program it? Answers for such philosophical questions have to be established before implementation.

I believe that, although the meanings in the terminologies of agents from DS and AI are different and apply to different levels, they can easily complement each other. Thus, programming languages can support this integration.

4.3 Mobility and some related concepts

At this point, I am not interested in mobility of matter, nor even at the *logical* level and, because of this, I do not explore this particular subject here. The notion of mobility, as viewed as a sequence of different places at different times, which in turn can be seen as a sequence of very short discrete intervals, has been published in books of popular sciences. I prefer not to see people and objects as space and time, but instead, I define computation in this way. The opposite view could also be considered: without mobility, i.e., if nothing changes in the world, even planets do not move, past is totally equal to future and, hence, there is no perception of time. Since nothing changes, there is no thought. Each of these two approaches corresponds to a different philosophical view, although I choose the former view. However, it is important to observe up to what extent computer science and technology can make use of a particular philosophical view as a starting point.

Therefore, regarding computation, the present chapter explores some physical properties of computation together with mental and philosophical ingredients. In other words, computation in this chapter transcends pure mathematics.

Thus, this section is a conceptual discussion on the foundations of computing science in the presence of mobility. There are good surveys on code mobility such as [75, 123, 240]. There are other important contributions. In chapters 5 and 6, I technically discuss code mobility. Briefly, from the technological standpoint, code mobility came from a refinement of the client-server paradigm of distributed systems. The well-know paradigms for code mobility are: remote evaluation, code on demand and mobile agents. Additionally, there have existed two forms of code mobility: week and strong. In the present chapter, I am particularly interested in strong mobility, which requires the implementation by the mobile agents paradigm, although there are mobile agents systems that provide a weak form of mobility. The first symmetrically secure solution[1] for mobile agents systems was published in [105].

On the one hand, mobile agents technology was initially developed to solve or minimize technical problems, in particular in a distributed environment where performance is regarded as important. Furthermore, the Internet is a shared resource, users want to share their resources in a controlled way, and this technological scenario contributes to development of mobile agents technologies. Thus, transactions usually need several messages between partners and, when this case holds, they ought not to be performed remotely but mainly by local communication[270, 301] between a mobile agent and another agent. This requires new programming languages concepts and constructs, and some have been designed considering agent migration as priority, such as [72, 229].

As already written, the present notion of computation provides four forms of mobility. In addition to code mobility, the movements of a robot in a cor-

[1]The term symmetrically secure is used in the sense that the solution protects both hosts and visiting agents in an equally satisfactory way.

ridor, and the movements of a portable computer running a program in a transport on the move, at least for computer science, should *not* be examples of the same form of mobility for, although the computation moves as a consequence of hardware mobility, robots move *intentionally*, in contrast with portable computers, although some technologies can permit both implementations to be *aware* of physical positions of the underlying physical machine.

"*Mobile computation*"[58] has been used informally to mean the computation supported by mobile code applications. The longer the distance the stronger the argument towards mobile code systems. However, although electronic commerce, for example, can be conceived without code mobility, *CPU loan or rental* is a kind of application impossible to be done *on-line* on a computer that does not provide code mobility. Mobile agents typically come and use someone's CPU, or even many CPUs in parallel. Although very simple, this example is evidence that code mobility provides a physical and probably more general meaning of what can be computed in the real world, and not simply a new technology.

Conceptually, one can observe that *mobility* has always been a basic concept in computer science, as well as part of some models of computation. For example, although there are books that provide a symbolic definition of a Turing machine such as [259], a well-accepted metaphor since Turing himself is based on one head that *moves* along a tape as a result from the current state and the transition function.

There are examples in other models. In a Petri net the control also *moves* in a similar way to a finite-state machine, either deterministic or not. The β-reduction rule in λ-calculus is also a move in a sense. Furthermore, any von Neumann machine, any sequential computer, provides mobility at several levels: a variable assignment is a move and a copy. The digital and analogical circuits also move bits and electrons, and so on. In order to generalize, every bit that moves from one part of the computer to another may be conceived as the simplest form of code mobility: it has source, content and destination. Because these models of computation provide some form of simple mobility,

code mobility should be a *primitive* of a more general notion of computation. The fact that some mobile agent can simulate a Turing machine write/read head or a token in a Petri net is only one example of the generality of the notion of mobility.

Summarizing the forms of mobility dealt with in the present chapter, I itemize four different ones:

- Strong mobility: the destination is stated explicitly and hardware or computational environment (CE) does not move (existential software movement in a sense).
- Broadcast mobility, **SpreadOut** primitive: the destination is not explicit and hardware or CE does not move (universal software movement in comparison to the strong mobility).
- Non-intentional hardware mobility: the hardware moves (or it is moved) and the software might be aware or not.
- Intentional unite mobility, **wemove** command: it is an intentional form of mobility of the computational environment (CE), in a sense. Robots movements and people walking in the streets, might also be seen as particular case here.

There are other forms of mobility.

4.4 Other Concepts

In [16], the author presents a model of distributed computation which is based on a fragment of π-calculus relying on asynchronous point-to-point communication. The same author then enriches the model with some features. In general, nowadays, many researchers who work in the code-mobility community are in some distributed systems group, and most call for papers of conferences on distributed systems includes mobile agents technologies. The connection between the two notions is indeed very strong.

Here, an important issue is: are mobile agent systems distributed systems? If one thinks carefully, the answer may be *no*. Research on mobile-code technology includes research on programming languages, design and implementation, and such languages make programmers be aware of resources, which contrasts with the philosophy of distributed systems in its traditional sense.

But, returning to the primary and conceptual level, the terms centralization and distribution are often related, and they can oppose or complement. Although we might prefer to use distributed computing for technical reasons such as efficiency, robustness and security, the concept of centralization is necessary even in computing. Taking the human body as a metaphor, the human circulatory system consists of *one* heart, veins spread out in the body and blood. Some of us know that the movement of blood in veins towards the heart represents centralization while the opposite movement away from the heart to the parts of the body represents distribution. This is a natural example showing that centralization and distribution can be mutually beneficial, and even be necessary for each other. Here, mobility is a third important component, represented by the movement of blood in both directions.

Similarly, although complex systems are normally distributed, *mobile code systems are not distributed systems* but they are related[37, 311]. Likewise, *mobility is not distribution*, but these concepts can coexist. These concepts exist at different levels of abstraction. For example, one can implement a distributed system using code mobility.

The concept of centralization is present in mobile code systems. For example, the concept of centralization applies to the level of programming. In this case, a central component is the programming language, as it provides standards and imposes constraints to the whole system. As another example, agents can move to a central place, a specific interpreter, and communicate with each other locally. This independence from centralization and distribution, together with the possibility of implementing the last two, makes mobility a more general concept and, hence, a good candidate for a primitive in this model of computation.

Another pair of concepts is individuality and what is from the collective, which is also relevant for global computers. Physically, every person is individually unique. There are refinements of physical characteristics that depend on genes, e.g. groups due to family factors. However, at the collective level, all people have common features, such as two eyes, two ears and one mouth. Accordingly, every individual has his or her own personality, and yet they share many collective standards: for instance, it is commonly felt that Marilyn Monroe was beautiful. These psychological standards vary according to place and time, and some of them change more slowly, others more quickly. Culture and fashion are two examples of collective standards. Family psychological characteristics, including those due to education, is an example of psychological characteristics shared by groups. Rules of good behavior exemplify group or collective common sense, and is normally conscious behavior.

So far computation is relatively simple for us, computer scientists, but when one investigates deeply human psyche and starts thinking about unconscious, the subject becomes with synthetic nature. The psychologist Carl G. Jung[175], for instance, studied deeply what he called the *collective unconscious*, which is a theory based on science, but also a theory that contains elements from his philosophical view.

An analogy can be made between the hierarchy of characteristics as described above, and mobile agents systems. Because agents cannot view the internal parts of the interpreter implementation, and because all interpreters of the system should be the same or at least compatible with each other from a minimum extent, the interpreter corresponds to the collective unconscious, while the mobile agent corresponds to the conscious part of an individual. It turns out that such a psychological model can be somehow simulated by computers. The power of the collective unconscious can be illustrated in the following way: if there is some change or mistake in the implementation of the interpreter, like a social revolution, all mobile agents of the system may be critically affected at once. On the other hand, common sense and other kinds of information ought to be ubiquitous resources, provided by the mobile-

agent system, i.e. mobile agents do not need to carry such established knowledge about the world, nor even any established belief system. Although the work of Jung form one of the most general psychological models, the Jung's model seems to apply to mobile agents, including the four psychological types, namely, reasoning, five-sense perceptions, intuition and feeling. The exceptions are probably intuition and feeling, although one can develop software which imitates human behavior.

Within the scope of AI, neural networks also make use of mobility. A neural network tends to represent what is learned from perception and possibly intuition. Using some mobile agent technology, a neural network may be spread out even over the globe, in such a way that the perception is also spread out, which contrasts with human perception. Because of this observation, this implementation can be seen a novel hybrid model of computing. One can observe that, whereas deductive systems are closer to the western way of thinking[183], neural networks with fuzzy systems[189] are opposite models closer to the eastern view, although this difference tends to disappear. I would like to stress that such applications use the technology of mobile agents to implement mobility, but *mobility is an essential property of networks in general*. And since the whole universe is on the move, mobility is a relative concept.

Regarding computation, as well as *mobile computation*, there is another modality of mobility, namely *mobile computing*[249]. The latter modality comes from wireless networks and portable computers, a subject somewhat close to robotics in a sense. Both forms are described in [60] or [61], and both are orthogonal to each other, for instance, one application does not affect the other[288], at least directly. As an example, to be general I must consider that a mobile agent can move from one craft to another, both on the move, and such a double movement is part of the general notion of computation.

4.5 An intuitive notion of computation

Traditionally, the models of computation are: Turing machines, Church's λ-calculus, Post production systems, Kleene's μ-recursion schemes, Herbrand-Gödel equational definability, Shepherdson-Sturgis register machines, the while programming language, and flow charts. Mobility is becoming part of candidates to the next generation of established models of computation.

Taking the example of on-line CPU loans or rentals, agents come to the host and, after identification and/or negotiation, use the CPU and possibly other resources, depending on the agreement. If it becomes expensive, some agents move to another host. Besides practical concerns, any application that depends on the presence of the "computational entity" acting locally is an example that there are computations which cannot be done without some form of interpreted code mobility. Should space and time are regarded, it is not difficult to find other examples. In particular, after having transported an agent to some virtual machine, while that agent interacts locally, some connections may be interrupted but the agent's computation might not be affected by that (temporary) interruption.

Therefore, in this chapter, I define a notion of computation from an operational standpoint, in particular, I am concerned with time and space as part of a somewhat general notion and, therefore, part of the model that I shall present. The present formal model is only an extension of the *while* programming language.

Thus, some conceptual issues are: *what is a computational entity? what do we mean by 'code' and 'interpretation'?* In some sense, 'code' can be any data, and the present discussion on computation leads one again to philosophy. There is the same for the term "executable code" which, depending on the adopted meaning, there are two different views in computer science. Therefore, code mobility introduces a philosophical view to computer science.

In [57], Cardelli briefly and informally defines *global computation* and points out several related issues, such as how multiple global computers can

interact effectively. As he says, the main characteristic of global computation is the geographical distribution. Although every planet has its globe, the term global computation refers to this planet. Although I keep the term "global computer" from his article, I prefer to use the term global computing instead of global computation. The part II of this thesis is dedicated to programming languages that can be used for global computing.

In this way, I shift the notion commonly referred to as "computation" to be referred to as *mathematical computation* to accommodate mobility as a primitive of the notion which I refer to as *computation*, for mobility is the focus of attention in this chapter. Alternatively, one may prefer to refer to the same notion as *physical computation* while keeps the traditional meaning of computation.

Parallel computation is another abstract and theoretical concept that is modeled in π-calculus[212]. In that article the author shows that names of channels can be passed from one process to another, for instance, and that was a major step in the foundations of mobility and computation. Technologically, parallel computing has been linked to super-computing and powerful machines, but code mobility can also implement parallel computation over a local-area network or wide-area network or both. Thus, *parallel computation is simply computation in parallel.*

In [119], the authors introduce the distributed Join-calculus, which is an extension of the Join-calculus[118] for mobile agents. Both are asynchronous variants of π-calculus with the same expressive power as the latter, but the former provide better locality and better static scoping rules[119]. To represent mobile agents, the distributed Join-calculus introduces locations, and for unreliable environments, that calculus also provides a simple model of failure.

Ambient calculus[60, 61] also captures the notion of code mobility, and, because the calculus is also partially based on the π-calculus, it also describes parallel processes. The Ambient model is also inspired by Telescript but almost dual to it, according to the authors. Other important contributions have been made since then, such as [48]. However, like other calculi and unlike

Distributed Join-calculus and the Seal calculus[307] which is another extension of π-calculus, it does not abstract other details associated to mobility, such as resources and uncertainty, e.g. due to unreliability of physical media. On the other hand, the Seal calculus and others[154] are somewhat practical calculi, in the sense that they are dependent on current structures such as the Internet. Although the notion of process[192] migration[213] is not new, the term *mobile computation* was coined by Cardelli in [58].

An interesting feature of a model of computation with strong mobility is that they *transcend* term rewriting systems. In comparison to models based on π-calculus, the **flyto** primitive (i.e. an instruction needed in every mobile agents programming language) moves not only the remaining symbols but also its surrounding context. Here I present an example of a rewriting system-like rule, in Ambient calculus, for moving an agent composed of two parallel processes ($\mathbf{flyto}(B).P$ and Q) from the place A to the place B:

$$A[\ (\mathbf{flyto}(B).P)|Q\ ]\ \ \|\ \ B[\ ]\quad \xrightarrow{\tau}\quad A[\ ]\ \ \|\ \ B[\ P|Q\ ]$$

The above rule cannot be applied using, for instance, a context-free grammar[22, 157, 165]. Moreover, at a more practical level, if one considers that places have their local resources and that agents typically use them locally, the order in which agents move *does* matter. For instance, in an unreliable environment, one cannot think in terms of a more general sense of the Church-Rosser property[143].

Parallel computation is a more general and abstract notion in comparison with recursive functions, that is, the latter is like a thiner granule. The linear operator $\otimes$, for instance, captures the notion of parallel operands, but it is still purely mathematical. Because I am looking for generality, my notion of computation includes both parallel computation and mobility, in addition to the physical[265] nature of this form of computation. In comparison to Ambient calculus, for instance, I consider timeouts in the model.

Although the mobility community in computer science established global

computing for structures such as the Internet, certain issues related to computation are not limited to this planet. For example, a mobile agent can migrate from a spacecraft and continue its computation at another spacecraft no matter where they are. Agents can also travel from one planet to another no matter the distance between them. Because of this, I can perceive another term to refer to another model that includes code mobility. I use the term *physical computation* to stress the physical nature of computation, and *computation in the real world* to stress the philosophical, physical and psychological aspects of computation together. Additionally, I also use the term *computing in the real world* to include programming languages, technologies and applications.

Larger distances and time intervals for mobility are two fundamental characteristics of code mobility in comparison to the computation local to a single hardware. In [57], Cardelli describes the main characteristics of global computing and I summarize them below:

- Parallel or concurrent processes.
- Code mobility.
- Latency and bandwidth are directly addressed.
- The availability of resources are distributed geographically, which requires that programmers be aware of locality of resources. This in turn replaces a established law in distributed systems.
- Higher level of interaction between users and machines.
- Security and privacy are particularly critical.

Thus, physical computation is a more abstract and theoretical notion than global computing, because physical properties of time and space are not limited to this planet or Internets. Physical computation and global computing almost entail mobility. However, while mobile code systems necessarily produce physical computation, this can be done locally or remotely, but not necessarily on a global environment.

If one starts considering mobile computation as a specialization of computation, we can see a kind of ontological paradox[64] in the traditional notion of computation: *from the moment that one conceives the idea of moving computation, one can observe that an effective general notion of computation transcends pure mathematics and meets the physical world*. Furthermore, objects in the real world are far from being perfect as the mathematical objects which one idealizes. In other words, code mobility changes the notion of computation.

The terms *real* and *ideal* have been used in philosophy concerning realism and idealism, respectively. Computation in the real world is at somewhere between the two, for I consider that the real world includes the material world, the psychological world and so on. Examples of fundamental questions are:

- What is the meaning of computation?
- What is the meaning of computing?
- What are the differences between mankind and machines?
- Can machines succeed in the Turing test? What does *intelligence* mean?
- Humans compute while they are dreaming, and sometimes they may even make some arithmetic calculation, but they are not conscious nor have any control over the thought. Should I consider this phenomena in a more general notion? Is not it evidence that machine intelligence is different from human intelligence?
- Etc.

Such questions show the importance of having some knowledge on psychology and philosophy. In particular, the above kind of issues is evidence of the relevance of philosophy in foundations of computer science. Regarding the question on whether or not machines may think, in [273], it is presented a number of pieces of evidence that can explain how a machine may simulate insight

and/or intuition. In contrast, there are other philosophical opinions which have been defended by others. The author of the present dissertation, from a different philosophical standpoint, makes note that such a simulation should not be regarded as necessarily similar to real thought for, if simulation had been really thought, that would simply mean that natural sciences and philosophy already know all about human beings and the universe (This observation is mainly conceptual and not necessarily related to results that machines may give, for this in turn can be a matter of up to what extent human beings can perceive differences in the results). There is an important difference between to be able to explain a hypothesis from some skeptical point of view over the reality and to prove the same hypothesis, an issue a little similar to what is briefly discussed in the present dissertation, section 1.3, on relevance logics. Moreover, it is already known that there exist notions that cannot be proved or refuted, as perhaps Gödel incompleteness theorem[144] is the best example that truth and proof, among other notions, do not necessarily go together. Therefore, in the lack of evidence that we humans know all about the universe, the opinion that machines think should be regarded as part of particular belief system and a purely philosophical matter.

Considering that humans have unconscious, the task of simulating human behavior with computers becomes dramatically more difficult. Here I give one more example of this view. It can be perceived that the sensation of *pleasure* conceptually is a mechanism of which the nature makes use to preserve both the individual and their species. At the individual level, a delicious (or beautiful) meal is sometimes able to create the wish in some person to feed themselves, sometimes even without any need. At the collective level, sexual pleasure can make a person pregnant. According to Jung, the human unconscious is often projected on things and people that one deals in one's daily life, and that mechanism can work individually or collectively. Thus, *projection* is a view that the individuals have of their unconscious, both personal and collective, and the rôle of projection seems to be to make them aware that the source of the standard of their interactions with the world is in themselves,

and possibly that the standard needs change.

Projection is a mechanism of which the psyche makes use to advise the individual concerning what they do not perceive in themselves. In some sense, although projections are not normally pathological, the rôle of projection is somewhat similar to physical pain, which advises the individual that he or she is sick and, therefore, should seek treatment. Some other physical symptoms seem to play this rôle but they act at the collective level especially when the disease is contagious.

The collective unconscious makes sense as a *psychological* mechanism that seems to protect the individual and the species. To date, in addition to the approaches in natural sciences, there are different philosophical positions concerning the nature of psyche, some more complex than others. A recent selection on the rôle of analogy in the context of cognitive science is in [136].

To establish what computation is, I find a philosophical question: does any computation exist that cannot be perceived by humans at all? The answer also depends on the philosophical position. Although I use the term "real", I also adopt some idealistic ideas, i.e. computation in the real world regards human as a central component, but this does not exclude beliefs in God, either internal or external, and this is another subject in philosophy. The previous question can be used to provide foundations for a theoretical computer science based on physics and philosophy, as well as on mathematics. The relationship between mathematics and computer science is certainly very strong, like the relationship between mathematics and physics[110].

Here, as an example of computation, I present the reduction steps in some computation using some version of λ-calculus with some syntactic sugar:

$$(\lambda f.f\ 10)(\lambda x.x+1) \overset{\lambda}{\rightsquigarrow} (\lambda x.x+1)10 \overset{\lambda}{\rightsquigarrow} 10+1 \overset{\lambda}{\rightsquigarrow} 11$$

An interesting introductory and long study on λ-calculus and models of untyped λ-calculus is in [32]. Although these steps of computation are described as purely abstract objects, at the moment that I read them the *actual*

computation is carried out mentally, in a context at some time and at some place.

4.6 A notion of computation

Although there are other notions of computation, using local symbol definition, here I consider that computation can be ϕ in the following signature:

$$\phi : \pi \times \rho, \quad \rho : \tau \times \mathcal{U} \times \psi$$

where π is the sequential concept of computation, and ρ is an extra philosophical component, which in turn is defined as a product of time τ, space $\mathcal{U}$ and some possible psychological component ψ. Here, place and time can be those traditional physical dimensions, while ψ is with respect to some observer. In this piece of work, I do not use the above signature, which was shown as illustration.

4.6.1 A view of time, a representation

This section describes a view of time in accordance with what is left as parameters in chapter 2. In many articles on temporal logics[11, 12, 128] commonly applied to AI planning systems[13] and other fields, time is often represented by using real values where, as time goes by, *the present moment* normally increases. There might be branches along these lines to represent "futures". There are other approaches, such as in [257] that can also be useful for applications, including system specification, but also to express natural sub-languages by using particular cases of modality.

In this chapter, I adopt a form of representing time by making use of a flow. Thus, let us define $\mathbb{T}$ as an infinite set of temporal moments and let the flow be linear in $\mathbb{R}$. I use relational operators over the real numbers to state temporal relations.

The longer the distance is, the more significant modern physics is. Thus

the present model is only a simplification of my intuitive notion i.e. this notion depends upon a number of factors that do not appear in these semantic rules, such as those caused by gravity and bodies, as well as what can be discovered in physics.

Let $\mathbf{C}$ be the set of five logical values as defined in chapter 2. In accordance with the two operators $<_t$: $\mathbb{T} \times \mathbb{T} \longrightarrow \mathbf{C}$, $-_t : \mathbb{T} \times \mathbb{T} \longrightarrow \mathbb{T}$ and so on, which I introduced somewhat informally in chapter 2, the operators over time instants refer to the daily-life temporal concepts, e.g. $a <_t b$ refers to *"a happens before b"* and so on. Apart from such order operations, there is no interval relationship over indexes. Thus, if $t \in \mathbb{T}$ is used as time variable, $t_j >_t t_i$ always holds for $j > i$ but $t_{i+1} -_t t_i$ is not necessarily equal to $t_{j+1} -_t t_j$. If $\mathbb{T}$ is $\mathbb{R}$, therefore $<_t$ is $<$ and $-_t$ is $-$ without further formalization.

I do not use $<_s$ here. $-_s : \mathbb{S} \times \mathbb{S} \longrightarrow \mathbb{S}$ is some approximation of the Euclidean distance from the first operand to the second one

$$(x_i, y_i, z_i) -_s (x_j, y_j, z_j) \stackrel{def}{=} \text{approx}\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

by using some mathematical method, already known for example.

One may want to represent time in a different way.

Here I define computation by defining the set of operations of some abstract machine.

4.6.2 States of the Real World

In this subsection, both space and time[34] are defined as continuum[78]. Much current theoretical work in computer science, such as [46], makes use of some form of continuous time. Because it is not my intention to present and discuss more than one philosophical view in the current chapter, I choose and present one as an example, having in mind that it is not necessarily a proposal for all. One subjective view is needed for supporting explanations in next subsection. Therefore, since such a model supports the idea of computing in $\mathbb{R}$, it may

also support the idea of computing in $\mathbb{Z}$ with possible minor adaptation. In parallel to this, as an example that philosophy is a basis for computer science, in this subsection, I am regarding a hypothetical situation where agents travel to some country, for instance, China and, therefore, are culturally exposed to their very ancient wisdom. However, in India for instance, although it is another country in Asia, agents would be exposed to some different cultural background. In general, computer science has been based on modern western culture, but mobile agents on a global environment is making one think about the meaning of computation from different perspectives. Furthermore, on the one hand, complexity theory states what computers can do. On the other hand, ethics[179] (which is a branch of philosophy) asserts what computers should and what they cannot morally do, i.e. there are two complementary approaches. Internethics may be viewed as a good term for this new area.

Following this, during computation, each *real state*, or here I simply refer to it as "state", is not only the context of the program, in its traditional sense. It is also behaviors, including actions, which are unique in time and space. Behavior is the part of a state that can be perceived externally, while the internal state contains the perception of the external world. A state is a set that can contain behavior and internal state, which in turn can contain locations, the characteristics of the physical world at a specific time, e.g. the whole mankind and all computers, the whole world, although not everything is accessible by the agent in question. In this sense, there is a mismatch between real states and the machine set of behaviors. The former is uncountable, the latter is countable[110]. Human experience is continuous in both space and time[2]. I tend to perceive time and space as in the structure of the set of real numbers.

Without defending a particular view, an *analogy* can be drawn between this discussion and in the relationship between I Ching and Taoism. I Ching

[2]The notion of time might also be even individual, e.g. for a 2-year-old child, one year corresponds to the experience during half of life, and, perhaps because of this, I may feel that time goes by quicker as I get older.

hexagrams are pictorially composed of six lines. Some lines might move from *yin* to *yang*, others might move from *yang* to *yin*. Indeed, the I Ching is based on the binary system. On the other hand, although Taoism is also based on this pair of concepts, they are pictorially shown inside the Tao circle in such a way that one is gradually (but without granularity) being moved to the other. This analogy suggests that the I Ching is only a simplification of the Taoist view of reality as much as machines, by nature, are simplifications of the real world.

As another analogy, the classical musical language as well as the capability of some instruments can also be seen as simplifications of music. For some instruments, there are only 12 notes, although the scale is cyclical and can be repeated with higher or lower pitches. However, between any two subsequent notes rests a continuum interval of frequencies.

Likewise, from a somewhat similar point of view, although digital computers are useful and much can be improved on them, human computation might not be a concept limited to the set of integers, or perhaps it is better to define the notion of computation more precisely. That is, it is important to make clear up to what extent one is talking about computation, whether e.g. feeling and intuition are really computation, or whether this pair of psychological concepts and others, say *synthetic* concepts, can only be a matter of analytical simulation. Moreover, $\mathbb{R}$ certainly suggests the potential for future discoveries, as the infiniteness of the tape in the Turing machine model represents this potential, which is infinite. As a concrete example, if I want to conceive a circumference in a Cartesian plan, it is sufficient to have its equation, i.e. $x^2 + y^2 = r^2$ where r denotes its radius, but, to calculate its coordinates, I must convince myself that between any two points there exist infinite points. Although the infiniteness of the tape in the Turing machine model suggests this potential, proposing a model in $\mathbb{R}^4$ may be more natural and easier to conceive from a different view of computation. As one example of other work on a continuous three-dimensional space, in [163], the authors introduce an approach to the solution of the pursuit problem in the Euclidean $\mathbb{E}^3$ space.

It is known that humans reason and do research in science and mathematics in many ways, while the mathematical world is pure, exact or precise and perhaps very clean, tidy and structured in comparison to the psychological world, with dreams and the unconscious, for instance. An example of this kind of issue is input-output in purely functional languages, solved by using monads[145, 310], but a problem that can also be seen from a physical standpoint instead. There are other different views regarding the match between mathematical and real worlds, and this is also another piece of evidence that computation is a philosophical notion. This almost entails that philosophy is a theoretical basis of computation and computer science.

To compute in $\mathbb{R}$ is not a novel idea, and there are good references such as [298].

In this discussion, I can still see a finite computation in a discrete interval in time as a sequence of states $s_{t_0}s_{t_1}...s_{t_n}$, where $\{t_0, ..., t_n\}$ are chosen instants, and I simply add the notions of time and space for every state. To define computation I need to define an abstract machine. My machine is a virtual machine that supports mobile agents, i.e. a virtual machine for a subset of an imperative language, such as Pascal or C, in addition to the ability to move the code, data and state of the computation in accordance with the definitions here.

Firstly, I define a state as a tuple composed of an internal state (ι), the external state of behavior (β), a place $p \equiv \langle x, y, z\rangle$ and an instant (τ). Thus,

$$s \in S,\ s \stackrel{def}{=} \langle \iota, \beta, x, y, z, \tau\rangle$$

or, in the @-logic,

$$s \in S,\ s \stackrel{def}{=} @p \cdot \tau[(\iota, \beta)]$$

where S is the set of all possible states. In this chapter, I define ι and β as *sets* of propositions. I define the empty state as $\mho \in S,\ \mho \stackrel{def}{=} \langle \emptyset, \emptyset, 0, 0, 0, 0\rangle$. Thus, for $s_1 \equiv \langle \iota_1, \beta_1, x_1, y_1, z_1, \tau_1\rangle \wedge s_2 \equiv \langle \iota_2, \beta_2, x_2, y_2, z_2, \tau_2\rangle \wedge \tau = \tau_1 = \tau_2$,

and ζ being a condition (a predicate as a singleton), there are six properties, which are the following:

P1:

$$\zeta \in s_1 \leftrightarrow \zeta \in \iota_1$$

That is, if a condition is in a state, it means that it is part of its internal state. Similarly, two predicates for sets:

P2:

$$s_1 \overset{s}{\subset} s_2 \leftrightarrow (\iota_1 \subset \iota_2 \vee \beta_1 \subset \beta_2) \wedge p_1 =_s p_2 \wedge \tau_1 =_t \tau_2$$

Accordingly, $s_1 \overset{s}{\supset} s_2 \leftrightarrow s_2 \overset{s}{\subset} s_1$.

P3:

$$s_1 \overset{s}{=} s_2 \leftrightarrow \iota_1 = \iota_2 \wedge \beta_1 = \beta_2 \wedge p_1 =_s p_2 \wedge \tau_1 =_t \tau_2$$

Accordingly, $s_1 \overset{s}{\subseteq} s_2 \leftrightarrow s_1 \overset{s}{\subset} s_2 \vee s_1 \overset{s}{=} s_2$.

P4:

$$s_1 \overset{s}{\neq} s_2 \leftrightarrow \iota_1 \neq \iota_2 \vee \beta_1 \neq \beta_2 \vee p_1 \neq p_2 \vee \tau_1 \neq_t \tau_2$$

and two functions for the spatial sets, in the @-logic from now on.

P5:

$$s_1 \overset{s}{\cup} s_2 \equiv \langle \iota_1, \beta_1, x, y, z, \tau \rangle \overset{s}{\cup} \langle \iota_2, \beta_2, x, y, z, \tau \rangle \leftrightarrow @p \cdot \tau[(\iota_1 \cup \iota_2, \beta_1 \cup \beta_2)]$$

P6:

$$s_1 \overset{s}{\cap} s_2 \equiv \langle \iota_1, \beta_1, x, y, z, \tau \rangle \overset{s}{\cap} \langle \iota_2, \beta_2, x, y, z, \tau \rangle \equiv @p \cdot \tau[(\iota_1 \cap \iota_2, \beta_1 \cap \beta_2)]$$

I also take the liberty to use ε to denote the idle state. There is only one occurrence of the idle state in any computation. Notice that ε and $\mho$ are not the same notion.

In this chapter, because ι and β are closely related, I collapse ι and β and refer to them as *virtual state*. Thus, from now on a state is the tuple $\langle r_t, x, y, z, t \rangle$ or the proposition $@p \cdot t[r_t]$, which is the shorthand for

$$@p \cdot t[state = r_t]$$

where r_t is the corresponding virtual state at the time t.

As already mentioned, although I informally consider the possibility of continuous flow-like multi-dimension set of states with respect to space and time, here I define a simplified version of computation, i.e. computation in the real world is a sequence of states, $s_{t_0}s_{t_1}...s_{t_n}$ that one selects according to a particular focus of attention.

Let S be the set of all machine states, S isomorphic to $\mathbb{N}$. I define a (non-reflexive except for one case, and) anti-symmetric relation $\xrightarrow{c}$: $S \times S \longrightarrow Bool$ to indicate the existence of two subsequent states associated to the computation. I define $\xrightarrow{c}$ in terms of its properties as follows:

- $s \xrightarrow{c} s_1 \wedge s \xrightarrow{c} s_2 \Rightarrow s_1 \overset{s}{=} s_2$, $s_1 \xrightarrow{c} s \ \wedge \ s_2 \xrightarrow{c} s \Rightarrow s_1 \overset{s}{=} s_2$.
- $s \xrightarrow{c} s \Rightarrow s \overset{s}{=} \varepsilon$.
- Its particular case $\varepsilon \xrightarrow{c} \varepsilon$ which always holds.

I define $\xrightarrow{+c}$ as the transitive relation as follows:

$$s_{t_0} \xrightarrow{+c} s_{t_n} \overset{def}{=} (s_{t_0} \overset{s}{\neq} \varepsilon \wedge s_{t_0} \xrightarrow{c} s_{t_n} \xrightarrow{+c} \varepsilon) \ \underline{\vee} \ (\exists s_{t_1} \in S : s_{t_0} \xrightarrow{c} s_{t_1} \xrightarrow{+c} s_{t_n} \xrightarrow{+c} \varepsilon)$$

where $\underline{\vee}$ is the *exclusive or*, and $\forall s_0, s_1, s_2 \in S : s_0 \xrightarrow{+c} s_1 \xrightarrow{+c} s_2$ is defined as $s_0 \xrightarrow{+c} s_1 \wedge s_1 \xrightarrow{+c} s_2$ and the same holds with $\xrightarrow{c}$. Among others, three important properties of $\xrightarrow{+c}$ are the following:

- $s \xrightarrow{+c} s \Rightarrow s \overset{s}{=} \varepsilon$.
- $\forall s, s \xrightarrow{+c} \varepsilon$ which is my practical view of computation.
- $s_i \xrightarrow{+c} s_j \wedge s_i \xrightarrow{+c} s_k \Rightarrow (s_j \overset{s}{=} s_k \ \underline{\vee} \ s_j \xrightarrow{+c} s_k \ \underline{\vee} \ s_k \xrightarrow{+c} s_i)$, i.e. $\xrightarrow{+c}$ is unique with respect to $\xrightarrow{c}$.
- $s_i \xrightarrow{+c} s_j \wedge s_i \overset{s}{\neq} \varepsilon \Rightarrow \neg(s_j \xrightarrow{+c} s_i)$

In this way, *computation in the real world* can be defined as any finite sequence of states over the time, where every two subsequent states are linked with an application of $\xrightarrow{c}$ relation, where the last state of that sequence is the only inactive state of that computation. The intention might be the same, the place might be the same but, if the instants are not the same, the external world is no longer the same. Therefore, from this point of view, computations performed at different times cannot be the same. With some simplification, human thought can be an example of $\xrightarrow{+c}$, where ε corresponds to the individual's death. It is very difficult, if possible, if I want to mathematically establish when a computation starts and when it finishes for this case. So I state in this example that human computing starts when they are born and finishes when they die, and it is also always finite. I can also think in terms of collective computing which may never finish due to (human) communication, or may finish due to some colliding asteroid, for instance.

An analogy can be made between computation and a melody being played, where not only the sequence of notes is relevant but also the duration of every note among other variables. More generally, there is a subtle difference between a melody and its performance, whereas, likewise, there is some subtle difference between the mathematical and physical forms of computations.

In comparison with any rewriting system[23], except for the idle-state case here, although $\xrightarrow{+c}$ is also transitive, such a relation is neither symmetric nor reflexive as time never goes by backwards nor it stops, i.e. joining together two of the above properties we obtain the following one:

$$s_{t_i} \xrightarrow{+c} s_{t_j} \Rightarrow (s_{t_i} \overset{s}{=} s_{t_j} \overset{s}{=} \varepsilon) \ \underline{\mathrm{v}} \ (s_{t_i} \overset{s}{\neq} s_{t_j} \wedge \neg(s_{t_j} \xrightarrow{+c} s_{t_i}))$$

Moreover, this relation implies some measure of uncertainty due to the somewhat unpredictable nature of the real world. While any computation happens, the probability of its success gradually increases over time. To add some probability between states is enough to introduce a more general and physical model, in some sense. Hence, I alternatively define $\xrightarrow{+c}$ as follows:

$$s_{t_0} \xrightarrow{+c} s_{t_n} \stackrel{def}{=} \exists m \in \mathbb{R}, 0 \leq m \leq 1 : (s_{t_0} \stackrel{s}{\neq} \varepsilon \wedge \Psi(m : s_{t_0} \xrightarrow{c} s_{t_n}) \xrightarrow{+c} \varepsilon) \ \underline{\mathrm{v}}$$
$$(\exists s_{t_1} \in S : \Psi(m : s_{t_0} \xrightarrow{c} s_{t_1}) \xrightarrow{+c} s_{t_n} \xrightarrow{+c} \varepsilon) \tag{4.1}$$

where $\Psi(n : \varphi)$ represents the assertion φ with probability n. Therefore, $\forall s : \Psi(1 : s \xrightarrow{+c} \varepsilon)$ and also

$$\begin{aligned} &\forall s_0, s_1, s_2 \in S, mn \in \mathbb{R}, 0 \leq mn \leq 1 : \Psi(mn : s_0 \xrightarrow{+c} s_2) \stackrel{def}{=} \\ &\exists m, n : \mathbb{R}, 0 \leq m \leq 1, 0 \leq n \leq 1, mn = m \times n : \\ &\Psi(m : s_0 \xrightarrow{+c} s_1) \wedge \Psi(n : s_1 \xrightarrow{c} s_2) \end{aligned} \tag{4.2}$$

The $\downdownarrows$ relation indicates that two states coexist independently. Formally using the @-logic...

$$@p_1 \cdot \tau_1[r_1] \downdownarrows @p_2 \cdot \tau_2[r_2] \stackrel{def}{=} @p_1 \cdot \tau_1[r_1] \stackrel{s}{\cap} @p_2 \cdot \tau_2[r_2] \stackrel{s}{=} \mho \wedge p_1 \neq_s p_2$$

Let $\mathbb{U}$ be the set of computations and S be the set of states. I define the function $\stackrel{s}{\rightsquigarrow}$: $\mathbb{U} \times S \longrightarrow Bool$, which informs whether the second operand is the last non-idle state *during* a computation, which in turn is given as the first operand.

The function $\stackrel{s}{\rightsquigarrow}$ may have included the influence from the behavior of other computations, not only interaction with other computations. Thus, a program running twice produces two different computations (and perhaps two different behaviors). There are two more significant properties:

$$(\forall C_1, C_2 \in \mathbb{U}, \forall s \in S)\ C_1 \stackrel{s}{\rightsquigarrow} s \wedge C_2 \stackrel{s}{\rightsquigarrow} s \Rightarrow C_1 \stackrel{c}{=} C_2$$

where $C_1 \stackrel{c}{=} C_2 \stackrel{def}{=} \exists n \in \mathbb{N} : C_1 \equiv s_0 s_1 \ldots s_n \wedge C_2 \equiv s'_0 s'_2 \ldots s'_n \wedge \forall (i \in \mathbb{N}, i \leq n) : s_i \stackrel{s}{=} s'_i$. And also:

$$(\forall C \in \mathbb{U}, \forall s_1, s_2 \in S)\ C \stackrel{s}{\rightsquigarrow} s_1 \wedge C \stackrel{s}{\rightsquigarrow} s_2 \Rightarrow s_1 \stackrel{s}{=} s_2$$

That is, computation has the property of being a unique object with respect to its final state. I also define α as the set of all programs, i.e. the set

of all mobile agents (or mobile processes), which are the potential for computation. Let $\mathbb{U}$ be the set of computations and S be the set of states. The function $\lceil\ \rceil : \mathbb{U} \longrightarrow S$, that indicates the last active state, is defined as the following:

$$\lceil C \rceil = \begin{cases} s_0 & \text{if } C \equiv s_0\varepsilon, \text{ where } s_0 \neq \varepsilon \wedge s_0 \xrightarrow{c} \varepsilon \\ s_n & \text{if } C \equiv s_0 ... s_n\varepsilon, \text{ where } ((\forall i \in \mathbb{N})\ s_i \neq \varepsilon) \wedge s_n \xrightarrow{c} \varepsilon \end{cases}$$

for every computation C which entails some $n \in \mathbb{N}$.

4.6.3 The Present Semantics of Computation

In this subsection, I attempt to formalize a simplified version of the previous notion of computation, informally introduced above in this section, by using the @-logic.

Let π be a program in some object language with some computation ϕ_c and let all of my definitions in this subsection apply to the scope π, unless stated otherwise. To avoid being exhaustive, I consider that variables have their separate scopes in each rule although they have the same names and meanings, except for variables defined explicitly as global for all rules. Let $\mathcal{A}^R$ be isomorphic to $\mathbb{R} \cup \{\boldsymbol{uu}\}$ and $\mathcal{A}^L$ stand for the set of Boolean values, and use these sets as carriers of the algebra[96, 210] that is going to be defined here.

I also write $\langle x, y, z \rangle$ explicitly when I want to stress the relationships between each of these coordinates and the time, although I do not use them individually in the semantic rules. There is one global definition, in symbols, $p \stackrel{def}{=} \langle x, y, z \rangle$ and it may be indexed, e.g. $p_i \stackrel{def}{=} \langle x_i, y_i, z_i \rangle$. As well as the notation in rules, I make implicit use of first-order predicate logic in these rules. When t is not the present the predicate expression does not hold. Thus, the states of the predicate expressions change as time passes. Thus, in the @-logic, I write logical expressions as well as operational expressions.

The present model is over the following definitions:

- There is a common three-dimensional space in $\mathbb{E}^3$, which is the universe.

I equate $\mathbb{R} \equiv \mathbb{E}$ and use Cartesian coordinates $\langle x, y, x\rangle$ to refer to the points in (Euclidean) space $\mathbb{E}^3$. Thus, let $\mathcal{U} \stackrel{def}{=} \mathbb{E}^3$, be the universe;

- There is an infinite but countable set of possible agents $\triangle \stackrel{def}{=} \{a_1, ..., a_i, ...\}$ written in the language α;
- Every agent has its input queue;
- Every point $\langle x, y, z\rangle$ in $\mathcal{U}$ contains a finite set $Obj \stackrel{def}{=} \langle v, \{a_1, ..., a_n\}\rangle$ where $v \in Val, Val \equiv \mathcal{A}^R$, that is, v stands for either a real number or the uu value, and $\{a_1, ..., a_n\} = Y \subseteq \triangle$.

Then, I define the following Σ-algebra

$$\begin{gathered}\mathcal{A} \stackrel{def}{=} \langle \mathcal{A}^R, \mathcal{A}^L, Var, SObj, Loc, S, \mathcal{U}, 0, 2, \boldsymbol{uu}, par, p, p_0, p_i, q,\\ r, r_0, r^p, r^{p_0}, r_t, r_{t_0}, r_{t_0+I_b}, t_1, r_{t_i}, s, s', t, t_0, t_\varepsilon, t_f, t_i, \Delta t,\\ A_t^p, A_{t_i}^p, A_{t_0}^{p_0}, A_t^{p_0}, A_t, A_{t_0}, A_{t_i}^{p_i}, A_{t_0+I_b}^{p_0}, A^d,\\ u, v, V, x, y, z, x_0, y_0, z_0, x_i, y_i, z_i,\\ \Delta l, \Delta l_1, \Delta l_2, \Delta s, \Delta s_1, \Delta s_2,\\ I, I\mu, I_a, I_{a_1}, I_{a_2}, I_b, I_e, I_i, I_{:=}, IC, I_{th}, I_s, I_{wem}, \omega, \Psi\rangle\end{gathered}$$

for signature Σ here, where Var is the set of all variables internal to ϕ_c, and also to π; and since $Obj = \langle v, Y\rangle$ where $v \in \mathcal{A}^R$, $Y \subseteq \triangle$ as stated, $SObj = \mathcal{A}^R \times \mathcal{P}(\triangle)$; Loc is the set of internal locations, and S is the set of states. Let $u, v \in Val,\ V \in Var,\ p \in \mathcal{U}, r, s \in S, t \in \mathbb{R}$. Then,

$$\Sigma \stackrel{def}{=} \langle \{\mathcal{A}^R, \mathcal{A}^L, Var, Val, Loc, S, SObj\}, F\rangle$$

where F is consisted by $+$, $-$, $/$, $=$, $\neq$, $<$, $>$, $\wedge$, $\vee$, r^2 as usually defined in mathematics plus the following functions:

$$(locate) \quad \gamma : Var \longrightarrow Loc$$

$$(lookup) \quad \rho : S \times Loc \longrightarrow Val$$

$$(update) \quad \Delta : S \times Loc \times Val \longrightarrow S$$

$$(value) \quad \Theta : S \longrightarrow SObj$$

$$(first) \quad \xi : S \longrightarrow \mathcal{A}^R \times S$$

$$(del\ queue) \quad \chi : S \longrightarrow S$$

$$(ins\ queue) \quad \eta : S \times \mathcal{A}^R \longrightarrow S$$

$$(fault) \quad f\pi : \mathbb{R}^3 \times \mathbb{R} \longrightarrow \mathcal{A}^L$$

$$(physical\ move) \quad \ddot{\mu}x, \ddot{\mu}y, \ddot{\mu}z : \ \mathbb{R} \longrightarrow \mathbb{R}$$

Intuitively, γ maps a variable to its location; ρ results in the content of a location in some particular state; Δ updates the memory according to its parameters: location and value; Θ, provided a state, results in the value of the corresponding point in space and time; ξ gives the first element of the input queue and the state after the operation; χ removes the first element from this queue; η inserts a value as the last element of this queue; $f\pi$ is a predicate that, given some point p and time t, informs whether there is some fault (e.g. the presence of another agent) between the current point (an implicit parameter that denotes the point at the three current coordinates of the running agent) and p at t; and $\ddot{\mu}x, \ddot{\mu}y, \ddot{\mu}z$ are postfix functions that, given one coordinate results in the new corresponding coordinate due to possible physical hardware movement. That is

$$p\ddot{\mu} \stackrel{def}{=} \sqrt{(x\ddot{\mu}x)^2 + (y\ddot{\mu}y)^2 + (z\ddot{\mu}z)^2} \quad \text{(for an approximation of the real)}$$

For each semantic rule, as it is already clear here, to simplify a little the notation, I shall assume that the existence of more than one occurrences of the functions $\ddot{\mu}x, \ddot{\mu}y, \ddot{\mu}z$ and $\ddot{\mu}$ is not relevant, that is, I do not formally consider that the physical shifts of the machine for this tiny intervals might be at different velocities. However, in this case, to state this independence of velocities, I could index the occurrences of theses operators, for example, $\ddot{\mu}x_1, \ddot{\mu}x_2, \ddot{\mu}_1, \ddot{\mu}y_1, \ddot{\mu}_2, \ddot{\mu}y_2$ and so on. Additionally, let $s, s', r, r_t, r_0, r_{t_0}, r_{t_0+I_b}, r_{t_i}, r^p, r^{p_0} \in S$, as well as let r_0 be the initial state of the computing agent.

For the level of abstraction to capture computation, I regard the meaning of defining operations in terms of some sequence in some microcode. There are other alternatives. My choice is to divide the rules into two levels. One of them

concerns the object of computation, as usual, and I call *object rules*. The other concerns the application of the object rules. This applies only for interaction and mobility operations. Let us start with the three rules as follows:

$$\ddot{a}_{t_0} \wedge (\forall t,\ (t_0 <_t t <_t t_0 +_t q_i) \Rightarrow \ddot{a}_t) \wedge K_{t_0+q_i} \wedge \forall t, t >_t t_0 +_t q_i,\ (\neg\ddot{a}_t \wedge \neg K_t)$$

where $\ddot{a}_t$ is the attempt to perform a remote operation (**wemove** although for this level of detail there is no timeout, **flyto** or **view** or **throw**) at time t by using the rule 4.5 (**wemove**) or 4.6 (**flyto**) or 4.9 or 4.12 (both rules for **view**), or 4.15 (**throw**); K_t is the action of skipping the executing operation of rule 4.8 (**flyto**) or 4.14 (**view**) or 4.17 (**throw**) at time t; $q_i = \min(q, d)$, d is the delay due to the operation in question, q is the timeout for this occurrence of operation.

I also need to formalize the notion "the sooner the better". Thus, if t_i and t_j are time variables, $\ddot{a}_t$ is some action of type A to be performed at t, then the rule

$$(\ (t_i \leq_t t_j) \wedge \Diamond\ddot{a}_{t_i} \wedge \Diamond\ddot{a}_{t_j}\ ) \rightarrow \Box\ddot{a}_{t_i}$$

states that, given the above conditions, the action $\ddot{a}_{t_i}$ must be performed first.

Every state is a tuple $\langle r_t, x, y, z, t\rangle$ where r_t is the virtual state at some time t, and $\langle x, y, z\rangle$ are Cartesian coordinates at that locality. I use the notation $r[V/u]$ to mean that the variable V contains the value u in the virtual state r. I adopt A^p_t, for *ambient state*, to refer to a particular state at some place p and time t. Ambient states are important to stress the mobility of virtual states. Therefore, $r_t \cap A^p_t$ simply indicates that the virtual state r_t is placed in the ambient A^p_t at place p and time t, once the premise $r_t \subseteq A^p_t$ exists. I use this notation for mobility and other related operations, and simply write r_t to denote the same situation in other rules. In the semantic rules, I use the definition $s \stackrel{def}{=} \langle r, x, y, z, t_0\rangle$ which can also be represented as $@p \cdot t_0[r]$ in the @-logic. Thus, two or more states $s, s'...$ can happen at the same time, i.e. in parallel. In this case, I denote this as $s \Downarrow s' \Downarrow ...$.

Given that $\varepsilon \xrightarrow{+c} \varepsilon$, any agent segment of computation can also be com-

posed in $S_1 \| S_2$ i.e. in parallel:

$$\frac{S_1 \models \langle r_t^{p_1}, x_1, y_1, z_1, t\rangle \xrightarrow{+c} s_{t_i} \quad S_2 \models \langle r_t^{p_2}, x_2, y_2, z_2, t\rangle \xrightarrow{+c} s'_{t_i}}{S_1 \| S_2 \models (\langle r_t^{p_1}, x_1, y_1, z_1, t\rangle \Downarrow \langle r_t^{p_2}, x_2, y_2, z_2, t\rangle) \xrightarrow{+c} (s_{t_i} \Downarrow s'_{t_i})}$$

where S_1 and S_2 denote two segments of computation for some t_i. For any two computations, the rule for composing them is different from the previous one:

$$\frac{C_1 \models \langle r_1, x_1, y_1, z_1, t_1\rangle \xrightarrow{+c} s_{t_i} \quad C_2 \models \langle r_2, x_2, y_2, z_2, t_2\rangle \xrightarrow{+c} s_{t_j}}{C_1 | C_2 \models (\langle r_1, x_1, y_1, z_1, t_1\rangle \Downarrow \langle r_2, x_2, y_2, z_2, t_2\rangle) \xrightarrow{+c} (s_{t_i} \Downarrow s_{t_j})}$$

where C_1 denotes a computation that finishes at t_i and C_2 denotes a computations that finishes at t_j.

Parallel computation presents behaviors in parallel. Notice that computations might interfere with each other.

My abstract model of computation is an extension of the while language [2], which is an abstract model of computation. Nonetheless, although that model has the *if* statement, I observe that this statement can be definable: that is, for p and q as a Boolean expression and statement, respectively, IF p THEN q is defined as

$x := 0;$
while $x = 0 \wedge p$ **do**
  $q;$
  $x := 1;$
endwhile

for a new variable x.

For defining the *if-then-else*, roughly speaking the reader can view two similar loops, and this change by removing *if* from the present model will simplify my effort. I am formulating a model in terms of one set of operations. It is important to see that these operations do not necessarily correspond to their suitability in programming languages that permit proactive move because here I am only defining an abstract model. Chapter 6 contains sufficient material of programming languages in the presence of global computers.

The coordinates $\langle x, y, z\rangle$ of $\mathbb{E}^3$, for instance, are not normally provided at the language level, but they are in the model because they provide a suitable notion of space for my purpose.

In this way, the operations form a somewhat minimal set of primitives for the present semantics of computation, i.e. it corresponds to an extension of the while language presented in the literature as in [151, 185, 219, 228, 275, 318, 319]). Thus, I formalize the operational semantics that complement theirs and give a few local primitives. Thus my working objects are:

- Real constants.
- The space initialization.
- The **create** statement, which creates another copy of the computation at another place.
- The arithmetical operators: $+, - : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$.
- The relational operators: $=, <: \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$.
- The assignment statement.
- The **while** statement.
- Intentional unity mobility: the **wemove** statement.
- Broadcasting mobility: the **SpreadOut** statement.
- Strong mobility: the **flyto** statement.
- Communication: the **view** and **throw** statements for remote communication, and **read** and **write** for local communication.
- The *halt* operation. The $f\pi$ predicate.
- Physical mobility: no operators, but the $\ddot{\mu}x$, $\ddot{\mu}y$ and $\ddot{\mu}z$ postfixed functions defined in the above algebra. Further,

$$p\ddot{\mu} \stackrel{def}{=} \sqrt{(x\ddot{\mu}x)^2 + (y\ddot{\mu}y)^2 + (z\ddot{\mu}z)^2},$$

more precisely, some approximation of this number.

Variables can range over an address space, that is, $Var \stackrel{def}{=} \{V_0, V_1, ...\}$, but here I shall use only the V symbol to denote any variable in Var.

From the **while** statement, one builds the *if-then*, *if-then-else* and *case* as I did or in the same manner. From these definitions, one builds *min* and *max*, then builds the logical operators, and so on.

Now I can present the initial axiom before one places agents and values in $\mathcal{U}$:

$$Initial: \quad \frac{}{(\exists t_0 \in \mathbb{T})\ (\forall s \in S,\ \forall t \in \mathbb{T},\ t_0 \leq_t t)\ s \equiv \langle r_{t_0}, x, y, z, t_0 \rangle,\ \Theta s = \langle \boldsymbol{uu}, \emptyset \rangle}$$

and another rule with the $\stackrel{\text{eval}}{\rightsquigarrow}$ relation:

$$\frac{\rho(s, \gamma V) = u}{@p \cdot t_0 [\![V, s]\!] \stackrel{\text{eval}}{\rightsquigarrow} @p\ddot{\mu} \cdot t_0 +_t \Delta t[u]}$$

where Δt is the time for accessing a variable at place $\langle x, y, z \rangle$ and time t_0. Notice that, in the signature, I defined $\gamma : Var \longrightarrow Loc$ as the function that, given a variable, locates its storage.

An agent can create another agent by executing the **create** statement. The new agent is a clone but it executes from scratch at a given coordinate. The semantics for this statement can be as follows:

$$\frac{r_{t_0}^{p0} \not\subseteq A_{t_i}^p \quad r_{t_0}^{p0} \subseteq A_{t_i + \Delta t}^p \quad @p_0 \cdot t[t <_t t_0 +_t q \wedge \neg f\pi(p, t)]}{\begin{array}{c} @p_0 \cdot t_0 [\![\textbf{create} \langle x, y, z \rangle \textbf{ time } q, r_{t_0}^{p_0} \cap A_{t_0}^{p_0}]\!] \stackrel{\text{exec}}{\rightsquigarrow} \\ @p_0 \ddot{\mu} \cdot t[r_{t_0}^{p_0} \cap A_{t_0}^{p_0}] \downdownarrows @p \cdot t_i +_t \Delta t[r_0^p \cap A_t^p] \end{array}} \tag{4.3}$$

where $t_i =_t t_0 +_t \Psi +_t I_c$, I_c is the time for interpreting the **create** statement. Ψ is defined globally as

$$\Psi = \text{approx}\ \frac{\sqrt{(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2}}{\omega}$$

where ω is the velocity of light and, for the rule 4.3, $t \geq_t t_i +_t \Psi$ and Δt is the time spent installing the agent at the destination, once the transmission has completed.

For the purpose of simplification and visibility, I allow myself a slight abuse of notation by not writing the temporal subscript in operations, such as $+_t$, when they are already part of a temporal expression appearing in subscript of a formula, for instance $A^p_{t_i+\Delta t}$, which appears above. The same is valid for spatial relations in subscript, as the meanings of the operators are clear.

The semantics for the corresponding timeout situation is as follows:

$$\frac{@p_0 \cdot t[t =_t t_0 +_t q \wedge f\pi(p,t)]}{@p_0 \cdot t_0[\![\textbf{create } p \textbf{ time } q, r^{p_0}_{t_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_0\ddot{\mu} \cdot t_0 +_t I_c[r^{p_0}_{t_0} \cap A^{p_0}_{t_0}]} \tag{4.4}$$

The *halt* operation is not a statement of the present language, that is, after the execution of the last statement in the program, the computation goes to the idle state. Although it is not a statement, I regard as if it were, as, here, I am interested in the computation itself:

$$(\exists k \in \mathbb{N})\ \frac{}{@p_0 \cdot t_0[\![halt, r]\!] \overset{\text{exec}}{\rightsquigarrow} @p_0\ddot{\mu} \cdot t_0 +_t k[\varepsilon]}$$

where *halt* corresponds to the implicit last operation.

For unity (CE or hardware) mobility, the command **wemove** moves the set of agents from the current point $p_0 : \mathbb{S}$ to another $p : \mathbb{S}$ (coordinates $\langle x, y, z\rangle$) as a unity, along with the corresponding value, given the coordinates of the destination. I assume that sets of agents can share a common place and move slower or at the speed of light, but robots movements can be a particular case of this form of mobility as long as I formalize some physical laws as well as constrain and regard the destination close enough to the source place for uniform and straight movements, in such a way that more complex movements could be obtained from a sequence of this simpler physical movements here. Thus, let p_0 represent the coordinates of the original place. While the unity moves (say, after some delay ζ), the original place becomes without value in the problem domain. The subexpression $\frac{2\times(p-_s p_0)}{\omega}$, below, represents the minimum interval with value of twice the space through which the light would traverse: Time for observation and, then, for moving the unity; I_{wem}, for interpreting

the **wemove** command:

$$\frac{\begin{array}{c}\exists(t \geq_t t_0 +_t I_{wem} +_t \frac{2\times(p-_s p_0)}{\omega}) \\ p \equiv \langle x,y,z\rangle \wedge \Theta\langle r,x,y,z,t\rangle = \langle u, A^d\rangle \wedge \neg f\pi(p,t)\end{array}}{@p_0 \cdot t_0[\![\textbf{wemove } p, A^{p_0}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_0 \cdot t_0 +_t \zeta[\langle uu, \emptyset\rangle] \Downarrow @p \cdot t[A^d \cup A^{p_0}_{t_0}]} \quad (4.5)$$

where $p_0 \equiv (x_0, y_0, z_0)$. For strong mobility, given the source and destination ambient states A^{p_0} and A^p, time t_0 and the current state $\langle r, x_0, y_0, z_0, t_0\rangle$, an operational semantics for the **flyto** p **time** q statement, which moves the computation to the position $\langle x, y, z\rangle$ using a set timeout q, can be as follows:

$$\frac{@p_0 \cdot t_0[r \subseteq A^{p_0}_{t_0}] \quad @p \cdot t[r \subseteq A^p_t] \quad @p_0 \cdot t[t -_t t_0 <_t q \wedge \neg f\pi(p,t)]}{@p_0 \cdot t_0[\![\textbf{flyto } p \textbf{ time } q, r \cap A^{p_0}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p \cdot t[r \cap A^p_t]} \quad (4.6)$$

where $(p_0 \neq_s p \wedge r \not\subseteq A^{p_0}_t)$ $\underline{\text{v}}$ $(p_0 =_s p \wedge r \subseteq A^{p_0}_t)$ and

$$t \geq_t t_0 +_t \Psi +_t \Delta l +_t \Delta s +_t I\mu \quad (4.7)$$

where Δl is latency, Δs is the time interval due to the code size, and $I\mu$ is the time for interpreting the **flyto** instruction. The semantics of the timeout for this operation is as follows:

$$(\exists!\ t \in \mathbb{T})\ \frac{@p_0 \cdot t_0[r \subseteq A^{p_0}_{t_0}] \quad @p_0 \cdot t_0 +_t q[r \subseteq A^{p_0}_t \wedge f\pi(p, t_0 +_t q)]}{@p_0 \cdot t_0[\![\textbf{flyto } p \textbf{ time } q, r \cap A^{p_0}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_0 \cdot t_0 +_t q[r \cap A^{p_0}_t]} \quad (4.8)$$

Here is the semantics for the + operation:

$$\frac{\begin{array}{c}@p_0 \cdot t_0[\![a_1, s]\!] \overset{\text{eval}}{\rightsquigarrow} @p' \cdot t_0 +_t I_{a_1}[\langle u, s\rangle] \\ @p' \cdot t_0 +_t I_{a_1}[\![a_2, s]\!] \overset{\text{eval}}{\rightsquigarrow} @p \cdot t_0 +_t I_{a_1} +_t I_{a_2}[\langle v, s\rangle]\end{array}}{@p_0 \cdot t_0[\![a_1 + a_2, s]\!] \overset{\text{eval}}{\rightsquigarrow} @p\ddot{\mu} \cdot t_0 +_t I_{a_1} +_t I_{a_2} +_t I[\langle u + v, s\rangle]}$$

where s is any state, I is the time for evaluating the operation once the machine has the operands. $\boldsymbol{uu}$ permits that almost all operators of the underlying machine are lazy, according to the following rule:

$$\frac{@p_0 \cdot t_0[\![a_1, s]\!] \overset{\text{eval}}{\rightsquigarrow} @p \cdot t_0 +_t I_{a_1}[\langle \boldsymbol{uu}, s\rangle]}{@p_0 \cdot t_0[\![a_1 + a_2, s]\!] \overset{\text{eval}}{\rightsquigarrow} @p\ddot{\mu} \cdot t_0 +_t I_{a_1} +_t I[\langle \boldsymbol{uu}, s\rangle]}$$

The pair of rules for $-$, $=$ and $<$ are the same as for $+$, except that the operator is different. Notice that I am assuming that the above operands

do not have any side-effect and, therefore, their evaluations do not move the computation, although the mobility can exist due to physical movement, $\ddot{\mu}$. Now here the assignment statement:

$$\frac{@p_0 \cdot t_0[\![a, s]\!] \overset{\text{eval}}{\rightsquigarrow} @p \cdot t_0 +_t I_e[\langle u, s\rangle] \quad @p \cdot t_0 +_t I_e[\Delta(s, \gamma V, u) = s']}{@p_0 \cdot t_0[\![V := a, s]\!] \overset{\text{exec}}{\rightsquigarrow} @p\ddot{\mu} \cdot t_0 +_t I_a +_t I_{:=}[s']}$$

where $s' \overset{def}{=} \langle r[V/u], x\ddot{\mu}x, y\ddot{\mu}y, z\ddot{\mu}z, t_0 +_t I_a +_t I_{:=}\rangle$, and I_a is the interpretation time for evaluating the expression, and $I_{:=}$ is the time for assigning the value u to the variable.

The other local expressions and statements are similar to the above. A semantics for the **while** statement is as follows:

$$\frac{\begin{array}{c} r_{t_0} \subseteq A^{p_0}_{t_0} \quad r_t \subseteq A^p_t \quad @p_0 \cdot t_0[\![b, r_{t_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{eval}}{\rightsquigarrow} @p_0\ddot{\mu} \cdot t_0 +_t I_b[\langle v, r_{t_0+I_b}\rangle] \\ v \neq 0 \quad @p_0\ddot{\mu} \cdot t_0 +_t I_b[\![C, r_{t_0+I_b} \cap A^{p_0}_{t_0+I_b}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_i \cdot t_t[r_{t_i} \cap A^{p_i}_{t_i}] \\ r_{t_i} \subseteq A^{p_i}_{t_i} \quad @p_i \cdot t_i[\![\textbf{while } b \textbf{ do } C, r_{t_i} \cap A^{p_i}_{t_i}]\!] \overset{\text{exec}}{\rightsquigarrow} @p \cdot t[r_t \cap A^p_t] \end{array}}{@p_0 \cdot t_0[\![\textbf{while } b \textbf{ do } C, r_{t_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p \cdot t[r_t \cap A^p_t]}$$

where $t_i =_t t_0 +_t I_b +_t IC_{t_0+I_b}$, $t >_t t_i$ and $IC_{t_0+I_b}$ is the time for executing the statement C at time $t_0 +_t I_b$, where I_b is the time for evaluating the Boolean expression b. A semantic rule for finishing the **while** loop is as follows:

$$\frac{@p_0 \cdot t_0[\![b, r_{t_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{eval}}{\rightsquigarrow} @p_0\ddot{\mu} \cdot t_0 +_t I_b[\langle 0, r_{t_0+I_b}\rangle]}{@p_0 \cdot t_0[\![\textbf{while } b \textbf{ do } C, r_{t_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_0\ddot{\mu} \cdot t_0 +_t I_b[r_{t_0+I_b} \cap A^{p_0}_{t_0}]}$$

Once two or more agents are at the same coordinate, they usually communicate locally. To provide local communication, every agent has its input queue, which is part of the state. As briefly defined in the algebra signature, $\Delta : S \times Loc \times Val \longrightarrow S$ updates the queue with a value of type Val, given a state in S and location in Loc. Moreover, $\chi : S \longrightarrow S$ removes the first element from the agent queue by receiving a state and producing another one. $\xi : S \longrightarrow \mathcal{A}^R \times S$ informs the first element of the agent queue (removing it), which is a pair that contains the value and the new state. As before, $\gamma : Var \longrightarrow Loc$ is the function that, given a variable, locates its storage. Thus, here I am not concerned with delays or faults:

$$\frac{\xi@p_0 \cdot t_0[r_{t_0} \cap A^{p_0}_{t_0}] = \langle u, r_{t_i}\rangle \quad u \neq \textbf{\textit{uu}} \quad \xi@p_0\ddot{\mu} \cdot t_i[r_{t_i} \cap A^{p_0}_{t_i}] = \langle v, r_{t_j}\rangle \quad @p_0\ddot{\mu}\ddot{\mu} \cdot t_i +_t I2_r[\Delta(\chi\Delta(\chi r_{t_j}, \gamma par, v), \gamma V, u) = r_t]}{@p_0 \cdot t_0[\![\textbf{read to } V, r_{t_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_0\ddot{\mu}\ddot{\mu}\ddot{\mu} \cdot t[r_t \cap A^{p_0}_t]}$$

where *par*, which stands for *partner*, is a system variable that informs the sender id ϕ_{par}, $t =_t t_i +_t I2_r$ and $I_r =_t I1_r +_t I2_r$ is the time for interpretation of the **read** statement, composed of these two parts, and $t_i \geq_t t_0 +_t I1_r$. $I1_r$ is the interval of time from t_0 until obtaining $\langle u, r_{t_i}\rangle$, while $I2_r$ indicates the interval of time from t_i until receiving $\langle v, r_{t_j}\rangle$.

If the queue is empty, the evaluation results in $\textbf{\textit{uu}}$, and there is no modification in the state of the queue:

$$\frac{\xi@p_0 \cdot t_0[r_{t_0} \cap A^{p_0}_{t_0}] = \langle \textbf{\textit{uu}}, r_{t_i}\rangle \quad @p_0\ddot{\mu} \cdot t_1[\Delta(r_{t_0}, \gamma V, \textbf{\textit{uu}}) = r_t]}{@p_0 \cdot t_0[\![\textbf{read to } V, r_{t_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_0\ddot{\mu}\ddot{\mu} \cdot t[r_t \cap A^{p_0}_t]}$$

where $t_0 <_t t_1 <_t t$.

Given $\eta : S \times \mathcal{A}^R \longrightarrow S$ from the signature, which inserts a value in $\mathcal{A}^R$ in the queue, the opposite operation is **write**, which writes the real number stored in the variable denoted by a, to the channel denoted by h, with the following semantics:

$$\frac{@p_0 \cdot t_0[\![a, r_{t_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{eval}}{\rightsquigarrow} @p_0\ddot{\mu} \cdot t_0 +_t I_a[u] \quad @p_0\ddot{\mu} \cdot t_0 +_t I_a[\eta(\eta(r^h_{t_0+I_a}, u), c) = r^h_t]}{@p_0 \cdot t_0[\![\textbf{write to channel } h \textbf{ from } a, r_{t_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_0\ddot{\mu}\ddot{\mu} \cdot t[r_{t_0}] \;\Downarrow\; @p_0 \cdot t[r^h_t]}$$

where r^h_t is the state r of some computation π_h at time t; and c is the Id of the computation, which corresponds to ϕ_{par}, i.e. the sender Id, which is also used in the **read** statement. Therefore, the **write** statement writes in the receiver's queue the value in $\mathcal{A}^R$ and c in this order. As stated in the corresponding rules, the **read** statement obtains these elements in the same order. As well as the $\ddot{\mu}$ shift, code mobility and local communication, an additional facility for mobile agents is remote communication among agents. To define remote communication formally, I first ought to set the following rules:

- $\kappa \in SObj$ at $\langle x, y, z\rangle$ is visible by all agents if there is no agent at $\langle x, y, z\rangle$;

- $\kappa \in SObj$ at $\langle x, y, z\rangle$ is not visible by any agent at a different point if there is some agent at $\langle x, y, z\rangle$. In this case, any attempt to access this value results in ***uu***.

- $\kappa \in SObj$ at $\langle x, y, z\rangle$ is visible by agents at $\langle x, y, z\rangle$. In this case, I say that κ is local to those agents. And, since $\kappa \equiv \langle v, Y\rangle$, for all r in Y, v is visible by r.

In this model, to see the content of a point $\in \mathbb{R}$ at $\langle x, y, z\rangle$, agents execute the statement **view**$\langle x, y, z\rangle$ **time** q, whose semantics in the @-logic is:

$$\frac{\begin{array}{c}@p_0 \cdot t_0[p \neq_s p_0 \wedge \Theta @p \cdot t_1[r^p] = \langle u, \emptyset\rangle \wedge u \in \mathbb{R}] \\ @p_0 \ddot{\mu} \cdot t_f[t_f -_t t_0 <_t q \wedge \neg f\pi(p, t_f)]\end{array}}{@p_0 \cdot t_0[\![\textbf{view } p \textbf{ time } q, r^{p_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{eval}}{\rightsquigarrow} @p_0 \ddot{\mu}\ddot{\mu} \cdot t_f[\langle u, r^{p_0}\rangle]} \tag{4.9}$$

where

$$t \geq_t t_1 \geq_t t_0 +_t \Psi +_t \Delta l_1 +_t \Delta s_1 +_t I_i \tag{4.10}$$

and

$$t_f \geq_t t_1 +_t \Psi +_t \Delta l_2 +_t \Delta s_2 \tag{4.11}$$

where I_i is the time for interpreting the **view** statement. For remote attempt to see a value that is local to other agents, there is another rule:

$$\frac{\begin{array}{c}@p_0 \cdot t[p \neq_s p_0 \wedge \Theta @p \cdot t_1[r^p] = \langle u, Y\rangle \wedge Y \neq \emptyset] \\ @p_0 \ddot{\mu} \cdot t_f[t_f -_t t_0 <_t q \wedge \neg f\pi(p, t_f)]\end{array}}{@p_0 \cdot t_0[\![\textbf{view } p \textbf{ time } q, r^{p_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{eval}}{\rightsquigarrow} @p_0 \ddot{\mu}\ddot{\mu} \cdot t_f[\langle \textbf{\textit{uu}}, r^{p_0}\rangle]} \tag{4.12}$$

That is, the value in the problem domain is not accessible and, therefore, ***uu*** is provided instead.

For local access in permitted time, let $\{a_1, ..., a_n\} \subset \triangle$, and c is the Id of the agent who is currently computing. Thus, the semantics is as follows:

$$\frac{@p_0 \cdot t_1 +_t \Delta t_1[\Theta @p \cdot t_1[r^p] = \langle u, \{c, a_1, ..., a_n\}\rangle \wedge \Delta t +_t I_i <_t q]}{@p_0 \cdot t_0[\![\textbf{view } p \textbf{ time } q, r^{p_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{eval}}{\rightsquigarrow} @p \ddot{\mu} \cdot t_0 +_t \Delta t +_t I_i[\langle u, r^{p_0}\rangle]} \tag{4.13}$$

where $t_1 \geq_t t_0$. The semantics for the timeout situation during the **view** request follows the next rule:

$$\frac{@p_0\ddot{\mu} \cdot t_f[t_f -_t t_0 =_t q \wedge f\pi(p, t_f)]}{@p_0 \cdot t_0[\![\textbf{view } p \textbf{ time } q, r_{t_0}^{p_0} \cap A_{t_0}^{p_0}]\!] \overset{\text{eval}}{\rightsquigarrow} @p_0\ddot{\mu} \cdot t_f[\langle \textbf{\textit{uu}}, r_{t_f}^{p_0}\rangle]} \tag{4.14}$$

where t_f is constrained in accordance with the expression 4.11.

In this model, to provide some form of communication, agents can throw a real number at any place in space with timeout q, and this operation takes some time to be completed, as described below. A place occupies one point in $\mathbb{E}^3$. The value $\textbf{\textit{uu}}$ can also be thrown and indeed can be part of programming languages constructs as introduced in[106]. Chapters 6 and 7 contain the same material.

The operational semantics of the **throw** instruction is as follows:

$$\frac{\begin{array}{c}@p_0 \cdot t_0[\![a, r^{p_0} \cap A_{t_0}^{p_0}]\!] \overset{\text{eval}}{\rightsquigarrow} @p_0\ddot{\mu} \cdot t_0 +_t I_a[\langle v, r^{p_0}\rangle] \\ @p_0 \cdot t[t -_t t_0 <_t q \wedge \neg f\pi(p, t)] \quad @p_0\ddot{\mu} \cdot t[\Theta @p \cdot t[r^p] = \langle u, \emptyset\rangle]\end{array}}{@p_0 \cdot t_0[\![\textbf{throw } a \textbf{ to } p \textbf{ time } q, r^{p_0} \cap A_{t_0}^{p_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_0\ddot{\mu} \cdot t[r^{p_0}] \Downarrow @p \cdot t[\langle v, \emptyset\rangle]} \tag{4.15}$$

where

$$t \geq_t t_0 +_t \Psi +_t \Delta l +_t I_a +_t Th$$

where Th is the time for interpreting the throw instruction.

In the case that there is already some agent at $\langle x, y, z\rangle$, if an agent throws a number from another place, there is no effect. Therefore, if

$$\{a_1, a_2, ..., a_n\} \subseteq \triangle$$

is a non-empty set of agents, and $t_0 \leq_t t_1 \leq_t t$:

$$\frac{\begin{array}{c}@p_0 \cdot t_0[\![a, r^{p_0} \cap A_{t_0}^{p_0}]\!] \overset{\text{eval}}{\rightsquigarrow} @p_0\ddot{\mu} \cdot t_0 +_t I_a[\langle v, r^{p_0}\rangle] \\ @p_0 \cdot t[t -_t t_0 <_t q \wedge \neg f\pi(p, t)] \quad @p_0\ddot{\mu} \cdot t[\Theta @p \cdot t_1[r^p] = \langle u, \{a_1, a_2, ..., a_n\}\rangle]\end{array}}{\begin{array}{c}@p_0 \cdot t_0[\![\textbf{throw } a \textbf{ to } p \textbf{ time } q, r^{p_0} \cap A_{t_0}^{p_0}]\!] \overset{\text{exec}}{\rightsquigarrow} \\ @p_0\ddot{\mu} \cdot t[r^{p_0}] \Downarrow @p \cdot t[\langle u, \{a_1, a_2, ..., a_n\}\rangle]\end{array}} \tag{4.16}$$

The timeout condition for the **throw** statement has the following rule:

$$\frac{\begin{array}{c}@p_0 \cdot t_0[r^{p_0} \subseteq A_{t_0}] \quad @p_0\ddot{\mu} \cdot t[t -_t t_0 =_t q] \quad @p_0 \cdot t_0[r^{p_0} \subseteq A_t] \\ @p_0 \cdot t[f\pi(p,t)] \quad @p_0\ddot{\mu} \cdot t[\Theta @p \cdot t_0[r^p] = \langle u, \emptyset \rangle]\end{array}}{\begin{array}{c}@p_0 \cdot t_0[\![\mathbf{throw}\ a\ \mathbf{to}\ p\ \mathbf{time}\ q, r^{p_0} \cap A_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} \\ @p_0\ddot{\mu} \cdot t[r^{p_0} \cap A_t] \downdownarrows @p \cdot t[\langle u, \emptyset \rangle]\end{array}} \tag{4.17}$$

Notice that real numbers can contain the codification of some finite mobile agent. This means that the **throw** statement is able to model this model. I formalize the sequence of statements:

$$\frac{\begin{array}{c}@p_0 \cdot t_0[\![C_1, r_{t_0} \cap A_{t_0}^{p_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_i \cdot t_i[r_{t_i} \cap A_{t_i}^{p_i}] \\ @p_i \cdot t_i[\![C_2, r_{t_i} \cap A_{t_i}^{p_i}]\!] \overset{\text{exec}}{\rightsquigarrow} @p \cdot t[r_t \cap A_t^p]\end{array}}{@p_0 \cdot t_0[\![C_1; C_2, r_{t_0} \cap A_{t_0}^{p_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p \cdot t[r_t \cap A_t^p]}$$

In section 4.5, I consider a possible collective rôle for generating mutations, while in this section I introduce the operation **SpreadOut** for agent mobility, and mutation becomes merely individual and random, as in Darwin's view. Mutation can be used by agents e.g. for playing chess, thus implementing an evolutionary model of computing. This model can be used to solve combinatorial problems in general. In practice, models for general computation with mobility and some global environment should assume, for example, that in the future computers will have receptors of mass electronic media. The essential difference between this form of mobility and strong mobility is that, in the latter, the machine has to know the destination address before performing the operation, while in the former, the machine simply broadcasts the agent. As a practical consequence, the broadcasting mobility lets agents migrate to a much greater number of places almost at the same time. Because of these differences, here I regard that they require two different primitives, with or without destination address. In this way I introduce *broadcasting mobility* and its operational semantics.

The crucial aspect here is that, unlike the **flyto** statement, this new proactive move is broadcasting publicly, and without any address specification for the destination. The only condition for the continuation is the agent be accepted by the destination, and for which I should set the predicate *receptive*, which states that the destination is receptive to the present computation. In

this way, *rec(r)* indicates that the surrounding ambient is receptive to the state r, and now the **SpreadOut** operation can be conceived and included in the present model. Because the operation does not require any operand, here I use the syntax **SpreadOut**. The semantics is as follows:

$$\frac{\forall p \in \mathcal{U} \quad @p_0 \cdot t_0[r \subseteq A^{p_0}_{t_0}] \quad @p \cdot t_i[rec(r) \in A^p_{t_i}] \quad @p \cdot t_i[r \not\subseteq A^p_{t_i}] \quad @p \cdot t[t =_t t_0 +_t q \wedge \neg f\pi(p,t) \wedge r \subseteq A^p_t]}{@p_0 \cdot t_0[\![\textbf{SpreadOut}\ q, r \cap A^{p_0}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_0 \ddot{\mu} \cdot t_0 +_t I_s[r \cap A^{p_0}_{t_0+_t I_s}] \Downarrow @p\ddot{\mu} \cdot t[r \cap A^p_t \setminus \{rec(r)\}]}$$

and, finally, one additional rule for the case that there is no receptions:

$$\frac{\forall p \in \mathcal{U} \quad @p_0 \cdot t_0[r^{p_0} \subseteq A^{p_0}_{t_0}] \quad @p \cdot t_i[(rec(r^{p_0}) \notin A^p_{t_i}) \vee r^{p_0} \subseteq A^p_{t_i}]}{@p_0 \cdot t_0[\![\textbf{SpreadOut}\ q, r^{p_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_0 \ddot{\mu} \cdot t_0 +_t I_s[r^{p_0} \cap A^{p_0}_{t_0+_t I_s}]}$$

where both $t =_t t_i +_t \Delta t$ and $t_i \geq_t t_0 +_t \Psi +_t \Delta l +_t \Delta s +_t I_s$ hold for every p, and I_s is the time for interpreting the **SpreadOut** statement. Notice that this operation is asynchronous.

It is *not* impossible to have timeout for the **SpreadOut** operation, and the corresponding rule is as follows:

$$\frac{\forall p \in \mathcal{U} \quad @p_0 \cdot t_0[r^{p_0} \subseteq A^{p_0}_{t_0}] \quad @p \cdot t_0 +_t q[f\pi(p, t_0 +_t q) \vee r^{p_0} \subseteq A^p_{t_0+_t q}]}{@p_0 \cdot t_0[\![\textbf{SpreadOut}\ q, r^{p_0} \cap A^{p_0}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @p_0 \ddot{\mu} \cdot t_0 +_t I_s[r^{p_0} \cap A^{p_0}_{t_0+_t I_s}]}$$

Also notice that, from the application point of view, there may exist a form of mobility in the opposite direction, which would implement the concept of centralization at a higher level of abstraction by using strong mobility.

As well as efficiency concerns, there are important philosophical issues here: like sperm cells in a female animal, mobile agents can *compete* for the best solution, which leads designers to questions in *ethics*. "What should agents be allowed to do?" This is an open question. Ethics for mobile agents on a global environment may be even more complicated because agents often transcend boundaries of countries.

Applications such as chess form a model of computing to solve problems that include the possibility of combinatorial explosion. One can provide an interesting algorithm to find a mobile agent on a network.

Another issue is how should a mobile agent system prevent agents from multiplying in an uncontrolled manner? What is the reasonable cost, overhead and burden of such an operation? To date, there has not been a technology that guarantees total security for a host that receives a broadcasted mobile agent, but any mobile code system that support such operations should see the network as public. Chapter 5 introduces a secure model for mobile computing with agents. A similar problem is to access WWW hosts in parallel by software modules, while this idea might be desirable from some people's point of view but could also cause traffic jams if performed by programs too often. Therefore, it is clear that there is an open door to philosophy in computer science, while AI gets closer to the foundations of computer science. In the following section, I demonstrate that this also holds for applications of computation.

4.7 Computing in the real world

Examples of mobile code languages are: Telescript[316] for strong mobility, and Java[147] and Obliq[55] for weak mobility, without taking into consideration the current stage of the technologies that support code mobility.

In comparison to other paradigms for code mobility, a common criticism of the mobile agent paradigm is that mobile agents do not maintain connections upon migration[58]. This criticism may be appropriate or not, depending on the philosophical point of view, the metaphor, the model and, occasionally, how well the system is implemented. For example, I think that the key concept for agents and distributed systems is *communication* and that connection is an implementation issue. Connection also is a matter of abstraction. In some models, agents maintain connections during their move by communicating to each other when appropriate, but this happens at the programming layer of abstraction, and is not provided by the programming language. Some agents are sent only for communicating and then halt, for example. This might not be seen as a problem as humans without mobile phones do not maintain connections either. Dangling pointers as a result of strong mobility, for example,

at some level is normal, not a problem: as an analogy, what happens when we call someone and he or she is not around at that moment? No matter whether he or she can use a mobile phone, we might want to try again later or leave a message or dial another number. Such decisions are at the application level rather than the mobile agent system level.

Regarding communication, the language and its agent-based system are responsible for providing general and suitable features for programming agents, and suitability depends on the programming model.

In [105], metaphorically we follow a modern world composed of objects such as individuals, airplanes, airports, places and providers, and, because of this, the model for global computing with strong mobility is technically simple to conceive and use, but there are open philosophical questions that deserve careful consideration before designing any global system. Some laws can be established for the whole global system, and these laws ought to be public, declarative, and fixed.

In a public environment, nature-based models, such as evolutionary computing with individual mutations simulated, ought also to be linked to the human-based approach because there are a number of peoples with different cultural, religious and philosophical backgrounds. This also leads the foundations of computer science to *ethics* and *politics*, more generally, to philosophy.

We might be able to perceive that the industrial revolution, as well as positive concepts of liberty, equality and fraternity, reflect a change that underlies the collective consciousness and have gradually influenced the mankind, and that computer science is part of this evolutionary process. Nowadays a person inserts a few coins in a coffee machine, chooses what he or she prefers, picks up his or her preferred coffee, and receives the change. This reflects the observation that *individuality* is another keyword in this trend, for programs and systems must be adaptable to fit the needs of the individuals. Indeed, as already mentioned in this chapter, one of the applications of mobile agents is to personalize clients and servers. Somehow, both individuality and generality are complementary key concepts in technology, since machines usually

produce many copies of the same product. However, programs can deal with these concepts very well adapting themselves to specific needs.

For almost everyone, the source of human intelligence is individual, but one who travels around abroad can more easily perceive that every country has its own *identity*. Thus, I also conceive the ideas of collective intelligence and qualitative intelligence.

Since global computers are relatively complex machines, among many notions from human sciences, philosophy and *religion* have consequences on software development for this scenario. As an example, if designers or users believe in God, they may think that spirits are individual intelligent entities that are maintained after human beings death. Furthermore, if a designer or user believes in reincarnation (whose doctrine, as well as being found in Indian religions and philosophy, was held in western philosophy by Plato and apparently by others including Pythagoras), for example, he or she thinks that some individual inheritance is brought to human beings from their previous existences[262]. In most religions, individuality is maintained after life. The soul is seen as an intelligent entity whose individuality is also manifested during one's life.

Platonist scientists normally believe in God, although intuitionist scientists are not necessarily atheist.

A mobile agent system for the Internet ought to take into consideration that different peoples mix together. The fact that designers establish their own laws entails diversity of global systems. I believe that systems that establish fairer laws attract more users. I also think a model that provides a definition of agent by establishing rules synthesized from the real life, is easier, both to implement and to use. The main involved technical difficulty is to implement a suitable programming language and efficient system with strong mobility together with a satisfactory level of security[220] and suitable AI-based features, because an inference engine needs control for its efficient reasoning process, without interruption from the user's program. Inferences can trigger imperative execution which in turn can contain the **flyto** state-

ment, described in the previous section. The efficiency problem is much more difficult to solve than to move computation from imperative programs, where virtual machines interpret simple instructions. On the other hand, there are AI techniques that make it easier to provide an agent-based model inspired by human beings, as well as providing the programmer with a more suitable way of representing knowledge on the real world. Uncertainty and reasoning with partial information are two examples of AI techniques useful in global computing.

If the global system is public, laws should not be limited to laws of users' etiquette but preferably laws that constrain agents behaviors. In this way, agent laws should follow from international human laws.

To design a global computer is similar to creating a simplified world. The interactions between the computer and the real world become so complex that subjective factors enter the computer science.

For considering that a notion of computation is more complete than what has been used to be presented, as previously stated, both psychology and philosophy have to be considered. As an example of the relevance of the former, if by any chance a person makes some calculation sleeping, he or she computes without being conscious or having a precise control over the calculation. Nonetheless, such an action might be from a different model of computation. Yet, if that calculation is performed while the person is on a train, conceptually, there is a new kind of mobility. The person may also speak as sleeps, or perform another kind of action on the train. Moreover, since human beings spend around one third of their lives sleeping, this kind of experience is not so rare in the daily life. And since the presented discussion on computation is conceptual, such observations under potentially different computational models become fundamental.

Many models impose constraints upon programming languages. In addition to *uu*, a constraint was solved by the adoption of declarative uncertainty representation, as humans usually make use of subjective judgment based on feeling. Uncertainty can also be used to represent confidence on a particu-

lar public piece of information, which is another subjective feature. Thus, programmers can declare that some particular data is 80% reliable while the remaining 20% corresponds to technical faults, for example. Uncertainty is also present in rules, as some antecedents contribute more than others to validate a hypothesis.

4.8 Conclusion

Among other conclusions, I discussed mobility and a more general notion of computation that includes physical, psychological and philosophical factors. I propose four forms of mobility in this physical and simplified model.

Up to some level of abstraction, a simple and different model in $\mathbb{E}^3$ has been presented here by defining a machine together with a somewhat minimal set of constructs that complements the traditional notion of computation, as well as an operational semantics for these constructs.

The Internet is used by people from different places from different backgrounds as philosophy increases its importance for computer science. Furthermore, this change from mathematics to also include other studies also opens a door to changes in scientific methods. Science in computer science has a broader sense.

Chapter 5

A Complete Solution for Mobile Agents on Global Structures

In the present chapter I introduce a complete and general solution for mobile agent systems on global environments such as the Internet, informally using a top-down approach without irrelevant details, and discussing all problems as well as presenting connected sub-solutions.

While in chapter 4 I introduce a physical semantics of computation regarding a simplification of the real world as the context, here the solution is based on global computers and internets. Here I consider human beings. In particular, I believe that without centralizing the security problem and closing the software from the public, one cannot guarantee a satisfactory level of security to the users in a feasible way. This can be viewed as an odd academic proposal, and I would certainly agree, but it is also a consequence of the nature of the human condition in the real world.

5.1 Introduction

Any subset of an Internet-like network can be abstractly seen as a very large, inefficient and non-reliable computer, instead of a network. As an example, [256] is an updated bibliography for data mining research whose approach is clearly different from ours. Thus, one global computer[57] (GC) consists of an homogeneous set of interpreters (also called virtual machines) on a global network where application programs can run. I use here *Programming for a Global Computer* (PGC) to stress particular features of such a network, e.g. to be public. However, it is a good idea to keep in mind that a *general* purpose programming language ought to consider a global environment.

5.1.1 Mobility and its Paradigms

In this sub-section I make comparisons between paradigms of code mobility. Although these paradigms are technical, I am demonstrating that only one of them is enough for the present theoretical definition later in this chapter. As a side effect, this discussion might also be useful with respect to general-purpose programming languages.

So far, as more general than the client-server paradigm of distributed systems, there are three paradigms of code mobility, namely, remote evaluation (REV), code on demand (COD) and mobile agents (MA). REV and COD are based on the concept of *weak mobility* while MA is ideally and normally based on the concept of *strong mobility*. In this chapter, I refer to MA as having strong mobility. Although this classification[3] is important for technological reasons, both forms of weak mobility can be implemented by using only

[3]Briefly, for those who are not familiar with those terms, strong mobility is the ability of a mobile code language to permit the computation to move from one interpreter to another, possibly remote, while weak mobility is the ability of a mobile code language to permit a running program to be bound dynamically to some coming code, either because the local interpreter requested previously or because the code was sent. In the latter case, the coming code can either be linked or run from scratch.

strong mobility. That is, both forms of weak mobility are particular cases of strong mobility. This is the main subject of this sub-section.

REV can be implemented as MA in which the first instruction is *moveto*, which allows the agent to move and evaluate. COD can also be implemented with MA together with communication between mobile agents: one agent moves to call another agent to come. The latter agent is the piece of code that would run in the COD paradigm. Because I am looking for primitives powerful enough to capture code mobility in the most general sense, I am going to consider only strong mobility in the present model of computation.

As opposed to *client-server* (C-S), in [123], the authors present an interesting metaphor to explain the paradigms to support code mobility. There, the authors explain that C-S, REV, COD and MA are the main design paradigms. Here I initially consider that they are the possible paradigms, then I introduce one and include it in the present model of computation.

Here, at a more practical level, I state that MA is probably the most suitable paradigm for general-purpose programming languages by explaining that they are probably the most general for communication and mobility. Several mobile agents languages and systems were conceived to be based on Java[147] and JVM[199], developed by Sun Microsystems. Such agents move code and data but they do not move the execution state. Application programs in *Aglets*[194], for instance, reinitialize after migration. Java with COD and JVM are certainly amazing and deserve respect, among other important technologies. But although Java is a well designed programming language, its virtual machine architecture, instruction set and program format are open, which means that security and privacy are important issues concerning public mobile agents. Moreover, as JVM does not support a low-level instruction that implements proactive migration along with state migration, to date technologies based on JVM is somehow limited, from the point of view of mobile agents based applications such as electronic commerce. Therefore, a statement that MA is the most suitable paradigm in programming languages ought to be regarded in the context of applications in general as *a* design goal. Special-

purpose languages are designed to be more appropriate to certain specific tasks than a general purpose language. Moreover, in this chapter I am regarding only MA with strong mobility.

Clearly, MA is a paradigm for code mobility not less general than C-S, REV or COD. Thus, because applications such as distributed CPU usage, on-line transactions with subsequent use of CPU as well as the presented model cannot be implemented by one paradigm other than MA, therefore, MA is more general than C-S, REV or COD.

The possibility of mixture of paradigms is not being neglected. But here, because my main motivation is the (informal) semantics of computation, I only make individual and one-to-one comparisons between MA and C-S, REV and COD.

5.1.2 Contents of the Chapter

To date, there are not many mobile agent languages and systems, and information on them is easily found on WWW. Telescript[316, 317] by General Magic is pioneering and probably the best example of mobile agent technology. However, in addition to being suitable for different programming languages, the concept of identity in the present solution contrasts with the solution adopted by Telescript, where identity is simply a name. More recently, a few mobile agent technologies have been designed and implemented atop the Java Virtual Machine (JVM), such as Aglets Workbench[194] by IBM. However, because JVM is open, such systems are not secure enough for public applications on the Internet.

Section 5.2 briefly discusses programming for a global computer. Section 5.3 presents a general model for computation on a global environment by discussing problems and introducing solutions, and section 5.4 explains why I leave communication between and among agents untouched. Finally, section 5.5 contains some concluding remarks.

5.2 Programming for a Global Computer

As global computers will never be so reliable nor efficient as local interpreters only, traditional programming might not be appropriate for global environments. Decisions must be made very often on WWW due to some delay, and this requires specific language constructs.

I recast here the characteristics of PGC from the programmers point of view: *long distances* is the first characteristics of global computers, hence *delays*. Programmers are *aware of locations* of resources. This not only allows the use of *resources* spread out on the global computer but also allows the control of distances between places, because the slowness of light is relevant here in performance. The programmer should be able to set a *time limit* under which his or her statement will wait for a remote operation to complete. If the operation was not completed by then, his or her program can try the same operation on a different site. *Fault* in connection is no longer regarded as exception but instead as a normal situation. Other requirements of PGC are *security* guaranteed by the underlying system, *code mobility*, *robustness* and *communication*, both synchronous and asynchronous. *Slowness* of the sequential computation is another characteristic. *Heterogeneity* of hardware and operating systems have been mentioned as typical but they are irrelevant in programming. *Parallel computation* is strongly suggested, in such a way that computing in a global environment may become even more *efficient* than local computation only.

I am regarding the above characteristics as being what defines PGC and, hence, any programming language or system for general purpose must provide at least all of those characteristics. Because of this, it turns out that no programming language or system to date is suitable for global environments, although there are languages on the top of mobile-code systems. Surprisingly, *concurrent programming* is not one of the requirements for PGC.

5.3 Problems

In this section, I analyze problems together with their solutions, because they are linked concepts and one solution to a problem may create other subproblems and so on. On the other hand, key words are emphasized for being searched.

I define *provider* a stationary agent in the present solution. Here, I define *Individual* as a mobile agent to stress that an Individual moves and are not split, although there are exceptional cases, as are explained later. To generalize, I refer to either *provider* and *Individual* as *agent* or *module*. The former is dynamic whereas the latter is static.

For philosophical reasons and to keep the solution uniform, both providers and Individuals contain themselves, that is, they cannot be broken into smaller modules but instead they are able to create other agents. The present solution always guarantees a unique identity for different agent although some attributes are shared after that creation. Moreover, providers do not create Individuals, and Individuals do not create providers. Symmetrically, providers and Individuals cannot be dynamically linked but instead they often *communicate*, providers with Individuals. In addition, providers can communicate with other processes designed by using different technologies. Thus, I refer to providers and Individuals in this solution as *self-contained modules* to distinguish them from *mobile agents* in some other solutions.

Some mobile code technologies have been designed and implemented but nowadays there are few applications that offer *security* in such a way that the public feel secure. Extending from [308] to PGC, it becomes even more critical. Thus, we provide security to protect agents and their interactions as follows: communication between hosts and between CEs; the host from malicious visitors; visitors from malicious hosts; communication between Individuals; communication between an Individual and a provider; Individuals from being bothered by other agents; Memory space. CPU from time abuse. However, incoming agents do not need to be granted access rights, as will be

explained in this section.

The interpreter controls accesses by Individuals and providers to operating system calls.

The easiest and safest way to guarantee security in the solution is to initially forbid all critical operations and then allow only those operations that are necessary and do not violate the security policy. Input/output operations are regarded as such. Thus, the system guarantees that *no Individuals, by definition, perform directly input/output operations, but instead, communicate with providers that* may *perform the operations for the Individuals*. Therefore, providers must be able to recognize Individuals, which in turn, require the definition of the concept of *agent identity*, or *identity* for short.

Authentication of messages is easily solved by digital signature, but this technique is used only when the sender is not the receiver. In the present solution, at this level, we can safely regard the system as being both the sender and the receiver, which simplifies the solution, also from the user's point of view. Here, there is no need for users to have public and private keys to use the system, because the authenticity is already guaranteed by the mobile agent system (MAS) by keeping one secret key (or pair of keys) for the whole system. I call this key *system key*, from now on. This also keeps the solution consistent because the person responsible for a host is not necessarily responsible for the computations that are sent from his or her host, as Individuals can move from host to host more than once. Notice that an Individual is not a static document.

Researchers adopt the idea that incoming computations must be granted access rights[308], which may be consistent but produces practical problems: 'access' is a very broad concept when we talk about security in global computers. The concept of access should not be limited to the level of operating system but, instead, be extended to services at higher levels. To allow an agent to access files is too risky, for example. We want to state a permission such as "Abelardo is allowed to read *xyz.abc* file on Thursdays", for example. At such a level of detail and flexibility, there are typically many permissions in an

application system and, because of this, I do not adopt the approach of giving permissions beforehand. Instead, this approach allows providers to respond to Individuals requests in a programmable way, by accessing directly their identities and checking authorization on demand. Thus, all levels of access and services are controlled in a uniform way. Because there are typically so many permissions, declarative knowledge bases, e.g. composed by logic programming clauses, are natural candidates for representing such permissions.

In the present solution, the system has to provide a way of distinguishing Individuals from providers. This can be easily solved statically: if a program contains some **flyto** statement, the statement that causes the Individual proactive move, the compiler stamps the status of Individual in the generated code. On the other hand, if there are critical operations in the program, the compiler stamps the status of provider instead. A program cannot be Individual and provider, and, in case of both kinds of operation in the same program, the compiler reports an error message.

Because of the **flyto** special nature, it has to be a statement in the language, neither a method nor a library function. The same is true in the interpretable byte code: the **flyto** statement has to be generated as a corresponding virtual machine **flyto** instruction. This not only permits the compiler and interpreter to distinguish Individuals from providers, but also allows the interpreter to deliver Individuals to the local *airport* for departure. A library function call does not normally provide information at that level of detail. Therefore, *all* critical operations in providers are no longer traditional library functions or methods, no matter their syntaxes: the compiler and interpreter have to be able to recognize such operations, otherwise the system will probably not provide satisfactory security in the real world.

This creates another problem: how to protect the programming system from potentially malicious compilers that generate critical operations in Individuals. My solution is to hide the virtual machine architecture and the Individuals format. Although this solution may cause surprise for being proprietary, it closes the architecture to the world in the same way as purely

interpretable languages do. Furthermore, although it prevents other compiler designers from implementing other programming languages directly on the runtime system, it allows compilers to translate a program from another language to the source language owner of the interpreter. Moreover, the present solution does not require checking whether an Individual contains critical operations.

Identity in the present solution allows the public to even identify the programmer of an Individual, if necessary, as compilers could also have identities to be stamped in the generated code. The present solution does not discard the possibility of having different worlds, say global computers, that communicate, each world with its own programming language and implementation.

When a message arrives at a site, a program of the system called *airport* recognizes the format as an Individual, decrypts the incoming message by using the system key, verifies the Individual identity and verifies the integrity of the Individual. If everything is correct, then the airport verifies the Individuals passport, updates the passport and keeps a record about that arrival. If the message is not an Individual or the Individual integrity is not certified, the airport records the event and ignores the message.

When an Individual departs, the airport updates its passport, keeps a record about its departure, encrypts the Individual by using the system key, adds some header, and finally sends the package.

I divide *resources* in three classes: temporal, material and service. The first is CPU time plus some overhead due to instruction interpretation. The second may be persistent data while the third corresponds to responses to requests by agents. When an Individual arrives at a site, before running, the runtime system assigns a time interval for the Individual to leave the host, and this time is controlled by the system before interpreting every instruction. Thus, when an Individual leaves the host, its intervals of time in the host, including its effective time and total response time spent by providers, are available. A similar solution is adopted for memory allocation, controlled by the interpreter.

Another way to prevent from time abuse in some specific applications is to allow or forbid, as a local policy, Individuals whose programs contain iterative statements, such as **while**. Another policy is to inhibit some interpretable operations, such as system library calls or method invocation. For example, square roots might not be calculated on a host that offers a simple and public commercial service. The *airport* can inspect for both policies at arrival time.

A kind of resource that can be regarded as both material and service is executable code, which also can be delivered by agents as any material resource. An Individual can carry and deliver such code in such a way that it can be executed remotely as long as the parties wish. This partially solves the problem of a few applications that require more efficiency than interpreters can provide.

A particular case of porting executable code is when a new version of a mobile agent system is released. Mobile agents can then visit sites to update all copies as long as this is part of some contract. In other words, unlike the real world, the approach does not prevent an Individual from carrying the whole mobile agent system, including the interpreter and the airport, installing it and even continuing running on it. In fact, an Individual can install any application.

Identity is partially generated by the compiler, some other fields are filled in when the computation starts, and passports are updated by airports when Individuals arrive and depart. Therefore, the format of object code is not the same as an Individual format.

Although some systems make use of passwords, for public applications, identity is not intended to be explicitly passed by the caller as parameter but instead its passing must be implicit and always guaranteed by the runtime system. I initially define identity as an abstract data type that contains the following fields:

Entity flag (whether Individual or provider); Program owner's identification; Home city; Internet Home Address (notice the country code); Postal Home Address (optional); File name of the Individual at home; Initial inter-

preter version; The initial interpreter Id; (Local) Date-Time when the program started running; The initial interpreter time zone (optional); Latitude and longitude of the first interpretation (optional); User's cryptographic key (optional); User's password for the application (optional); Password of the application (optional); The Individual passport (list of tuples about departure or arrival).

Identity is a class in the adopted language. Notice that this concept of identity is unchangeable and its presence is guaranteed by the present solution. This concept, however, is normally extended at the level of application, which will depend on the set requirements for security. Agents use *communication* to identify others at the level of application. Here, passwords can be used to identify Individuals.

An agent is not allowed to move an Individual but instead to request the latter to move, because every Individual is the responsible for its move. The exception is a situation when the MAS punishes an Individual because it violated some security policy. For *remote evaluation*, a programmer can write an Individual with enough code and data to accept the request and perform the operation remotely, while another program simply invokes the Individual. Therefore, there is no statement in the system for *shipping* an Individual. Accordingly, in the present solution, there is no concept of *downloading*.

An Individual is globally referred to by the ⟨Program owner's Id, Internet Home address, file name at home, sequence⟩ tuple which is unique.

For many applications, a provider normally sleeps and, when awaken by another program, performs some operations and sleeps again. An Individual, in its turn, moves to a host, requests some resources, perhaps it blocks while services are being provided, pays for services, moves on to another host and so on. Besides the communication is local to a host, I adopt a mechanism somewhat similar to method invocation. But its granularity is wider, i.e. the communication is between different agents, possibly written by different companies, with no assumption about static type correctness, nor even the existence of particular methods. Thus, in accordance with the present holistic

philosophical view, compilers and linkers cannot see the whole picture, and this helps make global MAS feasible. The lack of a static global view is compensated by the ability to deal with partial information at the programming level, which is very important in any case.

In this solution, the concept of identity also allows a flexible scheme for porting material resources. As one example, an Individual I_1 might migrate from host H_1 to host H_2 without resources and, since at the destination, requests H_1 resources *from a provider* at H_2 which, in its turn, sends an Individual I_2 to H_1 that locally requests the resources from H_1 and take them back to H_2. The provider finally delivers the resources to I_1.

To date, almost all mobile-code languages adopt one or two fixed strategies to manage resources. These strategies are seen in chapter 6. Because I am looking for generality, in the present solution, whether the strategy is replication or sharing, or whether the replication is static or dynamic, or whether the latter is by copy or by move is entirely up to the programmer. For example, the Individual I_1 might migrate from a host H_1 to H_2 taking the resources eagerly.

Still regarding *communication* between programs, it can be *synchronous* or *asynchronous*. When the former is applied, it is possible for an Individual to migrate as part of its response to some message, while the calling process, the running interpreter, is waiting. If the Individual comes back before the time has expired, it might give the response. Otherwise, the calling agent receives the special signal indicating that the value is unknown. Global MAS should take this mechanism into account. Thus, the solution guarantees safety and robustness on communication and migration.

From the point of view of a provider, a simple comparison between two references is enough to recognize an Individual, but some pattern matching between the identity fields can also be done. Identity cannot be changed by application programs at all, but it is public, which means that any program can access the caller's identity fields directly.

The Individual identity uniqueness is extremely important for both philo-

sophical and security reasons. By no means, the language allows assignment to identity fields. Individuals never share identities and this is guaranteed by the system clock along with other fields.

Each interpreter has its unique Id, which is generated when it is delivered, and used by the system to authenticate Individuals *flights*. The same for compilers, thus allowing programmer's identification.

From the designer's point of view, a dynamic search for symbolic names in a program to identify the called object requires that the compiler writes the identifiers of the interface in the agent. Although Individuals get fat with so many names, it is a good idea for mobility and persistence to generate all identifiers, to be used by the virtual machine when saving and restoring contexts. Generating symbolically all identifiers also provides flexibility to the language.

By accessing methods and fields declared in the dynamic interface, providers and Individuals usually establish local communication according to the application. This requires the concept of *sender* and its key identifier. In the present language, **Sender** is used by the called program to refer to the calling one in the same way, and the presence of this key identifier in the calling program refers to the called one, i.e. the rôles are often reversed during communication, thus establishing a synchronous protocol. Therefore, every Individual or provider contains internally a stack of computations. In the present solution, a response may modify values that are used later by other responses.

Thus, the program interface (public objects, methods and fields) describes the objects of the program that can be accessed by another agent, local to the same host. This allows invocation of an external method that does not exist, for example. The same for accessing an external variable. Thus, there might be type mismatch between programs while they are running, and this condition must be dynamically checked and reported to the calling program. However, in the present solution, dynamic type mismatch is not regarded as an error, although the corresponding operation is not performed.

The concept of *home* in Individual identity permits any agent to recognize

the person who is responsible for that Individual by his or her Internet address. This person is also responsible for *all* messages sent by his or her Individuals.

The system can provide security against tempered interpreters and airports, although total security cannot be guaranteed. I adopt the following solution: one who develops the system has his or her own *police*. He or she often sends an Individual to each host that has an interpreter and then performs a privileged operation in the same programming language that permits access to the interpreter byte code, the same for the airport. Such Individuals can run an algorithm that checks whether the interpreter (or the airport) was tempered and then returns to the police host with the result along with some other pieces of information, such as the interpreter identity. The algorithm to check interpreters and airports can vary along the time. The checking result together with date and time is also of general interest because it *certificates* that the former host was not tempered at that time. Because many Individuals might wish to consult the police before critical transactions, the police itself should be organized in hierarchy, that is, the developer's police should be spread out over the network in such a way that every interpreter host belongs to some geographical region which is under its local police. Indeed, the police can provide a sophisticated and efficient distributed mobile system for the whole network.

In order to protect Individuals from malicious hosts, there can be two alternative solutions in this framework. In the first solution, the runtime system keeps a unique file of contexts for all blocked agents. While saving and restoring agents, the MAS always encrypts and decrypts the file by using the system key. The only kind of attack that can be done is to delete the file (therefore all saved agents at once), but the airport records information on both arrivals and departures of every Individual. The airport file is also encrypted and decrypted by using the system key, although the information is available by queries, both locally and remotely. The second solution for the problem of protecting Individuals from malicious hosts consists in assigning each Individual to one running interpreter, both forming one process of the

operating system. Thus, two Individuals run in two processes, and so on. When an Individual sleeps, the whole process sleeps and the security of the Individual is shifted to the operating system level. It is desirable that new versions of operating systems will provide the concept of *public process* and forbid the normal user to kill it. Only privileged users ought to be able to kill such processes. I tend to adopt the latter solution.

In the security sub-model, I suggest that virtual machines and Individuals format should be hidden, at least from normal users. In this way, the system protects Individuals against malicious hosts and malicious implementations. If operating systems provide and manage *public process* and forbid normal users to kill such processes running locally, then Individuals are pretty protected from malicious hosts, although not totally protected from malicious interpreters. The latter protection lies on that interpreters are general programs whose object code can be acquired from a public Network and that their developers are publicly known. Thus, this solution centralizes the responsibility of security towards MAS, while propose diversity. While in different models[102] encryption is seem as almost pointless if it is intended to protect the agent from the interpreter, here there is only one key for the whole system, to be used when it encrypts and decrypts agents, besides application keys. In security, I make use of the concept of centralization which considerably simplifies the solution because all users assume that the mobile-code system itself is reliable. In this way, some applications that require security do not normally require an extra agent to act as a *trusted third party* (TTP).

To protect Individuals while they are flying, airports encrypt them before departure key and decrypt them after arrival.

Privacy is another issue in this setting. Solutions that have been adopted by other programming systems can be used, for example, Java type modifiers are adopted as part of the present solution. In this scenario, privacy is also guaranteed at runtime. Dynamically, each variable (method) is public or otherwise.

For applications among known partners, programmers can write passwords

in the **flyto** statement that is checked at arrival. This solution is similar to a login session on FTP, which can be anonymous or not, depending on the Internet account. Depending on the system implementation, this password may be encrypted and sent in some message before sending the whole Individual, as part of the system protocol.

Another problem that arises from programming for a global computer is *naming*, that is, to use services, the user who writes an Individual must know beforehand names in the interface of the providers that he or she will use in his or her program. If the Individual visits many hosts it must keep different names for the same service. On the other hand, a provider that wants to control accesses of resources by different kinds of Individuals, it has to keep a large database (or knowledge base) of identities and authorizations. This is a concern on programming languages design or AI techniques[104].

I consider *parallel computation* as one of the requirements for a general programming system for global environments and the reason is straightforward. Because communication to some Individual can be asynchronous, a provider may trigger more than one Individual that fly over the Internet to run in parallel. It makes *strong mobility* one of the requirements.

One of the criticisms to the mobile agent paradigm is the fact that agents do not maintain connections upon migration. I do not think that this is a problem: first, connection is a matter of abstraction, that is, a connection that was interrupted at a lower level might be rescued in such a way that, for an upper level of abstraction, the interruption did not even occur. Therefore, an Individual can keep states of a connection at a lower level of programming in order to maintain the corresponding connection at a higher level. Although connections are not implicitly maintained by agents during migration at the language level, conceptually, what is more important is *communication*. I think that mobile agents is the most general paradigm, also when communication is the concern. Some remote communication can be indirect by local communication with providers that might perform the requested communication. This scheme improves security. Because Individuals can move more than

once, in some cases, another alternative for keeping connections is to move the Individual to the host in order to communicate locally where the resource is.

Another characteristic of global computers is the existence of *failures and delays*. Developers want to program for such environment considering that failure or delay may be the result from remote operations. A similar problem that can be solved by this special value is due to special conditions during the communication between agents, for example, type mismatch or absence of symbolic identifier in an external program. The solution for these problems consists of adding a special value for each data type. In this PhD thesis, this value is called *uu*. As we know, this value is a key notion. *uu* in this context means that a value in the problem domain is not available at moment for some specific reason. The reason can be available as a MAS variable. Thus, unlike many distributed systems, I shift failures and delays to the programming level, and hence I provide language constructs for that.

In order to program with delays, *timeouts* are part of all language constructs that perform external operations.

5.4 Communication Between Individuals

As I discuss problems of dealing with resources here, communication may also be viewed as a kind of resource. In fact, communication is one of the keywords with respect to agents at the level of application, such as AI agent systems. Here I introduce a general solution for systems and, therefore, communication at such a level of application is outside the scope of the present thesis. For my proposal, at the language level, to consider communication a resource suffices, while without going into those details.

Mobile agent communication is one of the issues where AI has much to contribute, and also because of this reason, now I leave the problem almost untouched.

However, briefly speaking, communication can be represented by representing *curiosity* properly for AI systems. Curiosity generates communica-

tion. It is a relatively complex task which I dare not comment here in this thesis dissertation. Nonetheless, in the remaining chapters, it will be clear that the intuition reveals that the constant uu is also essential for representing notions from AI, including curiosity. Thus, one can represent more easily *the motivations for learning* in such systems by recognizing that, in some specific situation, the system does not have the piece of information. Then, communication becomes a natural consequence in the presence of uu. Thus, uu works well at a meta-level for communication.

5.5 Conclusion

I define a global computer in terms of its requirements. Taking them, it turns out that no programming language or system to date has been suitable for this kind of distributed computer.

In particular, I think that,

- by centralizing the sub-system for security,
- by making the agent format be private to the developers,
- and by closing the architecture of the global computer,

developers may provide *the only feasible way* to guarantee a satisfactory level of security on an open environment. In particular, it is desirable that the organization who propose their "club" also act as the unique trusted third party.

The present solution is based on a metaphor and describes typical situations with which travelers are used to dealing. Airports, arrivals, departures, passports, security and such natural concepts are in the present solution. Because the metaphor is based on modern life, it is expected that its implementation will be easy to use. Except for communication and language concepts and constructs, almost all problems were addressed here with a solution. In

general, systems suggest or impose programming language concepts and constructs, and the present solution is no exception. However, I believe that I have explained the solution at such a level of detail that implementors can write other systems based on Chiron, in particular, providing at least the same level of security as described. However, programming language concepts and constructs for this context is the issue in part II of this thesis dissertation, and I make use of *uu* throughout it. In particular, chapter 6 is specific for global computers.

Part II - Concepts of Programming Languages

Chapter 6

Programming Language Concepts and Constructs for Global Computers

In part I, I presented one theorem whose slogan is "programs are not functions". I have also informally discussed global computing and defined a framework for mobile-agent systems on global environment. We now work on language features other than functional programming. Thus this chapter is essentially a summary of part II, however, here I am more focused on mobile-code languages. After this chapter, chapter 7 concentrates on **`uu`** *in imperative and functional paradigms.*

This chapter presents programming language constructs that are useful for mobile agents and for the Internet. In particular, I explore the ability to program with the unknown value, **`uu`***, by giving examples of these constructs and of some possible combinations with others. Although the presented characteristics are not specific for mobile-code languages, the underlying global environment can be a unified one.*

6.1 Introduction

Code mobility is a relatively new field of research that has inspired intriguing ideas on programming techniques to improve related software. New programming languages have been presented and discussed in workshops and conferences aiming at providing better standards and good examples for future language designs, commercially or otherwise.

To propose a better model for mobile agent programming for the World Wide Web, for example, the language designer should consider that the underlying connections often fail or delay. A neutral state, which would represent the lack of result, could be assigned to the variable that takes part in the request in such a way that the program carries on running safely. Mobile agents need to be robust and make their own decisions remotely.

Although the present programming language features are not exactly for mobile-code languages, the underlying environment can be the same for the sake of generality. Here I explain how this additional value can be useful in programming for a global environment where the mobile agent paradigm and technology have become increasingly important. In logics, this value has been traditionally referred to as ***uu*** for providing an alternative value to $true$ and $false$. Like many imperative features, the present ones also apply to functional languages.

Broadly, some pieces of work, such as [216, 217], have indicated similarities between technologies of code mobility and persistence, and some persistent languages are being explored at some universities. In the present approach, because I am looking for generality and efficiency, persistence is not provided directly by the language, but instead by programmable constructs. Because of that, this approach here is at least as applicable to mobile agents as persistent languages. As well as persistence, communication is another very active topic of research and much work has been done regarding fault tolerance and communication between mobile agents. However, relatively few satisfactory results have been achieved in terms of language facilities and abstractions.

Here we concentrate on the use of programming features in the context of a global environment and mobile agents programming.

The ability to represent and reason with partial information is well understood in the artificial intelligence and logic communities. However, very little of this work has been related to programming techniques. An exception is Extended Logic Programming, introduced by Gelfond and Lifschitz[133, 134], that can be used for the same purpose as that I am discussing here. Extended Logic Programming makes use of two forms of negation. In [177], the author suggests that an important practical problem in Extended Logic Programming is how the programmer distinguishes whether a negative condition is to be interpreted as explicit negation, or as negation due to the absence of any clause in any closed world as an assumption.

The *unknown* value, ***uu***, extends the semantics of other logics, such as classical and intuitionistic logic, according to the Łukasiewicz [116, 186] 3-valued logic. Here I extend this value for each data type in programming for a global environment or for mobile agents. In arithmetic and relational expressions, ***uu*** as a resulting operand implies the expression to result in ***uu***. Accordingly, statements have to be adapted to make use of this value.

Most Expert System Shells made use of an *unknown* symbol to represent lack of information in boolean variables, in a very restrictive way however[107, 108]. I generalize this concept to programming languages in general, although I apply it here to mobile agents programming.

Agents have to be robust and, because of this, when connections fail or delay, programs should carry on running despite the lack of information. ***uu*** is a constant in programming languages that can be assigned to any variable of any datatype. This new constant guarantees both safety and robustness at the same time, because variables are never committed to any value that is not in the problem domain. A specific discussion on ***uu*** is in chapter 7, for general purpose programming, not necessarily in the context of global computers. An excellent introduction on types for functional programming languages can be found in [267].

In section 6.2 I review some recent programming languages for such an environment, while section 6.3 is dedicated to a programming language that I have called PLAIN. Section 6.4 introduces the concept of *unknown* together with other related concepts, and explains how they can be used to achieve the proposed goals. As a consequence, section 6.5 complements those concepts by stressing the importance of any form of lazy evaluation in programming, as well as timeouts, no matter the adopted paradigm. Section 6.6 briefly discusses logic programming on global computers while section 6.7 also describes strong mobility. Section 6.8 contains other relevant features that are relevant enough to be mentioned. Section 6.9 contains a local conclusion.

The examples in this chapter are written in PLAIN and, because the notation has not been established, in section 6.3, I discuss the syntax of the relevant subset of this language beforehand.

6.2 Some Current Mobile Code Languages

In the past few years researchers have seen the Internet as a popular environment for systems. Some of us would like to program and compute using this structure, i.e. to view parts of an Internet-like network as *global computers*[120]. Many companies, for example, are starting to have their own internal global computers for their specific purposes.

In 1995, Sun Microsystems presented Java[21] programming language with the stress on the interesting idea of permitting code mobility on the World-Wide Web. Portability of code has become critical to software development. Some *online portable* languages (i.e. taking a running program and port it to a different architecture while it is running) have recently been designed[59], whose characteristics will be discussed in this section. Type systems, scoping, name resolution and dynamic linking are some of the key concepts in this context. According to Cardelli, "languages that are not on-line portable will be abandoned because they do not provide what is increasingly perceived as basic functionality: mobility"[59].

However, one of the most interesting ideas is not only to move code, as Tcl[232] and Java[21] do, but also computation (code along with context) over the network, that is, a computation which starts at some location may continue to execute at some other location. Synchronous connections to the original site may be set while a program is running remotely in such a way that any change in some variables transparently causes the value to be stored in the original site. Alternatively, new values of variables can be sent to the original site with no need for synchronous connections. Other paradigms of mobile computation already exist and they depend on the kind of entities that are transfered over the network, with respect to what is moved (code, data, connections, etc).

When the code is moved, what happens if the names it contains are bound to resources in the source virtual machine? This issue defines two classes of strategy, *replication* and *sharing*. The first strategy may be either static or dynamic. Concerning static replication strategy, constants, system variables and libraries, for example, are regarded as *ubiquitous resources*[188] and they can adopt such strategy, where bindings are deleted and set after arrival. As for dynamic replication strategy, the code migrates to another virtual machine along with bound resources and the original bindings are deleted. The original resource in the source virtual machine may be either deleted (*replication by move*) or kept (*replication by copy*). In the *sharing strategy*, the original resource is kept and remotely accessed through network references and connections.

In both strong and weak mobility[75], security[308, 309] is a very important matter. Locations must check for authorization and capabilities in order to prevent malicious software running. By the way, the notion of what is "malicious" itself is a subject in philosophy. However, as long as that is ensured by the system, a global network can be a very interesting and natural platform for computation. Thus, a new challenge emerges: how to provide these facilities and prevent the related problems?

Mobility should also be considered not only during the execution of programs but also during the elaboration of software. The emphasis in perfor-

mance is no longer in the run-time code generated by compilers, but in the (dynamic) compilation process itself (when applicable), transmission and additional overheads to guarantee security and other requirements.

Acharya, Ranganathan and Saltz[4, 5] during the design of Sumatra, an extension of Java, consider three requirements for **individuals**: *awareness*, which is the need to monitor the level and quality of resources in their operating environment; *agility*, which is the ability to react to changes in resource availability, and *authority*, which is the ability to control the way that resources are used. Although they are important concerns, I think that these concerns should be treated at the application level, not at the language level.

Some programming languages for mobile computation are described and analyzed in [75] and other articles, and briefly described here:

- Java[21] is a strongly-typed object-oriented language. Java deals with security[86] and allows transmitting program byte-code to be interpreted by the Java Virtual Machine[199], but does not migrate computation. It supports weak mobility with dynamic linking. Security level is increased by the byte-code verifier at loading-time. Some security problems have been found[85]. It was shown that the ability to break Java type system leads to an attacker being able to run arbitrary machine code[84]. Static and dynamic type checking.

- Telescript[316] is an agent-based and object-oriented language that explicitly deals with locality, strong mobility, and finiteness of resources. There are two kinds of Execution Units: agents and places. Typically, when an agent is running on an interpreter, the instruction *go* causes the agent execution to be suspended, its code and current state are transmitted to a remote virtual machine and, there, the computation is resumed. However, agents do not maintain connections to remote agents. The Telescript run-time code is interpreted without security checking since security is ensured at the language level. The replication strategy is dynamic, by move. Static scoping and name resolution. Static and dy-

namic type checking. In spite of the historical reasons for mentioning Telescript here, that technology was replaced by Odyssey[72], which is a Java-based version of Telescript, briefly speaking.

- Tycoon[208] provides thread migration like Telescript. It is a polymorphic, higher-order functional language with imperative features, which may support other paradigms indirectly, including object orientation. Tycoon provides strong mobility and support for persistent programming. All objects in this language have first-class status. Static and dynamic type checking, dynamic replication with strategy by copy, besides static replication strategy.

- Agent Tcl[148] provides strong mobility where the whole image of the interpreter can be transfered to a different site by executing a *jump* instruction. Agent Tcl also provides weak mobility by executing a *submit* instruction which allows transmission of procedures along with part of their global environment, to a remote interpreter. Typeless language therefore no type checking. Dynamic replication strategy, both by copy and by move. Agent Tcl is a PhD thesis[149].

- Safe-Tcl[43] supports active e-mail, where messages may include code to be executed when an interpreter reads the message after receiving it. However, Safe-Tcl does not support active e-mail code mobility at the language level but, instead, code mobility is achieved through a dynamic code loading mechanism. Typeless language therefore no type checking.

- Obliq[35, 56] is an object-based language that encourages distribution and mobility. While a mobile object is migrating from one place to another, new connections are automatically open between source and destination places in order to guarantee that any change in the variables will update the state in the source place. Therefore, object references are transformed into network references. Although a simple language, there is some loss of efficiency and robustness due to some possibly very

large number of connections in an unreliable environment. Dynamic type checking, sharing strategy.

- Facile[292] is a functional language, a superset of ML with primitives for distribution, concurrency and communication. Mobile code programming was later added to this extension[188]. Static and dynamic type checking. Dynamic replication strategy by copy, besides static replication strategy.

- TACOMA[169, 170], the Tcl language plus primitives to allow a running Tcl script to send another script and initialization data to another host in order to execute the script remotely. Typeless language therefore no type checking. Dynamic replication by copy.

- M0[75, 297] is a stack-based interpreted language which provides weak mobility and run-time type checking. Dynamic type checking, dynamic scoping rules, dynamic replication by copy.

Aglets Workbench, developed by IBM, is a mobile agent system based on Java. Like others, such as ObjectSpace Voyager, the system security and other issues depend on the Java system[181].

As mentioned before, in the present chapter, I discuss some of the features of the PLAIN language.

6.3 Plain

PLAIN[103, 104] is a language that supports mobile agents, syntactically somewhat similar to Java. It supports strong mobility, as well as some forms of knowledge representation, reasoning, and uncertainty treatment. As an ongoing experimental project, the language has not been scaled up and security has not been a concern. Communication between agents has not been implemented either. The PLAIN VIRTUAL MACHINE interprets byte-code and the language provides both replication strategies by programmable handlers[106].

BNF legend: boldface letters are keywords; italic words with initial capital letter are other terminal symbols; words in lower-case letters are non-terminal symbols; meta-symbols: | indicates alternative, ε is the empty symbol of the grammar. Other terminal symbols: { (, ;) } are used in the grammar. Here, I introduce a very simple subset of PLAIN in Backus-Naur Form, with "aprog" as its grammar starting symbol:

aprog $\longmapsto$ classlist commandlist

classlist $\longmapsto$ ϵ | classdef classlist

type $\longmapsto$ **int** | **list**

modifier $\longmapsto$ **private** | **public** | ε

onevardef $\longmapsto$ *Id* | assignment

idlist $\longmapsto$ onevardef | onevardef ',' idlist

vardef $\longmapsto$ modifier type idlist ';'

handler $\longmapsto$ evaluator | reactor

evaluator $\longmapsto$ **when** *Id* ',' **do** command

reactor $\longmapsto$ **when** *Id* ':=' **do** command

classdef $\longmapsto$ **class** *Id* '{' defs '}'

defs $\longmapsto$ ε | vardef defs | function defs | handler defs

command $\longmapsto$ assignment | '{' commandlist '}' | functioncall |
ifcommand | **return** | **return** expression

assignment $\longmapsto$ Id ':=' expression

ifcommand $\longmapsto$ **if** expression **then** command |
if expression ',' command |
if expression ',' command **ifnot** command |
if expression ',' command **else** command |
if expression ',' command **otherwise** command |
if expression ',' command **ifnot** command **otherwise** command

commandlist $\longmapsto$ ϵ | command ';' commandlist

where non-terminal symbols, namely function, functioncall and expression are as usual. In PLAIN, they are somewhat syntactically similar to C++ or Java. The main difference is that the symbol $ can be placed where a variable identifier is expected, as it will be explained below. There are other details that will be explained together with the examples. The appendix B formalizes the semantics that will be explained.

6.4 *uu* in Global Computers

I start this section copying a few words from chapter 7, which is specific for ***uu***. Although repetitions are often not very pleasant, because this chapter is an article[106] and the present dissertation is only a PhD thesis, I shall not remove text. For every data type, the language designer can add a special value, namely ***uu***, to represent lack of some *domain value*, i.e. some known value in the problem domain. For integers, there is $\boldsymbol{uu}^i$; for real numbers,

there is $\boldsymbol{uu}^r$ and so on. I simply write $\boldsymbol{uu}$ to mean that the type is irrelevant in such context. Accordingly, I write *value* in the singular form to mean that its type is not important in the sentence. Grammatically, $\boldsymbol{uu}$ or *unknown* is a constant. Variables either contain $\boldsymbol{uu}$ or some domain value.

Some languages adopt a default value as initial variable contents. But since one now has $\boldsymbol{uu}$, we ought to adopt this value as the initial one for every variable. The programmer should certainly want to initialize some variables with different values.

For any variable in the program, *handlers* can be attached. They can be one *evaluator* and/or one *reactor*, independently. As well as other purposes, one handler can protect a variable. The idea of evaluator is to allow the programmer to write a piece of code to produce and provide some domain value for the corresponding variable, while the idea of reactor is to inspect and protect the variable against assignments. Thus, a reactor allows the programmer to write a piece of code to react instead of letting values be stored unconditionally in the corresponding variable.

int x, y;

when x, **do** { $x := 3 * y$; }

when $x :=$ **do** { $x :=$ \$; }

In the above example, two handlers are defined for the variable x. The first time that the value of x is being requested in an expression, the above evaluator is triggered, which in turn computes the triple of the value of the variable y assigning it to x. From the second time on, the computed value $3 * y$ is already available in x and, because of this, the evaluator is not triggered. This idea is not limited to *exception handling*, a mechanism supported by some other languages, and this will become clearer soon.

An evaluator can contain **return** statement (similar to C) as an alternative to assigning a value to the requested variable. In the case of the **return**

statement and no prior assignment in the evaluator, the evaluator is always triggered when that variable is being used, unless some domain value has been assigned to that variable outside the evaluator.

Whenever a value is to be stored in x, the control is jumped to the corresponding reactor. Notice that the $ symbol above is used in reactors to represent the value that, in other languages, would be stored unconditionally. In the above example, the value is accepted.

Built-in predicates can be provided to check whether a variable contains ***uu***, for example, **known** and **unknown**. In these cases, the value is accessed directly and the *boolean* result from the condition is provided by the interpreter without evaluating the handler of that variable.

The use of variables in expressions can have innovative semantics:

If the variable contains some value in the problem domain, the semantics is exactly the same as in imperative languages. However, if the variable contains ***uu***, the semantics is divided by two separate sub-cases: if there is an evaluator, it is executed. Otherwise, i.e. when there is no evaluator, ***uu*** is used instead. However, the semantics of the execution of evaluators is not similar to the semantics of function calls, because the latter are always executed. In the case of Remote Procedure Call or Remote Method Evaluation, this unconditional call is probably inconvenient or inefficient in programming.

In terms of design, ***uu*** and handlers replace exception handling in other languages. This might relatively simplify the language. Handlers are very useful during program *testing* and *debugging* phases, by inspecting what is being stored and, since mobile agents might escape from the user, ***uu*** together with handlers can be used in mobile agent programming. For example:

```
class mycl {
  public int x;
  private list queue := [ ];
  when x := do {
    x := $;
```

```
        queue := queue +
        [ [ #self + ".x := " + $ + " at " + LocalTime() ] ];
    }
}

mycl c;   c.x := 10;   c.x := 20;   c.x := 30;
```

In the above class or its subclasses, whenever x receives a value, it is also stored in the queue together with the name of the object ($\#self$), the name of the field (x) and the current local time ($LocalTime()$). The '+' operator concatenates lists or strings, besides the arithmetic addition, as usual. The square brackets are used to construct a list of values of any type. Here the programmer chose list of lists for programming reasons.

For it is known that it is very difficult to implement the mobile agents debugging system in a satisfactory way, the above simple code can be written since we have handlers. To generalize, when a mobile agent dies, the local runtime system ought to provide a way of returning the agent to its home. By some local query, the programmer can inspect the contents of such queues, including a general queue for all classes. By writing some declaration (*trace*) in the language, a mobile object support system can internally maintain these queues.

If one thinks of mobile agents that can deal with resources that cannot move, the difference from other paradigms might become decisive in language design. On the one hand, a variable in an evaluating expression may cause its value to be read from a data base or requested from a remote process, provided that its current value is ***uu***. Thus, a variable may have a *cache* because in the subsequent uses, some domain value is available locally and the handler is not triggered. On the other hand, to assign a value to a variable may cause its value to be stored on a data base or sent to a remote host.

The following piece of code exemplifies a persistent field p and a remote field r that can live together in the same class:

```
class remoteandpersistentcl {
  public int p, r;

  public void ini(int i, int j) {
    inttodb("p",i);
    p := i;
    inttourl("www.aaa.bbb.ccc/cgi/server/r.txt",j);
    r := j;
  }

  when p := do {
    p := $;
    inttodb("p",p);
  }

  when p, do {
    return intfromdb("p");
  }

  when r := do {
    r := $;
    inttourl("www.aaa.bbb.ccc/cgi/server/r.txt",r);
  }

  when r, do {
    r := intfromurl("www.aaa.bbb.ccc/cgi/server/r.txt");
  }
}

remoteandpersistentcl c;
```

$c.p := 20$; // also store the value 20 locally on data base.

sendlocally(**home**, $c.p$); // send $\langle c.p \rangle$ to the agent home.

$c.r := 30$; // also update remotely.

sendlocally(**home**, $c.r$); // send $\langle c.r \rangle$ to the agent home

Notice that, according to the evaluator definitions, while the p field is retrieved from a data base whenever its value is requested, the r field is programmed to behave as cache over the global environment. I could write a method reassigning ***uu*** to r to cause the r integer value to be retrieved from network at the next time that it is requested in some evaluating expression. The functions *inttourl*, $intfromurl$ and *sendlocally* tell the underlying system to generate internally mobile agents to take part in the protocol. There has been a general criticism concerning mobile agents because they do not maintain connections. I agree that a programming language should hide connections from the agent, but the mobile agent support system should provide remote communication in an appropriate way. This produces positive effects and abstractions in the programming language.

As stated, a much more convincing discussion on ***uu*** in the context of general-purpose programming languages is in chapter 7. Here I concentrate on features for global computing and present some new aspects of ***uu***.

6.5 Lazy Evaluation and Timeout

Lazy evaluation is one of the most interesting characteristics of programming, in particular in applications where time is regarded as important. In this chapter, I am not regarding lazy evaluation as being only *call by need* of functional languages. If the language provides functions, lazy evaluation can also be very useful in the same platform, from the same point of view of the present section.

Programming for mobile agents on a global environment tends to be more personal. One of the reasons is that patience and mood vary for different

people as well as for the same person at different instants, and one of the purposes of agents is to represent users.

As an example of a situation, a mobile agent *ma* can communicate with a stationary agent *s* which in its turn can send a small agent remotely to *ma*'s home in order to return some piece of information to *s* which in turn can hand it to the mobile agent *ma*. To deal with faults and delays in communication, a timeout can be set, implicitly or explicitly, for every input operation. After that time, the result is *unknown* (***uu***) and the computation continues normally. Similarly, every output operation has a timeout.

In this way, the same statement can be executed at different locations[57], either sequentially or not. This situation happens often. Cache-like variables might produce a similar result as lazy evaluation. Computing with timeouts together with ***uu*** and handlers is not lazy evaluation, but it can give a somewhat similar impression of *impatience* and, because the resulting value in this case of exception is ***uu***, variable values in programs are always sound and finally this scheme improves *agent robustness*.

Another way of dealing with faults and delays is to provide a standard semantics for basic operations such as arithmetic and relational. In particular, if the first operand is ***uu***, the expression might result in ***uu*** without the evaluation of the second operand. This is a form of lazy evaluation. Finally, ***uu*** in "call by need" is presented in section 7.7.

6.6 Logic Programming

In comparison to imperative programming languages, logic programming is easy because the former requires a more difficult form of reasoning, for instance, sequences of statements. Here a program is ideally a set of facts and rules, and these are notions with which all of us are used to dealing in our daily lives. In agent-based languages and systems, the programmer typically needs to state permissions of access for users to resources and this can be done in logic programming quickly, perhaps automatically. On the other hand,

logic-based languages that are boolean, such as the famous Prolog, are not very compatible with global computing because there are delays and failures in connections in the real world. Perhaps because of this, logic programming has not been interesting for code mobility or global computers. Prolog negation as failure is dependent on the closed-world assumption while, in contrast, global computers contain those mentioned characteristics. This presents an important problem if one wants to use Prolog in applications other than with non-monotonic reasoning, similar to locally answering whether there exists a flight leaving London for New York on Wednesday afternoon, as illustrated in [138].

6.7 Strong Mobility

Global computers almost imply code mobility, whose most general form is *strong mobility*, which in turn can be implemented by mobile agents. To date, for all mobile agent languages and systems, an instruction that causes mobility is required, although strong mobility can be easily conceived declaratively, instead of in the form of an imperative statement. In Telescript, for instance, this statement is called **go** while in Agent Tcl the equivalent statement is called **jump**. In PLAIN, as described in chapter 5, the instruction is the **flyto** statement. There might be small variations in this statement, in comparison to the semantics of the μ operation presented in chapter 4. So, given the current place *cur* and initial time t_0, given the current and destination ambient states A^{cur} and A^{dst}, also indexed with time, and given the current state $@cur \cdot t_0[r]$, an operational semantics for the **flyto** *dst* **timeout** q; statement, with timeout q, in the language and semantics presented in chapter 2, can be the following:

$$\frac{@cur \cdot t_0[r \subseteq A^{cur}_{t_0}] \quad @dst \cdot t[r \subseteq A^{dst}_{t}] \quad @cur \cdot t[t -_t t_0 <_t q]}{@cur \cdot t_0[\![\textbf{flyto } dst \textbf{ timeout } q, r \cap A^{cur}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @dst \cdot t[r \cap A^{dst}_{t}]} \tag{6.1}$$

where $t_0 <_t t$. The semantics of the timeout in this operation can be the following:

$$\frac{@cur \cdot t_0[r \subseteq A^{cur}_{t_0}] \quad @cur \cdot t[t -_t t_0 =_t q \wedge r \subseteq A^{cur}_t]}{@cur \cdot t_0[\![\textbf{flyto}\ dst\ \textbf{timeout}\ q, r \cap A^{cur}_{t_0}]\!] \overset{\text{exec}}{\rightsquigarrow} @cur \cdot t[r \cap A^{cur}_t]} \tag{6.2}$$

The statement for strong mobility makes the agent execution freeze and the thread continues at the destination address, which is its operand. There may be some password and other details, depending on the technology.

6.8 Other Features

Because generality is desirable, choices among various strategies for binding resources should normally be programmed. Handlers may be used to implement different strategies for variables that are resources, either local or remote.

During the compilation, in order to support higher-level communication between agents, names of objects in the source program can be written in the object code, which increases the agent size but it is still a good idea. A possibility is to generate only names defined in the dynamic part of the interface. If the language supports artificial intelligence techniques, perhaps it is even interesting to consider the idea of generating all names. Communication between agents can be set from a prefix in function calls containing the name of the destination agent. For example, in $x := prov : func(params)$, the string variable *prov* is a name that indicates the agent which in turn might contain the $func$ function definition. The *prov* value is an absolute (global) or relative (to the local host) address. If such a matching name of $func(params)$ is undefined in that agent when the call is executed, x receives ***uu***. In every function call (or method invocation) between two agents, a timeout can be attached. For example, in $x := prov : func(params)$ **timeout** 3, if the operation is not completed before 3 seconds, at that time it is interrupted and x receives ***uu***.

The concepts of **home** and **Id** of agents ought to be key words in the programming language, in a similar way as exemplified above, outside the class $remoteandpersistentcl$.

Surprisingly, although *concurrent programming*[52] is an important technique that can help in certain applications, it is not a specific feature for mobile agent programming languages, as concurrency can be achieved at the operating system level.

However, ***uu*** permits a large number of parallel operations, not only parallel *and* and parallel *or*. For example, to evaluate op_1 $(+)$ op_2, the operands are evaluated first, possibly in parallel, and if both result in a known value the sum is finally performed, otherwise the result is ***uu***. See semantic rules for $(+)$ in the appendix B. There are no side effects in the operation, as the language can guarantee syntactically the presence of only pure function applications and pure expression evaluations. Although I do not list all parallel operations, which lexically I would similarly surround all those sequential operators with parentheses, very similar semantic rules would apply for the other operators. In other words, in the appendix B, I only present rules for $(+)$ and not for $(-)$, $(*)$, $(/)$ etc. The systematic use of these parallel operators radically improves programming, as well as improving efficiency of agents running globally.

As mentioned, the examples in the present chapter is written in PLAIN, a general purpose language that supports strong mobility. A previous language, Lidia, was initially conceived to integrate forms of knowledge representation and programming paradigms. Although I have not implemented, during 1997, I also defined a three-valued logic programming sublanguage, GLOBALLOG, that might be suitable for global or wide-area applications. I intend to include the implementation of GLOBALLOG in the current PLAIN implementation. The whole PLAIN language has only one negation. Although its security model and the logic paradigm have not been implemented, some applications in the main language have been written. Concerning its virtual machine, to guarantee agents privacy, PLAIN virtual machine decrypts byte code in some specific format, obtains the agent and starts interpreting the agent.

6.9 Conclusion

At a more refined level, for avoiding repetition in computation when the variable value is requested more than once, there can be two kinds of unknown states: the first one represents the initial lack of information with potential for substitution. The second also represents the same lack and is internally assigned to variables after having attempted to provide some value. From this chapter, it is easy to deduce both forms of ***uu***.

Local inefficiency is an issue of the features discussed in this chapter. Assuming that, in practice, mobile agent support systems entail code interpretation, the interpreter has to check the presence of ***uu*** whenever a variable is being requested in an evaluating expression. However, as hardware is getting faster and larger, this is not considered a significant problem. Moreover, this problem can be compensated for the fact that mobility and remote accesses are the bottleneck in applications, and that variables can behave as cache and operations can be lazy. This combination is encouraged by the language.

Chapter 7

uu for Programming Languages

The Internet has motivated new programming languages features. I consider that programming for such a global environment requires the ability to deal with what is unknown because connections often fail or delay and programs should be robust.

In this chapter I present **uu***, a value that can represent lack of information in programming languages. A thesis is that this value is a good feature towards the unification of some programming paradigms. In this chapter I explore constructs as consequences of* **uu** *in such languages together with examples where this value helps in programming, and I explain the relationship between* **uu** *and another programming paradigm. The* **uu** *implementation is also briefly discussed.*

7.1 Motivation

High school students usually solve systems of algebraic equations. They try to find values, either numeric or symbolic, to be assigned to variables. Once they find a value for a variable, this value remains constant. Variables in such a system have two general states: unknown and (subsequently) known. These states inspire a new programming paradigm, where variables in a program initially contain the *unknown* state, and later, might change their state to

known as it receives a value in the problem domain. In imperative languages, if a variable is in the former state, it contains a special value that is called ***uu***. Otherwise, it contains a value in the problem domain.

As another example, a program presents a question to the user, but the user does not know or even does not want to answer the question. What state should we assign to the variable after the input operation?

One more example, in programming for a world-wide network such as the Internet, the languages designer should take into account that underlying connections often fail or delay. In such cases, a neutral state could be assigned to the variable that takes part in the request, in such a way that the program carries on running safely, that is, the variable could contain a value that would represent the lack of that piece of information, instead of being committed to some known value in the problem domain.

A question arises from the above situations: what are the programming languages concepts and constructs to provide a better abstraction for variables regardless of the programming paradigm? In this chapter, I partially provide an answer to this question. Here, I also refer to a non-***uu*** value as *known value*.

Section 7.2 addresses somewhat related work. Section 7.3 introduces the concept of the *unknown* value together with other related concepts, and section 7.4 explains how ***uu*** can replace specific constructs for exception handling. Section 7.5 explains the applicability of ***uu*** in OOP while section 7.6 contains a discussion on the use of the *unknown* value in the integration with other paradigms, in particular logic programming. Section 7.7 explains how ***uu*** can support lazy evaluation. Section 7.8 contains a comment on implementation, section 7.9 presents a synthesis and, finally, the appendix B presents an operational semantics.

7.2 Related Work

The ability to represent and reason under lack of information is well known in the artificial intelligence and logics communities, and there has been much work on this subject. However, from the best of my knowledge, none of them is related to programming techniques: Extended Logic Programming, introduced by Gelfond and Lifschitz[134] is one exception, where the core idea is the negation itself, and *abstract negation*, to provide only one negation in the whole unifying language. Our approach here consists in using the ***uu*** value when a specific piece of information is unknown to the program. In this way a set of variables with their corresponding values can represent lack of information. Another somewhat related work is *boxed* representation[173], implemented for functional languages such as Haskell. Here, my main concern is in pragmatics of a programming language, although such technique can implement ***uu***. The present ideas do not apply to pure functional languages, in which variables do not contain values but instead denote them.

In the late 80's, most commercial Expert System Shells also made use of similar unknown value to represent lack of information over Boolean variables, although in a very restrictive way. I generalize the concept to the programming language level regardless of the application area.

Current programming languages, such as Java[147] and ML, provide the concept of *exception handling* and constructs for it. I present more general constructs that replace this concept at the language level.

The ANSI/IEEE Standard 754-1985 also supports the concept of "Not-a-Number" or NaN[167] instead of floating-point values to represent indeterminate quantities. Although it also propagates NaN, the approaches are not the same since that standard does not concern programming languages constructs. Moreover, our approach is not limited to some specific data type.

7.3 uu: the Unknown Value

In this section, I introduce uu for programming languages design. For every data type, the languages designer can add a special value, namely uu, to represent the lack of information. For integers, we have uu_i; for real numbers, we have uu_r and so on. In this paper however, I simply write uu to mean that the type is irrelevant in the context. I apologize for such slight abuse of notation. Accordingly, I use the term *value* to refer to a value regardless of its type, and a *known* value to mean a value in the problem domain.

The *unknown* value, uu, extends the semantics of the classical and intuitionistic connectives according to the Łukasiewicz[186] 3-valued logic. In arithmetic and relational expressions, the presence of uu as an operand implies that the expression results in uu without evaluating the other operand.

As a first example, I present a program to find the roots of an equation of the second degree:

float a, b, c;
float $delta := b * b - 4 * a * c$;
float $x1 := (-b - sqrt(delta))/(2 * a)$;
float $x2 := (-b + sqrt(delta))/(2 * a)$;
write $x1$, " ", $x2$;

I believe that the above syntax is intuitive or familiar for programmers. In comparison to other programming languages, it is not syntactically much different and here, a bit like Prolog, the corresponding system tries to find values for variables when they are being requested in an evaluating expression. Initially, all variables contain or denote uu. The programmer, then, uses the variables, $x1$ and $x2$, in the expressions of the **write** command and then their known values are found according to the formulas in the program. Although *delta* is used by both $x1$ and $x2$, it is desirable that its value be calculated only once. Thus, the program execution asks for the values of b, a, c, in this order, before giving the answer.

In terms of Domain Theory, ***uu*** is a result and there is no order relation between ***uu*** and other values in the domain. Thus, ***uu*** is a value in the object language, and not the bottom ($\perp$). However, although ***uu*** is not "undefined", it can be used where a function is undefined to transform a partial function into a total one. Thus, the factorial function can be written as follows:

int $fact$(**int** x) :=
  if $x == 0$, 1
  ifnot
    (**if** $x > 0$, $x * fact(x - 1)$
    ifnot ***uu***)
  otherwise ***uu***;

In the above example, if the value of the parameter x is ***uu*** or negative, the result is ***uu***. Notice that the conditional was extended to accommodate a 3-valued logic. I address conditionals as well as lazy evaluation later in this chapter.

Variables either contain ***uu*** or a known value. Most imperative programming languages adopt a default value as initial variable contents. Here, since I am introducing ***uu*** in this context, I adopt this value as initial for each variable according to its type. The programmer might want to initialize some of the variables according to the application.

7.3.1 Evaluators and Reactors

In imperative programming, the present idea is to allow the programmer to write a piece of code to "discover" the value in the problem domain whenever a variable that contains ***uu*** is being used in some evaluating expression. I call this piece of code an *evaluator*. Additionally, the programmer can write a piece of code, called *reactor*, to be triggered instead of letting values be stored in the variables. For every variable in a program, one can attach *handlers*. They can be one *evaluator* and/or one *reactor*, independently. Among other

purposes, such handlers protect the variable. One can write handlers in the following way, intuitive for Pascal or C++ programmers:

```
int x, y;

when x do { // this is an evaluator
   x := 2 * y;
}

when x := do { // and this is a reactor
   x := $;
}
```

In the above example, two handlers were defined for x. At the first time that the value of the variable x is being requested in an expression, the above evaluator is triggered, which in turn computes the double of the value of the variable y assigning it to x. From the second time on, that computed value $2 * y$ is already available and the evaluator is not triggered.

The concept of evaluator can contain the **return** statement (similar to C) instead of assigning a value to the requested variable. In the case of the **return** statement, the evaluator is always triggered when that variable is used, unless a known value has been assigned to that variable outside the evaluator. Such an assignment can be statically allowed and dynamically allowed or forbidden, or simply statically forbidden.

On the other hand, whenever a value is to be stored in x, the control is jumped to the corresponding reactor. In the above example, the value is accepted: notice that the $ symbol is used in reactors to represent the value that, in other languages, would be stored unconditionally.

Two predicates are used to check whether a variable contains ***uu***, namely, **known** and **unknown**. In these cases, the value is accessed directly and the *Boolean* result from the condition is provided by the interpreter without evaluating the ***uu***-valued variable handler.

7.3.2 Comparing Handers with Methods and Functions

A pragmatic comparison between the use of handlers and the use of functions and methods can be done: after implementing an application system, handlers can always be added and updated without changing the system elsewhere. However, unlike functions and methods that are called elsewhere, handler definitions can always be removed as the system provides a default semantics for the absence of a handler.

The semantics of using a variable containing a known value is exactly the same as for other languages. However, the semantics of using a variable containing ***uu*** is not: if there is an evaluator, it is executed. Otherwise, i.e. when there is no evaluator, ***uu*** is used. On the other hand, the semantics of the use of variables is not the same as the semantics of function calls either, because the latter are always executed. Therefore, in some sense, ***uu*** combined with handlers are semantically somewhere between use of variable and function.

Before using ***uu*** as an operand in the expression evaluation, PLAIN[103], a proposed language for the Internet, optionally tries to get the value from other sources, e.g. asking the end-user, in such a way that in the first run programmers can be reminded that they forgot to initialize some variable.

Besides using some languages constructs to hide variables ($x2$, $y2$, uux, and uuy, below), the equivalent semantics is then achieved without a built-in ***uu*** in the following way:

int $x2,\ y2$;
logic $uux := true,\ uuy := true$;

int x(**void**) {
  if uux, { // the evaluator for x
    $x2 := 2 * y()$; // instead of $x\_eq(2 * y())$;
    $uux := false$;
  }
  ifnot return $x2$;

```
}

int x_eq(int dollar) { // the reactor for x
  x2 := dollar;
  uux := false;
}
```

Notice that the *Boolean* type, built-in in some languages, has been replaced by (3-valued) **logic**. Also, the parentheses that surround conditions in the **if** statement (and **while** statement in other contexts), which might contain commas in C, C++ and Java, have been replaced by the comma to mean **then**. Here, **ifnot** is being presented instead of the reserved word *else*, adopted in other imperative and functional languages.

In an object-oriented context, although it is possible to write a class, say $C1$, with the above code, and then to define many instances of that class and its subclasses, the use of classes does not encourage the use of these ideas as much as a programming language does. Moreover, the syntax for assignment and for handlers are more suggestive of the programmers intention, and the occurrence of variables in expressions should not need to have the same syntax as function calls. More importantly, handlers contain different code, and this diversity makes classes and subclasses definitions much more complicated and less readable. Another important difference: in object-oriented languages, programmers should take care when they are defining a public field because the field can be used outside the class and programmers can no longer change the field definition, e.g. to be private to that class, without considering elsewhere. Here, although it is not possible to change the field definition either, it is still possible to insert code related to that variable in a handler without changing the rest of the system. The use of public fields in the current scheme is surprisingly harmless. If we think in terms of classes as being downloaded and linked dynamically on a public network and of mobile agents that deal with resources that cannot move, this difference itself might become decisive in the

language design. Using a variable in an evaluating expression might cause its value to be read from disk or requested from a remote process, provided that its current value is *uu*. Thus, a variable may be a kind of cache, because in the subsequent uses of that variable its value is already locally available and the handler is not triggered. Conversely, assigning a value to a variable might cause its value to be stored on disk or sent to a remote host. Storing values of variables on disk and restoring the values by using the same variables implements persistent programming.

7.4 *uu* in Exception Handling

uu, as being a more primitive and general concept, when combined with handlers, replaces exception handling, which simplifies the language. The prefix operator **code** gets the exception code for a variable in a context where the reason for its value be unknown is required. Suppose that I want to access a value for a variable at some place on the WWW, given the address www.somewhere.on.the.earth.

```
string addr = "www.somewhere.on.the.earth";
int x = getint(addr) timeout 10;
if unknown x then {
  int c = code(x);
  if c == exc_to then {
    write "Time-out in the attempt to access ",addr,nl;
  }
  ifnot {
    write "Exception ",c, "in the access of x at",addr,nl;
  }
}
```

Thus, if the requested integer value is not retrieved by 10 seconds, the program execution continues normally. In the above example, the case is

treated as an exception, but the lack of information in the problem domain that was expected to be in x allows the computation to continue normally, with or without the above treatment, and without x being committed to any value in the problem domain, which guarantees safety and robustness.

7.5 Object-Oriented Programming with *uu*

The ***uu*** value can be added to any programming language no matter its paradigm. In this section I give some examples in object oriented programming. Although a hybrid paradigm language can adopt only one construct for both frames and classes, I discuss them separately here.

7.5.1 *uu* and Frames

A programming language can adopt the following semantics for the use of a variable in an evaluating expression:

1. If the value is known, use this value.
2. otherwise, if the evaluator for that variable is defined, execute it. Then
   (a) if its value is now known, or the evaluator **return**s a value, use this value;
   (b) otherwise, use ***uu***.

   Success.
3. otherwise, if the variable is a field, look for the value of the corresponding field in the object class (and recursively, in its superclass), using the steps 1-3 of this algorithm, and then,
   (a) if a value in the problem domain is found, use this value in the expression instead of assigning to the requested variable;
   (b) otherwise, use ***uu***;

Success.

4. At this point, the variable is not a field, contains ***uu*** and there is no handler for the variable. If "possible", ask the application user for the value of that variable, and then,

   (a) If the value is entered, the value is assigned to the variable and the evaluation of the expression continues.

   (b) On the other hand, if the user does not want to answer, ***uu*** is used in the expression and its calculation carries on. The user will no longer be asked for a value of the same variable, unless ***uu*** is assigned to that variable.

5. If "not possible", e.g. the application is not interactive, use ***uu***.

Note. Steps 4-5, if implemented, require that the compiler generates the symbolic names of the variables. PLAIN has done so and recently, in spite of the size of the byte-code, it was realized that these names are also useful for symbolic communication between agents on an open system. Thus, because PLAIN was designed for knowledge representation, it has been relatively easy to adapt it to support mobile agents.

Now, looking back to the first example, of the equation of second degree, its coefficients a, b and c, do not have evaluators and hence their values can be asked at the terminal at the first time that they are requested. Assigning the result from an expression to a variable in its definition is a syntax simplification of writing an evaluator for that variable containing only that expression. Thus, initializations might be dynamic and lazy in a sense.

Handlers are very useful for *testing* and *debugging*, by inspecting what is being used, and this can also be done in mobile agents. For these programs, there can be handlers for other purposes that are outside the scope of this discussion, e.g. to be implicitly executed before departures and after arrivals.

7.5.2 Classes with *uu*

Using the Internet as an example, if a programmer wants to build a class to represent some specific measure, they should take into account the fact that some countries use certain measuring systems while others use different systems. Consider the temperature representation, either in Celsius or Fahrenheit:

```
class temperature {
  public float Fahr, Celsius;
  when Fahr do {
    Fahr := 9.0/5.0 * Celsius + 32;
  }
  when Fahr := do {
    Fahr := $; // it accepts the assignment
    Celsius := 5 * (Fahr - 32)/9;
  }
  when Celsius do {
    Celsius := 5 * (Fahr - 32)/9;
  }
  when Celsius := do {
    Celsius := $; // accepts the assignment
    Fahr := 9.0/5.0 * Celsius + 32;
  }
}
```

The above example is a form of constraint programming. It is represented here, almost declaratively, the relationship between two variables concerning measurement of the same concept, temperature, and the handlers keep the variables always consistent. Regardless of the syntax, this is somewhat similar to method invocation. However, methods are always executed.

As another example, I consider that classes and frames are concepts that can be integrated. But while classes come from the set theory, frames come

from the prototype theory. While a class is a shape and is conceived for reusability and other programming concerns, a frame represents concepts of the real world. Because both class frames and instance frames are defined, I can integrate them easily:

class *Human* {
  public logic *dies* := *true*;
  public int class *cardinality* := 6000000000; // not exact
  public string *handed* := "*Right*";
}
Human Socrates; // Therefore, Socrates.dies

However, in a frame system, classes are also treated as instances, e.g. it makes sense to compute *Human.cardinality++*; when someone is being born. The reserved word **class** is being used as a modifier in *cardinality* to remove the attribute from the instances of that class, although its subclasses can inherit the attribute and even change its value to represent exceptions. In the example, *Socrates.cardinality* might be meaningless. According to the step 3 of the algorithm presented in the previous section, the ***uu*** contribution here is that the field values of the instance *Socrates* is obtained from the class *Human* (or possible superclasses) dynamically when they currently contain ***uu***, i.e. if a definition is changed dynamically inside a frame, its instances will inherit the new value on demand.

According to the algorithm, ***uu*** does not necessarily depend on handlers, and it can be used as any other value, e.g. it can be assigned to a variable. *Handlers* are somewhat similar to "ties" in Perl, "triggers" in SQL and "tag methods" in Lua[164], which are languages that do not provide ***uu***.

Handlers and ***uu*** can be used to implement multiple inheritance in an object-oriented language that provides single inheritance and late binding: the programmer defines secondary conceptual super-classes as fields in the defining class as in the example below:

class *c*1 {

```
    public int i;
    public float f;

    public void m() { }

    when f do { f := 3.14; }
    when i do { }
}

class c2 {
    public int f;
    public float i;

    public void m() { write "Hello", nl; }

    when i do { }
    when f do { f := 15; }
}

class c3: c1 {
    private c2 sec; // c2 is a secondary super-class

    // here, the conflicts are solved by interception:
    public void m() { sec.m(); } // ... from c2
    when i do return sec.f; // ... from c2
    when i := do { sec.f := $; } // ... to c2
}
```

Thus, the language shifts responsibility to the programmer to define the interpretation of the conflicts among attributes from different conceptual su-

perclasses. The exceptions are treated by overriding methods and handlers, which can be used to rename conflicting attributes when they are needed in the defining class. In $c3$ in the above example, by default, the attributes are inherited from $c1$ and handlers and methods are written to implement inheritance from $c2$. Because multiple inheritance is not very often needed, single inheritance in this context seems to be an interesting solution, in particular, together with late binding.

Classes do not necessarily need the **new** operator to create objects as they might be created on demand: when a variable of any class is used in an evaluating expression and it contains ***uu***, the object referred to is created.

The program below exemplifies the use of ***uu*** when the programmer wants some default value to be assumed. For example, we normally assume that English ought to be generally used on the Internet when we want to communicate with the public. Spanish ought to be used in Latin-America mailing lists, and so on. With ***uu***, value inheritance can be a dynamic relationship:

```
class General {
  public string you := "world";
  public string hello := "Hello,";
  public string sayhello;
  when sayhello do
    return hello + " " + you+ "!";
}

class MailingList: General {
  initial {
    hello := "Ciao, ";
    you := "Italia";
  }
}
```

//...

MailingList b;

b.you := "caros brasileiros";

write *b.sayhello*, *nl*;

b.you := *uu*;

write *b.sayhello*, *nl*;

MailingList.hello := *uu*;

write *b.sayhello*, *nl*;

In the above example, after the definition of b, the value of the field *b.you* is customized as "caros brasileiros". In the following line, the field *b.sayhello* is requested and evaluated as "Ciao, caros brasileiros!", and this content is written followed by the newline: the constant **nl**. Then, ***uu*** is stored in *b.you*. When *b.sayhello* is evaluated in the subsequent line, it is evaluated as "Ciao, Italia!", that is, because *b.you* now contains ***uu***, its known value is picked up from its class (and so on, upwards, if it is also ***uu*** there). Then, this value is written followed by the newline. The next line assigns ***uu*** to the field *hello* of the class *MailingList*. Finally, the string "Hello, Italia" is written followed by the newline command. In this way, I represent default values. In this case, *sayhallo* is always evaluated.

7.6 Imperative and Logic-Based Features

uu can help combine imperative constructs in a hybrid language with other programming paradigms, such as Logic Programming. A Logic Programming system, besides answering a query with either *true* or *false*, can provide values for free variables. Some of these variables remain free after the computation from a query, which means that they represent "any answer". Thus, besides the ability to answer a query with ***uu*** to mean "unknown", ***uu*** can be used to implement free variables.

On the other hand, if a variable of the imperative paradigm is passed to a query of the logic paradigm and its value is ***uu***, the algorithm of the called paradigm will understand that the query includes a request for the value of that variable. After receiving the answer, the calling program interprets variables containing ***uu*** as an appropriate answer, and continues normally. If the answer list is empty, that means "no answer", while ***uu*** is normally interpreted as "unknown answer". As above, depending on the contract, ***uu*** can be interpreted as "any answer".

7.7 *uu* in Lazy Evaluation (Call by Need)

As stated in the section 6.8 with reference to appendix B, ***uu*** can support lazy evaluation in almost all sequential operations in a language, that is, when the first operand results in ***uu***, the second operand is not evaluated. Chapter 6 describes in detail the application of ***uu*** and lazy evaluation in programming for internets and mobile agents. Thus, in the rest of this section, I mention only call by need.

Some functional languages are eager (e.g. ML) and some are lazy (e.g. Haskell), but I can think of lazy and eager evaluations as concepts related to functions in the following way:

logic lazy f(**int** x, **int** y) :=
  if $x + x > y$, $true$
  ifnot $false$;

//...
logic $y := f(1 + 2 + 3, 2 + 2)$;

Here, although the parameter x is used more than once in the function *f*, x is evaluated only in its first occurrence in the expression. The **lazy** modifier postpones the evaluations of the parameters, e.g. 1+2+3 and 2+2. Regarding the implementation of this mechanism, it is not difficult with ***uu***: The compiler

generates code to skip the actual parameter list. It also creates a pointer to every expression in the actual parameter list, $1 + 2 + 3$ and $2 + 2$ in this case, including references to the activation record[7], transforming every actual parameter into a local subroutine, and then passes the expression pointer to the corresponding formal parameter handler, x or y. As already explained, for the first time that a ***uu***-ed parameter is being evaluated, the corresponding actual parameter, either $1+2+3$ or $2+2$, is evaluated. From the second time on, the value is known and, because of this, it is not evaluated. However, if the formal parameter is not used, the corresponding actual parameter is not even evaluated.

It seems to be bizarre to provide lazy evaluation and imperative features because of side-effects. However, a hybrid paradigm programming language can provide the concept of "pure function" as its compiler forbids assignments and global objects inside its code.

7.8 Implementation

Although efficiency is not the main issue in the present paper, inefficiency might be the only negative point of the present ideas, in comparison with imperative languages that do not provide object-oriented constructs, because the interpreter has to check the presence of ***uu*** whenever a variable is being used in an evaluating expression. References to handlers for a variable also increase the size of the object code. This detail is not really a languages feature, but instead it depends on the decision of implementation which might require some analysis on the source code in terms of frequency of use of handlers, which in turn depends on use and experiments. Therefore, the implementation of ***uu*** is a challenge. The comparison is relative to application because method invocation, for example, requires pattern matching and search algorithm. Moreover, like a mobile-code language, a ***uu***-based language entails some form of code interpretation, which is becoming a normal conduct in programming languages as hardware is getting larger and faster.

7.9 Conclusion

As a consequence of the adoption of ***uu***, expressions in a programming language have to be able to consider this special value. Assuming that *not uu* results in ***uu***, statements such as **if-then-else** and **while**, as well as their semantics are adapted to deal with three logical values. The conditional statement or expression in their full forms, for example, becomes **if-then-ifnot-otherwise**.

uu and handlers can simplify a programming language by replacing common constructs such as multiple inheritance.

Handlers and ***uu*** have been experimented within PLAIN for a number of years successfully. Their application, along with other features, to programming for mobile agents and the Internet has been investigated.

The idea of a hybrid paradigm for programming allows programmers feel free to choose their own way of working. Some definitions are better written in some particular paradigm while others are better written in other paradigms.

At a more refined level, there can be two kinds of unknown states: the first represents the initial lack of information with potential for later discovery. The second kind also represents the lack of information after having attempted to discover its value. PLAIN distinguishes one kind from the other, to allow the inference machine to recognize variables whose value was already asked.

Chapter 8

uu and Uncertainty for Global Computing

Humans often make decisions based on subjective factors, such as opinions, feeling, measure of confidence and so forth. On the other hand, decisions are basic characteristics of any programming language. This chapter applies the connection between **uu** *and uncertainty to programming.*

In chapter 2, I introduced a logic and calculus/system that provide both uncertainty and **uu***. Here, I use these two notions at a more practical level. The present chapter, not only the others, helps form the first study on* **uu** *in programming.*

Uncertainty is a research topic which is well known to the artificial intelligence community, and some commercial expert systems environments provide models for uncertainty[109], such as Bayesian networks, Dempster-Shafer and production systems with confidence factors. This chapter adapts an uncertainty model to programming for internets. One of the key motivations for the present model of uncertainty is to reduce the number of remote accesses making programs more efficient and robust in this environment. Perhaps more importantly, I argue that agents naturally require a different model of programming

and, therefore, different constructs at programming language level. One of the reasons for this is that, because agents are also conceived to autonomously represent individuals, programming agents is a task that is becoming increasingly based on subjective factors, such as personal taste.

8.1 Introduction

Uncertainty and fuzziness[278] are research topics which are well known to the artificial intelligence community, and some commercial expert systems environments even provide some model for uncertainty, most commonly, production systems with confidence factors. In the last two decades, most expert systems have been based on some uncertainty model, in particular, a few variations of the MYCIN model of uncertainty[272]. However, to date computation under uncertainty and under "the absence of pieces of information" have been little investigated in the programming languages community. In [153], the author classifies deductive logics as analytical. In this sense I agree with his philosophical viewpoint. However, in a sense, the synthetic and inductive form of reasoning is important in logics.

A thesis of mine is that, once there is a established paradigm for programming agents, uncertainty[191] is probably going to play an important rôle in programming. Models that require many probabilities are mathematically accurate and appropriate in many cases. I refer to such inferences as uncertainty-based inferences.

I observe that logical deduction and uncertainty-based reasoning form a pair of opposite but also complementary notions. The former represents some analytical reasoning while the latter represents some synthetic reasoning. On the one hand, humans should be able to deduce and make decisions, for instance, after perceiving new facts. On the other hand, although humans often make use of deduction while thinking, there are many situations where they use a more subjective way to deal with hypotheses and to act according to feeling and beliefs. For agents, *both* deduction and uncertainty are necessary and, therefore, a programming language for mobile agents has to provide constructs based on both forms of reasoning. Between this pair of notions, I can introduce another component, $\boldsymbol{uu}$, a constant which represents lack of information.

I believe that (mobile) agents and the Internet itself can provide a new support environment where agent-based systems and techniques of artificial

intelligence can succeed. In particular, while one of the original aims in artificial intelligence was to program and represent expert knowledge in systems, now, with agents, we humans not only expect intelligent machine behaviors, but also to simulate natural intelligence, for agents also represent their corresponding individuals in a complex society. In chapter 6, I observed that because people's patience varies, programming agents for global environment tends to be more personal. In general, regardless of the kinds of application, agents should contain personal knowledge about their corresponding users, such as their preferences, dislikes and so forth. Here, I can see the rôle of uncertainty-based programming in the technological scenario, and this can partially be due to subjectivity, but partially due to other factors, such as the nature of the subject being represented.

Individuals differ. As an example, some people might take a decision in a given situation when they are 60% sure that they will succeed, while others will only take the risk in the same situation if they are 70% sure that they will succeed. The evaluation itself, i.e. whether 60% or 70%, depends on different subjective perception and internal factors. In this context, it is easy to observe that judgments under thresholds can provide the flexibility needed to represent personal characteristics. The work of reviewing academic articles by others is an example of this kind of inference in science and, because of this, there are often a number of referees.

In this chapter, I present an uncertainty model suitable for programming agents or global computing. Given the simplicity needed for programming, the uncertainty model might not be new in the AI community, although I have not found another author for the same model, which is certainly based on the classical MYCIN[272]. The combination of significance with originality of the present work consists in:

- Adapting the MYCIN model and bringing the adapted model at the language level. I add two notions to the MYCIN model:
  - ***uu.***

  - The ability to evaluate each hypothesis as resulting in either $false$ or $true$ (or $unknown$) without exploring all of the premises of the hypothesis. And because the premises of a hypothesis might correspond to or include remote accesses, one of the practical motivations in this chapter consists in the ability that agents may have to reduce the number of remote accesses.

- Providing a formalization to the present model.

- Illustrating the importance of uncertainty in programming for global computing.

Moreover, at the time of writing this text, I am assuming that application programs with uncertainty at the language level are absent from the literature on programming languages.

In this chapter, the following sections are organized as follows: Section 8.2 introduces related concepts. Section 8.3 explains the relative concept of truth in this model, while section 8.4 explains forward and backward evaluation under uncertainty, including how a program with uncertainty can compute efficiently on the Internet. In addition to the present model, section 8.4.4 introduces a set of built-in inference operators that can provide other choices to programmers. Finally, section 8.5 contains the conclusion.

8.2 *uu* and Uncertainty

Here I present the formal syntax of a hypothesis declaration, another subset of PLAIN without handlers, with starting symbol hyp:

hyp $\longmapsto$ **hypo** Id factor '{' opthreshold **if** premlist ';' '}'

factor $\longmapsto$ ε | '(' $Number$ ')'

opthreshold $\longmapsto$ ε | **threshold** *Number* ';'

premlist $\longmapsto$ ε | prem cnf | prem cnf ',' premlist

prem $\longmapsto$ '{' premlist '}' | *Id* | '(' expr ')' |
inferop '(' premlist ')' | inferop '(' *Number* ',' premlist ')'

cnf $\longmapsto$ ε | **cnf** *Number*

inferop $\longmapsto$ **uand** | **uor** | **unot** | **dand** | **dor**

expr $\longmapsto$ ... – any three-valued logical expression.

Number $\longmapsto$ – any real number in $[-1.0, +1.0]$.

In the text, ?? can replace **cnf** here.

We all know that the real world is full of rules of causes and consequences and, because of this, as an alternative to deductive clauses such as Horn clauses or Prolog-like rules, uncertainty can also be used to permit an agent to make its own decisions. Like most expert systems environments, a PLAIN-like programming language can provide constructs for representing uncertainty as follows:

```
hypo awebfault {
  threshold 0.4;
  if {
    (!httpaccessok("www.aaa.bbb.ccc.dbf")) cnf 0.2,
    (!httpaccessok("www.ddd.eee.fff.dbf")) cnf 0.2,
    (!httpaccessok("www.ggg.hhh.iii.dbf")) cnf 0.2
  }
  when self true { /* make some decision... */ }
}
```

When an agent contains a great number of such hypotheses, the system is deemed to be intelligent, with the ability of making decisions given the complex and subjective nature of the objects (0.2 of certainty, in each of the above cases). Additionally, operators are provided to be used with certainty factors, as well as nesting expressions with their corresponding factors. In such contexts, parentheses surrounding an expression indicates that its negation does not contribute to refute the consequent, it is simply ignored instead.

In this model, I shall make use of two forms of *unknown* value for variables: uu and tf, although there is one constant in the language for both meanings, *unknown*. In this chapter, uu is a lexical sugar for either uu or tf at points where the difference does not matter. uu is the initial or partial unknown state while tf is the final unknown state after the evaluation is performed. I write uu where this difference is not relevant in a particular formula.

Initially, all hypotheses in the program contain uu. For every logical variable z in the program, if $z \neq uu$, there is an internal state composed by a pair of thresholds $(False, True)$ where $-1 \leq False(z) < True(z) \leq +1$ and which are stated by the programmer and will be clearer in this chapter later; a pair of certainty measures (at least, $x(z)$, and at most, $y(z)$) where $-1 \leq x(z) \leq y(z) \leq +1$, which are inferred dynamically; and three alternative external values that correspond to ff, $\mathbf{uu}$ and tt, namely $false$, *unknown* and *true* (internally, there are four values in $\{ff, uu, tf, tt\}$), one of which is the result from the following conditions intuitively written in PLAIN:

logic val(**logic** z) $=$
  if $z.x \geq z.True$ **then** *true*;
  else if $z.y < z.False$ **then** $false$;
  else *unknown*;

or, in a more mathematical style, still in PLAIN:

logic val(**logic** z) $=$ *true* **if** $z.x \geq z.True$,

$$false \quad \textbf{if } z.y < z.False,$$
$$unknown \textbf{ otherwise};$$

Thus, I write $z.x,\ z.y,\ z.False,\ z.True$ instead of $x(z),\ y(z),\ False(z),\ True(z)$. In this chapter, I omit the variable name when the context applies to all logical variables without the need to specify some particular variable. In this adapted model, by default, $False = 0$ and $True = +0.5$ in the range $[-1.0, +1.0]$.

There is a built-in unary prefix operator, ?, that can be applied to any logical variable. Thus, for a variable z, $?z$ is the certainty measure of z, whose value corresponds to the arithmetic mean of x and y of z. $?z$ is the form in PLAIN for any z while here, in this text, the $z?m$ form is used with the same meaning.

Logical variables are conceptually classified as hypotheses and pieces of evidence. A hypothesis has a premise list, while evidence does not have premises. Premises can be classified as hypotheses, evidence, logical expressions and inference operators (inferop). Besides the certainty measure, logical expressions and inferops can result in $\{ff, uu, tt\}$.

An example of syntax is as follows:

hypo h(-0.2,0.7) {
  if a **cnf** 0.3,
    c,
    $(b * b - 4 * a * c < 0)$ **cnf** 0.2,
    (c),
    uand (0.5, a **cnf** 0.2, b, c **cnf** 0.8) **cnf** 0.9,
    { a **cnf** 0.2, b **cnf** 0.2, c **cnf** 0.3 } **cnf** 0.8
  ;
}

The above hypothesis h contains the thresholds $False = -0.2$ and $True = +0.7$, and its list of premises as follows: a, a logical variable whose certainty

factor is 0.3 above (0.3 is actually the certainty factor for the relation with (a, h), but stated here in a simplified way); c, another logical variable whose certainty factor is 1.0 above, by default; the expression $(b*b-4*a*c < 0)$ whose certainty factor is 0.2; (c), another expression whose certainty factor equals 1.0, the default certainty factor; one instance of **uand** inference operator with truth threshold 0.5, certainty factor 0.9 and whose premises are variables a, b and c with certainty factors 0.2, 1.0 and 0.8 respectively; and a composition among the premises: a, b and c with certainty factors 0.2, 0.2 and 0.3 respectively, and the result from the composition is meant to be multiplied by the certainty factor 0.8 as will be explained later. As suggested in the above example, premises can be nested. Expressions are always between parentheses, and the semantic difference between a variable and an expression containing only the same variable is that when an expression results in $false$ its effect is null with respect to the hypothesis, i.e. it does neither contribute to prove nor refute the hypothesis. As for an occurrence of a variable in the list of premises, if the variable is false its certainty measure is negative. At first sight the use of expression might appear limited, but since an expression can be written in the list of premises more than once with different certainty factors, the use of expressions is flexible and even slightly more general. Thus, c ?? 0.8 corresponds to $\{(c)$?? 0.8, $(not\ c)$?? $-0.8\}$.

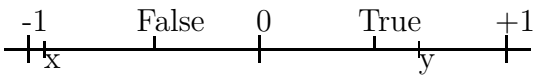
Like measure, certainty factors are in the real interval $[-1.0, +1.0]$ and are written by programmers to measure how the corresponding premises contribute to prove or refute the hypotheses, depending on the sign, positive or negative. While certainty factors are specified in the program, certainty measure is a dynamic value in the same real interval $[-1.0, +1.0]$. For simulating a subjective judgment, in this model, the concept of truth is relative to the programmers beliefs, as well as to beliefs of possible users when the corresponding system is applied. Therefore, each relation between one premise and one hypothesis or between one premise and one inferop has one certainty factor and one certainty measure attached. Syntactically, certainty factors can be floating-point expressions. I simplify the present model by allowing only

constants in that interval. During the evaluation, a certainty measure is multiplied by the corresponding certainty factor, and its result is finally an input to the hypothesis.

By combining hypotheses and premises, the programmer can represent complex knowledge forming an acyclic graph.

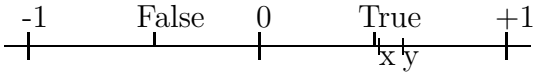
8.3 Uncertainty Handling

The four figures below represent the four possible states of a logical variable. A possible initial state in which a variable can be regarded as *uu* if the condition $x < False \wedge y \geq True$ holds is as follows:

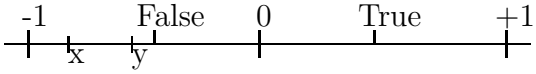


The initial state is not necessarily $x = -1 \wedge y = +1$, but instead both values can be calculated by the compiler.

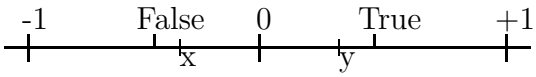
Eventually, a variable can be regarded as *true*, if the condition $x \geq True$ holds:



Or eventually, a variable can be regarded as $false$, if the condition $y < False$ holds:



Or eventually, a variable can be regarded as tf, if the condition $x \geq False \wedge y < True$ holds:



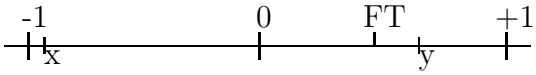
In an alternative model, there might be other values such as *non-False* ($x \geq False \wedge x < True \wedge y \geq True$) and *non-True* ($x < False \wedge y \geq$

$False \wedge y < True$). Both are regarded as *unknown* (uu) in the present model.

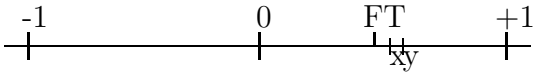
The difference $y - x$ means the amount of non-evaluated evidence. An inconsistent state does not occur in the present model since $x \leq y$ and $False \leq True$ always hold.

A simpler model equates the thresholds, $FT = False = True$. Further, $FT = 0$ is the default value. Graphically, it reduces a hypothesis to only three states:

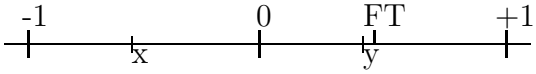
The initial *unknown* state (but here, the correspondence is with the possibly known, kk, value defined in chapter 2 and not the uu value) as the condition $x < FT \leq y$ holds:



Eventually, a variable can be regarded as *true* once the condition $x \geq FT$ holds:



Or eventually, a variable can be regarded as $false$ once the condition $y < FT$ holds:



Both schemes permit variables to contain ***uu*** in the premises and, because of this, hypotheses can also result in ***uu*** in the latter case.

The ability to permit premises and hypotheses to result in ***uu*** is a flexible characteristic of the present programming model.

Before describing the two directions of evaluations, I should set some default values for a premise z according to its category:

- Hypothesis: as described above, $FT(z)$ is statically given by the programmer. $x(z)$ and $y(z)$ are calculated as the agent is initialized, or at the time of compilation of the program;

- Logical evidence: $FT(z) = +0.5,\ x(z) = -1,\ y(z) = +1$;
- Expression: $FT(z) = +1,\ x(z) = 0,\ y(z) = +1$;
- Inferop: $FT(z) = 0,\ x(z) = -1,\ y(z) = +1$.

8.4 Evaluation

Here the present work provides forward and backward evaluations. The former starts from some evidence and goes upwards to the hypotheses while the latter starts from one hypothesis and goes downwards trying to calculate its certainty measure according to its premises. In the following subsections, I provide a scheme, for both forward and backward directions.

Let $[-1, +1] \subset \mathbb{R},\ \mathbb{F} \equiv [-1, +1], \neg, \ominus, \wedge, \&, \vee, \invamp, \rightarrow, \rightarrow, \twoheadrightarrow, \doteq, \leftrightarrow, \dot{\neq} = \{ff, uu, tf, tt\}$. For this section, for a logical variable, its *value and certainty measure* can be represented as a tuple in $\mathbb{O} \equiv \mathbb{F} \times \mathbb{F} \times \mathbb{F} \times \neg, \ominus, \wedge, \&, \vee, \invamp, \rightarrow, \rightarrow, \twoheadrightarrow, \doteq, \leftrightarrow, \neq \times \mathbb{F} \times \mathbb{F}$. The elements of such a tuple are the following, whose description order corresponds to left-right order in the tuple:

- The minimum certainty measure x;
- The maximum certainty measure y;
- The certainty measure;
- The logical value;
- The false threshold $False$;
- The true threshold $True$.

Let $(px, py, ??p, pv, pFalse, pTrue)$ be such a tuple, where px and py correspond to the certainty measures (*at least* and *at most*) as explained, and so on. Let $??p$ be $(px + py)/2$. It will always be in this way, computed on demand.

8.4.1 Forward Evaluation

Forward evaluation is invoked by a built-in function call, perhaps imperative, and usually after some data has been entered. After the evaluation has finished, some imperative actions are typically performed once hypotheses become true or false, and perhaps after other conditions.

The forward evaluation starts from the pieces of evidence and goes upwards. Because a hypothesis can be declared as a premise to another hypothesis, the forward evaluation continues recursively until no more hypotheses need to be calculated.

The composition of a list of premises, whose representation in PLAIN is between a pair of braces, has the same algorithm regardless of the number of premises. Before describing composition, I consider only two premises, namely P with logical value $(px, py, ??p, pv, pFalse, pTrue)$ and Q with $(qx, qy, ??q, qv, qFalse, qTrue)$, to the hypothesis H with $(Hx, Hy, ??H, Hv, HFalse, HTrue)$. Considering that $px = py = 0 \;\; if \;\; pv = uu$ in P and $qx = qy = 0 \;\; if \;\; qv = uu$ in Q, the certainty measures of H are defined by a set of formulae computed as follows:

$$\begin{array}{c} Hx = \\ \varphi(px, qx) = \end{array} \begin{cases} px + (1 - px) \cdot qx, & \text{if } px \geq 0,\ qx \geq 0. \\ px + (1 + px) \cdot qx, & \text{if } px < 0,\ qx < 0. \\ px + qx, & \text{otherwise.} \end{cases}$$

$$\begin{array}{c} Hy = \\ \varphi(py, qy) = \end{array} \begin{cases} py + (1 - py) \cdot qy, & \text{if } qy \geq 0,\ qy \geq 0. \\ py + (1 + py) \cdot qy, & \text{if } py < 0, qy < 0. \\ py + qy, & \text{otherwise.} \end{cases}$$

$$H.?? = mean(Hx, Hy) = \frac{Hx + Hy}{2}$$

and judgment for the value $Hv = H.v = A(H)$ in the following:

$$H.v = A(H) = \begin{cases} tt & \text{if } H.x \geq H.True. \\ ff & \text{if } H.y < H.False. \\ uu & \text{if } (H.x < H.False \vee H.y \geq H.True) \\ & \wedge H.x < H.y. \\ tf & \text{if } H.x \geq H.False \wedge H.y < H.True. \end{cases}$$

where $H.v$ denotes the attribute of logical value of H, and $H.x$ and $H.y$ correspond to Hx and Hy, respectively. For strict evaluation, which is an alternative form here (the default form in PLAIN), the reader shall see that there is a condition, $H.x = H.y$, in addition to the above one for the machine to consider $H.v = tf$. See later.

One of the positive points of these formulae is that the evaluation of the corresponding scheme is both commutative and associative. That is, although the premises are statically written in a sequence in the program, they become available dynamically in any order, and the order does not affect the result of the hypothesis certainty measure.

The structure for every hypothesis H follows: Let $\mathbb{P}$ be the finite set of all premises of the program, where every premise is of type $\mathbb{F} \times \mathbb{O}$. Let $\mathbb{H} \equiv \mathbb{O} \times \mathbb{N} \times \mathcal{P}(\mathbb{P})$ be the set of hypotheses of the program. For every hypothesis $H : \mathbb{H}$, besides its value, the list of premises that lead to H is of the form $\langle n, \{H.P_1, H.P_2, ..., H.P_n\}\rangle$. The whole list is addressed as $H.P$, of type $\mathbb{N} \times \mathcal{P}(\mathbb{P})$. The list length is addressed as $H.P.n$.

Every premise P is of type $\mathbb{F} \times \mathbb{O}$. The fist element of P, denoted as $P.??$, is the certainty factor between P and its corresponding hypothesis. From the hypothesis H, given some $i \in \mathbb{N}$, this certainty factor is referred to as $H.P_i.??$ while the certainty measure of P_i is referred to as $H.P_i?m$.

Thus, for this chapter, $\mathbb{H}$ is the set of hypotheses and $\mathbb{P}$ is the set of premises. Further, from now on I generalize the notation by writing $H.x$ and $H.y$, instead of writing Hx and Hy, respectively. And from now on I am going

to consider any number of premises. For every hypothesis $\mathcal{H}$, I have its finite list of premises, $P_1, P_2, ..., P_i, ..., P_n$. For every P_i, for $i \in [1, n] \subset \mathbb{N}$, the P_i value is in $\{ff, uu, tt\}$, as well as their certainty measures. Furthermore, in addition to the logical value and certainty measures, every premise contains the certainty factor to link to the hypothesis, that is, the certainty factor helps prove or refute the corresponding hypothesis. Given that every $P_i : \mathbb{F} \times \mathbb{O}$, the first element of that P_i is the certainty factor towards the corresponding hypothesis.

Letting $P_i.v = uu \Rightarrow P_i.x = 0$ and $P_i.y = 0$, the certainty measure of $\mathcal{H}$ is defined by a set of formulae, given new values p_i for some occurrences of $i \in [1, n]$, for some n, as follows:

Let $[-1, +1] \subset \mathbb{R}$, $\mathbb{F} \equiv [-1, +1]$, $\neg, \ominus, \wedge, \&, \vee, \parr, \rightarrow, \rightarrow, \twoheadrightarrow, \doteq, \leftrightarrow, \not\equiv \equiv$
$\{ff, uu, tt\}, \mathbb{O} \equiv \mathbb{F} \times \mathbb{F} \times \mathbb{F} \times \neg, \ominus, \wedge, \&, \vee, \parr, \rightarrow, \rightarrow, \twoheadrightarrow, \doteq, \leftrightarrow, \not\equiv \times \mathbb{F} \times \mathbb{F}$.

Suffix functions:

$.x : \mathbb{O} \longrightarrow \mathbb{F}; \qquad (x, y, c, v, f, t).x = x.$

$.x : \mathbb{F} \times \mathbb{F} \longrightarrow \mathbb{F}; \quad (x, y).x = x.$

$.y : \mathbb{O} \longrightarrow \mathbb{F}; \qquad (x, y, c, v, f, t).y = y.$

$.y : \mathbb{F} \times \mathbb{F} \longrightarrow \mathbb{F}; \quad (x, y).y = y.$

$?m : \mathbb{O} \longrightarrow \mathbb{F}; \qquad (x, y, c, v, f, t)?m = \ c.$

$.v : \mathbb{O} \longrightarrow \neg, \ominus, \wedge, \&, \vee, \parr, \rightarrow, \rightarrow, \twoheadrightarrow, \doteq, \leftrightarrow, \not\equiv; \quad (x, y, c, v, f, t).v = v.$

$.False : \mathbb{O} \longrightarrow \mathbb{F}; \ (x, y, c, v, f, t).False = f.$

$.True : \mathbb{O} \longrightarrow \mathbb{F}; \quad (x, y, c, v, f, t).True = t.$

$\alpha : \mathbb{F} \times \mathbb{H} \longrightarrow \mathbb{F};$

$\alpha(z, \mathcal{H}) \ = \ z \ \cdot \ \mathcal{H}.P_i.??$

where $\mathcal{H}.P_i.??$ corresponds to the certainty factor
between P_i and $\mathcal{H}$.

$\delta : \mathbb{P} \longrightarrow \mathbb{F};$

$$\delta(P) = P.y - P.x.$$

$$F : \mathbb{H} \times \mathcal{P}(\mathbb{P}) \times \mathbb{N} \longrightarrow \mathbb{F} \times \mathbb{F};$$

$$F(\mathcal{H}, P, n) = \begin{cases} (\ (F(\mathcal{H}, P, n-1)).x + \delta(P) \cdot \alpha(P_n.x, \mathcal{H}), \\ (F(\mathcal{H}, P, n-1)).y \ ) \\ \qquad if \ n > 0 \wedge \alpha(P_n.x, \mathcal{H}) \geq 0. \\ \\ (\ (F(\mathcal{H}, P, n-1)).x, \\ (\ (F(\mathcal{H}, P, n-1)).y + \delta(P) \cdot \alpha(P_n.y, \mathcal{H})) \\ \qquad if \ n > 0 \wedge \alpha(P_n.y, \mathcal{H}) < 0. \\ \\ (\ (F(\mathcal{H}, P, n-1)).x, \ (F(\mathcal{H}, P, n-1)).y) \\ \qquad if \ n > 0 \wedge P_n.v = \boldsymbol{uu}. \\ \\ (\ 0, \ 0 \ ) \qquad if \ n = 0. \end{cases}$$

and after having calculated the final values x and y,

$$\mathcal{H}?m = \frac{\mathcal{H}.x + \mathcal{H}.y}{2}$$

where $\mathcal{H}?m$ is the confidence measure of the hypothesis $\mathcal{H}$, calculated as follows:

$$\mathcal{H}?m = \frac{(F(x, y, \mathcal{H}, P_i, n)).x + (F(x, y, \mathcal{H}, P_i, n)).y}{2}$$

In the forward evaluation, because states can be changed, expressions in the list of premises are always evaluated when the virtual machine visits some hypothesis.

Forward evaluation is cyclical, and the process is repeated while hypotheses feed other hypotheses with new certainty measures. The process is repeated by possible imperative handlers whose actions can be triggered just after the

virtual machine has proven some particular hypothesis. Since such actions can change the program state, they can require an extra cycle of forward evaluation. The process finishes when the objects involved become stable, i.e. all set of hypotheses have the same values for more than one subsequent cycle.

This model has the advantage of representing and dealing with lack of information. Moreover, unlike Boolean models, the present model represents the condition of being unable to prove or refute a hypothesis.

8.4.2 Operational Semantics - Forward Evaluation

Here, I present an operational semantics for the forward evaluation during computation of a program Π. Given a program Π as a set of symbols, let $\mathbb{E} \subset \Pi$ be the set of pieces of evidence, $\mathbb{H} \subset \Pi$ be the set of hypotheses, $\mathcal{O}p \subset \Pi$ be the set of occurrences of inference operators, and $X \subset \Pi$ be the set of expressions. Let $\mathbb{P} = \mathbb{E} \cup X \cup \mathbb{H}$ and $E \cap X = \ E \cap H = \ X \cap H = \emptyset$. Let $\mathbb{S}$ be the set of states of computation.

Briefly, for our semantics of uncertainty-based inference, I define two relations for forward and backward evaluations, respectively: $\overset{\text{fw}}{\rightsquigarrow}$ and $\overset{\text{bw}}{\rightsquigarrow}$ leading to some state of computation. Another relation, $\overset{\text{eval}}{\rightsquigarrow}$ for expression evaluation, also leads to a state, although I only represent the resulting pair of values, the minimum and maximum certainty measures in $[-1.0, +1.0]$ and the logical value in $\{\mathit{ff}, \mathit{uu}, \mathit{tt}\}$.

$\iota : \mathbb{H} \times \mathbb{S} \times \mathbb{P} \longrightarrow \mathbb{O}$. Intuitively, $\iota(h, s, p)$ is the result from the forward evaluation from state s and premise p to the conclusion h.

Where it is suitable, I write $h.P$ to denote the list of premises of the hypothesis h. I define $\iota(h, s, P)$ to make the sequence of operations explicit in PLAIN, in terms of σ, which in turn results in a pair (x, y), which in its turn is a certainty measure (minimum,maximum), and to state that ι does not change the state, in the PLAIN syntax (although the words and other tokens are not the same):

$\alpha(p, h) \qquad = \ p?m \ \cdot \ h.p.??;$

$\delta(e, l) = \textbf{true if } [e] == head(l),$

$\delta(e, [h, t]) = \delta(e, t) \textbf{ if } [e] \; ! = \; h;$

$$
\begin{aligned}
\sigma(h, s, p) \quad &= \; (h.x \; + \; (h.y - h.x) \cdot \alpha(p, h), \; h.y \\
&\quad mean(h.x + (h.y - h.x) \cdot \alpha(p, h), h.y), \\
&\quad A(h), h.False, h.True) \\
&\qquad \textbf{if } \alpha(p, h) > 0, \\
\sigma(h, s, p) \quad &= \; (h.x, \; hy \; + \; (hy - hx) \cdot \alpha(p, h), \\
&\quad mean(h.x, \; hy \; + \; (hy - hx) \cdot \alpha(p, h)), \\
&\quad A(h), h.False, h.True) \\
&\quad \textbf{if } \alpha(p, h) < 0, \\
\sigma(h, s, p) \quad &= \; h \;\; \textbf{if } \alpha(p, h) = 0;
\end{aligned}
$$

$$
\begin{aligned}
\iota(h, s, p) \quad &= \; \sigma(h, s, P) \textbf{ if } \delta(p, h.P), \\
&\quad h \textbf{ otherwise};
\end{aligned}
$$

where $p \in \mathbb{P}$, $p?m$ is the certainty measure of a premise p and $h.p.??$ is the certainty factor from the premise p to the hypothesis h, which is static from the source program.

Since every hypothesis can have hypotheses, expressions, inference operators and pieces of evidence as premises (for proving or refuting the former hypothesis itself), let $Op(h) \in Op$ be the set of inferops in the list of premises of h, more precisely,

$$
\begin{gathered}
\forall i, j \in Op, \; (i \in h.P \Rightarrow i \in Op(h)) \wedge \\
(i \in j.P \wedge j \in Op(h) \Rightarrow i \in Op(h))
\end{gathered}
$$

Similarly, for some $h \in \mathbb{H}$, let $Obj = E \cup X \cup H$ be the set of all pieces of evidence and expressions and hypotheses of the program, and let $C(h) \subseteq Obj$ be the set of objects that can contribute to prove or refute h. Let $p \in Obj$.

Thus, $p \in h.P \Rightarrow p \in C(h)$, $(p \in C(g) \wedge g \in C(h)) \Rightarrow p \in C(h)$, and $(\exists i \in Op(h),\ (p \in i.P)) \Rightarrow p \in C(h)$.

Then, considering that the symbol a in the **forward** command is grammatically a piece of evidence, an operational semantics for the forward evaluation can be as follows:

$$\frac{a \in h.P \quad \langle h, s\rangle \overset{\text{fw}}{\rightsquigarrow} s'[h?m = \iota(h, s', a)]}{\langle \textbf{forward}\ a, s\rangle \overset{\text{fw}}{\rightsquigarrow} s'}$$

while, as usual, a state (s, s' or s'') followed by an expression between square brackets indicates that the expression holds in the state. For instance, $s[x = 0]$ means that the expression $x = 0$ holds in the state s. The **forward** statement can change the state of more than one hypotheses by propagating uncertainty from a hypothesis to another as follows:

$$\frac{\begin{array}{c} a \in C(h_1) \wedge h_1 \in C(h_2) \\ h_1 \notin C(h_2) \Rightarrow (\langle \textbf{forward}\ a, s\rangle \overset{\text{fw}}{\rightsquigarrow} s'[h_1?m = \iota(h_1, s', a)]) \\ \langle \textbf{forward}\ a, s'\rangle \overset{\text{fw}}{\rightsquigarrow} s''[h_2?m = \iota(h_2, s'', h_1)] \end{array}}{\langle \textbf{forward}\ a, s\rangle \overset{\text{fw}}{\rightsquigarrow} s''}$$

Finally, to state that **forward** is an imperative command in PLAIN, I can insert it in a rule for sequence of commands:

$$\frac{\langle c, s\rangle \overset{\text{fw}}{\rightsquigarrow} s' \quad \langle \textbf{forward}\ a, s'\rangle \overset{\text{fw}}{\rightsquigarrow} s''}{\langle c; \textbf{forward}\ a, s\rangle \overset{\text{fw}}{\rightsquigarrow} s''}$$

8.4.3 Backward Evaluation

While the forward evaluation is invoked by a command to discover those hypotheses that become proven and those hypotheses that become refuted, the backward evaluation is invoked for a particular hypothetic goal, more precisely, when a hypothesis is used in any evaluating expression such as from imperative constructs. The virtual machine then *tries* to decide that particular hypothesis (i.e. the machine tries to prove or refute it), visiting the premises backwards until it reaches the pieces of evidence, by running handlers to change values of the related evidence. After that, the control is returned to that hypothesis.

Finally, the control continues normally in the expression evaluation with the new requested result in $\{ff, uu, tt\}$ while the certainty measures are ignored for the context is deductive.

In a local context, the order of evaluation is not relevant for the result, but since I intend to apply the present model to programming for internet, I propose a way of predicting the best path in terms of efficiency.

Because certain pieces of evidence might not be available at that moment, the hypothesis h that has been expected to be proven or refuted might remain *unknown*, that is, $h.x < h.False \leq h.True \leq h.y$.

An expression as a premise can also result in $\{ff, uu, tt\}$. The corresponding certainty measures are -1 and $+1$ and 0. It is arguable whether it is worth allowing programmers to state some uncertainty inside expressions to allow the virtual machine to use the factors to compute the certainty measure of the expression, in particular if the expression may result in many possible alternative values, such as integer expressions, not only two or three values. Thus, for safety reasons, PLAIN adopts the conservative idea that only logical variables have certainty measures.

The Most-Interesting First Strategy

In this section I introduce *uncertain lazy computation with the most-interesting first strategy*, which is probably useful for global computers[57]. Although it is easy to evaluate all premises of the requested hypothesis sequentially, I introduce a strategy that tries to minimize the number of premises necessary to prove the hypothesis, or to refute it, or both (this case happens where some premises contribute to prove the hypothesis while other premises contribute to refute it. In this case, both subsets are relevant). Such a strategy becomes particularly significant in programming for a global computer because remote accesses are considerably much more expensive than use of local resources, although Internet 2 and other global networks in the future tend to minimize this difference. Thus, I classify three possible intentions for a hypothesis

containing uu:

- $\mathcal{I}_t$: To make the hypothesis be *true*.
- $\mathcal{I}_f$: To make the hypothesis *false*.
- $\mathcal{I}_{tf}$: To make the hypothesis either *true* or *false* (or tf).

All logical variables containing uu, as well as all non-lazy expressions, are valuable. For a valuable hypothesis or logical evidence or expression or inferop occurring as a premise p of a hypothesis h, in $\mathcal{I}_t$ what I look for is $(h.True - h.x)$ / $|h.p.??|$: for more than one hypotheses or pieces of evidence or expressions or inferops, the smaller this value is, the more interesting p is, but I only consider positive results for the premises. Similarly, in the intention $\mathcal{I}_f$ what I look for is $(h.y - h.False)$ / $|h.p.??|$: the smaller the value is, the more interesting p is, and here I consider only non-negative results for the premises. In $\mathcal{I}_{tf}$ what I look for is the value of $(h.True - h.x + h.y - h.False)$ / $|h.p.??|$: the smaller, the more interesting, and here only positive values are included in comparisons. For future work, every of these three values can be multiplied by the number of variables with uu and that play the rôle of p, for including the associated cost in the strategy. Sometimes this cost may be important because, as an example, connections might have to be set in order to associate values to variables.

Then, intuitively, the strategy consists in calculating these values for every premise of some hypothesis, and then to choose the smallest value obtaining the premise to be exploited first, in the backward direction. Then, the process is repeated until the hypothesis is no longer valuable, that it, until its logical value is either $true$, $false$ or tf. Although the strategy does not generally guarantee the most efficient proof or refutation, it is natural and efficient. Therefore, I refer to this final measure as the most *interesting*.

Among the valuable premises, the virtual machine can make use of $d(h, \mathcal{I})$ to identify the most interesting premise (that is, its index in $[1, n]$) in a list P with n valuable premises of a given hypothesis h, some intention $\mathcal{I}$. The

function definition is the following:

Valuable premises only

$\epsilon(\emptyset) \quad = \emptyset,$

$\epsilon(\{P\} \cup Q) \quad = \textbf{if } \psi(P) \textbf{ in } \{false, tf, true\}$

$\quad \textbf{then } \epsilon(Q) \textbf{ else } \{P\} \cup \epsilon(Q);$

$\psi(x, y, F, T) = true \textbf{ if } x \geq T,$

$\qquad\qquad\qquad\quad false \textbf{ if } y < F,$

$\qquad\qquad\qquad\quad tf \textbf{ if } F \leq x \wedge y < T,$

$\qquad\qquad\qquad\quad uu \textbf{ otherwise};$

Unary ψ

$\psi(P) \quad = \psi(P.x, P.y, P.False, P.True);$

$d(h, \mathcal{I}) \quad = \ -2 \textbf{ if } \epsilon(h.P) = \emptyset,$

$d(h, \mathcal{I}) \quad = \ d(h, \epsilon(h.P), \mathcal{I}) \textbf{ otherwise};$

Ternary d

$d(h, \{P_i\}, \mathcal{I}) \quad = \ i,$

with intention $\mathcal{I}_t$

$d(h, \{P_i\} \cup Q, \mathcal{T}) = \textbf{ if}$

$\quad 0 < (h.True - P_i.x) \ / \ |h.P_i.??| \leq$

$(h.True - h.P_{d(h,Q,\mathcal{T})}.x) \ / \ |h.P_{d(h,Q,\mathcal{T})}.??|$

$\quad \textbf{then } i \textbf{ else } d(h, Q, \mathcal{T}),$

with intention $\mathcal{I}_f$

$d(h, \{P_i\} \cup Q, \mathcal{F}) = \textbf{ if}$

$\quad 0 \leq (P_i.y - h.False) \ / \ |h.P_i.??| \leq$

$(h.P_{d(h,Q,\mathcal{F})}.y - h.False) \ / \ |h.P_{d(h,Q,\mathcal{F})}.??|$

then i **else** $d(h, Q, \mathcal{F})$,

with intention $\mathcal{I}tf$

$d(h, \{P_i\} \cup Q, \mathcal{U}) \ =$ **if**

$0 < (h.True - P_i.x \ + \ P_i.y - h.False) \ / \ |h.P_i.??| \leq$

$(h.True - h.P_{d(h,Q,\mathcal{U})}.x + h.P_{d(h,Q,\mathcal{U})}.y - h.False) \ /$

$|h.P_{d(h,Q,\mathcal{U})}.??|$

then i **else** $d(h, Q, \mathcal{U})$;

Finally, the backward evaluation finishes only when the hypothesis is no longer valuable. The c function repeats d until that condition holds, as follows:

$U(h, x, y) = (\varphi(h.x, x), \varphi(h.y, y), mean(\varphi(h.x, x), \varphi(h.y, y)),$

$\psi(\varphi(h.x, x), \varphi(h.y, y), h.False, h.True), h.False, h.True);$

$c(h, \mathcal{I}) \ =$ **if** $d(h, \mathcal{I}) = -2$ **then** h **else** $c(h, h.P, \mathcal{I})$;

Ternary c (every hypothesis has at least one premise)

$c(h, \{P_i\}, \mathcal{I}) \ = \ $ **if** $\psi(P_i) = tf$ **then** h **else**

let $p = newstate(P_i)$ **in** $U(h, p.x \cdot h.P_i.??, p.y \cdot h.P_i.??)$,

$c(h, P, \mathcal{I}) \ = h$ **if** $\epsilon(P) = \emptyset$,

$c(h, P, \mathcal{I}) \ =$ **let** $j = d(h, P, \mathcal{I}); \ p = newstate(P_j)$ **in**

$c(U(h, p.x \cdot h.P_i.??, p.y \cdot h.P_i.??), P \backslash \{P_j\}, \mathcal{I})$

otherwise;

Thus, the operational semantics for using a hypothesis in some lazy expression can be as follows:

$$\frac{\langle c(h, \mathcal{I}_{tf}), s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \langle \mathbf{lazy}(h), s'\rangle \overset{\text{bw}}{\rightsquigarrow} (s', \iota(h, s', c(h, \mathcal{I}_{tf})))}{\langle \mathbf{lazy}(h), s\rangle \overset{\text{bw}}{\rightsquigarrow} (s', \iota(h, s', c(h, \mathcal{I}_{tf})))}$$

Accordingly, there are also built-in functions in the programming language for the other intentions:

$$\frac{\langle c(h,\mathcal{I}_t),s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \langle h,s'\rangle \overset{\text{bw}}{\rightsquigarrow} (s',\iota(h,s',c(h,\mathcal{I}_t)))}{\langle \mathbf{lazy}(\mathbf{prv}(h)),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s',\iota(h,s',c(h,\mathcal{I}_t)))}$$

$$\frac{\langle c(h,\mathcal{I}_f),s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \langle h,s'\rangle \overset{\text{bw}}{\rightsquigarrow} (s',\iota(h,s',c(h,\mathcal{I}_f)))}{\langle \mathbf{lazy}(\mathbf{rft}(h)),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s',\iota(h,s',c(h,\mathcal{I}_t)))}$$

for trying to prove ($\mathcal{I}_t$) and refute ($\mathcal{I}_f$), respectively, some hypothesis h. The above rules are not complete.

Lazy and Strict Computations

In the backward evaluation, for the requested hypothesis, there can be lazy and strict computations. Lazy computation is the one which makes use of the most-interesting first strategy while strict computation evaluates all premises of z in any case until $z.v = tf$ for every logical variable z.

By default, the virtual machine adopts strict computation and, to specify lazy computation, the programmer writes the **lazy** keyword. Thus, **lazy** is a unary-prefix function with one of the highest precedences.

During the lazy computation of backward evaluation, for every variable z, one of the conditions $z.x \geq z.True$, or $z.y < z.False$, as stated previously, is enough to prove or refute the hypothesis, respectively. Thus, an operational semantics for uncertain lazy computation is as follows:

$$\frac{s[h.x \geq h.True]}{\langle \mathbf{lazy}(h),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s,tt)}$$

$$\frac{s[h.y < h.False]}{\langle \mathbf{lazy}(h),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s,ff)}$$

$$\frac{s[h.x \geq h.False \ \wedge \ h.y < h.True]}{\langle \mathbf{lazy}(h),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s,tf)}$$

$$\frac{\begin{array}{c}\text{let } z = \iota(h,s',h.P_{c(h,\mathcal{I})}) \quad \langle c(h,\mathcal{I}),s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \\ \langle h?m,s'\rangle \overset{\text{bw}}{\rightsquigarrow} (s',z) \quad s'[h.False \leq z.x \wedge z.y < h.True]\end{array}}{\langle \mathbf{lazy}(h),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s',uu_f)}$$

$$\frac{\langle c(h,\mathcal{I}),s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \langle h?m,s'\rangle \overset{\text{bw}}{\rightsquigarrow} (s',\iota(h,s',h.P_{c(h,\mathcal{I})})) \quad \iota(h,s',h.P_{c(h,\mathcal{I})}).x \geq h.True}{\langle \mathbf{lazy}(h),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s',tt)}$$

$$\frac{\langle c(h,\mathcal{I}),s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \langle h?m,s'\rangle \overset{\text{bw}}{\rightsquigarrow} (s',\iota(h,s',h.P_{c(h,\mathcal{I})})) \quad \iota(h,s',h.P_{c(h,\mathcal{I})}).y < h.False}{\langle \mathbf{lazy}(h),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s',ff)}$$

Some rules for **prv**:

$$\frac{s[h.x \geq h.True]}{\langle \mathbf{lazy}(\mathbf{prv}(h)),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s,tt)}$$

$$\frac{s[h.y < h.False]}{\langle \mathbf{lazy}(\mathbf{prv}(h)),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s,ff)}$$

$$\frac{s[h.x \geq h.False \ \wedge \ h.y < h.True]}{\langle \mathbf{lazy}(\mathbf{prv}(h)),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s,tf)}$$

$$\frac{\text{let } z = \iota(h,s',h.P_{c(h,\mathcal{I}_t)}) \quad \langle c(h,\mathcal{I}_t),s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \langle h?m,s'\rangle \overset{\text{bw}}{\rightsquigarrow} (s',z) \quad s'[h.False \leq z.x \wedge z.y < h.True]}{\langle \mathbf{lazy}(\mathbf{prv}(h)),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s',uu)}$$

$$\frac{\langle c(h,\mathcal{I}_t),s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \langle h?m,s'\rangle \overset{\text{bw}}{\rightsquigarrow} (s',\iota(h,s',h.P_{c(h,\mathcal{I}_t)})) \quad \iota(h,s',h.P_{c(h,\mathcal{I})}).x \geq h.True}{\langle \mathbf{lazy}(\mathbf{prv}(h)),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s',tt)}$$

$$\frac{\langle c(h,\mathcal{I}_t),s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \langle h?m,s'\rangle \overset{\text{bw}}{\rightsquigarrow} (s',\iota(h,s',h.P_{c(h,\mathcal{I}_t)})) \quad \iota(h,s',h.P_{c(h,\mathcal{I})}).y < h.False}{\langle \mathbf{lazy}(\mathbf{prv}(h)),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s',ff)}$$

and some rules for **rft**:

$$\frac{s[h.x \geq h.True]}{\langle \mathbf{lazy}(\mathbf{rft}(h)),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s,tt)}$$

$$\frac{s[h.y < h.False]}{\langle \mathbf{lazy}(\mathbf{rft}(h)),s\rangle \overset{\text{bw}}{\rightsquigarrow} (s,ff)}$$

$$\frac{s[h.x \geq h.False \ \wedge \ h.y < h.True]}{\langle \mathbf{lazy}(\mathbf{rft}(h)), s\rangle \overset{\text{bw}}{\rightsquigarrow} (s, tf)}$$

$$\frac{\text{let } z = \iota(h, s', h.P_{c(h,\mathcal{I}_f)}) \quad \langle c(h, \mathcal{I}_f), s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \langle h?m, s'\rangle \overset{\text{bw}}{\rightsquigarrow} (s', z) \quad s'[h.False \leq z.x \wedge z.y < h.True]}{\langle \mathbf{lazy}(\mathbf{rft}(h)), s\rangle \overset{\text{bw}}{\rightsquigarrow} (s', uu)}$$

$$\frac{\langle c(h, \mathcal{I}_f), s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \langle h?m, s'\rangle \overset{\text{bw}}{\rightsquigarrow} (s', \iota(h, s', h.P_{c(h,\mathcal{I}_f)})) \quad \iota(h, s', h.P_{c(h,\mathcal{I})}).x \geq h.True}{\langle \mathbf{lazy}(\mathbf{rft}(h)), s\rangle \overset{\text{bw}}{\rightsquigarrow} (s', tt)}$$

$$\frac{\langle c(h, \mathcal{I}_f), s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \langle h?m, s'\rangle \overset{\text{bw}}{\rightsquigarrow} (s', \iota(h, s', h.P_{c(h,\mathcal{I}_f)})) \quad \iota(h, s', h.P_{c(h,\mathcal{I})}).y < h.False}{\langle \mathbf{lazy}(\mathbf{rft}(h)), s\rangle \overset{\text{bw}}{\rightsquigarrow} (s', ff)}$$

On the other hand, for uncertain strict computation, the request for the value of a hypothesis can also mean to exploit all its premises in order to get a more accurate certainty measure. The order of computation in the list of premises is static as follows:

$b(h, \mathcal{I}) \ = \ \mathbf{if}\ d(h, \mathcal{I}) \ = \ -2\ \mathbf{then}\ h\ \mathbf{else}\ b(h, h.P, \mathcal{I});$

Ternary b

$b(h, P, \mathcal{I}) \ = \ h\ \mathbf{if}\ P = \emptyset;$

$b(h, P, \mathcal{I}) \ = \ \mathbf{let}\ p = newstate(P_1)\ \mathbf{in}$

$b(U(h, p.x \cdot h.P_1.??, p.y \cdot h.P_1.??), P \backslash \{P_1\}, \mathcal{I})\ \mathbf{otherwise};$

The operational semantics of the strict evaluation is as follows:

$$\frac{s[h.v = tf]}{\langle h, s\rangle \overset{\text{bw}}{\rightsquigarrow} (s, h.v)}$$

$$\frac{s[h.v \neq tf \wedge h.x < h.y] \wedge \langle b(h, \mathcal{I}), s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \iota(h, s', h.P_1).x \geq h.False \wedge \iota(h, s', h.P_1).y < h.True}{\langle h, s\rangle \overset{\text{bw}}{\rightsquigarrow} (s', tf)}$$

$$\frac{s[h.v \neq tf \wedge h.x < h.y] \quad \langle b(h,\mathcal{I}), s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \iota(h,s',h.P_1).y \geq h.True}{\langle h, s\rangle \overset{\text{bw}}{\rightsquigarrow} (s', tt)}$$

$$\frac{s[h.v \neq tf \wedge h.x < h.y] \quad \langle b(h,\mathcal{I}), s\rangle \overset{\text{bw}}{\rightsquigarrow} s' \quad \iota(h,s',h.P_1).x < h.False}{\langle h, s\rangle \overset{\text{bw}}{\rightsquigarrow} (s', ff)}$$

8.4.4 Inference Operator - Inferop

Inferops result in a pair of type $(\mathbb{F} \times \mathbb{F}) \times \neg, \ominus, \wedge, \&, \vee, \text{⅋}, \rightarrow, \rightarrow, \twoheadrightarrow, \doteq, \leftrightarrow, \not\doteq$, in which the first member is a pair which in turn consists of the *at least* and the *at most* certainty measures, in $[-1.0, +1.0]$, and the second member, in the outermost pair, is a logical value in $\{ff, uu, tt\}$. The operational semantics for the built-in inferops are:

Uncertain and:

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \alpha) \quad \langle b, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, \beta) \quad min(u.y, v.y) < T}{\langle \mathbf{uand}(T, a, b), s\rangle \overset{\text{bw}}{\rightsquigarrow} ((min(u.x, v.x), min(u.y, v.y)), ff)}$$

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \alpha) \quad \langle b, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, \beta) \quad min(u.x, v.x) < T \wedge min(u.y, v.y) \geq T}{\langle \mathbf{uand}(T, a, b), s\rangle \overset{\text{bw}}{\rightsquigarrow} ((min(u.x, v.x), min(u.y, v.y)), tf)}$$

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \alpha) \quad \langle b, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, \beta) \quad min(u.x, v.x) \geq T}{\langle \mathbf{uand}(T, a, b), s\rangle \overset{\text{bw}}{\rightsquigarrow} ((min(u.x, v.x), min(u.y, v.y)), tt)}$$

Uncertain or:

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \alpha) \quad \langle b, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, \beta) \quad max(u.y, v.y) < T}{\langle \mathbf{uor}(T, a, b), s\rangle \overset{\text{bw}}{\rightsquigarrow} ((max(u.x, v.x), max(u.y, v.y)), ff)}$$

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \alpha) \quad \langle b, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, \beta) \quad max(u.x, v.x) < T \wedge max(u.y, v.y) \geq T}{\langle \mathbf{uor}(T, a, b), s\rangle \overset{\text{bw}}{\rightsquigarrow} ((max(u.x, v.x), max(u.y, v.y)), tf)}$$

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \alpha) \quad \langle b, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, \beta) \quad max(u.x, v.x) \geq T}{\langle \mathbf{uor}(T, a, b), s\rangle \overset{\text{bw}}{\rightsquigarrow} ((max(u.x, v.x), max(u.y, v.y)), tt)}$$

where *min* and *max* are functions defined as usual as in table entitled "A Total min and max Definitions with *uu*." or formula in chapter 2,

Uncertain not:

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \gamma) \quad 1 - u.x < T}{\langle \mathbf{unot}(T, a), s\rangle \overset{\text{bw}}{\rightsquigarrow} ((1 - u.y, 1 - u.x), \not{f}\not{f})}$$

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \gamma) \quad 1 - u.y < T \wedge u.x \geq T}{\langle \mathbf{unot}(T, a), s\rangle \overset{\text{bw}}{\rightsquigarrow} ((1 - u.y, 1 - u.x), tf)}$$

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \gamma) \quad 1 - u.y \geq T}{\langle \mathbf{unot}(T, a), s\rangle \overset{\text{bw}}{\rightsquigarrow} ((1 - u.y, 1 - u.x), tt)}$$

Note that **unot** inverts the minimum and maximum values and swaps the positions.

Determinable and: (for any $f > 0$)

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \alpha) \quad \langle b, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, \beta) \quad u.y < T \vee v.y < T}{\langle \mathbf{dand}(T, a, b)\ \mathbf{cnf}f, s\rangle \overset{\text{bw}}{\rightsquigarrow} ((-f, -f), \not{f}\not{f})}$$

$$\frac{\begin{array}{c}\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \alpha) \quad \langle b, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, \beta) \\ (u.x < T \vee v.x < T) \wedge u.y \geq T \wedge v.y \geq T\end{array}}{\langle \mathbf{dand}(T, a, b)\ \mathbf{cnf}f, s\rangle \overset{\text{bw}}{\rightsquigarrow} ((0, 0), uu_f)}$$

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \alpha) \quad \langle b, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, \beta) \quad u.x \geq T \wedge v.x \geq T}{\langle \mathbf{dand}(T, a, b)\ \mathbf{cnf}f, s\rangle \overset{\text{bw}}{\rightsquigarrow} ((f, f), tt)}$$

Determinable or:

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \alpha) \quad \langle b, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, \beta) \quad u.y < T \wedge v.y < T}{\langle \mathbf{dor}(T, a, b)\ \mathbf{cnf}f, s\rangle \overset{\text{bw}}{\rightsquigarrow} ((-f, -f), \not{f}\not{f})}$$

$$\frac{\begin{array}{c}\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \alpha) \quad \langle b, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, \beta) \\ u.x < T \wedge v.x < T \wedge (u.y \geq T \vee v.y \geq T)\end{array}}{\langle \mathbf{dor}(T, a, b)\ \mathbf{cnf}f, s\rangle \overset{\text{bw}}{\rightsquigarrow} ((0, 0), uu_f)}$$

$$\frac{\langle a, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, \alpha) \quad \langle b, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, \beta) \quad u.x \geq T \vee v.x \geq T}{\langle \mathbf{dor}(T, a, b)\ \mathbf{cnf}f, s\rangle \overset{\text{bw}}{\rightsquigarrow} ((f, f), tt)}$$

for any $f > 0$.

Finally, users are able to define their own inferops.

8.5 Conclusion

The Internet has raised many issues in programming. The fact that connections are neither reliable nor efficient makes us consider the possibility of programming with uncertainty.

I formally introduced uncertainty as a programming language feature, considering the current scenario in the presence of code mobility and the Internet. Additionally, the present uncertainty model permits evaluation with partial information, by considering *unknown*[106] as part of the model in both the variables used as premises and in resulting hypotheses. I also introduced a number of inference operators. As a result, ***uu*** can permit the unification of this paradigm with others.

Chapter 9

uu in Globallog

In chapter 2, I introduced a logic and a calculus which provide the notion of $\mathtt{uu}$*. Here I apply this notion to logic programming. The present chapter helps form our first study on* $\mathtt{uu}$ *in programming. The full programming study includes the other chapters in part II of the present PhD thesis.*

This chapter presents a three-valued logic programming language which permits definitions of clauses under closed-world assumption or without it, due to the presence of a constant, $\mathtt{uu}$*, at the language level. A third truth value is used to provide only one negation, defined here as abstract negation, while Extended Logic Programs adopt two kinds of negation. I present an operational semantics for both propositional and the predicate forms, including variables. The language can be seen as an adaptation of Prolog capable of capturing lack of information. In particular, the language can be viewed as as an appropriate compromise solution between logic and a global structure such as the Internet. Little work has been done combining logic with such a platform.*

9.1 Introduction

Since Prolog was designed, several programming languages, techniques and paradigms have been proposed and developed. Much research work has been done combining logic programming with other paradigms. A few examples include the languages λProlog[225] and LIFE[8], which combine logic with functional programming. However, in spite of the great importance of such work, no logic programming language provides representation of lack of information together with only one negation. The proposed language (GLOBALLOG) has two relevant attributes, namely, the *abstract negation* and the ability to distinguish a refutable clause from the one which is not proven from a list of clauses. The need for such a distinction has become increasingly relevant for Internet programming, because connections sometimes fail and programs must be robust enough to continue running. Thus, in a distributed knowledge-based system over the Internet, a remote query that is not able to prove some predicate results in *unknown* instead of *false*.

Many important theoretical contributions[111, 112] have been made to logic programming by researchers, including well-founded semantics [113, 132] and stable models[244], also, the work of Przymusinsky and Gelfond[135], Ginsberg[141] among others. A more recent proposal is [25], which is a semantics for Prolog programs based on a 4-valued logic. Theories on negation have been proposed since the close-world assumption (CWA)[253]. [19] presents a survey on this subject. In *general* logic programs (GLP), negation as failure (NAF)[68] is used to infer negative information. In fact, NAF is a concept which depends on the CWA[1]. A GLP clause has the form

$$L_0 \leftarrow \mathcal{L}_1, \ldots, \mathcal{L}_m.$$

where $\leftarrow$ is an implication symbol (for instance the :– symbol in Prolog), $\mathcal{L}_i$, for $1 \leq i \leq m$, is a literal possibly with the **not** symbol, i.e. the negation as

[1]That is why the proposed negation is called "abstract negation" in the sense that its use does not depend on whether the world is assumed to be closed or not.

failure operator, while $\mathcal{L}_0$ does not accept negation, although there has been some recent work on this constraint[166].

A newer approach called Extended Logic Programming (ELP) provides two forms of negation: NAF and explicit negation[45, 97, 177]. As well as ELP, Gelfond and Lifschitz[133] present a proposal which is based on the stable model semantics and answer sets. The authors define an extended logic program as being a sequence of clauses of the form

$$L_0 \leftarrow L_1,\ \ldots,\ L_m,\ not\ L_{m+1},\ \ldots,\ not\ L_n$$

where $0 \leq m \leq n$ and each L_i is a literal, i.e. either a predicate or the explicit negation ($\neg$) of a predicate, and *not* is the negation as failure operator. Thus, in ELP, the CWA for a predicate p can be represented by $\neg p(X) \leftarrow not\ p(X).$, which, although flexible, makes use of two kinds of negation.

Contributions have been done by Pereira[9, 238], Kowalski and others[260], also in ELP. There are other new approaches, such as [314] based on Well-Founded Semantics. However, GLOBALLOG is outside the scope of ELP.

ELP was not conceived to allow programming under the choice of open- or closed-world assumptions. If the programmer uses NAF in a predicate p, it means that he or she is assuming a closed world. If he or she uses the explicit negation instead, it means that he or she is assuming an open world for *that* use of p. However, it is not clear how to use *positive* predicates choosing either assumptions, which is exactly what GLOBALLOG allows programmers to do. In other words, ELP is more closely related to the negations than the assumptions.

Like ELP, there exist three possible results: *false*, *true* and *unknown*. While what is stated as $false$ is simply regarded as $false$, what is not represented in the program (or what is unable to be proven) is simply regarded, by default, as *unknown*. The distinction between *false* and *unknown* allows the inference machine (also called *system* here), not only the application program, to recognize the *need* for learning. For example, if a query from a predicate p results in *unknown*, the system could ask the user whether he or she would

like to insert some definition in order to improve the knowledge base.

There are some situations, however, in which CWA with NAF are much more appropriate[18, 94]. For example, considering in Prolog the clauses for defining a member of a list, one (the programmer) can write the following rules:

$member(X, [X|\_]).$
$member(X, [\_|Z]) :- member(X, Z).$

Without the CWA, the programmer would have to explicitly write the interpretation of failure by stating what is not a member of the list, otherwise a query, such as *member(a,[b,c,d])*, would result in *unknown* in a three-valued logic language. This is true for many situations. However, in many other contexts, such as in constraints representation, even using CWA with NAF, it would be useful to regard what fails as $true$, instead of $false$, in such a way that the sequence of clauses becomes flexible, more readable and smaller, thus allowing faster proofs. Thus, GLOBALLOG provides both open- and closed-world assumptions. I am going to use the term *open-world assumption* as opposed to the CWA. Formally, I define both here as

$$CWA \stackrel{def}{=} (\forall\varphi)\ \frac{\neg(\Delta \vdash \varphi)}{\Delta \models \varphi = \mathit{ff}}$$

that is, closed-world assumption can be defined as for every formula φ, if φ cannot be proven from the set of formulae Δ, φ is interpreted as false, and accordingly,

$$OWA \stackrel{def}{=} (\forall\varphi)\ \frac{\neg(\Delta \vdash \varphi) \wedge \neg(\Delta \vdash \neg\varphi)}{\neg(\Delta \models \varphi = tt) \wedge \neg(\Delta \models \varphi = \mathit{ff})}$$

That is, if we cannot prove φ or its negation from Δ, we cannot interpret φ as tt nor as ff. Thus, I use uu for this intermediary value:

$$OWA \stackrel{def}{=} (\forall\varphi)\ \frac{\neg(\Delta \vdash \varphi)}{\Delta \models \varphi = uu}$$

Apart from this three-valued logic programming paradigm, I intend to integrate this paradigm with others including mobile agents[105]. Because

uu is present in every one of these sublanguages, it is particularly easy to integrate them. I therefore prefer the term *hybrid paradigm* instead of *multi paradigm*. Finally, the present chapter describes the programming language GLOBALLOG.

One can observe that logic and the Internet are two orthogonal concepts. Formal logics ought to match reality. The work on GLOBALLOG aims at reconciling both concepts. In fact, the name of the language, GLOBALLOG, is a shorthand for *global logic programming*. GLOBALLOG is part of the PLAIN programming language[103].

9.1.1 Conventions

In this chapter, the choice for the operational semantics is not only for being uniform regarding the previous chapters but also for stressing details that are relevant for possible implementations. Implicitly, the semantics is of space and time, defined in chapter 2. However, as the syntax is similar to Prolog, the referred to language is described intuitively during the text, together with the formalism. I use the constants *ff*, *tt* and *uu* to mean $false$, $true$ and *unknown*, respectively. I use both terminologies interchangeably.

9.1.2 Contents of this Chapter

Section 9.2 introduces the language together with its operational semantics. The semantic rules are defined for both propositional form and predicate form, i.e., our formulas capture the semantics of GLOBALLOG. The need for the concept of "intention" is also explained in this section. Section 9.3 contains an operational analysis of the language by comparing it with Prolog. Section 9.4 gives examples of queries in the language by tracing the programs whereas section 9.5 contains some remarks on consistency of the knowledge base and gives more examples. Section 9.6 concludes the chapter.

9.2 Syntactical and Semantic Definitions

Briefly speaking, like Prolog, GLOBALLOG is a sub-language of first-order predicate calculus. When proving the body of a rule, the system considers the subgoals from left to right and searches for solutions by using depth-first strategy and backward chaining. Before definition 6, I do not consider variables in the formal semantics, only propositions.

A program in GLOBALLOG is a sequence of clauses, and not an unordered set of clauses. This is because the flows of control are sequential and GLOBALLOG is deterministic. They start from the first clause and go downwards. Let Δ be a program formed by the sequence of clauses $c_1, ..., c_n$. Then it is said that a computation by Δ proves a goal g iff there exists some clause c_i in Δ such that g is an immediate consequence of c_i, assuming that the body of c_i, $b(c_i)$, can be proven. Formally,

$$\frac{(\exists i, 1 \leq i \leq n)\ b(c_i) \wedge c_i \vdash_i g}{c_1, ..., c_n \vdash g}$$

There might be zero or more solutions, that is, zero or more clauses that prove g. In the case of zero is the case where there is no solution. A solution s denotes one clause that proves the goal. Let S be the sequence of solutions, either $\emptyset$ or $sl_0, ..., sl_m$, $1 \leq m \leq n$, semantically defined as follows:

$$sl_0 \stackrel{def}{=} u \text{ if } (\exists i, 1 \leq i \leq n)(b(c_i) \wedge c_i \vdash_i g) \ \wedge \ (\forall j,\ 1 \leq j < i)\ \neg(b(c_j) \wedge c_j \vdash_i g) \wedge c_i = u$$

$$sl_{k+1} \stackrel{def}{=} u \text{ if } (\exists i, k,\ k < i \leq n)(b(c_i) \wedge c_i \vdash_i g) \wedge (b(c_k) \wedge c_k \vdash_i g) \wedge (\forall j,\ k < j < i)\ \neg(b(c_j) \wedge c_j \vdash_i g) \wedge c_i = u$$

For the definition of the operational semantics of GLOBALLOG, let Δ be a program in GLOBALLOG and Θ be the set of all states for some computation carried out by Δ.

Definition 6 *There are three predefined constants in* GLOBALLOG*, namely, ff, tt, and uu. Let* V^3 *be the set of truth values,* $\{ff, tt, uu\}$.

Remark: In terms of domain theory, *uu* is a value in the domain, and not the bottom ($\perp$). *uu* is a result, as *ff* and *tt* are, and does not represent non-

terminating computation. There exists no order relation between elements of $\{ff, tt, uu\}$.

Definition 7 *The unary symbol* **not** *is the only negation in* GLOBALLOG*, called abstract negation. Its basic semantics is defined in the following two paragraphs, although the chapter itself is necessary to describe details concerning its semantics.*

For the rules, let $\rightsquigarrow$ be the semantic relation for language constructs of GLOBALLOG as will be defined later, and let $\overset{s}{\rightsquigarrow}$ be the semantic relation for implicit actions by the system, resulting in another state.

Semantics 4 *The uu value, uu, extends the semantics of the classical connectives according to Łukasiewicz[116, 186]:* **not** $ff \rightsquigarrow tt$, **not** $tt \rightsquigarrow ff$, **not** $uu \rightsquigarrow uu$. *In the presence of the conjunctive operation (definition 10), as well as the Boolean results,* $uu \wedge ff \rightsquigarrow ff$ *and finally* $tt \wedge uu \rightsquigarrow uu$. *For details, see semantics 9.*

Definition 8 *A constant symbol is a term, a variable is a term, if f is an m-ary function symbol and $t_1, ..., t_m$ are terms, then $f(t_1, ..., t_m)$ is a term.*

Remark: A constant symbol is also called 0-ary function. Lexically, functions and variables are written as literals: a function symbol starts with a lower-case letter and a variable symbol starts with an upper-case letter.

Definition 9 *If p is an m-ary predicate symbol and $t_1, ..., t_m$ are terms, then both $p(t_1, ..., t_m)$ and* **not** *$p(t_1, ..., t_m)$ are predicative formulae, predicates for short: the former is a* positive predicate *and the latter is a* negative predicate. *If q is a predicate, then both $p^m(q)$ and* **not** *$p^m(q)$ are meta–predicates and the letter m here is not arity but instead part of the symbol. There exist three Boolean meta-predicates in the language, namely, istrue, isfalse, isunknown, with obvious meanings.*

Definition 10 *A program in* GLOBALLOG *is a sequence of clauses.* Clauses *have one of the forms below:*

$$[\textbf{not}]\ p(t_1,\ ...,\ t_n).$$

or

$$[\textbf{not}]\ p(t_1,\ ...,\ t_n)\ \Leftarrow [\textbf{not}]\ p_1(t_{1,1},\ ...,\ t_{1,r}), ...,\ [\textbf{not}]\ p_m(t_{m,1},\ ...,\ t_{m,s}).$$

where $\Leftarrow$ stands for an inference operator, either $\leftarrow$ (open) or $\leftharpoonup$ (closed). The first clause is a *fact* while the second is a *rule*. The square brackets are at the meta level and indicate that the **not** operator is optional in these positions. The predicate symbols are denoted as p and as p_i, for $1 \leq i \leq m$.

As usual, the predicate before the inference symbol is the *head* of the rule, while the sequence of predicates after the inference symbol is the *body* of the rule. The comma is the lazy non-commutative *three-valued and* operator with left association (semantics 9).

9.2.1 Syntactical Definitions

In this sub-section, I describe the GLOBALLOG syntax by using Backus-Naur Form, or BNF for short. In the current chapter, I have only symbolic constants or literals, and no numerical constants. In the definition of the non-terminal symbol *variable* below, like in Prolog, the symbol "_" stands for a new or anonymous variable which cannot be bound anywhere. The terminal symbols are in double quotation marks and keywords are in bold face. Thus, the syntactical definition is as follows:

$program \longmapsto clauses \mid clause\ clauses$

$clauses \longmapsto \epsilon \mid clause\ clauses$

$clause \longmapsto opNot\ lowercaseletter\ string\ '('\ listOfParameters\ ')'\ body\ '.'$

$\qquad\qquad \mid \textbf{close}\ contantOrvariable\ '.'$

$opNot \longmapsto \epsilon \mid \textbf{not}$

$constant \longmapsto lowerCaseLetter\ string \mid 'tt' \mid 'ff' \mid 'uu'$

$lowerCaseLetter \longmapsto 'a' \mid ... \mid 'z'$

$string \longmapsto \epsilon \mid lowerCaseLetter\ string \mid upperCaseLetterstring$

$$| \ '\_' \ string$$

$$upperCaseLetter \longmapsto \ 'A' \mid ... \mid 'Z'$$

$$listOfParameters \longmapsto \ factor \mid factor \ ',' \ listOfParameters$$

$$factor \longmapsto \ constant \ opParameters \mid variable$$

$$opParameters \longmapsto \ \epsilon \mid '(' \ listOfParameters \ ')'$$

$$variable \longmapsto \ upperCaseLetter \ string \mid '\_'$$

$$body \longmapsto \ \epsilon \mid ':-' \ listOfterms \mid '<-' \ listOfterms$$

$$listOfterms \longmapsto \ term \mid listOfterms \ ',' \ term$$

$$term \longmapsto opNot \ constant \ opParameters \mid metaPredicate \ '(' \ clause \ ')'$$

$$metaPredicate \longmapsto \mathbf{istrue} \mid \mathbf{isfalse} \mid \mathbf{isunknown}$$

$$contantOrvariable \longmapsto \ constant \mid variable$$

Key words: *tt*, *ff*, *uu* (definition 6), and **isunknown**, **istrue**, **isfalse** (definition 9), and **not**. The :– symbol corresponds to the CWA inference symbol, while the ⇐ symbol corresponds to the OWA inference symbol. However, in our analysis, as already defined, I refer to them as $\leftharpoondown$ and $\leftarrow$, respectively. I also make use of some syntactical sugar, e.g. the relational operators are infix with the usual notation, $\neq$ and $=$, instead of literal predicates.

Syntax 5 *An* `essential` *requirement: For a given predicate definition, there exists no list of rules with mixed inference symbols, and this is guaranteed by the syntax. Formally, in accordance with definition 11 below,* $\mathcal{R}[\leftarrow] \cap \mathcal{R}[\leftharpoondown] = \emptyset$. *Although the context-free grammar introduced above does not capture this, this syntactical constraint must be considered.*

9.2.2 Semantic Definitions

As we shall see, the semantic rules for GLOBALLOG make use of some notions and notations specific for defining the semantics whereas it is not common to present operational semantics for logic-based languages. I present some definitions in this subsection and subsequently use these notions.

Definition 11 *Let* $\Delta \in$ GLOBALLOG, $\mathcal{R}[\leftarrow]$ *be the set of all rules in* Δ *that contain the* $\leftarrow$ *symbol, and* $\mathcal{R}[\leftharpoondown]$ *be the set of all rules in* Δ *that contain the* $\leftharpoondown$ *symbol. Let -*$\mathcal{R}$ *be the set of all rules in* Δ *that contain the* **not** *connective in their heads, and +*$\mathcal{R}$ *be the set of all rules in* Δ *that do not contain the* **not** *connective in their heads. Let* $-\mathcal{R}[\leftarrow] = -\mathcal{R} \cap \mathcal{R}[\leftarrow]$, $+\mathcal{R}[\leftarrow] = +\mathcal{R} \cap \mathcal{R}[\leftarrow]$, $-\mathcal{R}[\leftharpoondown] = -\mathcal{R} \cap \mathcal{R}[\leftharpoondown]$, *and* $+\mathcal{R}[\leftharpoondown] = +\mathcal{R} \cap \mathcal{R}[\leftharpoondown]$.

For later definitions, I informally define an operator $\iota\ (x, (x_1, ..., x_n)) = (\exists i)\ x = x_i$ to check whether a particular value is an element of a tuple.

Let Var be the set of variables in the program Δ (each variable is denoted by a string), and let Val be the set of values in the program. I write here x/v to denote a variable x having value v as a syntax sugar for (x, v), which is an ordered pair in $Var \times Val$, since I shall not use the / arithmetic operator in this chapter. Let AnS denote an answer set, possibly with some subscript, with structure $\{f(x_0/v_{0,0}, ..., x_n/v_{0,n}), ..., f(x_0/v_{m,0}, ..., x_n/v_{m,n})\}$ where f is a predicate formula (possibly with **not**) or, depending on the focus, $\{sl_0, ..., sl_m\}$, with two important properties: $sl_i = sl_j \Rightarrow i = j$ and $i \neq j \wedge \iota(x_i, sl_k) \wedge \iota(x_j, sl_k) \Rightarrow x_i \neq x_j$, which states that the same variable cannot appear more than once in the same solution.

Let $\uplus$ be the concatenation of two answer sets and $\mathcal{A}$ be the set of answer sets. Formally,

$$
\begin{aligned}
&\uplus :\ \mathcal{A} \times \mathcal{A} \longrightarrow \mathcal{A} \\
&\uplus\ (ans_1, \emptyset) \ =\ ans_1 \\
&\uplus\ (\emptyset, ans_2) \ =\ ans_2 \\
&\uplus\ (\{e_1\}, \{e_2, ..., e_n\}) \ =\ \{e_1, e_2, ..., e_n\}, \quad n \geq 2 \\
&\uplus\ (\{e_1, ..., e_n\}, ans_2) \ =\ \{e_1\} \uplus (\{e_2, ..., e_n\} \uplus ans_2), \quad n \geq 2
\end{aligned}
$$

In the semantic rules, I use the rewriting relation $\rightsquigarrow$ to denote evaluation of some expression resulting in a three-tuple $\langle value, answer, state \rangle$ where *value* is in V^3, *answer* corresponds to the ordered answer set in $\mathcal{A}$, and *state* is the

state of the computation. The current answer set is part of the state in such a way that, in other contexts, I write $s[AnS]$ whenever I want to state explicitly that the answer set Ans is in state s.

Let the type V be $Var \times Val$. Thus, I define the operation $\ll: V \times \mathcal{A} \longrightarrow Boolean$ as $\ll (x/v, \{sl_1, ..., sl_n\}) = (\exists i)\ \iota(x/v, sl_i)$ to mean that the value of a pair x/v is in at least one solution from a given set $\{sl_1, ..., sl_n\}$. Like $\uplus$, I use $\ll$ as an infix operator in the semantics rules. I also use $\not\ll$ as syntax sugar for the $\neg(x \ll y)$ construct. This infix notation also applies to $\equiv$, the syntactical equivalence relation, which simply stands for the same variable in some context, e.g. $x_i \equiv y_j$ meaning that, although these variables are different in the formula, they denote the same variable in the program Δ, or, in other contexts, $\equiv$ is used to compare two operands of type V.

Substitution

Substitution is carried out implicitly in the GLOBALLOG machine. All free variables contain a typed uu. Here, the meaning of a typed uu is that the corresponding variable is universally quantified. Thus, substitution is informally defined as the operation which binds a variable to some value in the domain of the variable. Substitution is also done in the unification procedure.

I can define the substitution σ using an extension of PCF[218, 267] as follows. I initially define the $SList$ type:

$$SList = \mu\ t.\mathbf{unit} + ((Var \times Val) \times t)$$

where $\mu\ t.\mathbf{unit} + ((Var \times Val) \times t)$ is a syntax sugar for $\mathbf{fix}(\lambda t.\mathbf{unit} + ((Var \times Val) \times t))$.

$$\begin{aligned}\sigma : SList \times SList \rightarrow SList \ =\ & \lambda L_1 : SList.\ \lambda L_2 : SList. \\ & \mathbf{if}\ L_1 = * \mathbf{\ then}\ *\ \mathbf{else\ if}\ L_2 = * \mathbf{\ then}\ * \\ & \mathbf{else}\ \sigma(tail(L_1), \sigma'(head(L_1),\ L_2))\end{aligned}$$

and, for σ', given s containing one substitution $\langle x, v\rangle$, i.e. value v for variable x, and list L,

$$\sigma' : Var \times Val \times SList \to SList \;=\; \lambda sub : Var \times Val . \lambda L : SList .$$
$$\textbf{letrec } x : Var = \textbf{Proj}_1\ sub \textbf{ in}$$
$$\textbf{if } L = * \textbf{ then } *$$
$$\textbf{else if } x = \textbf{Proj}_1\ head(L) \textbf{ then}$$
$$cons(\langle x, v\rangle,\ \sigma'(\langle x, v\rangle, tail(L))) \textbf{ else}$$
$$cons(\langle x, uu\rangle,\ \sigma'(\langle x, v\rangle, tail(L)))$$

where *cons* constructs a list from its *head* and its *tail*. I also define substitution in a tuple from an answer set generating another answer set as follows:

$$AnswerSet = \mu\ t.\textbf{unit} + (SList \times t)$$

where $\mu\ t.\textbf{unit} + (SList \times t)$ is a syntax sugar for $\textbf{fix}(\lambda t.\textbf{unit} + (SList \times t))$.

$$\sigma\alpha : AnswerSet \times SList \to$$
$$AnswerSet \;=\; \lambda L_1 : AnswerSet.\ \lambda L_2 : SList.$$
$$\textbf{if } L_1 = * \textbf{ then } *$$
$$\textbf{else } \sigma'(head(L_1),\ L_2) \uplus \sigma\alpha(tail(L_1),\ L_2)$$

The same three functions can be defined in the PLAIN[103] syntax as follows:

$$\textbf{fnc } List\ \sigma\ (List\ L_1,\ List\ L_2)\ =$$
$$\textbf{if } L_1 == [\,] \textbf{ or } L_2 == [\,] \textbf{ then } [\,]$$
$$\textbf{else } \sigma(tail(L_1), \sigma'([head(L_1)],\ L_2));$$

and, for σ', given s containing one substitution (x, v), i.e. value v for variable x, and list L,

$$\textbf{fnc } List\ \sigma'\ (List\ s,\ List\ L)\ =$$
$$\textbf{let } Var\ x = Var(s), Val\ v = Val(tail(s)) \textbf{ in}$$
$$\textbf{if } L == [\,] \textbf{ then } [\,]$$
$$\textbf{else if } x == Var(List(L)) \textbf{ then}$$
$$[\ [x, v]\ ] + \sigma'([x, v], tail(L)) \textbf{ else}$$
$$[\ head(L)\ ] + \sigma'([x, v], tail(L));$$

where + here concatenates two lists as operands, and a construct of the form $[\langle exp \rangle]$ constructs a list from a sequence expression $\langle exp \rangle$, the only operand, possibly empty. In the particular case above, [$[x, v]$] constructs a list of only one element which in turn is a list of two values (hence, a pair containing) x and v. The corresponding $\sigma\alpha$ function can be written in PLAIN as follows:

fnc *List* $\sigma\alpha$ (*List* L_1, *List* L_2) =
  if $L_1 ==$ [ ] **then** [ ]
  else $\sigma'(head(L_1), L_2) \uplus \sigma\alpha(tail(L_1),\ L_2)$;

The Semantic Rules

Semantics 6 *I start presenting the operational semantics of a sequence of clauses of* Δ *as follows:*

Let $\mathcal{SEQ}$ be a sequence of clauses $\{c_1, ..., c_n\}$. Then, for any query q,

$$\frac{\begin{array}{c}\langle [p(x_0, ..., x_n) \Leftarrow b.], s\rangle \rightsquigarrow (tt, AnS_1, s') \\ \langle \mathcal{SEQ}, s'\rangle \rightsquigarrow (v, AnS_2, s'')\end{array}}{\langle \{p(x_0, ..., x_n) \Leftarrow b., \mathcal{SEQ}\}, s\rangle \rightsquigarrow (tt, \uplus(AnS_1, AnS_2), s'')}$$

where $\Leftarrow$ denotes either $\leftarrow$ or $\leftharpoondown$ and p denotes a predicate symbol. That is, the final answer set is formed by concatenating the answer sets of the clauses. The above rule applies to tt. For the other results,

$$\frac{\begin{array}{c}\langle [p(x_0, ..., x_n) \Leftarrow b.], s\rangle \rightsquigarrow (u, \emptyset, s') \quad u \neq tt \\ \langle \mathcal{SEQ}, s'\rangle \rightsquigarrow (v, AnS_2, s'')\end{array}}{\langle \{p(x_0, ..., x_n) \Leftarrow b., \mathcal{SEQ}\}, s\rangle \rightsquigarrow (v, AnS_2, s'')}$$

Therefore, intuitively, answers are obtained chronologically in the same order as the physical order of the clauses, from top to bottom.

Semantics 7 *The inference symbol '*$\leftarrow$*' has the following operational semantics for any query:*

$$\frac{\begin{array}{c}\langle b, s\rangle \rightsquigarrow (tt, AnS, s') \quad \forall(i, j,\ 0 \leq i \leq m,\ 0 \leq j \leq n) \\ w_{i,j} = v_{i,j}\ if\ x_{i,j}/v_{i,j} \ll AnS \vee w_{i,j} = uu\ if\ x_{i,j}/v_{i,j} \not\ll AnS\end{array}}{\begin{array}{c}\langle [p(x_0, ..., x_n) \leftarrow b.], s\rangle \rightsquigarrow \\ (tt, \{(x_0/w_{0,0}, ..., x_n/w_{0,n}), ..., (x_0/w_{m,0}, ..., x_n/w_{m,n})\}, s')\end{array}}$$

where b denotes the body and p denotes a predicate symbol with possible **not** operator (the condition $w_{i,j} = uu$ above is necessary because there can be parameters in the head of the rule that is not used in its body). I represent the other rules in a more simplified way as the following:

$$\frac{\langle b, s\rangle \rightsquigarrow (\bot, \emptyset, s')}{\langle [h \leftarrow b.], s\rangle \rightsquigarrow (\bot, \emptyset, s')}$$

$$\frac{\langle b, s\rangle \rightsquigarrow (\mathit{ff}, \emptyset, s') \vee \langle b, s\rangle \rightsquigarrow (uu, \emptyset, s')}{\langle [h \leftarrow b.], s\rangle \rightsquigarrow (uu, \emptyset, s')}$$

where h and b are the head and the body of a rule respectively; $s,\ s',\ s'' \in \Theta$, $v \in V^3$ and $\bot$ indicates infinite computation.

Intuitively, if the body of a rule is tt, so is its head (9.2.2). If the body diverges so does the head (9.2.2). If the body is either ff or uu (9.2.2), the system ignores the rule and the search carries on to the next rule on (9.2.2 and 9.2.2), with result in $\{tt, uu\}$ (syntax 5).

Remark: Regarding the resulting value uu, the above semantics also comes from the observation that, in both classical and intuitionist logic, a valid conditional has two alternative values for the consequent (either tt or ff) when the antecedent is ff.

Semantics 8 *The inference symbol '$\leftharpoonup$' has the following operational semantics for any query:*

$$\frac{\begin{array}{c}\langle b, s\rangle \rightsquigarrow (tt, AnS, s') \quad \forall\ i, j,\ 0 \le i \le m,\ 0 \le j \le n, \\ w_{i,j} = v_{i,j} \ \ if\ x_{i,j}/v_{i,j} \ll AnS \vee w_{i,j} = uu\ if\ x_{i,j}/v_{i,j} \not\ll AnS\end{array}}{\begin{array}{c}\langle [p(x_0, ..., x_n) \leftharpoonup b.], s\rangle \rightsquigarrow \\ (tt, \{(x_0/w_{0,0}, ..., x_n/w_{0,n}), ..., (x_0/w_{m,0}, ..., x_n/w_{m,n})\}, s')\end{array}}$$

$$\frac{\langle b, s\rangle \rightsquigarrow (\bot, \emptyset, s')}{\langle [p \leftharpoonup b.], s\rangle \rightsquigarrow (\bot, \emptyset, s')}$$

$$\frac{\langle b, s\rangle \rightsquigarrow (\mathit{ff}, \emptyset, s') \vee \langle b, s\rangle \rightsquigarrow (uu, \emptyset, s')}{\langle [p \leftharpoonup b.], s\rangle \rightsquigarrow (\mathit{ff}, \emptyset, s')}$$

Intuitively, if the body of a rule is tt, so is its head (9.2.2). If the body diverges so does the head (9.2.2). If the body is either ff or uu (9.2.2), the

system ignores the rule and the search carries on to the next rule on (9.2.2 and 9.2.2), with result in $\{ff, tt\}$ (syntax 5). The above rules keep GLOBALLOG semantics compatible with CWA and NAF.

Semantics 9 *The comma which separates goals in the body of a rule is the three-valued $\wedge$ operator, in accordance with the following semantics. The formulas 9.2.2-9.2.2 do not present the* **not** *operator in the semantics, as this level of detail is not necessary here. I also remove parameters when they are not necessary.*

$$\frac{\begin{array}{c}\langle p(x_0,...,x_n), s\rangle \rightsquigarrow (tt, AnS, s') \\ \langle \sigma\alpha(AnS, (y_0,...,y_m)), s'\rangle \overset{s}{\rightsquigarrow} s''[AnS_2] \\ \langle q(y_0,...,y_m), s''\rangle \rightsquigarrow (tt, AnS_3, s''')\end{array}}{\begin{array}{c}\langle [p(x_0,...,x_n),\ q(y_0,...,y_m)], s\rangle \rightsquigarrow \\ (tt, \{(x_0/w_{0,0},...,x_k/w_{0,k}),...,(x_0/w_{k,0},...,x_k/w_{m,k})\}, s'''[Ans_3])\end{array}}$$

where $n \leq k \wedge m \leq k \wedge k \leq m+n$, and also $(x_i/w_{i,j} \ll AnS \Rightarrow (x_i/\alpha \ll AnS_2 \Rightarrow \alpha = w_{i,j})) \vee (x_i/w_{i,j} \not\ll AnS \wedge x_i \equiv y_i \wedge y_i/w_{i,j} \ll AnS_2)$. Notice that, in the numerator, there are implicit substitutions such that $(\forall i,j)\ x_i/v \ll AnS \wedge x_i \equiv y_j \Rightarrow y_j/v \ll AnS_2$ holds, after the system action σ between the states s' and s''.

Removing those tuple repetitions in formula 9.2.2, I can also state here the order between those sets of solutions, in a way which is equivalent to lexicographic order. Thus, let $AnS(p,q)$, the answer set from the goal $(p \wedge q)$, be

$$\{(x_0/w_{0,0},...,x_k/w_{0,k}),...,(x_0/w_{k,0},...,x_k/w_{m,k})\}$$

as in formula 9.2.2, without parameters.

Furthermore, let the infix $\sqsubseteq$ be the subset relation between two tuples that can be defined here as $\sqsubseteq$: $V \times V \longrightarrow Boolean$. $t_1 \sqsubseteq t_2 \overset{def}{=} (\forall x)\ \iota(x, t_1) \Rightarrow \iota(x, t_2)$. Then,

$$\begin{array}{c}(\forall x/u, y/v \ll AnS(p,q))\ x/u \equiv sl_i(p,q) \wedge y/v \equiv sl_j(p,q) \Rightarrow \\ (i < j \Rightarrow (\forall a,b,c,d)(sl_a(p), sl_c(q) \sqsubseteq sl_i(p,q) \wedge sl_b(p), sl_d(q) \sqsubseteq sl_j(p,q)) \Rightarrow \\ (a < b \vee (a = b \wedge c < d)))\end{array}$$

and for equal solutions:

$$\forall(x/u, y/v \ll AnS(p,q))x/u \equiv sl_i(p,q) \wedge y/v \equiv sl_j(p,q) \Rightarrow$$
$$(i = j \Rightarrow (\forall a,b,c,d)(sl_a(p), sl_c(q) \sqsubseteq sl_i(p,q) \wedge sl_b(p), sl_d(q) \sqsubseteq sl_j(p,q)) \Rightarrow$$
$$(a = b \wedge c = d))$$

where sl_i is defined in the formulae at the beginning of section 9.2 for some goal g. Here I write the goal as an explicit parameter. Furthermore, for the cardinalities of these answer sets, namely $\beta = |AnS(p,q)|,\ m = |AnS(p)|$ and $n = |AnS(q)|$, all of $\beta < m, \beta > m, \beta < n, \beta > n, \beta = m, \beta = n$ may hold.

For failure, the rules are the following:

$$\frac{\begin{array}{c}\langle p(x_0,...,x_n), s\rangle \rightsquigarrow (tt, AnS, s') \\ \langle \sigma\alpha(AnS, (y_0,...,y_m)), s'\rangle \overset{s}{\rightsquigarrow} s'' \\ \langle q(y_0,...,y_m), s''\rangle \rightsquigarrow (\not{f}, \emptyset, s'')\end{array}}{\langle [p(x_0,...,x_n),\ q(y_0,...,y_m)], s\rangle \rightsquigarrow (\not{f}, \emptyset, s'')}$$

$$\frac{\begin{array}{c}\langle p(x_0,...,x_n), s\rangle \rightsquigarrow (tt, AnS, s') \\ \langle \sigma\alpha(AnS, (y_0,...,y_m)), s'\rangle \overset{s}{\rightsquigarrow} s'' \\ \langle q(y_0,...,y_m), s''\rangle \rightsquigarrow (uu, \emptyset, s'')\end{array}}{\langle [p(x_0,...,x_n),\ q(y_0,...,y_m)], s\rangle \rightsquigarrow (uu, \emptyset, s'')}$$

$$\frac{\langle p(x_0,...,x_n), s\rangle \rightsquigarrow (\not{f}, \emptyset, s)}{\langle [p(x_0,...,x_n), q(y_0,...,y_m)], s\rangle \rightsquigarrow (\not{f}, \emptyset, s)}$$

$$\frac{\langle p(x_0,...,x_n), s\rangle \rightsquigarrow (uu, \emptyset, s)}{\langle [p(x_0,...,x_n), q(y_0,...,y_m)], s\rangle \rightsquigarrow (uu, \emptyset, s)}$$

While goals are being satisfied in the body of a rule, the control continues from the left to the right of the rule as follows:

$$\frac{\begin{array}{c}\langle [listOfterms,\ q(y_0,...,y_m)], s\rangle \rightsquigarrow (tt, AnS, s') \\ \langle \sigma\alpha(AnS, (z_0,...,z_o)), s'\rangle \overset{s}{\rightsquigarrow} s''[AnS_2] \\ \langle q(z_0,...,z_m), s''\rangle \rightsquigarrow (tt, AnS_3, s''')\end{array}}{\langle [listOfterms,\ q(y_0,...,y_m),\ r(z_0,...,z_o)], s\rangle \rightsquigarrow (tt, AnS_3, s''')}$$

where $listOfterms$ corresponds to a non-terminal symbol of the GLOBAL-LOG grammar that denotes a sequence of satisfied goals in the above formula.

Notice that, from the above set of rules, there is no need to represent the backtrack when the second operand results in uu or $\not{f}$.

Definition 12 *A closed predicate* [2] *is a predicate defined by a list of rules in* $\mathcal{R}[\leftharpoondown]$ *or facts under CWA, or both. A fact is under CWA if there exists some rule in* $\mathcal{R}[\leftharpoondown]$ *whose head contains the same predicate as the fact does, or defined by the close statement, defined in the following paragraph:*

Definition 13 *To explicitly define a list of facts of the same predicate p as being under CWA, the programmer writes the clause* ***close*** *p. in the program. To set CWA for all facts that do not have some predicate defined in* $\mathcal{R}[\leftarrow]$*, the programmer writes the clause* ***close*** *X. in the program, where X is a valid name of variable.*

Definition 14 *An open predicate is a predicate which is not closed. In* GLOBALLOG*, a predicate defined by only facts is an open predicate by default.*

Definition 15 Clause *is the general term used for either* fact *or* rule*. A fact is syntactically a rule without the body nor the inference symbol. If p(...) is a fact, then it is equivalent to the rule* $p(...) \Leftarrow true$*, where* $\Leftarrow$ *is one of the two inference symbols. The fact* **not** $p(...)$ *is equivalent to* **not** $p(...) \Leftarrow true$*. If a fact or rule contains the* **not** *connective in its head, I generically refer to it as a negative clause, as usual. Let* $\mathcal{C}^-$ *be the set of all negative clauses in* Δ*. If a fact or rule does not contain the* **not** *connective in its head, I also generically refer to it as a positive clause. Let* $\mathcal{C}^+$ *be the set of all positive clauses in* Δ*. A closed clause is either a rule in* $\mathcal{R}[\leftharpoondown]$ *or a fact defined under CWA. An open clause is a clause which is not closed.*

Syntax 10 *The clause* ***close*** *p. is inconsistent with predicates p in* $\mathcal{R}[\leftarrow]$*, and such a consistency is guaranteed by the language, both statically and dynamically. Although the present context-free grammar does not capture this, this syntactical constraint must be taken into consideration.*

Syntax 11 *For a given closed predicate, both positive and negative clauses do not coexist. Formally,* $\mathcal{C}^+ \cap \mathcal{C}^- = \emptyset$*. Although the present context-free*

[2] This terminology is similar to *open/closed formula* and they have different meanings, but both have been used in the literature.

grammar does not capture this, this syntactical constraint must be taken into consideration.

Semantics 12 *If a closed predicate is defined by positive clauses, as* Prolog *does, failure for this predicate means ff. However, if a closed predicate is defined by negative clauses, failure for this predicate means tt.*[3]

Formally, $\neg(\mathcal{C}^+ \vdash_i q) \Rightarrow \mathcal{C}^+ \models (q = ff)$ where q is a closed predicate symbol in $\mathcal{C}^+$, and also $\neg(\mathcal{C}^- \vdash_i q) \Rightarrow \mathcal{C}^- \models (q = tt)$ where q is a closed predicate symbol in $\mathcal{C}^-$.

Definition 16 *The default logical value (δ) is the value assumed for the failure in the search for a given predicate. Failure leads to δ which is either tt, ff or uu.*

Definition 17 *A goal is an occurrence of a predicate when it is being used in a proof. When the predicate contains the* **not** *operator the goal is said to be negative, otherwise the goal is positive. A subgoal is a goal written in the body of a rule, while a main goal is a goal at the uppermost level. A query is the computation from a set goal. A subquery is a query from a subgoal. A positive query is a query from a positive goal. A negative query is a query from a negative goal. Because a computation can result in tt, ff or uu, I avoid using the term* proof*, which should also mean a computation that results in ff or uu.*

For the following semantic definitions, let $C^{][}$ be the set of clauses defined under OWA, and $C^{[]}$ be the set of clauses defined under CWA. In GLOBALLOG, $C^{][} \cap C^{[]} = \emptyset$ holds.

Semantics 13 *If no clause unifies a goal g, δ depends on whether the corresponding predicate is open or closed, and whether the goal is positive or negative: if the predicate is open, the query results in uu. On the other hand,*

[3]The idea is that the programmer ought to state only the exception cases for a predicate under CWA.

if the predicate is closed, the query results in ff in the case of a positive goal, and the query results in tt in the case of a negative goal. See also semantics 7, 8 and 12:

$$\frac{\langle (\nexists p)\ \mu(g,p) \wedge g \in C^{][}, s\rangle \overset{s}{\rightsquigarrow} (tt, \emptyset, s')}{\langle g, s\rangle \rightsquigarrow (uu, \emptyset, s')}$$

where μ is the unification algorithm (Section 9.2.3. The only difference between my unification algorithm and others' is described as follows: the algorithm receives in an additional parameter the intention, which is relevant during the unification, and returns the information about whether or not predicates with opposite signs are unified when the received intention is $\mathcal{I}_{tf}$ (see section 9.2.3, below). For the other intentions, the algorithm returns *false* in the latter parameter).

$$\frac{\langle (\nexists p)\ \mu(g,p) \wedge g \in C^{[]}, s\rangle \overset{s}{\rightsquigarrow} (tt, \emptyset, s')}{\langle g, s\rangle \rightsquigarrow (ff, \emptyset, s')}$$

$$\frac{\langle (\nexists p)\ \mu(\mathbf{not}\ g,p) \wedge g \in C^{[]}, s\rangle \overset{s}{\rightsquigarrow} (tt, \emptyset, s')}{\langle \mathbf{not}\ g, s\rangle \rightsquigarrow (tt, \emptyset, s')}$$

9.2.3 Intentions

As well as binding variables, a proof from a goal may be the result of one of the following different intentions:

$\mathcal{I}_t$: Proving that a predicate is true;

$\mathcal{I}_f$: Proving that a predicate is false;

$\mathcal{I}_{tf}$: Trying to find the truth value for a predicate, either uu, ff or tt.

If the intention is $\mathcal{I}_t$, the system will look for the positive clauses that unify the established goal. On the other hand, if the intention is $\mathcal{I}_f$, the system will look for the negative clauses that unify the established goal. With $\mathcal{I}_{tf}$, the system may unify both positive and negative clauses, and it is done at once, downwards. Thus, at a declarative semantics level, the first two cases combined do not form the third case because the order of the clauses is relevant.

Semantics 14 *For open predicates, if the subgoal is positive the system adopts the intention $\mathcal{I}_t$, otherwise, the system adopts the intention $\mathcal{I}_f$.*

Semantics 15 *For closed predicates, the system always adopts the intention $\mathcal{I}_{tf}$, although it finds either positive or negative clauses. If the subgoal g is negative, the sub-query results in the classical negation of the query whose goal corresponds to the positive predicate:*

$$\frac{\langle g \in C^{[]} \wedge (\exists[p \leftharpoondown b])\ \mu(g, p, \emptyset, \mathcal{I}_{tf}), s\rangle \overset{\mathrm{s}}{\rightsquigarrow} (tt, L, s') \quad \langle b, s'\rangle \rightsquigarrow (tt, L', s'')}{\langle g, s\rangle \rightsquigarrow (tt, \sigma(L', g), s'')}$$

$$\frac{\langle g \in C^{[]} \wedge (\exists[\mathbf{not}\ p \leftharpoondown b])\ \mu(g, p, \emptyset, \mathcal{I}_{tf}), s\rangle \overset{\mathrm{s}}{\rightsquigarrow} (tt, L, s') \quad \langle b, s'\rangle \rightsquigarrow (tt, L', s'')}{\langle \mathbf{not}\ g, s\rangle \rightsquigarrow (tt, \sigma(L', g), s'')}$$

$$\frac{\langle g \in C^{[]} \wedge (\exists[\mathbf{not}\ p \leftharpoondown b])\ \mu(g, p, \emptyset, \mathcal{I}_{tf}), s\rangle \overset{\mathrm{s}}{\rightsquigarrow} (tt, L, s') \quad \langle b, s'\rangle \rightsquigarrow (tt, L', s'')}{\langle g, s\rangle \rightsquigarrow (ff, \sigma(L', g), s'')}$$

$$\frac{\langle g \in C^{[]} \wedge (\exists[p \leftharpoondown b])\ \mu(g, p, \emptyset, \mathcal{I}_{tf}), s\rangle \overset{\mathrm{s}}{\rightsquigarrow} (tt, L, s') \quad \langle b, s'\rangle \rightsquigarrow (tt, L', s'')}{\langle \mathbf{not}\ g, s\rangle \rightsquigarrow (ff, \sigma(L', g), s'')}$$

Example 2 *A query p to the program $\{\mathbf{not}\ p.\}$ results in ff, while a query p to the program $\{\mathbf{not}\ q.\ p \leftarrow q.\}$ or to the program $\{q.\ p \leftarrow \mathbf{not}\ q.\}$ results in uu. See remark 9.2.2.*

In a case where there does not exist unifying clause:

$$\frac{\langle g \in C^{[]} \wedge (\nexists[p \leftharpoondown b])\ \mu(g, p, X, \mathcal{I}_{tf}), s\rangle \overset{\mathrm{s}}{\rightsquigarrow} (tt, Y, s) \quad (\nexists[\mathbf{not}\ p \leftharpoondown b])\ \mu(g, p, X, \mathcal{I}_{tf}), s\rangle \overset{\mathrm{s}}{\rightsquigarrow} (tt, Z, s')}{\langle g, s\rangle \rightsquigarrow (ff, \emptyset, s')}$$

$$\frac{\langle g \in C^{[]} \wedge (\nexists[p \leftharpoondown b])\ \mu(g, p, X, \mathcal{I}_{tf}), s\rangle \overset{\mathrm{s}}{\rightsquigarrow} (tt, Y, s) \quad (\nexists[\mathbf{not}\ p \leftharpoondown b])\ \mu(g, p, X, \mathcal{I}_{tf}), s\rangle \overset{\mathrm{s}}{\rightsquigarrow} (tt, Z, s')}{\langle \mathbf{not}\ g, s\rangle \rightsquigarrow (ff, \emptyset, s')}$$

Semantics 16 *In general, for any query with intention $\mathcal{I}_{tf}$, if a negative clause is proven from a positive goal (9.2.3), the query results in ff. Likewise, if a positive clause is proven from a negative goal (9.2.3), the query results in ff. If a negative clause is proven from a negative goal (9.2.3), or a positive clause is proven from a positive goal (9.2.3), the query results in tt.*

9.3 An Operational Analysis

In this section, I analyze the language at the operational level. Prolog will be used for comparisons because it is well known, but the comparisons apply to all languages that are based on GLP. Most of the proofs are by bissimulation[146] together with structural induction over the syntax, defined in section 9.2.1.

Proposition 3 *Let $\mathcal{P}$ be a Prolog program and $\mathcal{Q}^o$ be a program in* Globallog *formed by simply adding to $\mathcal{P}$ the clause* ***close*** *X.. Then, for every query, $\mathcal{Q}^o$ gives the same answer as $\mathcal{P}$.*

[Proof:] First, whenever a goal does not unify any clause in both $\mathcal{P}$ and $\mathcal{Q}^o$, both programs answer $false$. In both cases, if the goal unifies a fact, the answer is $true$ with the same values for the variables in the query, according to the unifications.

In both cases, if the goal unifies the head of a rule, both bodies are evaluated. In both $\mathcal{P}$ and $\mathcal{Q}^o$, whenever the body of a rule fails, its head becomes *ff* and the system backtracks. In both cases, whenever the body of a rule succeeds, its head becomes *tt*, with the same substitution set, according to the unifications.

Let **not** p be a negative subgoal in the body of a rule. As the rules are in $\mathcal{R}[\leftharpoondown]$, $\mathcal{Q}^o$ assumes the intention $\mathcal{I}_{tf}$. Since there are no negative rules in the program, the goal **not** p matches p clauses, which, like $\mathcal{P}$, the computation results in either *tt*, *ff* or the computation diverges. After obtaining a resulting value in $\{ff, tt\}$, the value is negated and then the computation produces the same effect as the negation in $\mathcal{P}$. In both cases, if the goal is *tt* the search

carries on by proving the next subgoal on the right. Otherwise, both systems backtrack, looking for another solution on the left, according to the depth-first strategy.

Finally, in the case of recursion, it is clear that $\mathcal{P}$ diverges iff $\mathcal{Q}^o$ diverges.

□

Proposition 4 *Let $\mathcal{P}^+$ be a Prolog program with only strict Horn clauses and $\mathcal{Q}^+$ be a program in* GLOBALLOG *formed by just replacing all instances of the '$\leftharpoondown$' symbol in $\mathcal{P}^+$ by the '$\leftarrow$' symbol. Then, for every query, $\mathcal{Q}^+$ gives the same answer as $\mathcal{P}^+$, except that whenever $\mathcal{P}^+$ answers $false$, $\mathcal{Q}^+$ answers unknown.*

[Proof:] Firstly, whenever a goal does not unify any clause in $\mathcal{P}^+$, the answer is $false$. On the other hand, whenever a goal does not unify any clause in $\mathcal{Q}^+$, the answer is *unknown*. In both cases, if the goal unifies a fact, the answer is *true* with the same values for the variables in the query, according to the unifications.

In both cases, if the goal unifies the head of a rule, both bodies are evaluated. In $\mathcal{P}^+$, whenever the body of a rule fails, its head becomes *ff* and the system backtracks. In $\mathcal{Q}^+$, whenever the body of a rule fails, its head becomes *uu* and the system backtracks. In both cases, whenever the body of a rule succeeds, its head becomes *tt*, with the same substitution set, according to the unifications.

Both unification algorithms provide the same response (including the same set of substitution) because there are neither negative goals nor negative heads of rules in either programs. These were the only cases where the Prolog unification algorithm was modified. Finally, in the case of recursion, $\mathcal{P}^+$ diverges iff $\mathcal{Q}^+$ diverges.

□

The proposition 4 shows that GLOBALLOG is sound with respect to Horn clauses.

For the propositions from 5 to 11, let $\mathcal{P}$ be a Prolog program and $\mathcal{Q}$ be a program in GLOBALLOG formed by just replacing all instances of the '$\leftharpoondown$' symbol in $\mathcal{P}$ by the '$\leftarrow$' symbol.

Proposition 5 *All rules in $\mathcal{Q}$ that contain some negative goal in their bodies either result in uu or diverge.*

[Proof:] Let **not** p be a negative subgoal. Then **not** p does not match any clause in $\mathcal{Q}$, because $\mathcal{Q}$ is obtained from $\mathcal{P}$ which does not contain negative rules and $\mathcal{Q}$ makes use of intention $\mathcal{I}_f$. In this case, **not** p results in uu and the $\mathcal{Q}$ system backtracks, looking for another solution. Therefore, no such rules in $\mathcal{Q}$ provide either solution or refutation, i.e. all rules in $\mathcal{Q}$ with some negative goal in their bodies either result in uu or diverge.

□

Proposition 6 *For any rule in $\mathcal{Q}$ no predicates on the right of the first negative goal are searched for.*

[Proof:] The same reasoning as the proof for proposition 5.

□

Proposition 7 *For every query, if $\mathcal{Q}$ diverges, $\mathcal{P}$ diverges.*

[Proof:] By definition, both systems adopt the same flow of control, except in one case: let **not** p_i be the first negative subgoal of a rule in both programs. In $\mathcal{Q}$, **not** p_i does not match any clause, because $\mathcal{Q}$ does not contain negative rules and $\mathcal{Q}$ makes use of intention $\mathcal{I}_f$. In this case, **not** p_i results in uu and the $\mathcal{Q}$ system backtracks. On the other hand, in $\mathcal{P}$, **not** p_i means that p_i is the subgoal to be proven. If p_i becomes tt, the flows of control are still the same. Thus, suppose that p_i becomes ff in $\mathcal{P}$. Then, the search continues after the predicate **not** p_i, i.e. either the predicate p_{i+1} or **not** p_{i+1}. This is the only situation where the flows of control separate from each other. However, $\mathcal{Q}$ has a pruned search tree in respect to $\mathcal{P}$ and hence, it is possible that $\mathcal{P}$ diverges

where $\mathcal{Q}$ simply answers *unknown* (by proposition 5). On the other hand, it is possible that $\mathcal{Q}$ diverges when $\mathcal{P}$ finds some solutions as a result under the CWA. Even in this case, after some more steps of inference, $\mathcal{P}$ backtracks to the point where the flows of control separated from one another and, after some more possible solutions, $\mathcal{P}$ diverges in the same way as $\mathcal{Q}$. Therefore, if $\mathcal{Q}$ diverges $\mathcal{P}$ diverges.

□

Proposition 8 *For every query, if $\mathcal{Q}$ answers true, $\mathcal{P}$ answers true.*

[Proof:] Proposition 4 has already showed that this holds for programs without negation. Now, I consider the case where negation occurs. But it follows from the proposition 5 that no rules with negation in their bodies in $\mathcal{Q}$ provide solutions. Therefore, if $\mathcal{Q}$ answers *true* $\mathcal{P}$ answers *true*.

□

Proposition 9 *$\mathcal{Q}$ never answers $false$.*

[Proof:] It follows directly from propositions 4 and 5.

□

The propositions 8 and 9 show that rules in $+\mathcal{R}[\leftarrow]$ are sound with respect to Prolog. The following proposition is related to the fact that Prolog always assumes a closed world while GLOBALLOG does not.

Proposition 10 *For every query, if $\mathcal{P}$ answers $false$, $\mathcal{Q}$ answers unknown.*

[Proof:] Proposition 4 already showed that it holds for programs without negation. Now, let **not** p be a negative sub-goal which occurs in the body of a rule in $\mathcal{R}[\leftarrow]$ of $\mathcal{P}$ and in the body of the corresponding rule in $\mathcal{R}[\leftarrow]$ of $\mathcal{Q}$. Then, in order for this $\mathcal{P}$ rule to become *ff*, it is necessary that some of its sub-goals fail. Without loss of generality, let **not** p be such a sub-goal. In this case, p had matched a rule in $\mathcal{R}[\leftarrow]$ of $\mathcal{P}$ whose query resulted in *tt*, causing **not** p be *ff*. In parallel to this and by proposition 5, a terminating rule in

$\mathcal{R}[\leftarrow]$ of $\mathcal{Q}$ with negation always results in uu. But $\mathcal{Q}$ does not diverge, by the contrapositive form of proposition 7. Therefore, for every query, if $\mathcal{P}$ answers $false$, $\mathcal{Q}$ answers $unknown$.

□

Proposition 11 *There exists some query to which $\mathcal{P}$ answers true and $\mathcal{Q}$ answers unknown.*

[Proof:] By presenting an example. Let $\mathcal{P}$ be a program with only one clause, namely, $p(a) \leftarrow$ **not** $q(a)$, and $\mathcal{Q}$ be the corresponding program $p(a) \leftarrow$ **not** $q(a)$: for the query $p(a)$, $\mathcal{P}$ answers $true$ and $\mathcal{Q}$ answers $unknown$.

□

The proposition 11 also follows from the fact that $\mathcal{P}$ always makes use of the CWA while $\mathcal{Q}$ does not. As $\mathcal{Q}$ offers only the abstract negation, **not** uu results in uu.

Proposition 12 *Let $\mathcal{P}$ be a Prolog program and $\mathcal{Q}^{-1}$ be a program in* GLOBALLOG *formed by replacing all instances of the '$\leftarrow$' symbol in $\mathcal{P}$ by the '$\leftarrow$' symbol, by replacing all occurrences of the predicates (both in the heads and in the bodies of the rules) in $\mathcal{P}$ by their negative forms, and by replacing all occurrences of negative predicates in $\mathcal{P}$ by their corresponding positive forms. Then, for every positive query q,*

a) $\mathcal{Q}^{-1}$ never answers true;

b) if $\mathcal{Q}^{-1}$ answers $false$, $\mathcal{P}$ answers true;

c) if $\mathcal{P}$ answers $false$, $\mathcal{Q}^{-1}$ answers unknown;

d) there exists some query to which $\mathcal{P}$ answers true and $\mathcal{Q}^{-1}$ answers unknown;

e) if $\mathcal{Q}^{-1}$ diverges, $\mathcal{P}$ diverges.

[Proof:] The proof is by applying the same reasoning as in the previous propositions obtaining symmetric results in $\mathcal{Q}$, and by applying symmetric intentions. Thus, sentence a follows from proposition 9, b from proposition 8, c from proposition 10, d from proposition 11 and e from proposition 7. □

9.4 Examples

Consider the classic non-flying bird example:

$fly(X) \leftarrow bird(X), \textbf{not}\ penguin(X).$

$\textbf{not}\ fly(Y) \leftarrow penguin(Y).$

$bird(tweety).$

$penguin(Z) \leftarrow bird(Z), polar(Z).$

To answer the query $fly(tweety)$, the system initially assumes the intention $\mathcal{I}_{tf}$.Then the system unifies the goal with the head of the first rule, binding X to $tweety$. Then, the system finds the subgoal $bird(tweety)$ which unifies the third clause according to the intention $\mathcal{I}_t$ (semantics 14). In the first rule, **not** $penguin(tweety)$ is the next subgoal to be explored with intention $\mathcal{I}_{tf}$ (semantics 15), as $penguin$ is a closed predicate. The subgoal unifies the fourth rule, binding Z to $tweety$. As the subgoal $bird(tweety)$ had already been proven, the next subgoal is $polar(tweety)$. To explore this subgoal, the system adopts the intention $\mathcal{I}_t$ because $polar$ is an open predicate by default (definition 14). The system does not unify any clause and, because of this, this subquery results in uu (semantics 13). The fourth body results in uu and hence the subquery *penguin(tweety)* results in ff (semantics 8 and 13), as the fourth rule adopts the CWA. Then, **not** *penguin(tweety)* results in tt (semantics 13), making the body of the first rule result in tt. Finally, the query results in *true*.

Now, suppose that a new clause, $polar(tweety)$. is asserted and the system places it at the end of the list of clauses. For the same query, $fly(tweety)$, the system now answers $false$.

In the example, a similar Prolog program would give the same answers for $fly(tweety)$, because the fourth clause alone ensures that only a polar bird is a penguin. However, for a query such as $fly(airplane)$, the above program answers *unknown* while Prolog would answer $false$ for a corresponding program.

As another example, suppose I know that Berne is the capital of Switzerland and that each country has just one capital. We can easily conclude that Zurich is **not** the capital of Switzerland. In GLOBALLOG, the programmer could write:

1 : $capital(berne, switzerland)$.

2 : **not** $capital(X, Y) \leftarrow capital(Z, Y),\ X \neq Z$.

According to the semantics of GLOBALLOG, a non-ground query, for example, $capital(bern, X)$. (note the Swiss spelling) would result in $X = unknown$, while a ground query such as $capital(zurich, switzerland)$. would definitely result in $false$. This goal would not unify the first rule but would unify the second one because the presence of the **not** operator does not fail during the unification process for $\mathcal{I}_{tf}$. In this case, X is bound to *zurich* and Y is bound to *switzerland*. Then, the system tries to prove the subgoal *capital(Z,switzerland)* and unifies the first rule, the fact *capital(berne,switzerland)*, binding Z to *berne*. Now the system evaluates the expression $X \neq Z$, which is tt as *zurich* is not *berne*. Since all the premises of the rule are tt, the system concludes that the head **not** *capital(zurich,switzerland)* is tt and, hence, that *capital(zurich,switzerland)*, the response of the query, is $false$.

9.4.1 A Global Extension

In this chapter, I am not formalizing the semantics of remote operations in GLOBALLOG. I give an example in this subsection that describes the informal semantics of a remote operation. Suppose that GLOBALLOG is implemented on the Web and that WWW directories contain GLOBALLOG programs. I can include global clauses in such programs, i.e. clauses that can be accessed by any other program anywhere on the Internet. By default, clauses are private. Therefore, I add the keyword **public** to indicate that the following list are the names of the clauses in the program that can be accessed by queries outside the program or, alternatively, that the transitive closure from those clauses can

migrate on the fly and be linked on demand. For example, in some host whose address is www.somewhere.on.earth , there can be the following program in a file *programming.txt*:

public $likes$.
$likes(X, \lambda\text{–}calculus) \ :- \ likes(X, fp)$.
$likes(X, logic) \ :- \ likes(X, \lambda\text{–}calculus)$.
$likes(X, philosophy) \ :- \ likes(X, logic)$.

The sequence of clauses above states publicly that every person who likes functional programming (probably) likes λ–$calculus$; that a person who likes λ–$calculus$ also likes logic, and so on. Now, a program at another site can have the following clauses:

$likes(X, Y) \ => \ 15s : www.somewhere.on.earth/programming.txt$.
$likes(X, fp) \ :- \ likes(X, haskell)$.
$likes(anna, haskell)$.

At the former site, an agent can make the query $likes(anna, philosophy)$. As the clause *likes* is public, an encrypted version of the file programming.txt can come from that site, www.somewhere.on.earth/programming.txt, and then be linked to the running program. After this, the execution continues by using clauses of that file and, after some more steps of computation, the given answer is "*yes*". As *likes* is a clause defined under CWA, the answer could be either *yes* or *no*. However, if the remote operation is not completed within 15 seconds, the answer is *unknown*. Although GLOBALLOG does not permit mixture of open and closed clauses in a program, it is possible to do this in other programs that are linked. This constraint is basically one structured programming technique that I impose if programmers adopt this language. Here, there are many details that are outside the scope of this chapter, but I hope this example provides a clear picture of the use of this language in such a global environment.

9.5 Consistency of a Knowledge Base

Due to the flexibility of the language, inconsistency might arise, e.g. the rules $\{p(a).\ \textbf{not}\ p(a).\}$. Over the last ten years, some proposals have been made to solve this problem[93], such as setting priorities, possibly in some implicit way, for all clauses. Some researchers have also defined four-valued logics in which the fourth value means "inconsistency" [30]. In the case of GLOBALLOG, it is easy to see that answer sets could be used to remove only inconsistent (i.e. both positive and negative) answers from the set. Therefore, inconsistency could also be interpreted by the system as *uu*. The literature on inconsistency in deductive databases and logic programs is large[268] but, naturally, there is little on abstract negation.

However, GLOBALLOG is a paraconsistent logic programming language, in the sense that dealing with inconsistency is left up to the programmer. The idea is that the language ought not only to represent inconsistency but also allow the program to give specific treatments to it. Mainly if the language is regarded as a subset of a hybrid language, where the fourth value is meaningless outside the scope of the rules, this makes the proposed approach particularly interesting.

In some modes of operation however, it is possible to check consistency easily. For instance, during the edition, whenever the user adds a new clause to the knowledge base, the system automatically checks consistency. Continuing using the previous example, suppose that while editing the rules, it is asserted that Brasilia is the capital of Brazil. The new clause, *capital(brasilia,brazil)*, is to be added to the knowledge base. However, before doing so, an implicit consultation may be automatically done by the system in order to guarantee consistency after the addition of this new fact. The insertion of a new fact may be allowed only if the result of the corresponding query is *uu*. If the result is *ff* the system may tell the user that the new fact could not be added to the knowledge base because, if it were added, it would lead to inconsistency. It is possible that the implicit consultation diverges. In this case, a time-out may

be configured in the editor in such a way that a warning be presented to the user.

In the current example, the predicate *capital(brasilia,brazil)* does not match the first rule but matches the second rule, binding X to *Brasilia* and Y to *Brazil*. Then the system starts to explore the body of the second rule, taking the first predicate *capital(Z,brazil)* as the current subgoal. Since the intention is $\mathcal{I}_t$, and there is no positive clause which matches this subgoal, the result of this sub-query is *uu*, causing failure in the second rule, and therefore, the answer of the main (implicit) query is *unknown*. The system now allows the insertion of the new fact:

1 : $capital(edinburgh, scotland)$.

2 : **not** $capital(X, Y) \leftarrow capital(Z, Y),\ X \neq Z$.

3 : $capital(brasilia, brazil)$.

Now, consider the addition of the fact $capital(rio, brazil)$. The system matches the second rule binding X to Rio and Y to Brazil. Then it makes the sub-query *capital(Z,brazil)* with the intention $\mathcal{I}_t$. The first two rules fail to match but the third rule matches the sub-query binding Z to Brasilia. Then control is returned to the body of the second rule and the system evaluates the next subgoal, the relational expression $X \neq Z$, now bound to $rio \neq brasilia$ which, in its turn, results in *tt*. As all clauses in the body of the second rule are *tt*, so is the head. Since the head is in the negative form, the implicit query results in *ff*. Thus, the system is able to interpret the straightforward rules and to prove that Rio is *not* the capital of Brazil, warning the user about the inconsistency.

In the last query, a similar result (from a slightly different list of rules) would also be given by GLP languages. However, a GLP program to answer queries like

$? - \ capital(vienna, austria)$.

in a realistic way would be much more difficult to write than in GLOBALLOG. Here, the system matches the second rule binding X to Vienna and Y to Austria. Then it makes the sub-query *capital(Z,austria)* with the intention $\mathcal{I}_t$, but nothing matches this sub-query. Thus, the second rule fails. The third rule does not match either and hence the system concludes that *capital(vienna,austria)* is *unknown*.

9.5.1 Dealing with Inconsistency

Because GLOBALLOG is a more expressive language, the programmer should take care to avoid inconsistency. However, the program itself can also catch contradictions easily and give the appropriate treatment. Suppose that the programmer wants to guarantee that no contradiction results from the predicate $p(X, Y)$. Then, they may rename p as pp and write a new and stronger version called $p(X, Y)$, which is not contradictory:

$$p(X, Y) \leftarrow pp(X, Y),\ isunknown(\textbf{not}\ pp(X, Y)).$$
$$\textbf{not}\ p(X, Y) \leftarrow \textbf{not}\ pp(X, Y),\ isunknown(pp(X, Y)).$$

Thus, pp is either uu, ff or tt, but not inconsistent. Note that if pp is inconsistent, p is uu. Inconsistency is actually a fourth truth value, but this leads to some practical problems when this value is part of the language, not only in terms of efficiency (any query would have intention $\mathcal{I}_{tf}$) but also in terms of integration with other programming paradigms, which was the original motivation for the design of GLOBALLOG. Furthermore, GLOBALLOG is more flexible for allowing the programmer to make use of the ability to treat contradictions. Another more subtle example:

$$p(a, X).$$
$$\textbf{not}\ p(X, b).$$

which holds for both $p(a, b)$ and **not** $p(a, b)$. In general, it is up to the programmer to avoid inconsistency. Sometimes, however, e.g. in planning systems, the notion of inconsistency may be useful. For example:

$might\_rain(tomorrow)$.
not $might\_rain(tomorrow)$.

which represents (with a slight abuse) the statements "it might rain tomorrow" and "it might not rain tomorrow". Both facts together may allow reasoning about possibilities, and once one of the above possibilities is no longer valid, the corresponding clause ought to be retracted from the knowledge base.

Although this model also works with predicates containing variables and constants, sometimes the result of a query is not as simple as those exemplified above. In general, the answer to a complete query results in an ordered set of *tuples*, either positive or negative (the exceptions), for the arguments in the query. For example, $\{p(a, b).\ p(b, c).$ **not** $p(c, a).\}$ and a complete query such as $?p(X, Y)$ binds $p(X, Y)$ to $\{p(a, b), p(b, c), \neg p(c, a)\}$ in this order. It means that $\{p(a, b), p(b, c)\}$ is the set of possible solutions for $p(X, Y)$ and that $\{p(c, a)\}$ is the set of possible solutions for **not** $p(X, Y)$.

9.6 Conclusion

I provide a semantics for GLOBALLOG which leads to the possibility of inconsistent programs. For agents, the ability to reason about inconsistency is essential. Thus, for any query, GLOBALLOG allows testing if there exist one proof and one refutation for the same goal, and trigger some action according to this.

In terms of practice, while GLOBALLOG provides only abstract negation, ELP provides two different negations which, although more flexible than GLOBALLOG negation, make the programming work harder. This is justified because while programmers define a predicate only once, they normally use it many

times in the program. On a global network, this simplification becomes even more significant. The negation is called "abstract" because, from the programmer's viewpoint, it does not matter whether the predicate of a negative query was defined as open or closed.

In comparison to Prolog, while NAF is a concept which depends on the CWA, GLOBALLOG adopts the idea that predicates may be defined as being either open or closed, which makes negation an independent concept of whether or not a closed world is assumed. GLOBALLOG adds a different inference symbol but I believe that this addition is more a matter of choice than a problem. Moreover, it is equally easy to define either open or closed clauses.

Like in ELP, the addition of the third value as representing what is *unknown* may increase the expressiveness of the language considerably, while allowing knowledge to be written in a more natural way. The ability to represent and reason despite a lack of knowledge is desirable in many applications. GLOBALLOG provides such representation by simply assuming *unknown* for non-existing clauses.

Because there are two forms of negation, ELP is technically more expressive than GLOBALLOG. However, there normally is only one meaning for negation in natural languages such as English[174] although there are different forms of negating a sentence. In other words, the expressiveness produced by two forms of negation is not necessary, since in ELP the choice of negation determines the assumption, either closed- or open world, which is not a normal situation in the real world. Furthermore, ELP possibly requires a higher level of intelligence.

Chapter 10

Conclusion

10.1 Another Approach

In this chapter, I summarize conclusions of each of the previous chapters and then draw my synthesis. I also add comments from section 10.7 on. However, before doing so, I would like to stress that, although this PhD thesis is, broadly speaking, in the area of philosophical foundations of computer science and programming languages, it is also closely connected to artificial intelligence. Moreover, the whole thesis can be viewed from the artificial intelligence standpoint, as AI may be at one level above the present discussion, i.e. the application level.

It is known that programming is one of the key subjects in computer science. However, if one observes the somewhat empirical characteristics in programming, the ideas contained here will become clearer. Yet, there seems to be nothing wrong in the way that programmers still work, and will probably continue doing. Furthermore, although one can prove that a given program is correct, there can be proofs of proofs of program correctness etc. Programs are either correct or not with respect to some representation of a relevant model from the real world. Therefore, although there are programming techniques including those suggested and imposed by programming languages, programming is a complex task that requires some basic synthetic skills.

10.2 Foundations of Computer Science

The operational semantics in chapter 4 as well as in part II make use of the notions of space and time, implicitly or explicitly. I introduced a space-time logic, a somewhat unified logic that includes the concepts of space, time, and uncertainty, as well as ***uu*** and inconsistency. I also defined a space-time deductive system for this logic. Both together represent and reason on real-world objects, capture mobility, and were helpful in many applications, and are part of the present PhD thesis. Both are also linked to AI and the author's background in AI.

Intuitively, both Turing machine and λ-calculus can be used to produce any natural number $f(x)$, given any input x. Additionally, for any input x, there exist an infinite number of sequences of steps that lead to the same result $f(x)$. On the other hand, composition is an important property of function application. However, unexpected effects produce unpredictability in the notion of computation, and while Turing machines have to deal with that problem, λ-calculus does not. Following this, I have demonstrated that the class of Turing machines is not isomorphic to the class of computable functions. Thus, from chapter 3, we obtain the following summary relatively relevant for the foundations of computing:

- Programs do not necessarily correspond to functions.
- There exists some computation of composition that does not correspond to any functional application.

As a consequence (and taking Turing machines as forming a true model of computation), it is also part of the present thesis the following: *The class of computations strictly contains the class of computable function applications.*

As regards code mobility, I demonstrated that traditional models of computation[155] are not as general as mobile agents with strong mobility.

A simple and more general model of computation, an extension of the while programming language, has been presented, by defining a virtual ma-

chine and a small set of operations that complements the traditional notion of computation.

I have introduced one operation, namely **SpreadOut**, which is not present in current models of computation, and formally included this operation in the notion of computation which I have presented.

Global computers need international laws, and because these laws depend on subjective factors, diversity of global and mobile-code systems should be encouraged. Ethics[279], a branch of philosophy, becomes a more important notion.

Philosophy is essential in the foundations of computer science. As an example, I have presented a philosophical view in computer science. However, there are others.

In chapter 5, a published paper in [105], I design a global computer. I introduce a framework for mobile agent system that can be regarded as *symmetrically secure*, in the sense that it equally protects visitors and hosts. In that model, I propose the centralization of the responsibility for the security in the whole system, and, in this way, such solutions become transparent to mobile agents. A more recent article which contains claims that are essentially the same as the author's PhD proposal in 1999 is [222]. In chapter 5, as well as in the already mentioned [105] above, I improved significantly those ideas in many points, and one can also observe that there is no virus in the system and no need for trusted third parties in applications, for instance. Designers of systems based on that model are already trusted third parties.

The present framework is based on a metaphor and describes typical situations which travelers face. Notions such as airports, arrivals, departures, passports and security are in the present framework. Because the metaphor is based on modern life, it is expected that its implementation will be easy, as well as the system will be easy to use. Except for details, the main problems were solved here, and the present framework suggests and imposes some programming language concepts and constructs.

10.3 Concepts for Programming Languages

This work on ***uu*** is based on the Łuckasievick and Kleene three-valued logics, as well as the Belnap four-valued logic. As it has become clear, ***uu*** is not partiality, for we can have both partial functions with infinite computation, as well as a sub-class of *total* functions, such as $\boldsymbol{uu} \rightarrow \boldsymbol{uu}$, $\boldsymbol{uu} \rightarrow 1, ..., \boldsymbol{uu} \rightarrow 1, \boldsymbol{uu} \rightarrow 2, ...$, for instance, using a sequence of two blank symbols from the Turing machine model as a previous convention for ***uu***.

Since a programming language designer adopts ***uu*** in their language, expressions and statements accommodate the presence of this value. For example, a version of the full conditional statement becomes **if-then-else-otherwise**, or using **ifnot** instead of the keyword **else**.

Local inefficiency is an issue of the present features. Assuming that the existence of mobile agent support systems almost entails code interpretation, the interpreter has to check the presence of ***uu*** whenever a variable is being requested in an evaluating expression. However, as hardware is getting faster and memory is getting larger in capacity, this is not considered a significant problem. Moreover, this problem can be compensated for by the fact that mobility and remote accesses are the bottlenecks in applications, and that variables can behave as a cache and operations can be lazy. This combination is encouraged by the discussed language concepts and constructs, in particular, lazy evaluation with timeouts, a larger repertoire of parallel operators and ***uu***.

Handlers and ***uu*** have been successfully experimented within Plain for a number of years.

The idea of a hybrid paradigm for programming allows programmers to feel free to choose their own way of working. Some definitions are better written in some particular paradigm whereas others are better written in a different paradigm.

At a more refined level, there can be two kinds of unknown states: the first one represents the initial lack of information with potential for later discovery, while the second kind represents the lack of information after having attempted

to discover its value. PLAIN distinguishes one kind from the other, to allow the inference machine to recognize variables whose values had already been requested.

The fact that global network connections are neither reliable nor efficient leads one to find ways to program despite this. Thus, as well as *uu*, I presented uncertainty as a programming language feature to cover this gap, among other purposes. The present scheme permits reasoning under lack of information, by both considering *unknown* as part of that uncertainty model in the variables used as premises, and in the resulting hypotheses. As a result, *uu* permits unification of this paradigm with others that include *uu*.

In chapter 9, I introduced GLOBALLOG, which makes *logic programming* and *mobile agents* compatible, and also provides *uu* as expected. While GLOBALLOG provides only abstract negation, extended logic programming provides two different negations. On the other hand, for any predicate, GLOBALLOG allows programming under either open- or closed-world assumption.

Whereas negation as failure is a concept which depends on the closed-world assumption in Prolog, this does not hold in GLOBALLOG. That is, in the latter language, predicates may be defined as being either open or closed as a matter of choice, and the negation has a semantics which is compatible with the definition.

Like in Gelfond and Lifschitz's extended logic programming, the addition of the third value as representing what is *unknown* increases the expressiveness of a Prolog-like language considerably, while allowing knowledge to be written in a more natural way. The ability to represent and reason under the lack of knowledge is desirable in many applications. GLOBALLOG provides such representation by only assuming *unknown* for non-existing clauses.

10.4 Further Work

It is interesting to open a book on future research directions such as [313], observe the time when the book was written and compare the contained ideas

with the current scientific context or with the state of the art in technology. Regarding computer science, it is particularly difficult to predict what will happen in the near future. In the present example, the book was written in 1996 and hardly contains comments on mobility. Nonetheless, I still share many of the discussed subjects without need to repeat them here.

Because this foundation work is broad and is placed at some proper level of generality, in this section, I state only a few ideas for possible further work. I also give some idea of what is not contained in the present PhD thesis dissertation.

- Philosophy of computer science as part of the foundations. This means that literature in this subject can continue to be written, not philosophy as a separate or satellite area in computer science, but instead as part of the theoretical work.

- The investigation of connections between what has been discussed in the present work and subjects that were not discussed in this thesis but are somewhat related, for instance, computational learning theory[182] and constraint programming[90, 205, 266], among many others.

- The logic introduced in chapter 2 for describing semantics in those chapters is only a simplification of the logic that I had conceived. Although the @-logic and some deduction for it have been developed, some more work is needed for checking their soundness, completeness and implementation.

- At the application level, a good amount of complementary work ought to be done regarding communication between agents in mobile-agents community, because in this PhD thesis, I left this issue almost untouched. This more detailed level of abstraction involves more technical issues, e.g. [29].

- Implementation of the ideas contained in chapter 5 as part of the Plain project. Although the **flyto** command is implemented in Plain, other

involved issues such as safety and security have not been implemented. Since those ideas are general and not limited to PLAIN, other virtual machines can implement the presented model.

- Further exploitation of the holistic view in computer science. More can be discussed and written. For instance, another volume might be conceived for investigating rôles and connections between *natural languages* and computing science from the present standpoint.

- Although the PLAIN compiler and virtual machine are efficient, much work can be done in terms of performance. One of the bottlenecks is in the communication between agents. Code mobility in global environments, in practice, requires special attention to efficiency, for, in general, there is much overhead.

The above list is far from complete. They are only examples of work to attempt to complete the subjects, issues or approaches that one deals directly with in the present PhD thesis.

10.5 Sciences and Deductive Logics

Based on observation, we can classify concepts related to computer science, such as methods, forms of inference, mental abilities and subjects of research, dividing them in two classes, i.e. synthetic and analytical, as explained here, in chapter 10. This classification, for being original, differs, for example, from the classification of Immanuel Kant[180]. Here, there is no formal nor precise definition of *synthetic* and *analytical*. Roughly speaking, analytical concepts are more or less motivated by exactness, whereas synthetic concepts are more or less motivated by generality. In this way, this classification itself is synthetic, fuzzy and incomplete. The analytical class is divided in two subclasses, namely, ω and π, whilst the synthetic class is divided in two other subclasses, namely, ϕ and ψ. Deductive logics is a key concept that belongs to the analytical class,

whereas induction and analogy are two of the key concepts that belong to the synthetic class.

Logics and exact sciences, in turn, as it is known, are partially based on deductions as well as observable and deduced facts, while such facts can be used as examples, which in turn can be used as proofs for existential propositions. On the other hand, propositions based on a finite number of cases are traditionally regarded as less relevant. As an example, it is known that belief, induction and analogy have not been accepted as valid methods of sciences[245], in particular, mathematics and exact sciences.

Nonetheless, synthetic concepts are significant in computing science and, among them, there are those empirical concepts, together with belief, induction, analogy and so on. A few forms of inference, such as the ability to weigh up possibilities, can be deeply studied in computing science, while the ability to weigh up possibilities is not traditionally regarded as a mathematical method.

Here it can be observed that computing science not only profits directly from logics and mathematics but that that science has a direct connection with the real world, while it belongs to the same world. Furthermore, the typical place where computing science has many notions of the synthetic side is in the interaction with the real world, including the interaction with our senses at work, and in the *applications* of the analytical notions. I see computing science or computer science as the organized knowledge about the world. In fact, while the logical, scientific and mathematical kind π in the foundations of computing science has been well studied since Babbage, no significant contribution was done in *philosophy of computing science*. Today, this is a new side of the foundations of computing science.

10.6 The Conceptual Diagram

This section and the following ones introduce one classification that can be used as a tool. The connection between this tool and the sample paradoxes that

I will present below is the following: there exist two large classes of concepts, analytic and synthetic. Roughly, the former is the concept of *smallness* while the latter is the concept of *greatness*. Thus, the former can be seen as smaller than the latter and, hence, if in some observation a subject from the latter is seen as belonging to the former, there can be some contradiction, which can lead to a possible refutation.

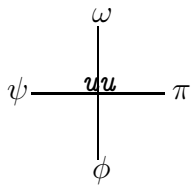
As one of the possible conclusions from this PhD thesis, I identify some of the usual concepts of computing science and place them in only four classes. Therefore, here I attempt to justify the present classification but knowing that it is a transcendental matter impossible to be either proven or refuted or both.

Most attempts to classify real world concepts necessarily lack precision, although attempts are often helpful. Because such classifications are empirical but takes a life time, here I present a classification from my standpoint, which illustrates the orthogonality of some paradigms and issues, skills, methods and approaches in the theoretical foundations. I illustrate how I identify the analytical and synthetic sides of computer science. Due to its empirical characteristic, our four-kind diagrams can be seen as simply a diagrammatic form of representation by some holistic view in computer science. Although fuzzy systems have been regarded as logics in their own right, I place these two notions as opponents in those diagrams, and this is because logics (which one can see as an analytical subject) is traditionally seen as the subject of the valid reasoning, hence, has some general meaning for all individuals. In contrast, fuzziness (which I see as a synthetic subject) typically requires individual truth threshold for each hypothesis, while fuzziness suggests, for every hypothesis, different truth threshold for different individuals. Furthermore, for I am placing logics and logic programming in the same analytical side in the referred to diagrams, and I am stating that any hypothesis that involves a synthetic subject cannot be generally proven, i.e. with a universally-quantified claim, the thesis of the validity of these diagrams, including my synthetic and empirical classification, is not provable or refutable. This itself seems to be in accordance with Gödel incompleteness theorem[276], as I shall explain briefly

later.

10.7 Synthesis in Programming

The work presented in the part II of this PhD thesis can be very briefly represented by the following diagram:



One of these hypotheses that I am presenting here is that the programming paradigms which I represent here in the Greek letters that were carefully chosen, π, ω, ϕ, ψ, for example, (functional-logic programming, equations, constraint programming, consistency can be placed in π), (imperative features including side-effects, states, communication between machines, interaction, small mobility can be placed in ω), (internet programming, strong mobility can be placed in ϕ), and (uncertainty, fuzzy systems, inconstancy can be placed in ψ), are orthogonal ways of seeing the world from the programming standpoint. Given this, PLAIN programming language takes into consideration this orthogonality, that is, different points of view.

Programming is a relatively complex task and, because of this, every programming language has features relating to all kinds. However, one can analyze and perceive predominance to one specific kind in most programming languages. The language Smalltalk, for example, can be associated to the ω kind; while Haskell could be the best example of the π kind. Concerning the ϕ kind, the mobile-code languages for global computers nowadays are still very emphatic on code mobility. More specifically, they have not solved all practical problems with respect to security, solved in chapter 5. There are other issues, e.g. regarding differences of cultures, that have not been deeply investigated.

In fact, the ϕ kind is too recent from the programming perspective and one misses comparisons on scientific basis. PLAIN tries to achieve a suitable balance between these four kinds. So, in the next subsections, I shall discuss the four kinds and make comparisons between them.

10.7.1 Concepts and Constructs of Kind π

This kind of constructs is not easy to describe because it seems that all programming languages are in here. Formalism, discipline, precision and details are extremely important, and they are key motivations here. Most logic programming languages and functional programming languages are typically in this kind, and because strong typing goes together with discipline and methodology, these languages are not of the same point of view as untyped programming languages, which are in ω, for instance. Objects are concrete and well formed here. Although most of object-oriented programming languages are imperative, inheritance is another example of constructs of kind π. There are exceptions such as C++, which in turn has a strong influence from C. Here, sub-classing can be sub-typing in terms of OOP. The senses of order and pureness are important here. And because of the importance of order and clarity in programming, I regard the kind π as compulsory.

For this synthesis, because there has been much work on logic programming and functional programming, I have preferred not to concentrate on languages of this kind. Instead, in the following subsections, I use them only for comparison.

10.7.2 Concepts and Constructs of Kind ω

Here flexibility and expressiveness are keywords. Others are interaction, communication, states, imperative features, and efficiency of time, for instance.

It is interesting to note that, although C programs do not tend to be robust, C is among the most successful programming languages and its historical importance is undeniable. Java is one of its successors.

The most traditional languages used in knowledge-based systems are Lisp and Prolog, which are untyped programming languages, although based on functions and logics, respectively, which are in kind π. In both languages, *list* is a basic data structure that provides flexibility.

Although PLAIN is a strongly typed language, because its designer and other users require flexibility, PLAIN also permits heterogeneous lists in programs. The head of a list is accessed by using the name of the type as the head function call. It has been a nice experience to program with heterogeneous lists, and programmers should be pleasant to program. The following is an example interpreting [ ] as an empty list:

$$\begin{aligned} &\mathit{public\ list\ reverse}([\,]) \;=\; [\,], \\ &\qquad\qquad \mathit{reverse}(\mathit{li}) \;=\; \mathit{reverse}(\mathit{tail}(\mathit{li})) + \mathit{head}(\mathit{li}); \end{aligned}$$

In the above example, the *public* keyword states that the function *reverse* can be used by another agent. The + operator here concatenates two lists. The *head* function gives a list with only the first element of its argument. Therefore, the above function reverses the order of the elements of a list regardless of its type.

For this kind of list, the programmer can infer types[151] or, alternatively, instead of the word *head*, use the type identifier to get its first element. In this way, at the point one writes the type identifier in the code, the type checking is made at runtime. In this way, that typical function definition becomes here as follows:

$$\begin{aligned} &\mathit{public\ int\ add}([\,]) \;=\; 0, \\ &\qquad\qquad \mathit{add}(\mathit{li}) \;=\; \mathit{add}(\mathit{tail}(\mathit{li})) + \mathit{int}(\mathit{li}); \end{aligned}$$

Thus, the above function computes the total value of the elements of a list. Such function definitions, together, illustrate that untyped lists and strong typing can coexist at the language level. A question is how to reconcile the flexibility of programming with the methodologies of pure languages, of kind π.

Another question is whether type inference[219, 267] is better or worse than otherwise with respect to programming methodology. For instance, we can see that type inference makes *programmers* infer types. Programmers write a piece of code only once and, later, will look at it many times. Therefore, programmers normally infer types in languages like ML. This can be seen as a contrast between kinds ω and π, for the former is predominantly based on assertions and *facts* (type declarations) whereas the latter is predominantly based on *deduction* (type inference). Therefore, this pair of different points of view, ω and π, often indicates the issue of facts (e.g. predicate declaration) and deduction under either open- or closed-world assumption.

10.7.3 Concepts and Constructs of Kind ψ

Here, uncertainty, inconsistency and relativity are some of the keywords for this kind of constructs, ψ. We can include here any constructs for knowledge representation that have vagueness. One approach is to avoid being impersonal and exact, in the programming paradigm, because of the consistency of this perspective. If so, surprisingly, probability theories may not enter this class, in spite of their undeniable importance in science, in general. In terms of programming and knowledge representation, exact probabilities are not easy to be found in the real world, let alone being represented using some formal language. We know that probabilities are based on assumptions concerning whether events are related to each other. But one underlying subjective issue is "when can we consider that two events are independent from each other?". There seems to be no objective or precise answer for this question. In fact, depending on the philosophical point of view, we have totally different answers. For programming with probabilities, there is one problem of computational complexity[63] and another of representing the web of events, which is a difficult one. A similar web is represented using ad-hoc models, where the basic difference is that probability is traditionally mathematics while ad-hoc models are not so recognized. This means that probabilities require too precise values

for representing imprecise knowledge about the real world.

Here, pictures, images, films, and diagrams can represent the imprecision and subjectivity of the real world in a more suitable way. Some proposed programming models for uncertainty extends MYCIN to predict the evaluation of hypotheses by introducing two variables, *the least* and the *most*.

If one user wants an agent to represent him or her, he or she wants to personalize the behavior of his or her agent if the task is not simple. Due to the huge number of different certainty factors, the community needs suitable languages to program agents.

10.7.4 Concepts and Constructs of Kind ϕ

In ϕ, among other notions, as humans, we learn by induction and analogy, and also use metaphors to communicate. We need to make comparisons and need broad and abstract views. And since we have had a broad or general perspective, we are able to start solving hard problems in an easy way, and then to investigate them deeply, top down. Aims are necessary, but they should be set only after having such a view and before solving the problem in question.

We humans cannot investigate some complex subject deeply without an initial broad view. Goal-driven programming, i.e. where programmers set goals and the underlying system does the job, is motivated by ϕ. In this sense, Prolog and relational languages have this feature in ϕ. As another example, inductive logic programming is a combination between π and ϕ because, while logical and formal propositions are there, induction and learning by experience are in this class.

Because of its synthetic nature, this class of notions is difficult to implement on a computer, but I am advancing in this direction. Mobile agents, for instance, are applications whose motivations fit in this kind ϕ since mobile agents are flexible and free enough to gain experience abroad. In chapter 5, I informally proposed a kind of framework for mobile agent technologies

discussing problems and their solutions, some of them open problems.

As regards "mobility", the difference between constructs of kind ω and ϕ is that, in the latter, differences of cultural and religious backgrounds are probably involved, while in the former, mobility essentially is related to changes of states, concurrence, unexpected effects on Turing machines computations and hardware circuits: everything happens in the same machine in ω. While ω is driven by issues such as flexibility, communication and curiosity, here truth is a key motivation. In comparison to π, ϕ is not mainly concerned with syntax and details but instead broader concepts, such as paradigms, and also semantics.

In comparison to functional languages in π that adopt type inference, for instance, from the ϕ standpoint, I observe that type inference makes programmers infer types. To solve this kind of question we would need experimental and empirical research, not proofs. PLAIN is experimental since it was not conceived for commercial purposes. This is validated by twenty years of experience in programming and compilers. The Unix system and C are two successful examples of what one or two professionals can do, in contrast with PL/I, which in turn was designed by some greater number of people. In this way, it can be better to rely upon one's own experience than to make experimental research among non-experienced programmers, while the result of one's work may happen months or years later.

The fewer the number of programmers the more independence a designer has for trying different constructs. The right moment to release a language to others is relevant, and I am nearly at this point. Yet, by using PLAIN, the author has experienced important insights concerning languages and paradigms. One of the key ideas in PLAIN is its hybrid and well-balanced paradigm, i.e., it tries to combine well-balanced features from different points of view, while accepting that divergences among people and, hence, researchers are natural. Although PLAIN is a large language in comparison to functional languages, it is meant to be concise and relatively smaller, as it provides common constructs. Here rests the difference between a multiple and a hybrid paradigm, at least in

comparison to a naïve approach. I have programmed and experimented with different kinds of constructs. The implementation should be sophisticated enough to hide complexity. On the other hand, a good hybrid paradigm is not hard to learn, since learning can be incremental, and makes programming much easier, since I have taken into consideration different kinds of motivations. Thus, if a person likes functional programming, it is easy to program in PLAIN, and if a person likes object-oriented programming, it should be equally easy to program in the same language. However, although the PLAIN philosophy is to be a hybrid language, the concept of "pure function", for instance, (or simply function. It is the opposite to imperative function) is present in the language, they are declarative, never making use of global objects, and this is guaranteed at compilation time. However, functions can be applied from any code of any paradigm. In this sense, PLAIN is different from PL/I where the key motivation was to bring together features for both engineering applications and commercial applications in *the same* paradigm. I believe[117] that, after some time, programmers naturally find that applications suggest the paradigm to use, and that they can benefit from a hybrid language after some time using a particular paradigm. However, at the present point it is an open issue that deserves future validation and perhaps even further work.

At the practical level, issues such as robustness and security are keywords in the mobile agents field of research. Even in programming, in society, laws have to be established, which stresses the relevance of philosophical issues in programming. There are other keywords, such as dynamic linking (which I did not adopt but may be necessary for efficiency), naming, and global-scope identifiers, i.e. "global" not in the old sense of global variables. The subjective aspects of such issues suggest new or open problems.

10.8 Synthesis in Knowledge Representation and Reasoning

For AI researchers, although the present classification is not complete (for instance, vision is important for AI and is a synthetic notion related to ψ up to some extent), the four-kind diagram can be seen from the knowledge/belief representation and reasoning/inference standpoint as follows:

- π: perception, precision, specialization, functional programming (traditionally LISP programs), logic programming (traditionally Prolog) and inheritance in class- or frame-based systems. Deductive rules, consistency, deductive reasoning and search. Analysis, complexity and efficiency. Closed-world assumption and negation as failure. Learning by deduction.

- ω: reasoning, in particular non-monotonic, natural negation, ambiguity and redundancy, knowledge, interaction, diversity, curiosity and learning by acquiring facts.

- ψ: feeling, fuzzy logics, uncertainty, partial information, incompleteness, subjectivity, inconsistency and belief.

- ϕ: intuition, perspective, monotonicity, absolute truth, speculation, axiomatization. Semantic web, inductive reasoning, generalization, analogy and metaphors. Objectivity as opposed to subjectivity. Open-world assumption, neural networks, inductive logic programming, broad view, synthesis and common sense. Learning by induction.

In terms of AI, this four classes can represent four types of intelligence. I observe that although the above concepts are interesting for AI, the classification is still the same as for programming languages and for computing science, which will be presented in the next section. While ω and π are analytical kinds, ϕ and ψ are synthetic ones. For instance, a country could well use

many deductive rules of logic programming to decide whether a person may be regarded as a citizen of that country or not. However, deductive logics is not very appropriate for, in the airport, deciding whether a person from abroad may enter the home country or whether the person will go back to the place from where he or she came, because the number of rules from the real world is practically impossible to count. Therefore, as programs need synthetic thinking, programming languages should also provide synthetic tools, in particular for mobile agents. Continuing, while ω and ψ are closely related to internal judgments (by reasoning and feeling, respectively), π and ϕ work like input devices (by five-sense perception and intuition, respectively). This classification has strong influence from Carl Jung psychological types[176], but the observation with respect to computer science is almost entirely based on my own experience in both areas as well as my empirical observation during my PhD studies. Perception above, in the π kind, corresponds to a refinement of what Jung called *sensation*. According to him, the four psychological functions are: thinking, feeling, sensation and intuition. More generally, he called the first two *rational functions* while he called the last two *irrational functions*. These four functions are somewhat similar to what I called ω, ψ, π, ϕ, respectively. Philosophically, Jung's work itself had some influence from Immanuel Kant.

In terms of humanity, the essence of ψ is seen in the romanticism. An atheistic view of ψ is exposed in [263].

Regarding mobility, I can classify its scopes as (individual,ω), (social,π), (geographic,ϕ) and (universal,ψ). With respect to languages, natural and artificial, I can still observe differences among the main concepts: (vocabulary, ω), (grammar and syntax, π), (semantics, ϕ) and (pragmatics, ψ), for instance.

10.9 Synthesis in Foundations of CS

Regarding the two-axis diagram, there probably is the same correspondence in computer science at a very high level. Respectively, psychology (ψ), sciences related to the machine, engineering and physical characteristics in general (ω),

mathematics (π) and philosophy (ϕ) form a four-leg table that can support computer science. Typical questions in science can also be placed under this classification e.g. *what*, *where*, *when* (factual and flat information in general, ω); *how* (π); *why* (ϕ); *for what* and *for whom* (ψ). *Who* can be either in ω or ϕ. These four legs are not sharp, too clear, exact or mathematical classification, e.g. in philosophy, the Platonist view can be placed in ϕ while the constructivist view can be placed in π. On the other hand, in logics, theories of truth, interpretations and model theory can be placed in ϕ while proof theory can be placed in π. These four kinds are somewhat fuzzy[189], relative, incomplete and personal. However, the hybrid semantics of that four-kind diagram contrasts with solely fuzzy views of the universe, e.g. [189].

I have presented two different philosophical views in computer science, that can be summarized in the following way, depending on the adopted meaning for "executable code".

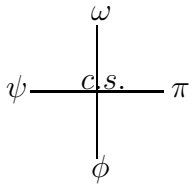
- Traditional view: computation has purely mathematical semantics. There are modal notions, such as mobile computation. Executable code is local to the machine, and a machine is only a physical notion. Correspondence between semantics[231]. Domain theory [3, 15, 150], denotational semantics[14, 286, 287]. Curry-Howard isomorphism[143], rewriting systems[23, 187], category theory[193, 197, 312], functions, logics in computer science and so forth. Functional programming, input and output operations as monads.

- An alternative philosophical view: computation by machines is a unique physical notion. Executable code is mobile, and a machine can be an interpreter. The equivalence between operational and denotational semantics does not necessarily hold from this point of view. In particular, although denotational semantics is still useful, the operational semantics is the most suitable semantics to capture this view of computation if the notions of space and time are part of the semantics. In the real world, uncertainty- or probability-based computation is performed. Computa-

tion can be incomplete. Space-time logical calculi. Input and output operations as physical interaction (by side-effect) in the real world.

Notice that these views are not necessarily holistic, although I use the second view to illustrate "computation in the real world" in this thesis.

I can refer to computation in the latter philosophical view as not only computation carried out by machines but also *computation by humans*. For example, interaction for a machine is roughly equivalent to human five-sense perception. However, as will become clearer in this section, while there is some (rough) equivalence between human thinking and machine computation, whether one can reproduce or only simulate human computation in a machine is one more philosophical issue in the foundations of computing science. Continuing the discussion with a diagram:



While I identify deductive logics with the π kind, the present piece of work aims at linking this kind with the others. In comparison with the ψ kind in terms of Bayesian or uncertainty work, in [233], there is a criticism on what psychological experiments on human inference have been achieving, where the author aims at showing that we must regard human probability judgments and decision making to describe human deductive ability correctly.

There is the ϕ kind since we all also learn by induction and analogy, as well as we make use of metaphors to communicate, mainly when the concepts are abstract, and we have only words. Possibly, that is why prophets and visionaries often use metaphors in their speech. And because we humans make use of induction, analogies and also use metaphors in our speech[53], a number of good examples are both didactically relevant and essential scientific

method, as computer science, in the broadest sense, is not totally mathematical. While in ω I talk about knowledge representation, in ψ, one regards with *belief* representation. Although the forms of representation can be the same, these two concepts are different. In ϕ, broad view and intuition are key words that normally lead to prediction[184].

In logics, while some of the main motivation in π would be proof and syntax, some of the main motivation in ϕ would be (perhaps informal) models and probably non-mathematical semantics. Since Gödel incompleteness theorem, it is known that proof (π), truth (ϕ) and incompleteness or inconsistency (ψ) are not equivalent notions in both the mathematical world and the real world. While, in π, *true* and *false* would be mere symbols that are manipulated according to some well-formed syntax, here in ϕ models would be closely connected to the real world. While in π a key motivation is to find differences between apparently similar objects, here in ϕ, a key motivation consists in finding similarities between objects apparently very different from each other. And while π can correspond to Aristotle and deductive reasoning, ϕ can correspond to Socrates and inductive reasoning[262]. Thus, we all need a broad view of the world to make good analogies: the broader the view, the better the analogy. Metaphorically speaking, if I place elements of a set in order and I want to connect elements that share a property different from the chosen order, I have to see distant objects, perhaps at once, to make comparisons. Following this reasoning, the notion of perspective and, therefore, distances and mobility, can be obtained. At a somewhat refined level, induction is an orthogonal notion in the sense that it depends on experience and, hence, time.

While in π the related quantifier is existential, here in ϕ the related quantifier is universal. In terms of games, while π may summarize the skills played by Eloisa ($\exists$), ϕ can represent the skills played by Abelardo ($\forall$). From the ϕ kind, in the chapter 1, I showed, by presenting two paradoxes, that human sciences and philosophy are in the foundations of computing science. In the chapter 4, I presented a model of computation based on space, time and mobility, as well as gave evidence as motivation and for showing that the foundations of com-

puter science are based on philosophical views, whether the theory is formal or informal.

I also informally presented, in chapter 5, a framework for global computing with strong mobility. Security has been recognized as one of the greatest problems if not the greatest, and whose solutions are difficult. This and other practical problems lead the interest of our community to ethics and other branches of philosophy. As an example, I centralize security in my solution in chapter 5 and, by hiding the implementation from users I reasonably provide security for all of them. One can observe that, from the mobile agents points of view, a MAS is omnipotent, omnipresent and omniscient. My proposal, at least in its broadest sense, rests on diversity of global computers, each of which proposing its corresponding view. As part of the approach, I followed analogy and found the solution in chapter 5 straightforward.

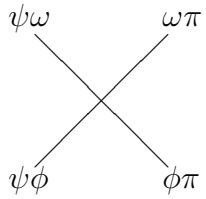
Many aspects of computer science have always had interesting philosophical components. As part of future work, one may want to define those bases explicitly, in a way that they can be identified by undergraduate students all over the world. As well as philosophy, there are other human sciences which are important for the foundations of computer science. In chapter 4, I also mentioned psychology, including the unconscious, as part of the ψ kind. Artificial intelligence is already connected or regarded as part of computer science or informatics. Much interesting work has been done in those areas and more is needed.

With respect to the ψ kind, for being a transcendental notion, while π is concerned with granularity and what can be perceived and countable, ψ, like a set of water molecules, although a countable set, suggests what is uncountable and is beyond one's five senses. Like in modern physics, ψ suggests hypotheses and observable models, but not what can be directly perceived. Thus, belief is an important issue here. ψ, as I lexically suggest, is a place where psychology has something to contribute to computer science. By comparing and stating differences between humans and machines, we humans also learn the limits of what can be computed by machines, an approach complementary to

complexity. As an example, if someone wants to build some application for identifying e-mail messages that are important or interesting, he or she has to have a very personal agent computation, probably based on some analogical measure, not only deductive systems such as natural deduction and Prolog, in which, for each consequent all pieces of its (conjunctive) premise must necessarily be tt. Thus, the notion of "importance" is not only personal but it also naturally requires some analogical interval. I can go further and state that it is not enough to have intelligent machines since we want them to be useful and relate them with human beings and match the real world. We may want machines to behave like humans up to some extent, and being able to show intelligence as natural as possible, but not up to the point of deceiving others. This is one of the greatest challenges in the future.

Returning to the π-ψ axis, I can state that ψ suggests computation in the set of complex numbers, while the π kind, although $\pi = 3.141592...$ in arithmetics, is mainly concerned with the set of integers. As a synthesis, Gödel incompleteness theorem, at another level of abstraction, is a piece of work which involves five different notions that I have identified in the above conceptual diagram: (π: proofs and consistency), (ϕ: truth and validity), (ψ: inconsistency), and finally (uu: incompleteness). The piece of work in chapter 3, on the other hand, leads me to place (π: proofs) and (ω: unexpected effects on Turing machines computations) in different classes of the diagram. At a different level of abstraction, due to the transcendental nature of ψ, $\boldsymbol{uu}$ is in the ψ class.

I can divide the same diagram in two diagonals as follows:



forming two divisions, each of which one can be referred to as *side*. I could refer

to ϕ and ψ as "synthetic side of the foundations of computer science" while ω and π would be "analytical side of the foundations of computer science". Therefore, this representation of those foundations of computer science would include the whole picture. The $\phi\pi$ side can be referred to as inductive and deductive reasonings, as well as knowledge/belief acquisition, whereas the $\psi\omega$ side can be referred to as knowledge/belief representation.

With respect to synthesis and analysis, which divide the diagram in two halves, $\psi\phi$ represents the purely subjective side of the foundations of computer science, while $\omega\pi$ represents the objective side of the same foundations. I simplify the language and refer to them as synthetic and analytical sides, respectively, having in mind that they are *not* independent from each other. The former is predominantly inductive while the latter is essentially deductive. In computer science, the analytical and concrete side has been well developed while the importance and even the presence of the synthetic side has been little perceived. This contrast is probably not simply due to the fact that deductions are very easy to implement in a machine. Indeed, while formal proofs are appropriate as a scientific method for the analytical side, *concerning the synthetic side, formal proofs of general claims do not work*. Frege[304] tried to prove that arithmetic was analytic[153]. As an example, tableau calculi are also called the method of *analytic tableaux*. From [36]: *Any calculus that starts with the formula to be proven, reducing it until some termination criterion is satisfied, is called* analytic*. In contrast, a* synthetic *calculus derives the formula to be proven from axioms*. In contrast, I am referring to *analysis* and *synthesis* as opposite mental processes. In fact, not only deductive logics, in its pure sense, is normally analytical at a higher level of abstraction, for Babbage himself called his work *analytical engine*, and so forth. These pieces of evidence are too informal and synthetic to be used as a proof of the validity of this classification. In contrast with logics, there are branches of mathematics that seem to be synthetic.

Because of this contrast between synthesis and analysis and because logics is closely linked to universality, for some logicians, inference based on uncer-

tainty might not be considered as part of logics, at least from the point of view of deduction and the universality of the classical logic. In the present work, I let them be opposites to form the same axis, as, in a sense, they complement each other because their natures are essentially inductive and deductive; synthetic and analytical; on open and closed word; subjective, and exact and with valid values for all; fuzzy and precise etc, respectively. Interestingly enough, logics may be seen as a branch of philosophy, which I classify as synthetic. It is true that nowadays, as there are many logics, to propose a logic one has to have a broader view than he or she would need to simply use the same logic.

I can also see the above diagram from the standpoint of the other diagonal and refer to $\phi\pi$ division as *perception* and *learning* while $\psi\omega$ division may be referred to as *reasoning* and *thought*. Perception here means not only by using five senses (programmers and logicians have to pay attention to forms and details) but also intuition (philosophers and researchers need to make use of their own insights and see the world abstractly, from a broad perspective).

An interesting explanation, extracted from [101], as regards intuitionistic logic: *"What distinguishes the intuitionists is the degree to which they claim the precedence of intuition over logical systems and the extent to which they feel their notions have been misunderstood by classically trained mathematicians and logicians"*. The idea of precedence of intuition over logical systems[235] is in accordance with the idea of trying to view the whole picture without details before starting concentrating on the latter, in a top-down fashion, for those who like software engineering. As known, intuitionism is a philosophical view[92].

As well as perception and intuition[202], in the above diagram, by completing the symmetry, thoughts are not only based on reason (to deduce hypothesis) but also based subjectively on feeling. Feeling is very personal. Even if the researchers decide not to implement such subjects as part of limitations of what computers can do, those subjects are still essential to the foundations of computer science. Nevertheless, it is easy to understand why this ψ kind has not been exploited in computer science, and an answer is that this is also a

natural consequence that computers have become increasingly complex. Deep Blue has already beaten the Grandmaster Kasparov in chess, but computer scientists took some time until the machine was able to beat him, a challenge which could be regarded as relatively simple, besides its computational complexity.

So far, there has not been any implementation running in the computer that could be regarded as representing feeling, at least in a universal way. Nonetheless, the notion of agent was introduced to represent people in their transactions in the daily life. The diagonals of the diagram also represent the idea that, among many other skills, human beings learn by communication and facts, perception and deduction, intuition and induction, feeling and belief. The diagram is not complete with respect to this either, for instance, motivation and pain are outside the present classification.

The computer science and AI communities have been discussing the differences between humans and machines in terms of the meaning of thoughts. For instance, in [36], the same applies to discovering proofs and deduction. Even with a set of wffs, deduction might be difficult for a machine due to the possibility of combinatorial explosion. Here one has two philosophical views supporting the answers for the question about whether we humans are machines, or whether we humans are much more than this. In the present PhD thesis, I presented, as examples, some skills that form the synthetic side in the diagram. Although specific answers for such questions are outside the scope of the present PhD thesis, one of the main results from my observations is that, *if one wants to adopt the view which equates any person with any machine, first it is a good idea to study intuition, feeling, analogy, induction, belief and other subjects which are parts of the human beings, and thus, philosophy and other human sciences.*

Finally, technology introduces useful insights into theoretical computer science. The introduction of mobile agent technologies has led the community to want agents to be autonomous and flexible to represent users. For simple tasks, there is relatively little to add to what we humans already know. But

users want to use complex application programs.

To truly represent people in our complex society, agents *have to* simulate subjective thought too, we humans want them to behave based on our personal tastes and personal opinions, for instance. Therefore, knowledge or belief representation can also be seen as a programming paradigm, not only as a subject in AI. The aim of introducing subjectivity in programming languages is difficult and ambitious. The author used to work on an expert system for diagnosis of heart diseases and was intrigued when observing that the users, experts, did not want to attach real numbers (in fact, floating-point numbers) to our rules as part of the certainty measure. Instead, I made use of words, carefully chosen to represent those real numbers. From that experience, the rôle of subjectivity in programming could be felt: "What do such subjective words mean?". There seems to be some common agreement which, in those cases, is more important and easier to understand in our every-day life than numbers. More recently, intelligent agents have been conceived to represent users and, for some reason, some of us also want them to be mobile. This scenario leads the present thesis to conclude that other more subjective, less exact sciences are also essential in the foundations of computer science.

In the following sections, a couple of examples of paradoxes in science and logics are presented. Each of them proves that philosophy and psychology are in the foundations of computing science.

10.9.1 A Paradox Example - Analogy

This section proves by contradiction that mathematical logics is not the only foundation of computing science.

An example of this is in the content of deductive proofs. Regardless of how mathematical and formal the problem be, what makes some given proof support some problem **X** and not some different problem **Y**, is the *analogy* which is made between a given problem and the representation of its relevant context, whether this representation is formal or not. Applied proofs are

regarded as correct only if

1. The deduction is logically valid, i.e. in accordance with the deductive rules of the used logic.

2. The representation of the problem is correct and, in particular, complete where the variable is universally quantified.

The second item describes a basic analogy, which in its turn is essentially an informal concept, somewhat personal. On the other hand, this concept is fundamental in the context of applied logics and, hence, of the computability theory, and so on. As a metaphor (analogy), this resembles the incompleteness theorem, because the fact that analogy is not part of the mathematical methods implies that those methods are not sufficient in computing science. That is, analogy supports mathematics while the former is not supported by the latter. Therefore, mathematics is not sufficient in this rigorous sense, while rigor is a kind of ideal in mathematics. As a consequence of this, philosophy, psychology and other human studies which deal with analogy[53], are new components in the foundations of computing science, whereas philosophy and psychology support mathematics and computing science. In this way, mathematics in computing science is a fundamental tool for letting the involved concepts be precise and clear.

10.9.2 Another Paradox Example - Induction

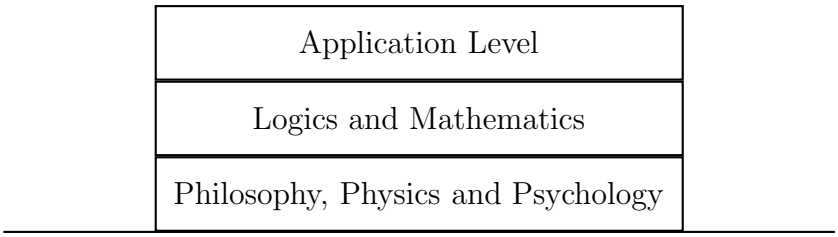
The previous example applies to the content of proofs. In contrast, the present example applies to the human inference forming generalizations, which one refers to as induction[139, 140]. As it is well known, both deductive logics and sciences, in a rigorous sense, reject not only analogy but also induction. By the way, both analogy and induction are referred to by philosophy as *irrational*. The fact is that, in rigorous way, both science and deductive logics reject the inductive method, in particular because its validity is not universally acceptable. Nonetheless, science itself works in the presence of panels (i.e.

committees) in Master or PhD courses/research, in selections for professorship, and considerations for publications. In the end, however mathematical the particular subject is, it depends on the present induction. For the same level of reputation, even taking it as the minimum, we humans tend to *believe* that the more the number of examiners, the more accurate the result is. However, this exemplifies that logics and science work by using some method rejected by them, which constitutes a paradox at the practical level. Establishments can rely upon the intellectual attributes of the examiners, but *belief* [282], as a synthetic concept, cannot be supported by logics, mathematics nor sciences in some rigorous sense.

The present author observes from both examples, above, that although the synthetic and human aspects are not entirely supported by logics, the latter is always supported by human beings in a synthetic way. Note that logics is traditionally philosophy. This shows the hierarchy of the large subjects inside the foundations of computing science, for philosophy and psychology can support the others. From this, different *theories* of computing science with synthetic notions can exist instead of a unique, mathematical and analytical theory. The novel area is called *philosophy of computing science*.

As well as containing a list of the results of the previous chapters, this chapter is a synthetic discussion, on philosophy of computing science, presenting some of the links between concepts, trying to describe a semantic network, which in turn is in accordance with the presented philosophical view.

Logics, in a rigorous sense, is not the only foundation for philosophy, psychology, computing science etc. More than this, it was shown that there can exist the following hierarchy of subjects in computing science:



Computing science has the interesting characteristic of being both exact and philosophy for being related to the reality. For these two facts, methods rejected by science that are applied in the daily life should be considered. In the above example of committee, in order to make the applied method be consistently scientific, one should consider not only the object, the criteria and grades, but also other variables of the real world, such as the names of the examiners, which in turn ought to be public. In this case, the scientific knowledge would necessarily be referred to as something broader.

As already stated, if one observes the somewhat empirical characteristics in programming, the ideas contained here will become clearer. Yet, there seems to be nothing wrong in the way that programmers still work, and will probably continue doing. Furthermore, although one can prove that a given program is correct, there can be proofs of proofs of program correctness etc. Programs are either correct or not with respect to some representation of a relevant model from the real world. Therefore, although there are programming techniques including those suggested and imposed by programming languages, programming is a complex task that requires some very basic synthetic skills.

My classification is itself synthetic and, as such, can be neither proved nor refuted. However, exceptions exist. Other synthetic subjects can be existentially proved, and some such subjects can be equally refuted. The classification here is essentially based on intuition, analogy and induction (because of its inductive nature, I had to present sample applications to programming languages and knowledge representation, not only to the foundations of computing science), as well as many observations on the real world. However, such

a classification is not scientific, only philosophic.

Finally, the two-axis diagram reflects only some particular philosophical view. Because of this, I do not expect that it can be used as a universal tool, nor accepted by the whole community as valid. However, sections 10.9.1 and 10.9.2, in a sense, show that the diagram somehow can be useful, by taking two concepts classified in ϕ, and because, according to the corresponding classification, logics belongs to the class π. Naturally, the classification can be used by others who like it.

The Appendices

Appendix A

The Space-Time Classical Logic and The Corresponding System

A.1 Axioms

$$\mathcal{CA}1: \ \Delta, \{A\} \vdash \Gamma, \{A\}$$

$$\mathcal{CA}2 \ \neg @ + s \cdot + t[A] = @ - s \cdot - t[\neg A]$$

$$\mathcal{CA}3 \ \neg @ + s \cdot - t[A] = @ - s \cdot + t[\neg A]$$

$$\mathcal{CA}4: \ \neg @ - s \cdot + t[A] = @ + s \cdot - t[\neg A]$$

$$\mathcal{CA}5: \ \neg @ - s \cdot - t[A] = @ + s \cdot + t[\neg A]$$

$$\mathcal{CA}6: \ @ + s \cdot + t[A] \wedge @ + s \cdot + t[B] = @ + s \cdot + t[A \wedge B]$$

$$\mathcal{CA}7: \ @ + s \cdot - t[A] \vee @ + s \cdot - t[B] = @ + s \cdot - t[A \vee B]$$

$$\mathcal{CA}8: \ @ - s \cdot + t[A] \vee @ - s \cdot + t[B] = @ - s \cdot + t[A \vee B]$$

$$\mathcal{CA}9: \ @ - s \cdot - t[A] \vee @ - s \cdot - t[B] = @ - s \cdot - t[A \vee B]$$

$$\mathcal{CA}10: \ @ + s \cup s_2 \cdot + t \cup t_2[A] =$$
$$@ + s \cdot + t[A] \wedge @ + s \cdot + t_2[A] \wedge @ + s_2 \cdot + t[A] \wedge @ + s_2 \cdot + t_2[A]$$

$$\mathcal{CA}11: \ @ - s \cup s_2 \cdot - t \cup t_2[A] =$$
$$@ - s \cdot - t[A] \vee @ - s \cdot - t_2[A] \vee @ - s_2 \cdot - t[A] \vee @ - s_2 \cdot - t_2[A]$$

A.2 Logical Rules

Implication:

$$\frac{\Delta, \{A\} \vdash \{B\}, \Gamma}{\Delta \vdash \{A \Rightarrow B\}, \Gamma} \quad \frac{\Delta \vdash \{A\}, \Gamma \quad \Psi, \{B\} \vdash \Omega}{\Delta, \Psi, \{A \Rightarrow B\} \vdash \Gamma, \Omega}$$

Conjunction:

$$\frac{\Delta \vdash \{A\}, \Gamma \quad \Psi \vdash \{B\}, \Omega}{\Delta, \Psi \vdash \{A \wedge B\}, \Gamma, \Omega} \quad \frac{\Delta \vdash \{A \wedge B\}, \Gamma}{\Delta \vdash \{A\}, \Gamma} \quad \frac{\Delta \vdash \{A \wedge B\}, \Gamma}{\Delta \vdash \{B\}, \Gamma}$$

Disjunction:

$$\frac{\Delta \vdash \{A\}, \Gamma}{\Delta \vdash \{A \vee B\}, \Gamma} \qquad \frac{\Delta \vdash \{B\}, \Gamma}{\Delta \vdash \{A \vee B\}, \Gamma}$$

$$\frac{\Delta \vdash \{A \vee B\}, \Gamma \quad \Psi, \{A\} \vdash \Theta \quad \Phi, \{B\} \vdash \Omega}{\Delta, \Psi, \Phi \vdash \Gamma, \Theta, \Omega}$$

Some asymmetry for eliminating ff:

$$\frac{\Delta \vdash \{\mathit{ff}\}}{\Delta \vdash \{A\}}$$

Negation:

$$\frac{\Delta, \{A\} \vdash \Gamma}{\Delta \vdash \{\neg A\}, \Gamma} \qquad \frac{\Delta \vdash \{A\}, \Gamma \quad \Psi \vdash \{\neg A\}, \Theta}{\Delta, \Psi \vdash \Gamma, \Theta}$$

A.2.1 Space and Time

In the following rules, s stands for either $-s$ or $+s$, and t stands for either $+t$ or $-t$. For all rules with $\cap$ on space or time, $s \cap s_2 \neq \emptyset$ and $t \cap t_2 \neq \emptyset$.

Space Weakening:

$$\frac{\Delta \vdash \{@ - s \cdot t[A]\}, \Gamma}{\Delta \vdash \{@\exists \cdot t[A]\}, \Gamma} - sW\mathcal{R} \qquad \frac{\Delta \vdash \{@\forall \cdot t[A]\}, \Gamma}{\Delta \vdash \{@ + s \cdot t[A]\}, \Gamma} + sW\mathcal{R}$$

$$\frac{\Delta \vdash \{@ + s \cdot t[A]\}, \Gamma}{\Delta \vdash \{@ - s \cdot t[A]\}, \Gamma} + -s\mathcal{R}$$

Space $\cup$ Introduction:

$$\frac{\Delta \vdash \{@ + s \cdot +t[A]\}, \Gamma \quad \Psi \vdash \{@ + s_2 \cdot +t[A]\}, \Omega}{\Delta, \Psi \vdash \{@ + s \cup s_2 \cdot +t[A]\}, \Gamma, \Omega} + s \cup \mathcal{IR}$$

Space $\cup$ Elimination:

$$\frac{\Delta \vdash \{@ - s \cup s_2 \cdot t[A]\}, \Gamma}{\Delta \vdash \{@ - s \cdot t[A] \vee @ - s_2 \cdot t[A]\}, \Gamma} - s \cup \mathcal{ER}$$

Space $\cap$ Introduction:

$$\frac{\Delta \vdash \{@ + s \cdot t[A]\}, \Gamma}{\Delta \vdash \{@ + s \cap s_2 \cdot t[A]\}, \Gamma} + s \cap \mathcal{IR}$$

Space $\cap$ Elimination:

$$\frac{\Delta \vdash \{@ - s \cap s_2 \cdot t[A]\}, \Gamma}{\Delta \vdash \{@ - s \cdot t[A]\}, \Gamma} - s \cap \mathcal{ER}$$

Space-Time $\cap$ Introduction:

$$\frac{\Delta \vdash \{@ + s \cdot +t[A]\}, \Gamma}{\Delta \vdash \{@ + s \cap s_2 \cdot +t \cap t_2[A]\}, \Gamma} + s + t \cap \mathcal{IR}$$

Space-Time $\cup$ Elimination:

$$\frac{\Delta \vdash \{@ + s \cup s_2 \cdot -t \cup t_2[A]\}, \Gamma}{\Delta \vdash \{@ + s \cdot -t[A] \wedge @ + s_2 \cdot -t[A] \vee @ + s \cdot -t_2[A] \wedge @ + s_2 \cdot -t_2[A]\}, \Gamma} + s - t \cup \mathcal{ER}$$

$$\frac{\Delta \vdash \{@ - s \cup s_2 \cdot +t \cup t_2[A]\}, \Gamma}{\Delta \vdash \{@ - s \cdot +t[A] \wedge @ - s \cdot +t_2[A] \vee @ - s_2 \cdot +t[A] \wedge @ - s_2 \cdot +t_2[A]\}, \Gamma} - s + t \cup \mathcal{ER}$$

Time Weakening:

$$\frac{\Delta \vdash \{@s \cdot -t[A]\}, \Gamma}{\Delta \vdash \{@s \cdot \exists[A]\}, \Gamma} - tW\mathcal{R} \qquad \frac{\Delta \vdash \{@s \cdot \forall[A]\}, \Gamma}{\Delta \vdash \{@s \cdot +t[A]\}, \Gamma} + tW\mathcal{R}$$

$$\frac{\Delta \vdash \{@s \cdot +t[A]\}, \Gamma}{\Delta \vdash \{@s \cdot -t[A]\}, \Gamma} + -t\mathcal{R}$$

Time $\cup$ Introduction:

$$\frac{\Delta \vdash \{@ + s \cdot +t[A]\}, \Gamma \quad \Psi \vdash \{@ + s \cdot +t_2[A]\}, \Theta}{\Delta, \Psi \vdash \{@ + s \cdot +t \cup t_2[A]\}, \Gamma, \Theta} + t \cup \mathcal{IR}$$

Time $\cup$ Elimination:

$$\frac{\Delta \vdash \{@s \cdot -t \cup t_2[A]\}, \Gamma}{\Delta \vdash \{@s \cdot -t[A] \vee @s \cdot -t_2[A]\}, \Gamma} - t \cup \mathcal{ER}$$

Time $\cap$ Introduction:

$$\frac{\Delta \vdash \{@s \cdot +t[A]\}, \Gamma}{\Delta \vdash \{@s \cdot +t \cap t_2[A]\}, \Gamma} + t \cap \mathcal{IR}$$

Time $\cap$ Elimination:

$$\frac{\Delta \vdash \{@s \cdot -t \cap t_2[A]\}, \Gamma}{\Delta \vdash \{@s \cdot -t[A]\}, \Gamma} - t \cap \mathcal{ER}$$

Appendix B

An Operational Semantics

An operational semantics is defined here to make the ideas presented in chapters 6 and 7 more precise. I define the semantics of certain language constructs, expression, assignment and conditional statements.

Let p be a program in the present object language $\mathcal{U}$ and let all of these definitions apply to the scope p. To avoid being exhaustive, I infer the $\boldsymbol{uu}$ type according to the operators, use $=$ and $\neq$ as polymorphic operators, and consider that variables have their separate scopes in each rule, although they have the same names in the set of rules. I apologize for this abuse of notation. Let A^I and A^L be isomorphic to $\mathbb{Z} \cup \{\boldsymbol{uu}\}$ and $\{ff, \boldsymbol{uu}, tt\}$ i.e. the set of logical values, respectively, and use these sets as carriers of the algebra that I am going to define. Because I use Łukasiewicz[116, 186] 3-valued logic in the rules, and it extends the semantics of the Boolean connectives using the same symbols, I use only the 3-valued connectives to avoid a mixture of logics, and write a dot over the symbols to stress that it refers to his logic. Thus, $\dot{\neg}\, ff \rightsquigarrow tt$, $\dot{\neg}\, tt \rightsquigarrow ff$, $\dot{\neg}\, \boldsymbol{uu} \rightsquigarrow \boldsymbol{uu}$, $\boldsymbol{uu} \,\dot{\wedge}\, ff \rightsquigarrow ff$, $\boldsymbol{uu} \,\dot{\wedge}\, tt \rightsquigarrow \boldsymbol{uu}$, $\boldsymbol{uu} \,\dot{\vee}\, tt \rightsquigarrow tt$, $\boldsymbol{uu} \,\dot{\vee}\, ff \rightsquigarrow \boldsymbol{uu}$, etc. Notice that although $\boldsymbol{uu} \neq \boldsymbol{uu}$ is false and $\boldsymbol{uu} = \boldsymbol{uu}$ is true at the rule level, both result in uu in the object language semantics. Now I can define an algebra $A \stackrel{def}{=} \langle A^I, A^L, Han, Var, Loc, Val, S, 0, ff, \boldsymbol{uu}, tt, u, v, x, y, ev, re, \$, def, +, -, \times, /, =, \neq, <, \dot{\neg}, \dot{\wedge}, \dot{\vee} \rangle$ for signature Σ in this analysis, where A^I is the set of integers, A^L is the set $\{ff, \boldsymbol{uu}, tt\}$, Han is the set of handlers, Var is the set of all vari-

ables, Loc is the set of locations, $Val \stackrel{def}{=} A^I \cup A^L$, and S is the store or state. Let $u, v \in Val,\ x, y \in Var$. I only consider variables of p and not constants or operands of another nature. Then, $\Sigma \stackrel{def}{=} \langle \{A^I, A^L, Han, Var, Loc, Val, S\}, F \rangle$, where F is consisted by $+$, $-$, $\times$, $/$, $=$, $\neq$, $<$, $\dot{\neg}$, $\dot{\wedge}$, $\dot{\vee}$ and the following functions:

$$(initialize) \quad \Omega : \ S \tag{B.1}$$

$$(locate) \quad \gamma : Var \rightarrow Loc \tag{B.2}$$

$$(lookup) \quad \rho : S \times Loc \rightarrow Val \tag{B.3}$$

$$(update) \quad \Delta : S \times Loc \times Val \rightarrow S \tag{B.4}$$

$$(handler\ is\ defined) \quad def : Han \rightarrow A^L \tag{B.5}$$

$$(evaluator) \quad ev : Var \rightarrow Han \tag{B.6}$$

$$(reactor) \quad re : Var \rightarrow Han \tag{B.7}$$

$$(intended\ value) \quad \$: Var \rightarrow Val \tag{B.8}$$

Intuitively, Ω initializes the whole memory; γ maps a variable to its location; ρ results in the content of a location in some particular state; and Δ updates the memory according to its parameters: location and value. As a syntactic sugar, I write $x.ev$ and $x.re$ to refer respectively to the evaluator $ev(x)$ and reactor $re(x)$ of some variable x, and use the following notation on the syntax: $def(x.ev)$ to mean that the evaluator of x exists, and $def(x.re)$ to mean that the reactor of x exists. Accordingly, I also write $x.\$$ to refer to the result from the evaluated expression that is always available during the evaluation of the reactor $x.re$ when it is defined and applied, that is, $x.\$$ is shorthand for $\$(x)$. Let $s_0, s, s', s'' \in S$, s_0 be the initial state. Then $\Omega \stackrel{def}{=} \forall x \in Var,\ \rho(s_0, \gamma x) = \boldsymbol{uu}$.

The operational semantics rules are:

$$Introduction: \quad \frac{}{\textbf{begin}} \quad \Omega \tag{B.9}$$

$$V_1: \quad \frac{\rho(s,\gamma x) = u \quad u \neq \boldsymbol{uu}}{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, s)} \tag{B.10}$$

$$V_2: \quad \frac{\rho(s,\gamma x) = \boldsymbol{uu} \quad def(x.ev) \quad \langle x.ev, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, s')}{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, s')} \tag{B.11}$$

$$Lazy\ +: \quad \frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu}, s')}{\langle x + y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu}, s')} \tag{B.12}$$

$$\frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \boldsymbol{uu} \quad \langle y, s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'') \quad v = \boldsymbol{uu}}{\langle x + y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu}, s'')} \tag{B.13}$$

$$\frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \boldsymbol{uu} \quad \langle y, s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'') \quad v \neq \boldsymbol{uu}}{\langle x + y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u + v, s'')} \tag{B.14}$$

$$(+)_1: \quad \frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu}, s')}{\langle x\ (+)\ y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu}, s')} \tag{B.15}$$

$$(+)_2: \quad \frac{\langle y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu}, s')}{\langle x\ (+)\ y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu}, s')} \tag{B.16}$$

$$(+)_3: \quad \frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \boldsymbol{uu} \quad \langle y, s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'') \quad v \neq \boldsymbol{uu}}{\langle x\ (+)\ y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u + v, s'')} \tag{B.17}$$

$$Lazy\ -: \quad \frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu}, s')}{\langle x - y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu}, s')} \tag{B.18}$$

$$\frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \boldsymbol{uu} \quad \langle y, s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'') \quad v = \boldsymbol{uu}}{\langle x - y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu}, s'')} \tag{B.19}$$

$$\frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \boldsymbol{uu} \quad \langle y, s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'') \quad v \neq \boldsymbol{uu}}{\langle x - y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u - v, s'')} \tag{B.20}$$

$$Lazy\ *: \quad \frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu}, s')}{\langle x * y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu}, s')} \tag{B.21}$$

$$\frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (0, s')}{\langle x * y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (0, s')} \tag{B.22}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \boldsymbol{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v = \boldsymbol{uu}}{\langle x * y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s'')} \tag{B.23}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \boldsymbol{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v \neq \boldsymbol{uu}}{\langle x * y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u \times v,s'')} \tag{B.24}$$

$$Lazy\ /: \quad \frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s')}{\langle x/y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s')} \tag{B.25}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \boldsymbol{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v = \boldsymbol{uu}}{\langle x/y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s'')} \tag{B.26}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \boldsymbol{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v \neq \boldsymbol{uu}}{\langle x/y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u/v,s'')} \tag{B.27}$$

$$Lazy\ =: \quad \frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s')}{\langle x=y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s')} \tag{B.28}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \boldsymbol{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v = \boldsymbol{uu}}{\langle x=y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s'')} \tag{B.29}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \boldsymbol{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v \neq \boldsymbol{uu}}{\langle x=y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u=v,s'')} \tag{B.30}$$

$$Lazy\ !=: \quad \frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s')}{\langle x\ !=y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s')} \tag{B.31}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \boldsymbol{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v = \boldsymbol{uu}}{\langle x\ !=y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s'')} \tag{B.32}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \boldsymbol{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v \neq \boldsymbol{uu}}{\langle x\ !=y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u \neq v,s'')} \tag{B.33}$$

$$Lazy\ <: \quad \frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s')}{\langle x<y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s')} \tag{B.34}$$

$$\frac{\langle x,s\rangle \overset{\text{eval}}{\rightsquigarrow} (u,s') \quad u \neq \boldsymbol{uu} \quad \langle y,s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v,s'') \quad v = \boldsymbol{uu}}{\langle x<y,s\rangle \overset{\text{eval}}{\rightsquigarrow} (\boldsymbol{uu},s'')} \tag{B.35}$$

$$\frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \textbf{\textit{uu}} \quad \langle y, s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'') \quad v \neq \textbf{\textit{uu}}}{\langle x < y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u < v, s'')} \tag{B.36}$$

$$\frac{\rho(s, \gamma x) \neq \textbf{\textit{uu}}}{\langle \textbf{not } x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\dot{\neg}\rho(s, \gamma x), s)} \tag{B.37}$$

$$\frac{\rho(s, \gamma x) = \textbf{\textit{uu}} \quad \dot{\neg}\, def(x.ev)}{\langle \textbf{not } x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\textbf{\textit{uu}}, s)} \tag{B.38}$$

$$\frac{\rho(s, \gamma x) = \textbf{\textit{uu}}}{\langle x \textbf{ known}, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{ff}, s)} \tag{B.39}$$

$$\frac{\rho(s, \gamma x) \neq \textbf{\textit{uu}}}{\langle x \textbf{ known}, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{tt}, s)} \tag{B.40}$$

$$\frac{\rho(s, \gamma x) = \textbf{\textit{uu}}}{\langle x \textbf{ unknown}, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{tt}, s)} \tag{B.41}$$

$$\frac{\rho(s, \gamma x) \neq \textbf{\textit{uu}}}{\langle x \textbf{ unknown}, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{ff}, s)} \tag{B.42}$$

$$\frac{\rho(s, \gamma x) = \textbf{\textit{uu}} \quad def(x.ev) \quad \langle x.ev, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, s')}{\langle \textbf{not } x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\dot{\neg} v, s')} \tag{B.43}$$

$$Lazy \;\; \dot{\wedge}\text{:} \quad \frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{ff}, s')}{\langle x \textbf{ and } y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{ff}, s')} \tag{B.44}$$

$$\frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \mathit{ff} \quad \langle y, s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'')}{\langle x \textbf{ and } y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u \mathbin{\dot{\wedge}} v, s'')} \tag{B.45}$$

$$Lazy \;\; \dot{\vee}\text{:} \quad \frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{tt}, s')}{\langle x \textbf{ or } y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (\mathit{tt}, s')} \tag{B.46}$$

$$\frac{\langle x, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u \neq \mathit{tt} \quad \langle y, s'\rangle \overset{\text{eval}}{\rightsquigarrow} (v, s'')}{\langle x \textbf{ or } y, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u \mathbin{\dot{\vee}} v, s'')} \tag{B.47}$$

$$\frac{\dot{\neg}\, def(x.re) \quad \langle E, s\rangle \overset{\text{eval}}{\rightsquigarrow} (v, s')}{\langle x := E, s\rangle \overset{\text{exec}}{\rightsquigarrow} s''} \Delta(s', \gamma x, v) = s'' \tag{B.48}$$

$$\frac{def(x.re) \quad \langle E, s\rangle \overset{\text{eval}}{\rightsquigarrow} (u, s') \quad u = \rho(s', \gamma(x.\$)) \quad \langle x.re, s'\rangle \overset{\text{exec}}{\rightsquigarrow} s''}{\langle x := E, s\rangle \overset{\text{exec}}{\rightsquigarrow} s''} \tag{B.49}$$

$$\frac{\langle ThVL, s\rangle \overset{\text{eval}}{\rightsquigarrow} (tt, s') \quad \langle C1, s'\rangle \overset{\text{exec}}{\rightsquigarrow} s''}{\langle \textbf{if } ThVL \textbf{ then } C1 \textbf{ ifnot } C2 \textbf{ otherwise } C3, s\rangle \overset{\text{exec}}{\rightsquigarrow} s''} \tag{B.50}$$

$$\frac{\langle ThVL, s\rangle \overset{\text{eval}}{\rightsquigarrow} (ff, s') \quad \langle C2, s'\rangle \overset{\text{exec}}{\rightsquigarrow} s''}{\langle \textbf{if } ThVL \textbf{ then } C1 \textbf{ ifnot } C2 \textbf{ otherwise } C3, s\rangle \overset{\text{exec}}{\rightsquigarrow} s''} \tag{B.51}$$

$$\frac{\langle ThVL, s\rangle \overset{\text{eval}}{\rightsquigarrow} (uu, s') \quad \langle C3, s'\rangle \overset{\text{exec}}{\rightsquigarrow} s''}{\langle \textbf{if } ThVL \textbf{ then } C1 \textbf{ ifnot } C2 \textbf{ otherwise } C3, s\rangle \overset{\text{exec}}{\rightsquigarrow} s''} \tag{B.52}$$

$$\frac{\langle ThVL, s\rangle \overset{\text{eval}}{\rightsquigarrow} (ff, s')}{\langle \textbf{while } ThVL \textbf{ do } C, s\rangle \overset{\text{exec}}{\rightsquigarrow} s'} \tag{B.53}$$

$$\frac{\langle ThVL, s\rangle \overset{\text{eval}}{\rightsquigarrow} (uu, s')}{\langle \textbf{while } ThVL \textbf{ do } C, s\rangle \overset{\text{exec}}{\rightsquigarrow} s'} \tag{B.54}$$

$$\frac{\langle ThVL, s\rangle \overset{\text{eval}}{\rightsquigarrow} (tt, s') \quad \langle C, s'\rangle \overset{\text{exec}}{\rightsquigarrow} s'' \quad \langle \textbf{while } ThVL \textbf{ do } C, s''\rangle \overset{\text{exec}}{\rightsquigarrow} s'''}{\langle \textbf{while } ThVL \textbf{ do } C, s\rangle \overset{\text{exec}}{\rightsquigarrow} s'''} \tag{B.55}$$

$$\frac{\langle C1, s\rangle \overset{\text{exec}}{\rightsquigarrow} s' \quad \langle C2, s'\rangle \overset{\text{exec}}{\rightsquigarrow} s''}{\langle C1; C2, s\rangle \overset{\text{exec}}{\rightsquigarrow} s''} \tag{B.56}$$

Appendix C

Symbols and Conventions

In addition to standard mathematical symbols, the present thesis dissertation contains formulas whose notations have the following meanings. The symbols are on the left and their respective meanings on the right.

Backus-Naur Form (BNF):

$\mathcal{L} \longmapsto \mathcal{R}$	A rule with left hand side $\mathcal{L}$ and right hand side $\mathcal{R}$
$\mid$	Alternative right hand side
...	Describes an infinite list of items, perhaps alternatives

Basics:

φ	A formula
$a, b, c, ...$	Variables
$x, y, z, ...$	Variables
v	In some contexts, v stands for a value
tt	The "true" value
ff	The "false" value
i, j, k, m, n	Used as integer variables
$\vdash$	A sequent relation
$\rightarrow, \Rightarrow$	Outside @-logic, classical or intuitionistic implications
$\Leftrightarrow, \equiv$	Classical or intuitionistic equivalence, syntax equivalence

$\mathbb{N}$	The set of natural numbers
$\mathbb{Z}$	The set of integers
$\mathbb{R}$	The set of real numbers
$\mathbb{E}^3$	Euclidean space
$\mathcal{P}$	The power set
t	Time, also as time instant variable, e.g. final time t_f
	t can also be a Turing-computable function
s	Space, also as place variable
p	Often used as a place variable, or a proposition
P	Often, a place or a proposition
$\langle\ \rangle$	Tuple for any number of elements separated by comma
$\langle a, b\rangle$	An (ordered) pair of values a, b
(a, b)	Another form: An (ordered) pair of values a, b
$\langle x, y, z\rangle$	The coordinates of a point in $\mathbb{E}^3$

@-Logic:

$\mathbb{T}$	Time set. @-Logic time flow. I define $\mathbb{T} \stackrel{def}{=} \mathbb{R}$ in chapter 4
$\mathbb{S}$	Space set. @-Logic space. I define $\mathbb{S} \stackrel{def}{=} \mathbb{R}^3$ in chapter 4.
$\varphi_1 =_t \varphi_2$	φ_1 and φ_2 happen at the same time
$\varphi_1 \neq_t \varphi_2$	φ_1 and φ_2 do not happen at the same time
$\varphi_1 <_t \varphi_2$	φ_1 happens before φ_2
$\varphi_1 \leq_t \varphi_2$	φ_1 happens before or at the same time as φ_2
$\varphi_1 >_t \varphi_2$	φ_1 happens after φ_2
$\varphi_1 \geq_t \varphi_2$	φ_1 happens after or at the same time as φ_2
$\varphi_1 -_t \varphi_2$	Duration of time from φ_2 to φ_1
$+_t$	Sum of a duration to a moment in time
$\varphi_1 =_s \varphi_2$	φ_1 and φ_2 happen in the same place
$\varphi_1 \neq_s \varphi_2$	φ_1 and φ_2 do not happen in the same place

$\varphi_1 <_s \varphi_2$	φ_1 happens before φ_2 (in accordance with some spatial order)
$\varphi_1 \leq_s \varphi_2$	φ_1 happens before or in the same place as φ_2
$\varphi_1 >_s \varphi_2$	φ_1 happens after φ_2 (in accordance with some spatial order)
$\varphi_1 \geq_s \varphi_2$	φ_1 happens after or in the same space as φ_2
$\varphi_1 -_s \varphi_2$	Distance between φ_2 and φ_1
$+_s$	Sum of a distance to a point in space
$(\forall x)\ \varphi$	For all x, φ holds. The formula binds x
$(\exists x)\ \varphi$	There exists x such that φ. The formula binds x
$(\exists! x)\ \varphi$	Syntax sugar: there exists a unique x such that φ
$\forall$	As well as the standard meaning, any time, or anywhere (depending on the context)
$\exists$	As well as the standard meaning, some time, or somewhere (depending on the context)
$@s \cdot t[\varphi]$	φ at place s and time t
$@s \cdot t[\![\varphi]\!]$	The meaning of φ (it requires that $\varphi \doteq tt$) at place s and time t

Models:

$\Delta \models \varphi$	Δ models φ

Other defined symbols:

$\stackrel{def}{=}$	Definition
$\bot$	Falsity, and in some contexts, infinite computation
$C \leftarrow A$	Denotes a Prolog or GLOBALLOG rule with semantics similar to $A \Rightarrow C$

$min(a, b)$	Function: the least value between a and b
$max(a, b)$	Function: the greatest value between a and b
$[\alpha, \zeta]$	Closed interval, from α to ζ (open interval not used here)

Types and signatures:

$f, g : T$	Function f and g of type T
$f : T$	Function f of type T
$\alpha \times \beta$	Cartesian product between α and β
$A \longrightarrow B$	Function with domain A and codomain B

Turing Machines:

F, G, H	Turing machines
f, g, h	Turing-computable functions
x	An integer number
X	The code of x as a null machine computation or input
Q	The finite set of states
q	A state
q_0	The initial state
F	The set of final states
Σ	The alphabet
s	A symbol of the alphabet Σ.
	s is also a state of computation in chapter 4.
$\circ$	The blank symbol. The same as s_0
T	The set of input symbols
P	The Turing machine program
$\mathcal{O}$	The set of operations $\{L, R, S, H\}$
	$\mathcal{O}$ is also the complexity function
γ	A transition function $\gamma : \ Q \times S \longrightarrow Q \times S \times \mathcal{O}$
	$\gamma : Var \longrightarrow Loc$ is used in chapter 4 as a function

	that, given a variable, locates its storage
$\odot$	The write/read head
U	The Universal Turing machine
$r(M)$	The representation of Turing machine M
$r(X)$	The representation of input X
$M(N[X])$	A composition where M is outside the tape and N and X are on the tape
$\uparrow M(N[X])$	The computation of $M(N[X])$

Process Algebra:

$A[P]$	A process P running in a place A
$P_1.P_2$	Sequential composition between processes P_1 and P_2
$P_1 \vert P_2$	Parallel composition between processes P_1 and P_2
$A \parallel B$	Parallel composition between places A and B
$flyto(B)$	The operation that moves the process to place B
$\xrightarrow{\tau}$	An atomic operation

Space-Time Operational Semantics (Programming Languages):

σ	A state represented in a semantics
$\sigma(m/X)$	State σ and $X = m$ holds in σ
c	A programming language construct
$\langle c, \sigma \rangle$	Construct c in state σ
$@s \cdot t[\![c]\!]$	The meaning of construct c at space s and time t
$\rightsquigarrow$	Evaluates to
$\mathcal{I}_t$	Proving that a predicate is true.
$\mathcal{I}_f$	Proving that a predicate is false.
$\mathcal{I}_{tf}$	Trying to find the truth value for a predicate, *ff*, *tt* or *uu*.

Computing in the Real World:

$s_1 \Downarrow\!\Downarrow s_2$	Two independent states
$C_1 \mid C_2$	Two computations in parallel
Ψ	Denotes probability. Also used as time for code mobility
μ	Strong mobility
$\ddot{\mu}$	Physical hardware mobility
π	Mathematics and related concepts.
	Also used as usual, i.e. radian constant
	Also used to denote a program in chapter 4
ϕ	Philosophy and related concepts
ψ	Psychology and related concepts.
ω	Physics and related concepts.
	Also used as the speed of light:
	$(\omega = c)$ where c is the speed of light

Bibliography

[1] Martin Abadi and Luca Cardelli, *A theory of objects*, Springer-Verlag, New York, 1996.

[2] S Abramsky, Dov M Gabbay, and T S E Maibaum (eds.), *Handbook of logic in computer science*, vol. 1 Background: Mathematical Structures, Oxford University Press Inc., 1992.

[3] Samson Abramsky, *Handbook of logic in computer science*, vol. 3: Semantic Structures, ch. Domain Theory, pp. 1–168, Oxford University Press, 1994.

[4] Anurag Acharya, Mudumbai Ranganathan, and Joel Saltz, *Dynamic linking for mobile programs*, Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag, April 1997, Lecture Notes in Computer Science No. 1222, pp. 245–262.

[5] ———, *Sumatra: A language for resource-aware mobile programs*, Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag, April 1997, Lecture Notes in Computer Science No. 1222, pp. 111–130.

[6] Peter Aczel, Harold Simmons, and Stanley S. Wainer (eds.), *Proof theory*, Cambridge University Press, 1992.

[7] Alfred Vaino Aho, Ravi Sethi, and Jeffrey David Ullman, *Compilers: principles, techniques, and tools*, Addison-Wesley series in computer sci-

ence, Addison-Wesley Publishing Company, Reading (Mass.). - London, 1986, Originally published as: Principles of compiler design.

[8] H. Ait-Kaci et al., *The wild life handbook*, Digital, Paris Research Laborator, 1994, http://www.isg.sfu.ca/life/.

[9] J. J. Alferes and Luís Moniz Pereira, *Reasoning with logic programming*, Lecture Notes in Computer Science. Lecture Notes in Artificial Intelligence, no. 1111, Springer-Verlag, Berlin, London, 1996.

[10] James Allen, *Towards a general theory of action and time*, Artificial Intelligence **23** (1984), 123–154.

[11] ———, *Time and time again: The many ways to represent time*, International Journal of Intelligent Systems **6** (1991), no. 4, 341–355.

[12] James Allen and George Ferguson, *Actions and events in interval temporal logic*, Journal of Logic and Computation **4** (1994), no. 5.

[13] James Allen and James Hendler (eds.), *Readings in planning*, Representation and Reasoning, Morgan Kaufmann, San Mateo, California, 1990.

[14] Lloyd Allison, *A practical introduction to denotational semantics*, Cambridge Computer Science Texts, no. 23, Cambridge University Press, 1986, Reprinted 1995.

[15] Roberto Amadio and Pierre-Louis Curien, *Domains and lambda-calculi*, Cambridge Tracts in Theoretical Computer Science, no. 46, Cambridge University Press, 1998.

[16] Roberto M. Amadio, *On modelling mobility*, Theoretical Computer Science **240** (2000), no. 1, 147–176.

[17] Alan Ross Anderson and Nuel D. Belnap Junior, *Entailment: The logic of relevance and necessity*, vol. 1, Princeton University Press, 1975.

[18] Grigoris Antoniou, Michael Maher, and David Billington, *Defeasible logic versus logic programming without negation as failure*, The Journal of Logic Programming **42** (2000), 47–57.

[19] Krzysztof R. Apt and Roland N. Bol, *Logic programming and negation: A survey*, The Journal of Logic Programming **19 & 20** (1994), 9–72.

[20] Ofer Arieli and Arnon Avron, *Frontiers of paraconsistent logic*, Studies in Logic and Computation, vol. 8, ch. Bilattices and Paraconsistency, pp. 11–27, Research Studies Press Ltd, 2000.

[21] Ken Arnold and James Goslin, *The java programming language*, Addison-Wesley Publishing Company, 1996.

[22] Jean-Michel Autebert, Jean Berstel, and Luc Boasson, *Hanbook of formal languages*, vol. 1, ch. Context-Free Languages and Pushdown Automata, pp. 111–174, Springer-Verlag, 1997.

[23] Franz Baader and Tobias Nipkow, *Term rewriting and all that*, Cambridge University Press, 1998.

[24] Henry Balen, *Distributed object architectures with corba*, Cambridge University Press, 1999.

[25] Roberto Barbuti, Nicoletta De Francesco, Paolo Mancarella, and Antonella Santone, *Towards a logical semantics for pure prolog*, Science of Computer Programming **32** (1998), no. 1–3, 145–176.

[26] Rosalind Barrett, Allan Ramsay, and Aaron Sloman, *Pop-11: A practical language for artificial intelligence*, Ellis Horwood series in computers and their applications, Halstead Press, Chichester New York, 1985.

[27] Diderik Batens, Chris Mortensen, Graham Priest, and Jean-Paul Van Bendegem (eds.), *Frontiers of paraconsistent logic*, Studies in Logic and Computation, vol. 8, Research Studies Press Ltd, 2000.

[28] Paul W. Beame and Samuel R. Buss (eds.), *Proof complexity and feasible arithmetics: Dimacs workshop 96*, DIMACS series in discrete mathematics and theoretical computer science, vol. 39, American Mathematical Society, 1996.

[29] Paolo Bellavista, Antonio Corradi, and Cesare Stefanelli, *Middleware services for interoperability in open mobile agent systems*, Microprocessors and Microsystems **25** (2001), no. 2, 75–83.

[30] Nuel D. Belnap Junior, *A useful four-valued logic*, Proceedings of the Fifth International Symposium on Multiple-Valued Logic (J Michael Dunn and George Epstein, eds.), Modern Uses of Multiple-Valued Logic, Indiana University, D Reidel Publishing Company, 1975, pp. 8–37.

[31] Paul Benacerraf and Hilary Putnam (eds.), *Philosophy of mathematics: selected readings*, Prentice-Hall, Inc., 1964.

[32] Chantal Berline, *From computation to foundations via functions and application: The λ-calculus and its webbed models*, Theoretical Computer Science **249** (2000), no. 1, 81–161.

[33] Paul Bernays, *Philosophy of mathematics*, ch. On Platonism in Mathematics, pp. 274–286, Prentice-Hall, Inc., 1964.

[34] Claudio Bettini, Sushil Jajodia, and Sean X. Wang, *Time granularities in databases, data mining and temporal reasoning*, Springer-Verlag, 2000.

[35] Krishna Bharat and Luca Cardelli, *Migratory applications*, Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag, April 1997, Lecture Notes in Computer Science No. 1222, pp. 131–149.

[36] W. Bibel and E. Eder, *Handbook of logic in artificial intelligence and logic programming*, vol. 1: Logical Foundations, ch. Methods and Calculi for Deduction, pp. 67–182, Oxford University Press, 1993.

[37] Lubomir F. Bic, Munehiro Fukuda, and Michael B. Dillencourt, *Distributed computing using autonomous objects*, IEEE Computer (1996).

[38] Richard Bird and Oege de Moor, *Algebra of programming*, Prentice-Hall International Series in Computer Science, Prentice Hall Europe, 1997.

[39] Richard Bird and Philip Wadler, *Introduction to functional programming*, Prentice-Hall International Series in Computer Science, Prentice-Hall International Ltd, 1988.

[40] Thomas Bittner, *Rough sets in spatio-temporal data mining*, Temporal, Spatial and Spatio-Temporal Data Mining, Lecture Notes in Artificial Intelligence, vol. LNAI 2007, Springer, September 2000, pp. 89–104.

[41] Wayne D. Blizard, *A formal theory of objects, space and time*, The Journal of Symbolic Logic **55** (1990), no. 1, 74–89.

[42] George S. Boolos and Richard C. Jeffrey, *Computability and logic*, third ed., Cambridge University Press, 1989.

[43] N. S. Borenstein, *Email with a mind of its own: The safe-tcl language for enabled mail*, Tech. report, First Virtual Holdings, Inc, 1994.

[44] Gerhard Brewka, *Principles of knowledge representation*, Studies in Logic, Language and Information, CSLI International: Center for the Study of Language and Information and FoLLI: the European Association for Logic, Language and Information, 1996.

[45] Gerhard Brewka and Jürgen Dix, *Logic programming and knowledge representation: Third international workshop / lpkr'97*, Lecture Notes in Artificial Intelligence, vol. 1471, ch. Knowledge Representation with Logic Programs, pp. 1–51, Springer, New York, October 1998.

[46] Manfred Broy, *Refinement of time*, Theoretical Computer Science **253** (2001), no. 1, 3–26.

[47] Glenn Bruns, *Distributed systems analysis with ccs*, Prentice-Hall International Series in Computer Science, Prentice-Hall Europe, 1997.

[48] Michelle Bugliesi and Giuseppe Castagna, *Secure safe ambients*, Proceedings of the POPL 2001, XXVIII ACM SIGPLAN - SIGACT, Symposium on Principles of Programming Languages, ACM SIGPLAN, SIGACT, 2001, Also SIGPLAN Notices 36(3):222–235, pp. 222–235.

[49] Robert Bull, *Logic and reality: essays on the legacy of arthur prior*, ch. Logics without Contraction I, pp. 317–336, Oxford University Press, 1996.

[50] Martin Bunder, *Logic and reality: essays on the legacy of arthur prior*, ch. Logics without Contraction II, pp. 337–349, Oxford University Press, 1996.

[51] John P. Burgess, *Proof, logic and formalization*, ch. Proofs About Proofs: a Defense of Classical Logic. Part I: the aim of classical logic, pp. 8–23, Routledge, 1992.

[52] Alan Burns and Andy Wellings, *Concurrency in ada*, second ed., Cambridge University Press, 1998.

[53] David Burrell, *Analogy and philosophical language*, Yale University Press, 1973.

[54] Stanley N. Burris, *Logic for mathematics and computer science*, Prentice Hall, Inc., 1998.

[55] Luca Cardelli, *A language with distributed scope*, Computing Systems **8** (1995), no. 1, 27–59, Also available as Digital Systems Research Center Research Report 122.

[56] ———, *A language with distributed scope*, Computing Systems. The MIT Press **8** (1995), no. 1, 27–59.

[57] ———, *Global computation*, ACM Computing Surveys **28A** (1996), no. 4.

[58] ———, *Mobile computation*, Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag, April 1997, Lecture Notes in Computer Science No. 1222, pp. 3–6.

[59] ———, *Mobile object systems*, Lecture Notes in Computer Science, no. 1222, ch. Mobile Computation, Springer-Verlag, Linz, Austria, 1997.

[60] Luca Cardelli and Andrew D. Gordon, *Foundations of software science and computational structures*, Lecture Notices in Computer Science, vol. 1378, ch. Mobile Ambients, pp. 140–155, Springer-Verlag, 1998, Also Proceedings of FoSSaCS'98.

[61] ———, *Mobile ambients*, Theoretical Computer Science **240** (2000), no. 1, 177–213.

[62] Alexander Chagrov and Michael Zakharyaschev, *Modal logic*, Oxford Logic Guides, vol. 35, Oxford University Press, 1997.

[63] ———, *Modal logic*, vol. 35, ch. Complexity Problems, Oxford University Press, 1997.

[64] T. S. Champlin, *Reflexive paradoxes*, Routledge, 1988.

[65] C. C. Chang and H. J. Keisler, *Model theory*, Studies in Logic and the Foundations of Mathematics, vol. 73, North-Holland Publishing Company and American Elsevier Publishing Company, Inc., 1973.

[66] David Chess, Colin Harrison, and Aaron Kershenbaum, *Mobile agents: Are they a good idea? – update*, Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag, April 1997, Lecture Notes in Computer Science No. 1222, pp. 46–48.

[67] Alonzo Church, *Introduction to mathematical logic*, Princeton Mathematical series, Princeton University Press, 1956, Tenth printing (1996) for the Princeton Landmarks in Mathematics and Physics series.

[68] Keith L. Clark, *Logic and data bases*, ch. Negation as Failure, pp. 293–322, Plenum Press, New York, 1978.

[69] A. G. Cohn, J. M. Gooday, and B. Bennett, *A comparison of structures in spatial and temporal logics*, Philosophy and the Cognitive Sciences (1994), Also http://www.comp.leeds.ac.uk/spacenet/gooday.html.

[70] Richard Connor and Keith Sibson, *Paradigms for global computation - an overview of the hippo project*, Proceedings of Computer Society International Conference on Computer Languages (Chicago), IEEE, 1998.

[71] B. J. Copeland (ed.), *Logic and reality: essays on the legacy of arthur prior*, Oxford University Press, 1996.

[72] General Magic Corp., *Odyssey white paper*, 1998.

[73] G. Crocco and Luis Farias del Cerro, *Conditionals: from philosophy to computer science*, Studies in Logic and Computation, Clarendon Press, Oxford University, 1995.

[74] Tristan Crolard, *Subtractive logic*, Theoretical Computer Science **254** (2001), no. 1–2, 151–185.

[75] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna, *Analyzing mobile code languages*, Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag, April 1997, Lecture Notes in Computer Science No. 1222, pp. 93–110.

[76] Nigel Cutland, *Computability: an introduction to recursive function theory*, Cambridge University Press, 1980, This book was reprinted.

[77] Subrata Kumar Das, *Deductive databases and logic programming*, International Series in Logic Programming, Addison-Wesley Publishing Company, 1992.

[78] Joseph Warren Dauben, *Georg cantor: his mathematics and philosophy*, Harvard University Press, 1979.

[79] J. H. Davenport, *Computing tomorrow: future research directions in computer science*, ch. Computer Science and Mathematics, pp. 66–87, Cambridge University Press, 1996.

[80] Antony J. T. Davie, *An introduction to functional programming systems using haskell*, Cambridge Computer Science Texts, no. 27, Cambridge University Press, 1992.

[81] Martin Davis, *The universal computer: the road form leibniz to turing*, ch. 2 Boole Turns Logic into Algebra, pp. 21–40, W. W. Norton & Company, 2000.

[82] ———, *The universal computer: the road form leibniz to turing*, W. W. Norton & Company, 2000.

[83] D.D.R. (ed.), *Classics in logic: Readings in epistemology, theory of knowledge and dialectics*, Peter Owen Limited, 1962.

[84] D. Dean, *The security of static typing with dynamic linking*, Proceedings of the Fourth ACM Conference on Computer and Communications Security (Zurich, Switzerland), April 1997.

[85] D. Dean, E. W. Felten, and D. S. Wallach, *Java security: From hotjava to netscape and beyond*, Proceedings of the Symposium on Security and Privacity, IEEE, 1996, pp. 190–200.

[86] Drew Dean, Ed Felten, and Dan Wallach, *Java security: From HotJava to Netscape and beyond*, Proceedings of the 1996 IEEE Symposium on Security and Privacy (Oakland, Cal.), May 1996.

[87] Giorgio Delzanno and Maurizio Martelli, *Proofs as computations in linear logic*, Theoretical Computer Science **258** (2001), no. 1–2, 269–297.

[88] René Descartes, *Meditations and other metaphysical writings*, Penguin Classics, Penguin Group, 1998.

[89] Kees Doets, *Basic model theory*, CSLI International: Center for the Study of Language and Information and FoLLI: the European Association for Logic, Language and Information, 1996.

[90] Jana Dospisil and E. Kendall, *Automated negotiation with agents*, Advances in Mobile Agents Systems Research. Proceedings of the 12th International Conference on System Research, Informatics & Cybernetics (George E. Lasker, Jana Dospisil, and Elisabeth Kendall, eds.), vol. 1: Theory and Applications, The International Institute for Advanced Studies in Systems Research and Cybernetics, August 2000, pp. 1–11.

[91] David Duffy, *Principles of automated theorem proving*, John Wiley & Sons, 1991.

[92] Michael Dummett, *The philosophy of mathematics*, Oxford Readings in Philosophy, ch. The Philosophical Basis of Intuitionistic Logic, Oxford University Press, 1996.

[93] P. M. Dung, *Negation as hypothesis: an abductive foundation for logic programming*, 8th International Conference on Logic Programming (Cambridge, MA), The MIT Press, 1991, pp. 3–17.

[94] P. M. Dung and P. Mancarella, *Production systems need negation as failure*, Proceedings of the XIII National Conference on Artificial Intelligence, vol. 2, AAAI Press and the MIT Press, 1996, pp. 1242–1247.

[95] J. Michael Dunn, *Handbook of philosophical logic*, Synthese library; v. 166, vol. III: Alternatives to Classical Logic, ch. Relevance Logic and Entailment, pp. 117–224, Kluwer Academic Publisher, 1986.

[96] John R. Durbin, *Modern algebra: an introduction*, fourth ed., John Wiley & Sons, Inc., 2000.

[97] Thomas Eiter, Wolfgang Faber, and M. Truszczyński (eds.), *Logic programming and nonmonotonic reasoning*, Lecture Notes in Artificial Intelligence, vol. LNAI 2173, Springer, 2001.

[98] E. Allen Emerson, *Handbook of theoretical computer science*, vol. B Formal Models and Semantics, ch. 16 Temporal and Modal Logic, pp. 995–1072, The MIT Press/Elsevier, 1990.

[99] Richard L. Epstein, *Predicate logic*, Oxford University Press, 1994.

[100] ———, *The semantics foundations of logic*, second ed., Oxford University Press, 1995.

[101] ———, *The semantics foundations of logic*, ch. Intuitionism, p. 277, Oxford University Press, 1995.

[102] William M. Farmer, Joshua D. Guttman, and Vipin Swarup, *Security for mobile agents: Issues and requirements*, Proceedings of the 19th National Information Systems Security Conference (Baltimore, Md.), October 1996, pp. 591–597.

[103] Ulisses Ferreira, *The plain www page*, URL http://www.ufba.br/~plain (1996–2002).

[104] ———, *Intelligent agents for the internet*, Proceedings of XIX Congress of SBC, ENIA'99-SBC, Sociedade Brasileira de Computação, July 1999.

[105] ———, *Chiron: a framework for mobile agent systems*, Advances in Mobile Agents Systems Research. Proceedings of the 12th International Conference on System Research, Informatics & Cybernetics (George E. Lasker, Jana Dospisil, and Elisabeth Kendall, eds.), vol. 1: Theory and Applications, The International Institute for Advanced Studies in Systems Research and Cybernetics, August 2000, pp. 12–22.

[106] ———, uu *for programming languages*, ACM SIGPLAN Notices **35** (2000), no. 8, 20–30.

[107] Ulisses Ferreira, Pedro S. Nicolletti, and Hélio M. Silva, *Lidia: Uma linguagem para sistemas especialistas bayesianos*, Proceedings of the V SBIA-SBC, Sociedade Brasileira de Computação, 1988, Published in Portuguese.

[108] ———, *Tratamento de incerteza na linguagem lidia*, Proceedings of the VI SBIA-SBC, 1989, Published in Portuguese.

[109] José Ulisses Ferreira Junior, Pedro S. Nicolletti, and Hélio M. Silva, *Tratamento de incerteza na linguagem lidia*, Proceedings of the VI SBIA-SBC, 1989, Published in Portuguese.

[110] Chris Fields, *Machines and thought: The legacy of alan turing*, Mind Association ocasional series, vol. 1, ch. Mensurement and Computational Description, pp. 165–177, Oxford University Press, 1996.

[111] Melvin Fitting, *A kripke/kleene semantics for logic programs*, Journal of Logic Programming **2** (1985), no. 4, 295–312.

[112] ———, *Kleene's logic, generalized*, Tech. report, City University of New York, Lehman College, Department of Mathematics and Computer Science, New York, 1990.

[113] ———, *Well-founded semantics, generalized*, Tech. report, City University of New York, Lehman College, Department of Mathematics and Computer Science, New York, 1991.

[114] Margaret M. Fleck, *The topology of boundaries*, Artificial Intelligence **80** (1996), no. 1, 1–27.

[115] Wan Fokkink, *Introduction to process algebra*, Texts in theoretical computer science, Springer-Verlag, 2000.

[116] Jan Łukasiewicz, *Jan łukasiewicz selected works*, Series on Studies in Logic and Foundations of Mathematics, North-Holland Publishing Company and PWN - Polish Scientific Publishers, 1970.

[117] Peter Forrest, *The dynamics of belief: A normative logic*, Philosophical Theory, Basil Blackwell, 1996.

[118] Cédric Fournet and Georges Gonthier, *The reflexive chemical abstract machine and the join-calculus*, Proceedings of 23rd ACM Symposium on Principles of Programming Languages, January 1996.

[119] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy, *A calculus of mobile agents*, 7th International Conference on Concurrency Theory (CONCUR'96) (Pisa, Italy), Springer-Verlag, August 1996, LNCS 1119, pp. 406–421.

[120] Eric Freeman, *Supercomputer earth: Massively parallel internet*, Tech. report, Yale University, December 1993, Supplement to the Yale Weekly Bulletin.

[121] Gottlob Frege, *The basic laws of arithmetic: exposition of the system*, University of California Press, 1964.

[122] Christian Freksa, Wilfried Brauer, Christopher Habel, and Karl F. Wender (eds.), *Spatial cognition* II*: Integrating abstract theories, empirical studies, formal methods and practical applications*, Lecture Notes in Artificial Intelligence, vol. 1849, Springer, 2000.

[123] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna, *Understanding code mobility*, IEEE Transactions on Software Engineering **24** (1998), no. 5.

[124] André Fuhrmann, *An essay on contraction*, Studies in Logic, Language and Information, CSLI Publications and FoLLI, 1997.

[125] Dov M. Gabbay, *Labelled deductive systems*, Oxford Logic Guides 33, vol. 1, Oxford University Press, 1996.

[126] ———, *Elementary logics: A procedural perspective*, Prentice Hall Series in Computer Science, Prentice Hall Europe, 1998.

[127] Dov M. Gabbay and Ian Hodkinson, *Logic and reality: essays on the legacy of arthur prior*, ch. Temporal Logic in the Context of Databases, pp. 69–87, Oxford University Press, 1996.

[128] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds (eds.), *Temporal logic: mathematical foundations and computational aspects*, vol. 1, Oxford University Press Inc., New York, 1994.

[129] Denis Gagné, Wanlin Pang, and André Trudel, *Spatio-temporal logic for 2d multi-agent problem domains*, Expert Systems with Applications **12** (1997), no. 1, 141–145.

[130] Antony Galton, *Machines and thought: The legacy of alan turing*, Mind Association ocasional series, vol. 1, ch. The Church-Turing Thesis: Its Nature and Status, pp. 137–164, Oxford University Press, 1996.

[131] L. T. F. Gamut, *Logic, language and meaning*, vol. 2 Intensional Logic and Logical Grammar, The University of Chicago Press, 1991.

[132] A. V. Gelder, K. A. Ross, and J. S. Schlipf, *Well-founded semantics for general logic programs*, Journal of the ACM **38** (1991), no. 3, 619–649.

[133] Michael Gelfond and Vladimir Lifschitz, *Logic programs with classical negation*, Proceedings of 7th International Conference on Logic Programming (Cambridge MA), The MIT Press, 1990, pp. 579–597.

[134] ———, *Classical negation in logic programs and disjunctive databases*, New Generation Computing. Ohmsha Ltd and Spring-Verlag (1991), 365–385.

[135] Michael Gelfond, Halina Przymusinska, and Teodor C Przymusinski, *The extended closed world assumption and its relationship to parallel circumscription*, Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, ACM, 1986, pp. 133–139.

[136] Dedre Gentner, Keith J. Holyoak, and Boicho N. Kokinov (eds.), *The analogical mind: perspectives from cognitive science*, The MIT Press, 2001.

[137] Carlo Ghezzi and Mehdi Jazayeri, *Programming language concepts*, third ed., John Wiley & Sons, 1998.

[138] Donald Gillies, *Artificial intelligence and scientific method*, ch. 4 A new framework for logic, pp. 72–97, Oxford University Press, 1996.

[139] ———, *Artificial intelligence and scientific method*, ch. 1 The Inductivist Controversy, or Bacon versus Popper, pp. 1–16, Oxford University Press, 1996.

[140] ———, *Artificial intelligence and scientific method*, ch. 5 Can there be an inductive logic?, pp. 98–112, Oxford University Press, 1996.

[141] M. Ginsberg, *Essentials of artificial intelligence*, Morgan Kaufmann Publishers, San Mateo, USA, 1993.

[142] Jean-Yves Girard, *Linear logic*, Theoretical Computer Science **50** (1987), 1–102.

[143] Jean-Yves Girard, Paul Taylor, and Yves Lafont, *Proofs and types*, Cambridge University Press, 1993.

[144] Kurt Gödel, *The undecidable - basic papers on undecidable propositions, unsolvable problems and computable functions*, ch. On Formally Undecidable Propositions of Principia Mathematica and Related Systems I, pp. 4–38, Raven Press, Hewlett, New York, 1965.

[145] Andrew D. Gordon, *Functional programming and input/output*, Distinguished Dissertations in Computer Science, Cambridge University Press, 1994.

[146] ———, *Bisimilarity as a theory of functional programming*, Theoretical Computer Science **228** (1999), no. 1–2, 5–47.

[147] James Gosling, Bill Joy, and Guy Steele, *The java language specification*, Addison Wesley Publishing Company, 1996.

[148] R. S. Gray, *Agent tcl: A transportable agent system*, Proceedings of the CIKM'95 Workshop on Intelligent Information Agent, 1995.

[149] Robert S. Gray, *Agent tcl: A flexible and secure mobile-agent system*, Tech. Report PCS-TR98-327, Dartmouth College, Computer Science, Hanover, NH, January 1998, Ph.D. Thesis, June 1997.

[150] C.A. Gunter and D.S. Scott, *Handbook of theoretical computer science*, vol. B Formal Models and Semantics, ch. 12 Semantic Domains, pp. 633–674, The MIT Press/Elsevier, 1990.

[151] Carl A. Gunter, *Semantics of programming languages: structures and techniques*, Foundations of Computing Series, The MIT Press, 1992.

[152] Anil Gupta and Nuel Belnap Junior, *The revision theory of truth*, ch. 2 Fixed Points: Some Basic Facts, p. 43, The MIT Press, 1993.

[153] Ian Hacking, *What is a logical system?*, Studies in Logic and Computation, no. 4, ch. What Is Logic?, pp. 1–33, Claredon Press, Oxford University, 1994.

[154] Matthew Hennesy and James Riely, *Type–safe execution of mobile agents in anonymous networks*, Secure Internet Programming: Security Issues for Mobile and Distributed Objects, and also Proceedings of the Workshop on Internet Programming Languages (WIPL 1998), Lecture Notes in Computer Science, no. 1603, Springer-Verlag, Berlin Germany, 1999.

[155] Rolf Herken (ed.), *The universal turing machine: A half-century survey*, Oxford University Press, 1988.

[156] Arend Heyting, *Intuitionism: An introduction*, second revised ed., Studies in Logic and the Foundations of Mathematics, North-Holland Publishing Company, Amsterdam, 1966.

[157] Juha Honkala, *On parikh slender context-free languages*, Theoretical Computer Science **255** (2001), no. 1–2, 667–677.

[158] John F. Horty, *Agency and deontic logic*, Oxford University Press, 2001.

[159] Paul Hudak, *The haskell school of expression: Learning functional programming through multimedia*, Cambridge University Press, 2000.

[160] Michael N. Huhns and Munindar P. Singh (eds.), *Readings in agents*, Morgan Kaufmann, San Francisco, California, 1997.

[161] Thomas W. Hungerford, *Algebra*, Graduate Texts in Mathematics, vol. 73, Springer-Verlag, 1974, Corrected eighth printing, 1996.

[162] Leon Hurst, Pádraig Cunningham, and Fergal Sommers, *Mobile agents — smart messages*, Proceedings of the 1st International Workshop on Mobile Agents (Berlin, Germany), April 1997.

[163] O. Ibidapo-Obe, O. S. Asaolu, and A. B. Badiru, *Generalized solutions of the pursuit problem in three-dimensional euclidean space*, Applied Mathematics and Computation **119** (2001), no. 1, 35–45.

[164] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes, *Lua - an extensible extension language*, Software: Practice and Experience **26** (1996), no. 6.

[165] Roberto Incitti, *The growth function of context-free languages*, Theoretical Computer Science **255** (2001), no. 1–2, 601–605.

[166] Katsumi Inoue and Chiaki Sakama, *Negation as failure in the head*, The Journal of Logic Programming **35** (1998), 39–78, North-Holland.

[167] Institute of Electrical and Electronics Engineers, *Ieee standard for binary floating-point arithmetic, ansi/ieee standard 754-1985*, 1985.

[168] Christian S. Jensen, Markus Schneider, Bernhard Seeger, and Vassilis J. Tsotras (eds.), *Advances in spatial and temporal databases*, Lecture Notes in Computer Science, vol. 2121, Springer, July 2001.

[169] Dag Johansen, *Mobile agent applicability*, Mobile Agents: Second International Workshop, MA'98, Lecture Notes in Computer Science, vol. 1477, Springer, 1998, pp. 80–98.

[170] Dag Johansen, Robbert van Renesse, and Fred B. Schneider, *An introduction to the TACOMA distributed system*, Tech. Report 95-23, Department of Computer Science, University of Tromsø, Tromsø, Norway, June 1995.

[171] Neil D. Jones, *Computability theory: An introduction*, ACM Monograph Series, Academic Press, New York and London, 1973.

[172] ———, *Computability and complexity: from a programming perspective*, Foundations of Computing, The MIT Press, 1997.

[173] Simon Peyton Jones, *The implementation of functional programming languages*, Prentice-Hall International Series in Computer Science, Prentice-Hall, Inc., 1987.

[174] M. P. Jordan, *The power of negation in english: Text, context and relevance*, Journal of Pragmatics **29** (1998).

[175] Carl Jung et al., *Man and his symbols*, Adus Books Ltda, Pan MacMillan, 20 New Wharf Road, London N1 9RR, 1964, Conceived and Edited by Carl Jung. Newer edition published by Picador, in 1978.

[176] Carl Gustav Jung and Anthony Storr, *Jung: Selected writings*, second ed., Fontana Pocket Readers, ch. Psychological Typology (1936), pp. 133–146, Fontana Paperbacks, 1986, Selected and Introduced by Anthony Storr.

[177] A. C. Kakas, R. A. Kowalski, and F. Toni, *The role of abduction in logic programming, in handbook of logic in artificial intelligence and logic programming*, vol. 5. Logic Programming, pp. 235–324, Oxford University Press, 1998.

[178] Immanuel Kant, *Crítica da razão pura*, Ouro, Martin Claret, 1781 and 2002, Translation into Portuguese from the original "Kritik der Reinen Vernunft".

[179] ———, *Logic*, Bobbs-Merrill, Indianapolis, 1974, Original in German Logik. Translation with an introduction by Robert S Hartman and Wolfgang Schwarz.

[180] Immanuel Kant and translation by Norman K. Smith, *Immanuel kant's critique of pure reason*, Macmillan Press Ltd, 1787, 1929.

[181] Gnter Karjoth, Danny B. Lange, and Mitsuru Oshima, *A security model for agents*, IEEE Internet Computing **1** (1997), no. 4.

[182] Michael J. Kearns and Umesh V. Vazirani, *An introduction to computational learning theory*, The MIT Press, 1994.

[183] Anthony John Patrick Kenny, *A brief history of western philosophy*, Blackwell Publishers, 1998.

[184] Fred N. Kerlinger and Howard B. Lee, *Foundations of behavioral research*, fourth ed., Harcourt College Publishers, 2000.

[185] B. Kirkerud, *Programming language semantics*, International Thomson Computer Press, 1997.

[186] Stephen C. Kleene, *Introduction of metamathematics*, D. Van Nostrand, Princeton, 1952.

[187] J. W. Klop, *Handbook of logic in computer science*, vol. 2 Background: Computational Structures, ch. Term Rewriting Systems, pp. 1–116, Oxford University Press, 1992.

[188] Frederick C. Knabe, *Language support for mobile agents*, Ph.D. thesis, Carnegie Mellon University, Paittsburgh, Pa., December 1995, Also available as Carngie Mellon School of Computer Science Technical Report CMU-CS-95-223 and European Computer Industry Centre Technical Report ECRC-95-36.

[189] Bart Kosko, *Fuzzy thinking: The new science of fuzzy logic*, HarperCollinsPublishers, Flamingo, 1994.

[190] Konrad B Krauskopt and Arthur Beiser, *The physical universe*, eighth ed., McGraw-Hill Companies, Inc, 1997.

[191] Henry Kyburgh Junior, *Handbook of logic in artificial intelligence and logic programming*, vol. 3: Nonmonotonic Reasoning and uncertain reasoning, ch. Uncertainty Logics, pp. 397–438, Oxford University Press, 1994.

[192] Leslie Lamport and Nancy Lynch, *Handbook of theoretical computer science*, vol. B Formal Models and Semantics, ch. 18 Distributed Computing: Models and Methods, pp. 1157–1199, The MIT Press/Elsevier, 1990.

[193] Saunders Mac Lane, *Categories for the working mathematician*, second ed., Graduate texts in mathematics, Springer, 1998, Previous edition: 1971.

[194] Danny B. Lange and Mitsuru Ishima, *Program and deploying java mobile agents with aglets*, Addison-Wesley, 1998.

[195] Ducan Langford (ed.), *Internet ethics*, MacMillan Press Ltd, Printed and Bound in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire, 2000.

[196] Jonathan Laventhol, *Programming in pop-11*, Artificial Intelligence Texts, Blackwell Scientific Publications Ltd, 1987.

[197] F. William Lawvere and Stephen Hoel Schanuel, *Conceptual mathematics: a first introduction to categories*, Cambridge University Press, 1997.

[198] Harry Lewis and Christos H. Papadimitriou, *Elements of the theory of computation*, second ed., Prentice-Hall, Inc., September 1997.

[199] Tim Lindholm and Frank Yellin, *The java virtual machine specification*, Addison-Wesley Publishing Company, Reading, Massachussetts, 1997.

[200] Zhaohui Luo, *Computation and reasoning: a type theory for computer science*, International Series of Monographs on Computer Science, edited by Gabbay, Hopcroft, Plotkin, Schwartz, Scott, Vuillemin and Galil, vol. 11, Oxford Science Publications, 1994.

[201] William G. Lycan, *Real conditionals*, Oxford University Press, 2001.

[202] Penelope Maddy, *The philosophy of mathematics*, Oxford Readings in Philosophy, ch. Perception and Mathematical Intuition, Oxford University Press, 1996.

[203] Grzegorz Malinowski, *Many-valued logics*, Oxford Logic Guides, no. 25, Claredon Press, Oxford University, 1993.

[204] Maurice Margenstern, *On quasi-unilateral universal turing machines*, Theoretical Computer Science **257** (2001), no. 1–2, 153–166.

[205] Kim Marriott and Peter J. Stuckey, *Programming with constraints: An introduction*, The MIT Press, 1998.

[206] Gerald Masini, Amedeo Napoli, Dominique Colnet, Daniel Leonard, and Karl Tombre, *Object-oriented languages*, A.P.I.C., no. 34, Academic Press, 1991, original in French: Les languages à objects by InterEditions 1989.

[207] Alexandru Mateescu and Arto Salomaa, *Hanbook of formal languages*, vol. 1, ch. Aspects of Classical Language Theory, pp. 175–251, Springer-Verlag, 1997.

[208] B. Mathiske, F. Matthes, and J. W. Schmidt, *On migrating threads*, Tech. report, Fachbereich Informatik Universitat Hamburg, 1994.

[209] John McCarthy, *Defending ai research: a collection of essays and reviews*, CSLI Lecture Notes, no. 49, CSLI Publications: Center for the Study of Language and Information, 1996.

[210] K. Meinke and J. V. Tucker, *Handbook of logic in computer science*, vol. 1: Mathematical Structures, ch. Universal Algebra, pp. 189–411, Oxford University Press, 1992.

[211] Robin Milner, *Computing tomorrow: future research directions in computer science*, ch. Semantic Ideas in Computing, pp. 246–283, Cambridge University Press, 1996.

[212] Robin Milner, Joachim Parrow, and David Walker, *A calculus of mobile processes. part* I *and* II, Information and Computation **1** (1992), no. 100.

[213] Dejan S. Milojíčić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou, *Process migration*, ACM Computing Surveys **32** (2000), no. 3, 241–299.

[214] Marvin Minsky, *Computation: Finite and infinite machines*, Prentice-Hall Series in Automatic Computation, Prentice-Hall International, Inc. London, 1972, Original American publication by Prentice-Hall Inc. 1967.

[215] ———, *A framework for representing knowledge*, Tech. report, Massachusetts Institute of Technology, Artificial Intelligence Laborator, 1974.

[216] Miguel Mira da Silva, *Mobility and persistence*, Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag, April 1997, Lecture Notes in Computer Science No. 1222, pp. 157–176.

[217] Miguel Mira da Silva and Malcolm Atkinson, *Combining mobile agents with persistent systems: Opportunities and challenges*, 2nd ECOOP Workshop on Mobile Object Systems (Linz, Austria), July 1996, pp. 36–40.

[218] John C. Mitchell, *Foundations for programming languages*, Foundations of Computing, ch. The Language PCF, pp. 45–144, The MIT Press, 1996.

[219] ———, *Foundations for programming languages*, Foundations of Computing, The MIT Press, 1996.

[220] ———, *Programming language methods in computer security*, Proceedings of the POPL 2001, XXVIII ACM SIGPLAN - SIGACT, Symposium on Principles of Programming Languages, ACM SIGPLAN, SIGACT, 2001, Also SIGPLAN Notices 36(3):1–3, pp. 1–3.

[221] François Monin and Marianne Simonot, *An ordinal measure based procedure for termination of functions*, Theoretical Computer Science **254** (2001), no. 1–2, 63–94.

[222] Rebecca Montanari, Cesare Stefanelli, and Naranker Dulay, *Flexible security policies for mobile agent systems*, Microprocessors and Microsystems **25** (2001), no. 2, 85–92.

[223] C. Moss, *Prolog++ the power of object-oriented and logic programming*, Addison-Wesley Publishing Company, 1994.

[224] A. Mycroft, *On integration of programming paradigms*, Computing Survey **28** (1996), no. 2, http://www.cl.cam.ac.uk/users/am/papers/.

[225] G. Nadathur and D. Miller, *An overview of λprolog*, Proceedings of the 5th International Conference on Logic Programming (Cambridge, MA), The MIT Press, 1989.

[226] Sara Negri and Jan Von Plato, *Structural proof theory*, Cambridge University Press, 2001.

[227] Anil Nerode and Richard A. Shore, *Logic for applications*, second ed., Graduate Texts in Computer Science, Springer-Verlag New York Inc., 1997.

[228] Hanne Riis Nielson and Flemming Nielson, *Semantics with applications: a formal introduction*, John Wiley & Sons, 1993.

[229] ObjectSpace Corp., *Voyager white paper*, 1998.

[230] Zoran Ognjanovic and Miodrag Raškovic, *Some first-order probability logics*, Theoretical Computer Science **247** (2000), no. 1–2, 191–212.

[231] C.-H. L. Ong, *Handbook of logic in computer science*, vol. 4: Semantic Modelling, ch. Correspondence Beteen Operational and Denotational Semantics: the full abstraction problem for PCF, pp. 269–356, Oxford University Press, 1995.

[232] J. K. Ousterhout, *Tcl and the tk toolkit*, Adison-Wesley, 1994.

[233] David E. Over, *Models and computability*, London Mathematical Society Lecture Notes, vol. 259, ch. Logic and Decision Making, pp. 313–338, Cambridge University Press, 1999.

[234] Christos H. Papadimitriou, *Computational complexity*, Addison-Wesley Publishing Company, 1995, Reprinted with corrections.

[235] Charles Parsons, *The philosophy of mathematics*, Oxford Readings in Philosophy, ch. Mathematical Intuition, Oxford University Press, 1996.

[236] Lawrence C. Paulson, *Ml for the working programmer*, second ed., Cambridge University Press, 1996.

[237] Z. Pawlak, *Rough sets*, International Journal Comput. Inform **11** (1982), 341–356.

[238] Luís Moniz Pereira, *Non-monotonic extensions of logic programming*, Proceedings of the Second International Workshop on Non-monotonic Extensions of Logic (Bad Honnef, Germany), 1996.

[239] I. C. C. Phillips, *Handbook of logic in computer science*, vol. 1: Mathematical Structures, ch. Recursion Theory, pp. 79–187, Oxford University Press, 1992.

[240] Gian Pietro Picco, *Mobile agents: an introduction*, Microprocessors and Microsystems **25** (2001), no. 2, 65–74.

[241] Benjamin C. Pierce, *Basic category theory for computer scientists*, Foundations of Computing Series, The MIT Press, 1993, Second print.

[242] Axel Poigné, *Handbook of logic in computer science*, vol. 1: Mathematical Structures, ch. Basic Category Theory, pp. 413–640, Oxford University Press, 1992.

[243] Lech Polkowski and Andrzej Skowron (eds.), *Rough sets and current trends in computing: first international conference / rsctc'98*, Lecture Notes in Artificial Intelligence, vol. LNAI 1424, Springer, Warsaw, Poland, June 1998, Proceedings.

[244] David Poole, *Abducing through negation as failure: Stable models within the independent choice logic*, Journal of Logic Programming **44** (1999), 5–35.

[245] Karl Popper, *The logic of scientific discovery*, Karl Popper, 1972.

[246] Arthur Prior, *Formal logic*, Oxford University Press, 1955.

[247] ———, *Past, present and future*, Oxford University Press, 1967.

[248] ———, *Logic and reality: essays on the legacy of arthur prior*, ch. Two Essays on Temporal Realism, pp. 43–51, Oxford University Press, 1996.

[249] Sanguthevar Rajasekaran and Panos Pardalos (eds.), *Mobile networks and computing: Dimacs workshop 99*, DIMACS series in discrete mathematics and theoretical computer science, vol. 52, American Mathematical Society, March 2000.

[250] David Randell, Zhan Cui, and Tony Cohn, *A spatial logic based on regions and connection*, Proceedings 3rd International Conference on Knowledge Representation and Reasoning (San Mateo), Morgan Kaufmann, 1992, pp. 165–176.

[251] Daniel W. Rasmus, *Rethinking smart objects: Building artificial intelligence with objects*, Advances in Object Technology Series, vol. 18, Cambridge University Press and SIGS Books, 1999.

[252] Stephen Read, *Relevant logic: a philosophical examination of inference*, Basil Blackwell, 1988.

[253] R. Reiter, *Logic and data bases*, ch. On Close World Data Bases, pp. 55–76, Plenum Press, New York, 1978.

[254] Greg Restall, *An introduction to substructural logics*, Routledge, 2000.

[255] Elaine Rich and Kevin Knight, *Artificial intelligence*, second ed., McGraw-Hill, Inc., 1991.

[256] John F. Roddick, Kathleen Hornsby, and Myra Spiliopoulou, *An updated bibliography of temporal, spatial, and spatio-temporal data mining research*, Temporal, Spatial and Spatio-Temporal Data Mining, Lecture

Notes in Artificial Intelligence, vol. LNAI 2007, Springer, September 2000, pp. 147–163.

[257] Rita Rodriguez and Frank Anger, *Logic and reality: essays on the legacy of arthur prior*, ch. Prior's Temporal Legacy in Computer Science, pp. 89–105, Oxford University Press, 1996.

[258] Kurt Rothermel and Fritz Hohl (eds.), *Mobile agents: Second international workshop / ma'98*, Lecture Notes in Computer Science, vol. 1477, Springer, Stuttgart, September 1998, Proceedings.

[259] Grzegorz Rozenberg and Arto Salomaa (eds.), *Handbook of formal languages*, vol. 1, Springer-Verlag, 1997.

[260] Paul Ruet and François Fages, *Combining explicit negation and negation by failure via belnap's logic*, Theoretical Computer Science **171** (1997), 61–75.

[261] Bertrand Russell, *Introduction to mathamatical philosophy*, Routledge, 1919.

[262] ———, *History of western philosophy*, vol. I, part 2, pp. 101–226, Routledge, 1946, Edition published in 2000.

[263] ———, *History of western philosophy*, vol. III, part 2, ch. 18, The Romantic Movement, pp. 651–659, Routledge, 1946, Edition published in 2000.

[264] Mark Ryan and Martin Sadler, *Handbook of logic in computer science*, vol. 1: Mathematical Structures, ch. Valuation Systems and Consequence Relations, pp. 1–78, Oxford University Press, 1992.

[265] Yu V Sachkov, *International congress of logic, methodology, and philosophy of science (6th : 1979: Hannover): Logic, methodology, and philosophy of science*, Studies in Logic and the Foundations of Mathematics,

vol. 104, ch. Foundations and Philosophy of the Physical Sciences, Probability in Classical and Quantum Physics, pp. 441–447, PWN-Polish Scientific Publishers-Warszawa and North Holland Publishing Company, 1982.

[266] Vijay Saraswat and Pascal Van Hentenryck (eds.), *Principles and practices of constraint programmming*, The MIT Press, 1995.

[267] David A. Schmidt, *The structure of typed programming languages*, Foundations of Computing Series, The MIT Press, 1994.

[268] Dietmar Seipel, *Logic programming and knowledge representation: Third international workshop / lpkr'97*, Lecture Notes in Artificial Intelligence, vol. 1471, ch. Partial Evidential Stable Models for Disjunctive Deductive Databases, pp. 66–83, Springer, New York, October 1998.

[269] Ravi Sethi, *Programming languages: concepts & constructs*, second ed., Addison-Wesley Publishing Company, 1996.

[270] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce, *Location-independent communication for mobile agents: a two-level architecture*, Internet Programming Languages, Lecture Notes in Computer Science, no. 1686, Springer-Verlag, 1998, pp. 1–31.

[271] Rajjan Shinghal, *Formal concepts in artificial intelligence: fundamentals*, Chapman & Hall Computing Series, ch. 10 Plausible Reasoning in Expert Systems, pp. 390–452, Chapman & Hall Computing, 1992.

[272] E. H. Shortlife, *Computer-based medical consultations: Mycin*, New York, 1976, Elsevier.

[273] Hebert Simon, *Machines and thought: The legacy of alan turing*, Mind Association ocasional series, vol. 1, ch. Machine as Mind, pp. 81–102, Oxford University Press, 1996.

[274] Michael Sipser, *Introduction to the theory of computation*, PWS Publishing Company, 1997.

[275] Kenneth Slonneger and Barry L. Kurtz, *Formal syntax and semantics of programming languages: a laboratory based approach*, Addison-Wesley Publishing Company, 1995.

[276] Raymond M. Smullyan, *Gödel incompleteness theorems*, Oxford Logic Guides, no. 19, Oxford University Press, 1992.

[277] Ernest Sosa and Jaegwon Kim (eds.), *Epistemology: An anthology*, Blackwell Philosophy Anthologies, vol. 11, Blackwell Publishers, 2000, With assistance of Matthew McGrath.

[278] John F. Sowa, *Knowledge representation: logical, philosophical, and computational foundations*, Brooks/Cole, 511 Forest Lodge Road, Pacific Grove,CA, 2000.

[279] Benedict de Spinoza, *Ethics*, Penguin Classics, Penguin Group, this edition 1996.

[280] Leon Sterling and Ehud Shapiro, *The art of prolog: advanced programming techniques*, second ed., MIT Press series in logic programming, The MIT Press, 1994.

[281] N. I. Styazhkin, *History of mathematical logic from leibniz to peano*, The M.I.T. Press, 1969.

[282] Marshall Swain (ed.), *Induction, acceptance, and rational belief*, D. Reidel Publishing Company, 1970.

[283] Alfred Tarski, *Logic, semantics, metamathematics*, Oxford University Press, 1956.

[284] Beverley Tasker, *The logic of space-time: (zero infinity becoming)*, The Loebertas Series of Philosophical Monographs, vol. 22, Loebertas, 1998.

[285] R Gregory Taylor, *Models of computation and formal languages*, Oxford University Press, 1998.

[286] R. D. Tennent, *Semantics of programming languages*, PHI series in computer science, Prentice Hall, Inc., 1991.

[287] ———, *Handbook of logic in computer science*, vol. 3: Semantic Structures, ch. Denotational Semantics, pp. 170–322, Oxford University Press, 1994.

[288] Do Van Thanh, Sverre Steensen, and Jan A. Audestad, *Mobility management and roaming with mobile agents*, Mobile and Wireless Communications Networks, Lecture Notes in Computer Science, no. 1818, Springer-Verlag Berlin Heidelberg, 2000, pp. 123–137.

[289] Pete Thomas and Ray Weedon, *Object-oriented programming in eiffel*, Addison-Wesley Publishing Company, 1995.

[290] Simon Thompson, *Laws in miranda*, ACM Communications **2** (1986), no. 3.

[291] ———, *Haskell: The craft of functional programming*, second ed., Addison-Wesley Publishing Company, 1999, Paperback.

[292] Bent Thomsen, Lone Leth, Sanjiva Prasad, Tsung-Min Kuo, Andre Kramer, Fritz C. Knabe, , and Alessandro Giacalone, *Facile antigua release programming guide*, Tech. Report ECRC-93-20, European Computer Industry Research Centre, Munich, Germany, December 1993.

[293] Chris Thornton and Benedict du Boulay, *Artificial intelligence through search*, Kluwer Academic Publishers and Intellect Books, 1992.

[294] Anne Sjerp Troelstra, *Lectures on linear logic*, CSLI lecture notes, no. 29, Center for the Study of Language and Information, CSLI/SRI International, 1992.

[295] Anne Sjerp Troelstra and H. Schwichtenberg, *Basic proof theory*, Cambridge Tracts in Theoretical Computer Science, vol. 43 Basic Proof Theory, Cambridge University Press, 1996.

[296] J. K. Truss, *Discrete mathematics for computer science*, International Computer Science Series, Addison-Wesley Publishing Company, 1991.

[297] Christian Tschudin, *The messenger environment M0 – a condensed description*, Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag, April 1997, Lecture Notes in Computer Science No. 1222, pp. 149–156.

[298] J. V. Tucker and J. I. Zucker, *Handbook of logic in computer science*, vol. 5: Logic and Algebraic Methods, ch. Computable Functions and Semicomputable Sets on Many-Sorted Algebras, pp. 317–523, Oxford University Press, 2000.

[299] Alan M. Turing, *Computability and λ-definability*, Journal of Symbolic Logic **2** (1936), 153–163.

[300] ———, *The undecidable - basic papers on undecidable propositions, unsolvable problems and computable functions*, ch. On Computable Numbers, With an Application to the Entscheidungsproblem, pp. 115–151, Raven Press, Hewlett, New York, 1965, A correction, pages 152–154.

[301] Asis Unyapoth and Peter Sewell, *Nomadic pict: Correct communication infrastructure for mobile computation*, Proceedings of the POPL 2001, XXVIII ACM SIGPLAN - SIGACT, Symposium on Principles of Programming Languages, ACM SIGPLAN, SIGACT, 2001, Also SIGPLAN Notices 36(3):116–127, pp. 116–127.

[302] Alasdair Urquhart, *Handbook of philosophical logic*, Synthese library; v. 166, vol. 3: Alternatives to Classical Logic, ch. Many-Valued Logic, pp. 71–116, Kluwer Academic Publishers, 1986.

[303] Franck van Breugel, *An introduction to metric semantics: operational and denotational models for programming and specification languages*, Theoretical Computer Science **258** (2001), no. 1–2, 1–98.

[304] Jean van Heijenoort, *From frege to gödel*, Harvard University Press, 1967.

[305] Michalis Vazirginiannis and Ouri Wolfson, *A spatiotemporal model and language for moving objects on road networks*, 7th International Symposium, SSTD 2001, Springer, July 2001, LNCS, 2121, pp. 20–35.

[306] Daniel J. Velleman, *How to prove it*, Cambridge University Press, 1994.

[307] Jan Vitek and Giuseppe Castagna, *Seal: A framework for secure mobile computation*, Internet Programming Languages, Lecture Notes in Computer Science, no. 1686, Springer-Verlag, 1998, pp. 47–77.

[308] Jan Vitek, Manuel Serrano, and Dimitri Thanos, *Security and communication in mobile object systems*, Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag, April 1997, Lecture Notes in Computer Science No. 1222, pp. 177–200.

[309] Dennis Volpano, *Provably-secure programming languages for remote evaluation*, ACM Computing Surveys **28A** (1996), Participation statement for ACM Workshop on Strategic Directions in Computing Research.

[310] Philip Wadler, *How to declare an imperative*, ACM Computing Surveys **29** (1997), no. 3, 240–263.

[311] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall, *A note on distributed computing*, Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag, April 1997, Lecture Notes in Computer Science No. 1222. Also published as Sun Microsystems Laborators Technical Report TR-94-29., pp. 49–64.

[312] R. F. C. Walters, *Categories and computer science*, Cambridge Computer Science Texts, Cambridge University Press, 1991.

[313] Ian Wand and Robin Milner (eds.), *Computing tomorrow: future research directions in computer science*, Cambridge University Press, 1996.

[314] D. S. Warren, *Web page*, http://www.cs.sunysb.edu/~warren/ (1998).

[315] David A. Watt, *Programming language concepts and paradigms*, Series in Computer Science, Prentice-Hall, Inc., 1990.

[316] J. White, *Telescript technology: the foundation for the electronic marketplace*, General Magic, Inc., 1994.

[317] James E. White, *Telescript technology: Mobile agents*, 1996, Also available as General Magic White Paper.

[318] Glynn Winskel, *The formal semantics of programming languages: an introduction*, fourth ed., The MIT Press, 1997.

[319] ———, *The formal semantics of programming languages: An introduction*, Foundations of Computing, The MIT Press, 1997.

[320] Ann Yasuhara, *Recursive function and logic*, Academic Press, Inc, 1971.

[321] Justin Zobel, *Writing for computer science: the art of effective communication*, ch. 5 Mathematics, Springer-Verlag Singapore Pte. Ltd., 1997.