PROBABILISTIC FINITE DOMAINS

NICOS ANGELOPOULOS

PH.D. THESIS

DEPARTMENT OF COMPUTER SCIENCE
CITY UNIVERSITY

April 9, 2001

THIS THESIS IS SUBMITTED AS PART OF THE REQUIREMENTS FOR A PH.D. IN COMPUTER SCIENCE, IN THE DEPARTMENT OF COMPUTER SCIENCE OF CITY UNIVERSITY, LONDON, UK.

Contents

List of Figures

List of Tables

Acknowledgements

For the first three years of this research I was supported my Centenary Scholarship by the School of Informatics at the City University (London, UK). My supervisor was Dr. David R. Gilbert of the Computer Science Department, whom I would like to thank for making me aware of the aforementioned Scholarship and for bringing my proposal to the Research Committee's attention.

During those three years at City I was fortunate to interact with two distinguished colleagues. Dr. Herbert Wiclicky was an inspiring influence and his presence a major reason for my decision to incorporate probabilistic notions in this thesis. Dr. Andreas Hadjiprokopis has been a researcher keen to discuss novel approaches and a source of vast technical knowledge and ability; whose approach to the day-to-day research was an inspiring factor in itself.

Last but not least I would like to thank Tarhata Orlanes for (among many other things) her advice on style and her invaluable corrections on the prose.

N.A.

London
August 2000

PS. I would also like to thank my external examiners; Dr James Cussens (York University) for his meticulous corrections and invaluable probabilistic comments, and Dr Mark Wallace (Imperial College) for his suggestions and observations on search algorithms and constraint research.

April 2001

Declaration

Abstract

This thesis presents a set of programming constructs that are capable of modelling probabilistic concepts and of computing with such concepts. The main objectives are to provide: a theoretically sound, practically achievable and notationally intuitive formalism. The probabilistic programming constructs are presented in the form of a system called probabilistic finite domains, which enhances the Logic Programming paradigm with a novel constraint solver. In doing so, we are able to take advantage of the knowledge representation power of probability. In particular we investigate: first, the duality of the two interpretations of probability to the problems researchers face when wishing to create a probabilistic formalism and second, the use of probability as a unifying model for computational derivations. Some programming examples and a simple implementation are also described.

Chapter 1

Introduction

The research described in this thesis is concerned with the study of probabilistic concepts in some computational context. Since this is a broad description that encompasses a variety of hybrid approaches, our aim in this introductory chapter is to give the basic theories and technologies that are pertinent to our work.

The objective of amalgamating these technologies is the formalisation of computation over quantities. We will thus continue our exposition by presenting some relevant research efforts from the literature. This also serves as motivation and background material, helping the reader to gain a basic appreciation of the technical issues involved.

With the broad boundaries of our study defined, we then present a new probabilistic formalism, which encompasses knowledge representation and computational methods capable of dealing with uncertainty. Here we will refer to this formalism as probabilistic finite domains (*Pfd*, for short). The constructs of *Pfd* are introduced as a set of primitives which are succinctly presented within a constraint Logic Programming setting. We will provide some justification for the choice of the particular paradigm and comment on the ability to embed these constructs to other paradigms.

The intended purpose for *Pfd*, is that of a conceptual tool for sensible automated reasoning for probabilistic problems. Where by sensible, we refer to a first step towards intelligence. The most important objective in our work has been the creation of an effective computational mechanism.

The sections in this Chapter are brief expositions of the various domains and technologies that are pertinent to this thesis. We also use them to introduce some terminology and allow clarifications of overloaded terms.

1.1 Probability Theory

By Probability Theory (PT) we will refer to the axiomatisation brought together by Kolmogorov ([Kol33]). In particular, the emphasis is placed in viewing Probability Theory as a mathematical tool, while avoiding to place onto its derivative terms any logical or philosophical connotations.

Logical and philosophical issues are of paramount importance when one considers the modelling and problem solving capabilities of formalisms based on probability. However, we choose to view the axiomatic definition of probability as a machinery for mathematical reasoning (much like we view Logic). This machinery in itself is not responsible for providing translations between the real-world and the mathematical objects. Although this is considered self evident in some circles, it is by no means the universal consensus of viewing or understanding probability. On the contrary, the attempts of computer scientists, seem to follow steps parallel to the ones taken by physicists and mathematicians, prior to the inception of Probability Theory, that is, when the theoretical explorations where very much interpretation driven.

Our exposition on interpretations and knowledge representation, using probability, assumes some familiarity with [Rei71] and [Car62]. These texts also place the development of Probability Theory in historical perspective (for a rigorous coverage on the historical perspective see [vP94]). This is important for our understanding of PT since it is the product of a long process of bringing together various fragments of research.

This thesis is self sufficient, in that it introduces most of the probabilistic concepts, that are necessary for a reader with a computer science background to follow. More general knowledge as discussed in most standard Probability Theory texts (such as [Fel59, Pap91]) may lead to much deeper appreciation of the issues involved and of their importance.

Even-though probability is a widely used modelling tool, there are strong reasons to believe that people have difficulties in reasoning with concepts involving probability based uncertainty ([GO94]). This problem becomes more acute since our objective is to provide a programming framework. This adds an extra complicating factor since the programmer needs to have a particular execution strategy in mind.

1.2 Logic for KR and Problem Solving

Logic is the single formalism that has the longest and strongest presence in Knowledge Representation (KR). Despite the fact that a number of alternatives have been suggested, both during the early development of the field and more recently, Logic is the stronghold at the common ground of a lot of research in Knowledge Representation.

Even more pertinent to our programming language orientation is Logic Programming, ([Kow79]) which notably bridges the space between Knowledge Representation and effective problem solving.

In this work we have diverted from purist approaches in embedding Probabilities in Logic, by virtue of placing such reasoning at the fringes i.e. constraint solving. This has been proven to be a fruitful combination strategy. (The technical argumentation for this diversion is in Chapter 2.) The integration of the two theories, has been considered as highly desirable by many researchers, and naturally a number of approaches have been suggested. Our contribution is a seamless and bi-directional symbiosis for the theories of Logic and Probability.

1.3 Modelling Uncertainty

Given the importance of Logic and PT, it is not a surprise that there exists a plethora of attempts which seek to integrate them. These range from purely theoretical abstractions to ad hoc expert systems. We will discuss some of the most well known approaches as a way of charting the relevant research areas, while pointing out the important issues in this field of study. However, we also include a brief discussion on formalisms which deals with uncertainty in non-probabilistic ways, in order to provide a more complete picture.

Here we give a basic categorisation for formalisms, which will be discussed in this thesis. It must be noted that the boundaries of the following areas are sometimes difficult to distinguish and often arbitrary.

Probabilistic

- Logics
  - Artificial Intelligence [Nil86]
  - Satisfiability [GKP88]
  - Statistical Knowledge [Bac90a]

- Logic Programming
  - Uncertainties [Sha83]
  - Quantitative Deduction [vE86]
  - Bilattices and Semantics [Fit91]
  - Horn Abduction and Bayesian Networks [Poo93b]

Bayesian Systems

- Evidential Support Logic Programming [Bal87]
- Bayesian Networks [Pea88]
- Expert systems [Nea90] (survey)
- Bayesian rule-based [DHN81]

Non Probabilistic

- Dempster-Shafer Theory [Sha76]
- Fuzzy Logic(s) [DP80, Zad65]

This thesis will also argue and try to establish, that the role of probability in computing with uncertainty, is analogous to logic's role in problem solving.

1.4 CLP

We have used and extended the CLP(X) model ([JL87]) (and more specifically CLP(FD)) for the experimental part of this work.

The particular choice provides :

- an abstract means for presenting the probabilistic reasoner,
- seamless movement between our preferred theories, (Logic and PT)
- a well defined and understood interaction between the two theories,
- support of effective programming environments,
- minimal effort in migrating to other programming paradigms and languages.

In particular, the clarity of separation and the ability to access both Logic and Probability Theory, from a single programming paradigm, have been invaluable not only in furthering the ideas of duality, comparison and integration of the two theories, but also in implementing evaluation prototypes within research constraints.

From a theoretical perspective, finite domains and CLP(FD) research ([Mac77, MR93, CD96]) allow us to cast the ingredients of the probability axioms to programming primitives. This is an important step, that allows the development of our probabilistic machinery to develop quite independently from any modelling restrictions. By modelling restrictions we mean the interpretations given to the probability values in the real world. Thus, we avoid the common practise of hardwiring a particular probabilistic interpretation to proposed programming constructs. The need for model-independent formalisms constitutes one of the central arguments of this thesis.

The extension we provide to the operational model of CLP continues on the tradition of generalising the interaction between the logic engine and the solver. The first wave of solvers treated the solver as a black box ([JMSY92]). These were followed by solvers which provided the ability to see through the solver (glass box approach) and better integration techniques ([DC93, CFS93]). The current trends include employment of more than one solvers (hybrid approaches, [WCJL+98]) and/or general systems for building constraint solvers ([Frü98, DBH+99]). With *Pfd* we introduce a solver that can be used for guiding computation (logic engine and constraints) in a general manner.

Finally, research from CLP has provided a convenient basis for the semantics of *Pfd*. Towards this end, we have utilised and extended concepts presented in [vE97]. Our alternative view of these concepts increases their applicability and maybe of importance to other CLP researchers.

1.5 Thesis Overview

The rest of the thesis is structured as follows. In Chapter 2 we give some of the fundamental results of Probability Theory and briefly review some of the existing research in the area of modelling and reasoning with uncertainty. Chapter 3 presents the syntax of probabilistic finite domains along with small examples that illustrate how the introduced constraints can be used to model uncertainty. The semantics of *Pfd* are then given in Chapter 4. This is followed by a number of example problem areas and a description of how they can be modelled in *Pfd*, in Chapter 5. These examples are then used: (a) to present a meta-interpreter that implements *Pfd* (Chapter 6) and (b) to provide some feedback on the general limitation of *Pfd* and the particular behaviour of the meta-interpreter (in Chapter 7). Finally, in Chapter 8 we give our concluding remarks and comment on possible directions for future work.

Chapter 2

Computing with Measures

In this chapter, we present the necessary background of the thesis. The first part describes and formalises our use of the term Probability Theory. We will state the axioms of the theory and some basic properties. While the later part is an overview of several existing methodologies which combine some computational model with notions of imprecision. This overview serves two main purposes. Firstly, as an implicit definition of the kind of issues we wish to address and secondly as the motivation for the rest of the work in this thesis.

2.1 Probability Theory

By Probability Theory we refer to the axiomatic definition put forward by Kolmogorov (in [Kol50]). The widespread acceptance of this formulation is primarily attributed to its independence from a particular interpretation. This was a novel property that was lacking from the preceding statistical theories. This quality of the theory plays a major role to this work, since our objective is to introduce a computational formalism, that although it provides probabilistic reasoning capabilities, still does not impose a particular interpretation on such reasoning.

2.1.1 Foundations

Following a statistical exposition, we will refer to an *experiment* as a basic (intuitive) concept describing a process or situation which we wish to model. For instance, the throw of a single dice where the top-face value is observed, constitutes a simple experiment. In each experiment we are interested in observing, predicting and describing the behaviour of all possible *outcomes*. Outcomes are the atomic units of formalisation. Thus, we condition the ability to apply the probabilistic machinery, on the existence of a set describing all the possible outcomes of an experiment. This ensures that three essential ingredients are in place. Firstly, that the outcomes are atomic, in the respect that they are indivisible. Secondly, that each outcome is unique, in as far as that no duplicates are allowed. Thirdly, that the set needs to be an exhaustive enumeration of all possible outcomes. This exhaustive set is defined as the *Sample Space* for the experiment (represented therein by S). In the single dice, single throw experiment, the obvious sample space is $S=\{1,2,3,4,5,6\}$

In order for the abstraction of physical phenomena to mathematical objects to become complete, we need an intuitive means of gathering atomic outcomes into meaningful groups. We will refer to such groupings as *events* and from an intuitive point of view, they can be seen as properties which can hold over parts of the sample space. Events provide an abstraction for reasoning over sets rather than over points in S. In the dice example an event (A) could be that of the appearance of an even face, thus $A = \{2,4,6\}$. More formally an *event* is defined as a subset of the Sample Space S.

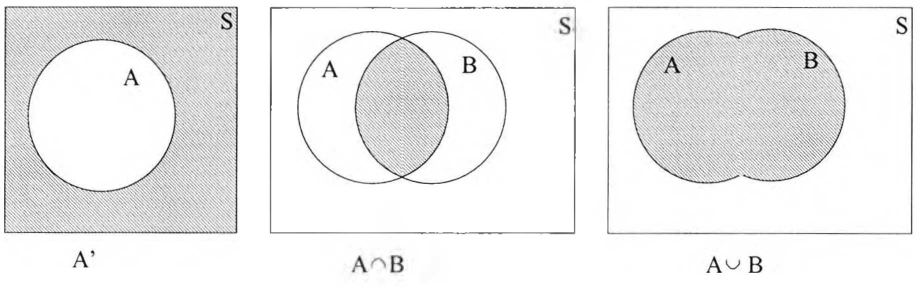


Figure 2.1: Composite Sets

This leads to the use of algebra of sets for the construction of composite events. For events A and B, (with A' denoting the complement to A with respect to S) we give a diagrammatic representation of the cases of, A', $A \cap B$, and $A \cup B$, composite events in Figure 2.1. A composition of particular interest is that between disjoint events. Events A and B are said to be disjoint if and only if $A \cap B = \emptyset$ ($\emptyset$ being the empty set).

2.1.2 The Axioms

The objective of the axiomatic definition of Probability, as for any concept defined axiomatically, is to formulate a number of axioms that describe the requirements on the mathematical objects of the theory.

Probability is a function P from sets to numbers. In particular, we are interested in the events (A_i) of S ($A_i \subseteq S$, as introduced in the previous section). Thus, we will call $P(A_i)$ the probability of event A_i. The axioms, which the function P need to satisfy, are:

i) $P(A) \geq 0$,

ii) $P(S) = 1$,

iii) for any k in the positive integers, and events $A_1, A_2, \ldots, A_k$ such that, if $A_i \cap A_j = \emptyset$ with $i \neq j$ then

$$P(A_1 \cup A_2 \cup \ldots \cup A_k) = P(A_1) + P(A_2) + \ldots + P(A_k) \tag{2.1}$$

The third axiom used here is that of *Finite Additivity* as opposed to the other options of *Countable* or *Infinite Additivity.* This is sufficient for the discrete and finite case which forms the basis of our computational apparatus.

2.1.3 The Properties

We briefly present some important properties that follow from the introduced axioms. For detailed expositions, proofs and motivation for the probabilistic significance of these properties the interested reader is referred to standard probability texts, such as [Fel59] and [Pap91]. The main properties of the theory that are of importance to this work are as follow :

Impossibility is ascribed as having a zero probability.

$$P(\emptyset) = 0 \tag{2.2}$$

Monotonicity of events to their corresponding probabilities.

$$if\ A \subseteq B\ then\ P(A) \leq P(B) \tag{2.3}$$

Complementarity

$$P(A') = 1 - P(A) \tag{2.4}$$

Coverage of intersection in union of arbitrary events.

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) \tag{2.5}$$

Finally, the *conditional* probability of an event A given the occurrence of event B, is defined as:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{2.6}$$

(for $P(B) > 0$).

2.1.4 Interpretations

With the mathematical abstractions of the theory in place, we will now take some space to present its three most established interpretations. This provides a perspective to the historic culmination towards the inception of probability and the need for an independent theory. It is also a natural link to the discussion in the second part of this chapter.

Frequency : This is the most common interpretation and is generally associated with statistics. It refers to well founded every day intuitions, where we can replace a point based approach to reasoning by an interval based one. In the frequency based interpretation, the probability of an event is perceived as being approximated by its relative frequency.

This is constructed by performing a number of repetitions (under identical or nearly identical conditions) of an experiment while counting the number of times an event (A) was observed to occur. The probability of the event is thought to be approximated by the quotient of the times event A was observed ($\#(A)$) over the total number of repetitions (N).

$$P(A) = \frac{\#(A)}{N} \tag{2.7}$$

Clearly, $\#(A) \leq N$. The approximation is thought to improve as the number of repetitions increase. The two values are assumed to converge as the number of repetitions tends to infinity.

As an example of this approach, consider a coin which is thrown a number of times ($N = 100$, for instance) with the number of tosses in which a head ($S = \{head, tail\}, A = \{head\}$) occurs being noted down (*e.g.* $\#(A) = 64$ for instance). The value of the quotient 2.7 is taken to be the probability of event A ($P(A) = .64$).

The frequency based interpretation is thought to be closer to the layman's intuitions and it usually precedes and motivates the formal treatment of the axiomatic definition in statistical text books.

Subjective : This is when the probability of an event is equal to the degree of personal (thus subjective) belief that the event will occur. A traditional way of quantifying such an immeasurable concept is through the acceptance of betting. So, someone may hold a personal belief that a game of backgammon between players *Andreas* and *Thanasis* is won by player *Andreas* with probability 3/4, if the following two statements are true. Firstly, that the person is willing to bet three units for each unit (or more) of pay-off in the event that player *Andreas* wins. Secondly, that the person is willing to bet one unit for a three unit (or more) of pay-off in the event that player *Thanasis* wins.

Let, $p(Andreas)$ be the probability of a win by *Andreas*. Trivially, we have

$$1 = p(Andreas) + p(Thanasis)$$

The number three, of the preceding paragraph is then derived, by:

$$\frac{P(Andreas)}{1 - P(Andreas)} = \frac{P(Andreas)}{P(Thanasis)} = \frac{3/4}{1/4} = 3 \tag{2.8}$$

For any event A, the quotient $P(A)/1 - P(A)$, is what in the knowledgeable circles called the *odds* of a bet. Alternative ways of quantifying the subjective probability have also been devised more recently (particularly via abstracting the notion of pay-offs to utility). An alternative approach which does not utilise the ideas of utility or betting odds, was developed by de Finetti [dF31].

One major difference of the subjective approach to the frequency based one, is that we can apply it in non-repeatable experiments. The basis of this approach was formed by work done primarily in Philosophy and Logic. An in-depth approach to the meaning of the two main interpretations of probability can be found in [Car62].

Classical : Historically speaking, the classical interpretation was the first approach to be developed. This goes way back before the time of formal mathematics. In recent times it is not presented independently, since it can be viewed as a a special case of the relative frequency approach. Its brief presentation here serves the two purposes of: (a) providing the historical perspective and (b) of showing that there are fundamental questions which are common to Logic as they are to probability.

The limitation relative to frequencies, comes from the requirement that the (N) outcomes in the sample space S, must all be equally probable alternatives. It then follows directly, that each outcome has a probability of $\frac{1}{N}$. The probability of an event, is thus formed from the enumeration of elements in the event (A) as a proportion to the total number of outcomes.

$$P(A) = \frac{|A|}{N} \tag{2.9}$$

The cardinality of set A ($|A|$) in (2.9) can be read as the outcomes favouring A, while $\#(A)$ in (2.7) is the observed occurrences of A. It is possible to link the two approaches by claiming that $|A|$ is the ***observed*** $\#(A)$ of a special experiment. This mapping is one directional, since there exists the obvious possibility of experiments that might not result to equal number of observations per outcome.

A fair dice can be used as an example for this approach. Normally, due to the lack of evidence to the contrary, one assumes that each of the faces ($\{1, 2, 3, 4, 5, 6\}$) is equally probable and thus can assign the probability value of $1/6$. The event of throwing an even face is then $P(A) = 3/6$. Note that this is not calculated in reference to the probability assigned to the actual outcomes. So, in this respect outcomes are treated as singleton events.

An important feature of the classical interpretation is that it promotes a less arithmetical evaluation in some domains. Instead of running experiments or gauging subjective approximations, one is encouraged to find symmetries in the underlying laws or structures of the problem domain. In cases where this is possible, one can derive very precise metrics, which come to the surface from the logical interactions. This is advantageous since there is no need to impose a mapping from approximations onto actual values.

From the perspective of knowledge management, the classical and subjective viewpoints are more suitable in cases where we want to move from axioms to theorems, whereas frequency approaches are used en masse in modern science to establish connections from theorems to more basic concepts in a mathematical theory.

2.1.5 Why a Measure Theory

When the additivity axiom is restricted to its finite version, as it is the case here (definition 2.1), the use of a measure theoretic approach is not always necessary. On the contrary, the use of the theory is often reduced to a glorified algebra for non-standard additions and multiplication. One alternative approach has been suggested by Nelson (in [Nel87]). The complete analysis of the respective merits are outside the scope of this work but we give the rationale behind our decision. With our target of a programming paradigm in focus, we believe that probability is the best available theory that can provide :

- a means for capturing people's intuitions about imprecision,
- a standardised lingua franca for communicating such notions,
- people who are trained in the theory,
- compositionality, and more so, sound compositionality.

2.2 Computational

In this section we present and comment on a number of computational methodologies which deal with notions of imprecision. The degree to which such notions are integrated is of course varied across the different methodologies. Our objective was to include these that are most likely to be neighbours of *Pfd.* The reader should note that this exposition does not claim completeness of the numbers or thoroughness of presentation. Furthermore, the categorisation used is contrived and by no means a standardised approach to the field.

2.2.1 Logics

There has been a long tradition of attempts whose target is to enhance logics with probabilistic concepts. The interest of philosophers and mathematicians (e.g. [Bor70]) has often been keen, and not surprisingly so, since they can be both viewed, as axiomatic theories of *knowledge.* Although these are deep engaging questions, here we will concentrate on work that is more akin to a computational oriented approach to knowledge, its representation and reasoning.

When one is mixing Logic and Probability for the purposes of representing knowledge, it is very often the case that the resulting formalism exhibits an in-built interpretation of probability. These are called, the epistemic and the statistical types of knowledge, which are perfectly aligned to the subjective and the frequency interpretations of Section 2.1.4.

Research in Propositional Probabilistic Logics [Nil86, GKP88, FHM90, FH94] has provided a clear perspective on their capabilities and limitations. The common ground is that the semantics are done via a possible worlds route. These languages are fairly expressive from a probabilistic perspective and have decision procedures that work well in the average case. The main drawback is that they seem to be better at (or more natural to form systems that are) dealing with epistemic knowledge and that the underlying Logics remain two-valued.

Work in First Order Logics include [Bac90a, Bac90b] which to some extent is based on and enhances the propositional case. This is again, using a possible-worlds semantics but manages to extend to the possibility of manipulating both types of probabilistic knowledge. Thus, it is possible to have the following sentences :

$$[fly(x)|bird(x)]_x > 0.5$$

$$[P(x) \land Q(x)|R(X)]_x = [P(x)|R(x)]_x \times [P(x)|R(x)]_x$$

The resulting formalism is very expressive but unfortunately it seems unlikely that a general procedure, which retains the computational properties of the propositional case, exists. Furthermore, it is not obvious how general probabilistic constructs can be derived for use within a programming paradigm. Two features that hinder this, are the use of detailed probabilistic constructs, which are only manageable by statisticians and also the obliqueness of the system's compositionality.

A rather different and less known approach was introduced in [Ale88]. Aleliunas at the first instance introduces a meta-theory for probabilistic Logics, which treat probabilities as uninterpreted formal marks. This results to a possibility of *non numerical* probabilities, which although an interesting concept; still reduces the appropriateness of probability as a name for these formal marks.

The meta-theory is formulated by a number of axioms for Propositional Probabilistic Logics. Some examples are then presented on how to instantiate this axiomatisation to more concrete theories. The examples are mainly used to illustrate that the formalism is a more general than existing ones, or that it can be used in novel ways.

This is a very insightful work that may have applications to common sense reasoning, knowledge representation and to decision theory, particularly so due to the fact that it might be capable of formalising some of the existing ad hoc approaches. The main two drawbacks are, firstly that it is concerned with the propositional case and secondly, that its impetus to practical problem solving has not been evaluated.

2.2.2 Logic Programming

The most effective use of Logic for programming, to date, is in the form of Logic Programming (e.g. [Hog90, SS86, NM90]). Thus, it is not surprising that there have been a number of suggestions towards a probabilistic version of Logic Programming. The first issue to be addressed in tackling such a task, is the mapping of statistical notions that underlie Probability Theory (such as experiment, outcomes and events as defined in 2.1) to Logic Programming constructs.

One of the issues which arises in various belief-oriented approaches is, that clauses of the (general) form :

$$A \; if \; B \; with \; P_1$$

$$B \; if \; C \; with \; P_2$$

must answer questions about the probability of A given C. Thus, as in any belief system, Probabilistic Logic Programming approaches need to define the combination rules which will provide answers to such questions.

In Logic Programs with Uncertainty ([Sha83]) we are presented with an extension to Logic Programming for reasoning with uncertainties. The semantics of normal logic programs are respected, although only a sketch of the extended semantics is given. The main novelty of this brief paper, is the use of multi-sets for the expression of degree in which the belief in one's premises influences the degree of belief to a particular conclusion. The perspective taken has a programming orientation and the multi-sets are shown to be allowing the construction of various meta-interpreters (each implementing different combination rules). Some trivial meta-interpreters are also sketched out. This approach is enlightening, in showing that it is quite easy to formulate combination rules for the limited case of subjective interpretation. By doing so, the generality of the term probability, is of course, lost.

Logic programs have also been shown to extend in a more general way through quantitative reasoning rather than through probability. There exist two approaches in this direction, this of van Emden (in [vE86]) and work by Fitting (see [Fit91]).

The idea was originally advocated by van Emden as an extension to standard Logic Programming (as in [vEK76]) motivated by the use of LP in implementing expert systems (in which one very often considers certainty measures rather than truth values). It is a general formalism for enhancing Logic Programming with quantitative reasoning, which contains both fuzzy and probabilistic reasoning as sub-cases. There are two Quantitative (as oppose to truth values which are considered qualitative) concepts introduced. The first associates a certainty factor to the implication operator, while the second extends the notion of interpretation (for relations) by regarding them as fuzzy sets ([Zad65]). From the computational point of view, this paper is a breakthrough since proofs are mapped into and/or trees and the *weights* attached to clauses (goals at time of derivation) are seen as heuristics for guiding the general purpose search algorithm (an $\alpha - \beta$ search algorithm in this case).

On the other hand, Fitting ([Fit91]) uses the notion of bilattices for the extension of logic programs, keeping the discussion primarily on the semantics while avoiding issues dealing with the proof procedure.

In both cases, the strongly founded semantics seem better suited for fuzzy reasoning. Also, due to the generality of these approaches, it will not be possible to take advantage of the special features of probabilistic measures for the purposes of efficient computation.

In a very similar vein, the most recent approach to Probabilistic Logic Programming in [Luk99] attempts to attach probabilities at the clausal level. The computability of this approach was further investigated using the principal of maximum entropy, which is a recurrent theme in uncertain reasoning, in [LKI99]. This approach is prototypical in illustrating why we have chosen to avoid defining probabilities at a clausal level.

A convenient bridge to the next section is Probabilistic Horn Abduction (in [Poo93b, Poo93a]). Which is not strictly speaking, within mainstream Logic Programming. This is an instrumental way of using a standard reasoning mechanism (that of Horn Abduction) to recast a special case mechanism (Bayesian Networks). Moreover, the resulting formalism subsumes (the propositional based) Bayesian Networks since it is predicate based. The main strength of Probabilistic Horn Abduction is in reasoning with conditional probabilities. As the author mentions,

> The aim was to create a simple Logic which is a compromise between epistemic and heuristic adequacy.

From our viewpoint there are two disadvantages. Firstly, that the probabilistic part is fused to the underlying reasoning strategy. Secondly, that although more general than Bayesian Networks, it still suffers from their main limitations.

2.2.3 Bayesian Systems

Evidential support Logic Programming is an approach that comes from the expert systems area, introduced by Baldwin (in [Bal87]). This is a generalisation of Logic Programming which allows reasoning with uncertainties (both probabilistic and fuzzy notions). The main extension to clauses is that each clause has a pair of numbers associated with it. These are the necessary and possible supports. In this framework derivations are not theorems which follow from axioms, but instead they are viewed as statements supported by evidence. The proposed theory was also implemented in two systems. One being a basic extension on top of Prolog, whereas another implementation sought to further differentiate the inference controls from the one available in Prolog. The advantage of these systems, is that they provide a solid basis which encompasses various alternatives to reasoning with uncertainty (fuzzy and probabilistic included). On the other hand, one of the main disadvantages, is that no formal semantics exist, since the semantics of Logic Programming cannot be applied to this formalism and no alternative semantics have been proposed by the author. This approach seems to be motivated by a statistical and expert systems background and it uses Logic Programming as a standardised form of rule inference.

There have been a variety of probabilistic methods used in expert systems (for some of the more well known ones see [Nea90]). Their common denominator is the use of Bayesian approaches, which normally boils down to taking advantage, in one form or another, of the conditional dependency definition (2.6). One of the more accepted representatives was proposed in [DHN81]. This is based in sound theoretical foundations and has been tested in some practical applications ([DGH81]). Its main target is to deal with subjective statements in a pragmatic way; accepting and trying to deal with problems such as inconsistencies found in subjective statements. Thus, although the basis is the normal Probability Theory, there were some generally applicable extensions introduced for dealing with networks of subjective inference rules. Reasonings in this system is done via a probabilistic updating procedure which can cope with peculiarities of subjective measures (there seems to be a slight confusion on the part of authors about the distinction, but this is no more than in other similar papers written by computer scientists). The applications presented include a hard-wired system for mineral exploration, as well as a more adaptive system that was employed in modelling areas such as medical diagnosis and securities analysis.

Bayesian Networks ([Pea88]) is a well researched formalism which can deal with probabilistic propositions given in a graphical form. The formalism has been particularly useful for exploratory analysis. That is, in areas where the involved variables and the exact topology of dependencies are still under investigation and a tool with sound inference is needed for exploring different hypothesis. Such areas include decision making (particularly in medicine) and economic modelling. Bayesian Networks are very well equipped to deal with such situations but they are not a very suitable formalism for general probabilistic programming; this is due to two reasons. Firstly because the mapping of the problem to (effectively) a global graph of variables requires a rigorous statistical understanding, and secondly because probabilistic dependencies can only be defined between values rather than random variables. This is beneficial to the problems of an exploratory nature where the researcher goes back and forth between the definition and the analysis of the *results* produced (for example of such interaction methodology see [CG98]). There is a plethora of publications on a variety of issues around Bayesian Networks, some of the ones pertinent to this thesis are, on complexity [BFGK96, DL93], on representation, [vdGM96b, Coo90, DP97, DvdG95].

2.2.4 Non Probabilistic

The Dempster-Shafer theory ([Sha76]) uses a non-probabilistic approach for modelling uncertainty. In some respect, it is an extension since it allows for non-exhaustive coverage of possibilities. The main drawback is that the combination rules are rather complicated and rigidly derived. These create a machinery to which results are commonly connected to prescribed facts in a un-intuitive manner. For a probabilistic view of the theory see [Kra97a].

Fuzzy systems ([Zad65, DP80, KY95]) have been very successful in control system applications. The success has been surprising to the theoretically inclined community since the foundations of Fuzzy *Logics* are generally considered as unsound. The success of the application domains with the control systems characteristics is not hindered by the unsoundness, because the rule base in such cases is quite limited and normally represent shallow knowledge (see [Elk93] for fuzzy systems' success contributing factors). Our main argument against fuzzy systems is not the unsoundness itself, but the fact that this leads to poor compositionality, which sets a hard limit on the size of problems to be tackled. Computing power on the other hard has been found to be a softer limit in comparison.

Chapter 3

Probabilistic Finite Domains

In this chapter we introduce the syntactic constructs of *Pfd* along with their intuitive meaning. We open the discussion by presenting the set of objectives sought to be fulfilled by this work. This set of objectives is also a good means of communicating the perspective through which we approach probabilistic computation.

The rest of the chapter describes the novel aspects of the constructs forming *Pfd*. For each of the constructs we provide its syntactic form, its intuitive meaning and the role it plays within the formalism. The intuitive meaning is conveyed via operational and logical arguments. We also briefly comment, on the algorithmic behaviour of the constructs and in particular identify the most expensive operations.

Although we present the *Pfd* framework within a CLP(X) context, its concepts can be integrated to different paradigms. The suitability of Logic Programming for presenting our work becomes apparent from the use of meta-programming constructs. In this chapter we comment on the concessions that have to be made, when considering *Pfd* within other programming paradigms.

Another reason for presenting *Pfd* through CLP is that this approach helps to concentrate on the novel aspects of the introduced framework. Particularly so since CLP(X) is declarative, well understood and modular in the choice of X; the domain of discourse.

3.1 Objectives

The main design objectives for *Pfd* were :

- sound measure theory,

- intuitive paradigm,
- programmability,
- language and interpretation independence.

The choice of these particular objectives, leads to a very interesting, challenging and open research topic. This is particularly highlighted by the fact that none of the approaches discussed in Chapter 2, is generally thought to address all of these objectives simultaneously.

3.1.1 Sound Measure Theory

The computations within the proposed formalism should be based on a consistent measure theory. The theory on which we base *Pfd* is Probability Theory. This is certainly considered as a consistent mathematical theory.

In particular, we require the computations to be sound with respect to Probability Theory. Thus the existence of a probabilistic formulation for every computation instigated within *Pfd* is necessary. On the other hand, we should note that completeness is not one of the requirements. The requirement of soundness and lack of completeness of *Pfd* with respect to PT is similar to the relationship of LP to Logic.

3.1.2 Intuitive Paradigm

As a modelling tool *Pfd* should present a uniform and intuitive paradigm. Through this paradigm the objects in the universe of discourse are mapped to mathematical entities. The converse should also hold; by this we mean, that the results of *Pfd* computations should intuitively map to objects in the underlying universe of discourse.

By intuitiveness we mean the proximity of constructs to the probabilistic notions that humans can handle in an immediate, error-free manner.

3.1.3 Programmability

One of our main objectives has been to create useful programming constructs. To this effect we aim towards:

- Predictable use of computational resources.
- Parameterisation for some of the provided basic primitives.
- Adaptability. The provided primitives in their entirety, should give convenient handles for a wide variety of probabilistic computations.

3.1.4 Independence

The core of the system should be independent of any particular language and easy to annex to as many programming languages as possible. Another form of independence should arise from the way the system deals with concepts of probability, which should be irrespective of the interpretation given to the probabilistic measures in the mathematical modelling stage.

3.2 Integrated Domains

At the very foundations of our system we have introduced a novel way of assigning probabilities to the elements of a domain. A *Probabilistic Finite Domain* (*Pfd*) variable has two parts. On one hand there is a *finite domain*, which at each stage holds the collection of possible values, while on the other hand, there is a *function* that can be used to assign probabilities to the (remaining) elements of the domain. The intuition behind this is that we do not need to mix the two parts until it is necessary and only for as long as this is required.

The finite domain is represented by a set of elements, while the probabilistic function operates on any of the subsets of the finite domain and valuates to a set of probabilities where each member belongs to the interval zero to one, and their summation is one.

The probability function is used to model the probabilistic behaviour of objects in the universe of discourse. As an abstraction mechanism, it provides a concept for communicating probabilistic behaviour, at a level were people are at ease. The separation of the probabilistic values from the elements in the finite domain, by the introduction of a context-dependent, transient function application has as a result a certain degree of non-monotonicity of assigned probabilities.

In what follows it is essential that the reader keeps in mind the dual nature of *Pfd* variables, in order to see how this duality is expressed by the syntactic entities and how these variables provide modelling capabilities for real world objects.

3.3 Variable definition

The basic computational units of *Pfd* are the probabilistic[1] variables. Probabilistic variables have two constituents: a finite domain and a function. We will refer to this function as the probability constructor method or simply as the variable's method to avoid confusion with other

[1]Probabilistic variables are in essence statistical random variables, but we shall use this alternative name due to the non standard way in which probabilistic variables form events.

kinds of functions. New probabilistic variables are introduced by,

$$Variable \in_{\mathbf{p}} Method(Domain\{, Arguments\})$$

where $Variable$ is an unbound logical variable which will only be available to *Pfd* constraints. *Method* is the name of the probability constructor method. *Domain* is a list representation of the variable's finite domain. Finally, Arguments stands for a number (zero or more) of optional arguments that will be passed to *Method.* Example usage,

$$Dice \in_{\mathbf{p}} uniform([i, ii, iii, iv, v, vi]) \tag{3.1}$$

$$Coin \in_{\mathbf{p}} biased\_coin([head, tail], 2/3) \tag{3.2}$$

On the first example (3.1) $uniform$ refers to a general purpose method. The intuition behind this method is that the given variable takes valuations from its finite domain in a uniform manner. Thus, this method assigns equal probabilities to the elements of the variable's finite domain. On the other hand, *biased_coin* refers to a program-specific method for which a definition has been locally provided. A possible meaning could be that for the domain represented by the set {a,b}, element *a* has a two-thirds probability attached. A more generic definition (for instance a method *biased_list*) could allow for the first element of the set-domain to have a probability of two thirds attached to it, with the probability for the rest of the elements uniformly spread over the remaining third.

The intuitive reading for Constraint 3.1 says that elements in *Dice's* finite domain are equiprobable in their chance of participating in a successful derivation involving *Dice.* In accordance with the dual interpretation of probability (Sect. 2.1.4) the distribution over the variable's domain has a double reading. Either that it expresses a programmer's belief in the likelihood of each element in the finite domain, or that it reflects a quantitative aspect of the variable.

Formulas formalising the methods discussed here, are presented in Table 3.1. In Table 3.1 we : (a) formalise the description and (b) give example usage for the methods described in this section. Note that $\mid Fd \mid$ denotes the cardinality of a finite domain and that the probabilities in the examples refer to sample subsets of possible finite domains for the purposes of illustration. The application of a single method over different subsets of a finite domain, illustrates how probabilistic distributions are affected by the pruning of a variable's finite domain. Note that for singleton sets the probabilistic value is one and of course there are no provisions for empty sets, since they denote failed computations.

By allowing locally defined probabilistic functions, we not only extend the scope of the formalism, but also do so in a way that allows the programmer to think about and model the behaviour of probabilistic notions in more static fashion.

Method	Domain	Options	Probabilities
Methods Description			
$uniform$	$[El_1, \ldots, El_n]$	[]	$[1/\mid Fd \mid, \cdots, 1/\mid Fd \mid]$
$biased\_coin$	$[El_1, El_2]$	$[HPrb]$	$[HPrb, 1 - HPrb]$
$biased\_list$	$[El_1, \ldots, El_n]$	$[HPrb]$	$[HPrb, 1 - HPrb/(\mid Fd \mid -1), \ldots]$
Examples			
$uniform$	$[i, ii, iii, iv, v, vi]$	[]	$[1/6, 1/6, 1/6, 1/6, 1/6, 1/6]$
$uniform$	$[i, ii, iv, v, vi]$	[]	$[1/5, 1/5, 1/5, 1/5, 1/5]$
$biased\_coin$	$[head, tail]$	$[2/3]$	$[2/3, 1/3]$
$biased\_list$	$[low, med, high]$	$[1/2]$	$[1/2, 1/4, 1/4]$
$biased\_list$	$[low, high]$	$[1/2]$	$[1/2, 1/2]$

Table 3.1: Definitions of Probabilistic Variables

Pfd facilitates program specific methods (e.g. *biased_coin*). At the conceptual level, this mechanism allows for the enrichment of probabilistic behaviours available to the programmer. The important feature is that this happens in a local fashion, where we can concentrate at the generic probabilistic behaviour of a variable. At a technical level, the exact syntax of how such methods are presented depends on the particular implementation of *Pfd*. It is sufficient to mention here that at each invocation of a program-specific method, the *Pfd* system will perform consistency checks (e.g. that the returning values are indeed forming a probability measure) which ensure their seamless integration.

3.4 Conditional

We now show how the introduced variables are combined to form probabilistic constraints. The constraint introduced here allows the modelling of probabilistic dependencies between variables. The basic probabilistic concept employed here is the notion of *conditionality*. This stems from Bayes' Theorem in Probability Theory and it has been a basic construct to almost all approaches in probabilistic reasoning. This is scarcely surprising and in effect the real question is not whether the concept of conditionality is used, but rather how it is used.

The syntactic form of a *Pfd* conditional (also referred to as a conditional constraint), is:

$$Dependent \mid Qualifier$$

where both the *Dependent* and the *Qualifier* (LHS and RHS of the conditional respectively) are predicates that involve one probabilistic variable and no unbound logical variables. We should note, that the probabilistic variable of *Qualifier* is considered as a constant when it appear in predicate *Dependent*. For the greatest part of this thesis we will restrict our attention to two predicates. The equality and difference predicates $=$ and $\neq$. (Note that the single probabilistic variable clause still applies.) One justification for our choice to concentrate on $=$ and $\neq$ comes from the fact that these predicates, are almost universally available in programming languages, thus allowing our discussions about *Pfd* to be more general. Another reason is that the two predicates are intuitively closer to the finite domain nomenclature.

The conditional constraint states that whenever *Qualifier* holds, then predicate *Dependent* also holds. Before elaborating how the previous statement interprets to probabilistic calculations, consider the following two examples :

$$Dice = 3 \mid \text{Coin} = \text{tail} \tag{3.3}$$

$$Coin \neq \text{tail} \mid \text{Dice} = \text{Random} \tag{3.4}$$

(Where Random is a non probabilistic variable.) The first constraint prescribes that, for the particular pair of $Dice$ and $Coin$, a toss of tail makes the $Dice$ to always stop its roll on a three. Whereas, the second example states that the $Coin$ cannot assume the value *tail* if the value of $Dice$ is equal to the value of the (non-probabilistic) variable Random. Random is required to be bound when this constraint is added to the store. Although it could be bound to any value, in this example it is only the values in $\{i, ii, iii, iv, v, vi\}$ which are significant.

There are two important issues involved. Firstly, that like conditionality in Probability Theory the introduced constraint is directional, i.e. it is not symmetric. (The effect of A on B does not tell us everything about the effect of B on A.) The main reason from a computational perspective for this choice, is that it provides an extra handle for producing effective computations. Secondly, the constraint provides a local way of affecting the probability assignments of the variable involved in the LHS. This is in contrast to the global way of affecting the assignments provided by $\in_{\mathbf{p}}$. Thus the probability distribution for the variable in the LHS of a conditional, prior to such a constraint, and the one after the constraint is introduced to the store, are in general different. It is also important to note that the effects of such changes are not calculated in an eager way, but rather on demand.

As an example of how a conditional constraint can influence the distributions of probabilistic variables, consider the following two variables :

$$Dice1 \in_{\mathbf{p}} uniform([i, ii, iii, iv, v, vi]) \tag{3.5}$$

$$Dice2 \in_{\mathbf{p}} uniform([i, ii, iii, iv, v, vi]) \tag{3.6}$$

Variable	Distribution
$Dice1$	$[i - 1/6, ii - 1/6, iii - 1/6, iv - 1/6, v - 1/6, vi - 1/6]$
$Dice2$	$[i - 1/6, ii - 1/6, iii - 1/6, iv - 1/6, v - 1/6, vi - 1/6]$
$Dice2 = iii \mid Dice1 \# iv$	
$Dice1$	$[i - 1/6, ii - 1/6, iii - 1/6, iv - 1/6, v - 1/6, vi - 1/6]$
$Dice2$	$[i - 1/36, ii - 1/36, iii - 31/36, iv - 1/36, v - 1/36, vi - 1/36]$

Table 3.2: Variations of Distributions

For the two variables $Dice1$ and $Dice2$ we give, in Table 3.2, their respective static distributions[2]. In rows two and three are the distributions immediately after the declarations of Constraints 3.5 and 3.6 respectively. In rows five and six, are their distributions after the addition to the store of the following constraint :

$$Dice2 = iii \mid Dice1 \# iv \qquad (3.7)$$

The intuitive reading is that all faces of $Dice1$ apart from face iv are only compatible to face iii of $Dice2$. The finite domain elaboration of what *compatible* refers to, is that combinations of elements such as ii ($Dice1$) and v ($Dice2$) are deemed inconsistent. Furthermore, *Pfd* adds a probabilistic elaboration, in that it changes the probabilistic distributions of variable $Dice2$. The variable's distribution before the constraint is $[1/6, 1/6, 1/6, 1/6, 1/6, 1/6]$, while the one following the addition of the conditional constraint to the store is $[1/36, 1/36, 31/36, 1/36, 1/36, 1/36]$. The reduction to most of the values reflect the change in how probable elements are to appear in a solution given the new constraint. Each of the one-thirty-sixths is derived from :

$$P'(Dice1 = iv) * P'(Dice1 = X) \; (X \in \{i, ii, iv, v, vi\})$$

(where P' is the probability prior to the constraint.) Whereas, for the third face we have :

$$\sum_{x} (P'(Dice1 = X) * P(uniform([iii])) + P'(Dice2 \neq iv) * P'(Dice1 = iii)$$

$$= (\frac{1}{6} * 1 + \frac{1}{6} * 1 + \frac{1}{6} * 1 + \frac{1}{6} * 1 + \frac{1}{6} * 1) + \frac{1}{6} * \frac{1}{6} = \frac{31}{36}$$

When it is the case that $Dice1 = iv$ then the prior distribution of $Dice1$ is used. This is because *Qualifier* predicate is false, thus the distribution of $Dice2$ remains unchanged. On the other hand, when the *Qualifier* holds, the distribution of *Dependent* is re-evaluated

[2]broadly speaking consider this as the distribution at a particular point of the computation, (fully defined in Section 3.8)

$(P(uniform([iii])))$. Note that the only elements of $Dice2$ which are considered in this case are the ones that make the predicate in the LHS true.

An interesting reading for the distribution of $Dice1$, is that it assigns to each element its share of times it is expected to appear in the solution space. Possible interpretations of this statement include : (a) the expected relative frequencies (belief or statistics oriented) to a single solution or (b) the expected relative frequencies over multiple solutions. We should also note that there is no implied ordering of events, the two variables continue to act independently.

In a similar vein, we define the *conditional difference* constraint which is of the form :

$$DependentVar \mid\!\ast\ QualifierVar$$

where $DependentVar$ and $QualifierVar$ are probabilistic variables. The intuitive reading for the constraint is that the values of the variables have to be different. (The directionality actually only constrains the value of $DependentVar$ to be different than that of $QualifierVar$ - once this has a specific value.) Like in the case of the conditional constraint, this can potentially influence the probabilistic distribution of $DependentVar$.

The corresponding formulation in terms of the conditional is :

$$\forall X \in fd(V_1) \cdot V_2 \neq X \mid V_1 = X$$

3.5 Conditional Variable definition

There is also an alternative way for variable definitions, which covers some special cases. This constraint combines the variable definition with a conditional constraint. The conditional definition constraint takes the form:

$$Variable \in_{\mathrm{p}} Method(Domain) \mid\!\ast\ QualifierVar\ \ with\ \ Probability$$

where the probability distribution of a variable hinges on a conditional difference. More specifically $Variable$ takes values from *Domain*, with the value of the $QualifierVar$ being assigned the probability of (1 - Probability), and $Method$ is used to construct the probability distribution for the remaining elements of the domain.

Unlike the effect of conditional constraint which is local, the conditional definition is used to define a global, fundamental causal relationship between two variables. The behaviour of the conditional definition follows that of the simple definition of variables and of the conditional constraint. An alternative way to view this constraint is by considering it as a special case of a probabilistic method that allows for another probabilistic variable as one of its arguments.

3.6 Probability of Events

The constraints introduced so far deal with the declaration of variables and that of constraints upon variables; in short, with adding information to the store. In this section we describe how information can be derived from the store. *Pfd* provides a single construct to this effect,

$$Probability\ is\ \mathbf{p}(Predicate) \tag{3.8}$$

where *Predicate* is similar to the predicates in the conditional constraint, except for the single probabilistic variable restriction. In detail, *Predicate* is an arbitrary term that has no unbound logical variables. Again for reasons of stability and uniformity across different programming languages, we will mainly consider the identity and difference predicates ($=$ and $\neq$). The logical variable Probability, after the encounter of Constraint 3.8, will be bound to the total probability with which the probabilistic arguments to *Predicate* satisfy it. The term *satisfy it*, stands for the applications that return true and in contrast to those that return false. (Clearly, the assigned value must satisfy, $0 \leq Probability \leq 1$.)

This constraint provides in *Pfd* a concept that intuitively maps to the statistical concept of events (as defined in Sec. 2.1). For example, we can ask for the probability of the following two events :

$$SingleFace\ is\ \mathbf{p}(Dice{=}2)$$

$$TwoFaces\ is\ \mathbf{p}(Dice1{=}Dice2)$$

Provided that all of $Dice$, $Dice1$ and $Dice2$ are defined as shown earlier and that these variables are not participating in any other constraints, then both $SingleFace$ and $TwoFaces$ will assume the value of one-sixth. In the first example, this follows from the number of elements in the domain (six) and the variable's distribution (uniform). The second example is more interesting, since it demonstrates how *Pfd* uses simple intuitive concepts to model notions that are not as intuitive to the layman. A large proportion of people when presented with the question of the likelihood of two dice having the same value give the answer of one-thirty-sixth.

3.7 Variable Instantiation

The next construct is concerned with the assignment of specific values to a probabilistic variable. It is of the following form,

$$label(Variable, Select, Value, ValProb, CurrentProb)$$

The overall behaviour of *label* is to instantiate $Value$ to an element of $Variable's$ finite domain and $Probability$ to the element's probability. This is done by using Select (an identification for the selection method to be used) for choosing between the different elements and their assigned probabilities. In a Logic Programming environment, this is best captured by a backtrack-able predicate. Each time a different Value is returned CurrentProb holds the summation of values in Probability encountered up to the current stage.

As an example of a selection method we have,

$$label(Coin, domain\_order, Value, Probability, Current)$$

Provided $Coin$ refers to a well behaved coin, the two ways in which the above succeeds, are :

$$label(head, domain\_order, head, 1/2, 1/2)$$

$$label(tail, domain\_order, tail, 1/2, 1)$$

Although in the Logic Programming case the use of Value might seem superfluous (Coin is instantiated to the same term) it is included because (a) it is a useful device for other programming paradigms, and (b) it allows alternatives implementation within Logic Programming.

In a similar vein we define :

$$label(Variables, Select, Values, ValPrbs, BranchPrb, CurrPrb)$$

This variation is particularly suited for specialised algorithms, which work better when considering more than one variable while labelling. Seen from a constraint programming point of view these valuations are a general form of non-binary constraint (an active area of research in constraint programming). The production of different selection methods is meant to cater for general probabilistic problem solving and their evolution is much harder than the other constructs of *Pfd*. As part of this research we have completed two algorithms implementing multi-variable labelling. One is an accurate but inefficient method. While the other is an efficient but approximate labelling method (based on a generic heuristic).

The first method has as its selection identifier the *asc_probability* token. It can be used via a call of the form :

$$label([Coin1, Coin2], asc\_probability, Vals, VlPrbs, BranchPrb, CurrPrb)$$

The results of this call for the two coins defined with the biased_coin method (Constraint 3.2) are given in Table 3.3. This selection method is guaranteed to give the most likely combinations first.

$Coin1 \in_p biased\_coin([head, tail], 2/3)$.
$Coin2 \in_p biased\_coin([head, tail], 1/3)$.

*Coin*1	*Coin*2	Value1	Value2	*Prb*1	*Prb*2	*BrPrb*	*CurPrb*
head	tail	head	tail	2/3	2/3	4/9	4/9
head	head	head	head	2/3	1/3	2/9	6/9
tail	tail	tail	tail	1/3	2/3	2/9	8/9
tail	head	tail	head	1/3	1/3	1/9	9/9

Table 3.3: Heuristic Labelling

In contrast, the second selection method does not guarantee that the valuation will occur in descending order of likelihood. The identifier for the selection is *max_unique_alt* . This identifies an approximating method that uses, what we here call the maximum alternative heuristic, to assign values to each of the variables. The second method can be used by a similar call :

$$label(Coins, max\_unique\_alt, Vals, ValPrbs, BranchPrb, CurrentPrb)$$

The main difference from the first method is that the results, rows in Table 3.3, are not necessarily retrieved in a top-to-bottom fashion. This is because the bookkeeping required lead to very expensive computations. Instead, the heuristic approach reduces the amount of bookkeeping but cannot guarantee that the combinations will be assigned in the most likely order.

3.8 Variable Interrogation

Here we present some predicates that retrieve information about the state of probabilistic variables in the current store. These are important computational units that can help determine branching decisions.

Static Distribution. At any point of the computation all the current information associated with a variable can be represented by a list of pairs. Each pair holding an element value along with its probability. We will refer to this list of pairs as the static probability of a variable.

Using the predicate static_distribution($+PVariable$, $-ElemPrbs$), the programmer can access this information for a particular variable.
ElemPrbs include all remaining elements of the *PVariable* 's finite domain paired to its

corresponding probabilities. The probabilites are calculated by taking into consideration all the current conditionals that involve *PVariable* at their LHS (*Dependent* predicate). The complexity of finding a variable's static distribution, closely matches the complexity of calculating the probability of events. The main difference is that we need to replace the variables in the event *Predicate* by the single variable, for which we seek to determine its distribution. This becomes more apparent when considering the following definition for the distribution of variable X:

$$\{< El, ElPrb > \cdot \forall El \in fd(X) \ \wedge \ ElPrb\ is\ \mathbf{p}(X = El)\} \qquad (3.9)$$

where $fd(X)$ is the finite domain of variable X.

Derived. A whole suit of derived information concerning a probabilistic variable can be calculated from the static distribution in the usual Logic Programming fashion. In what follows we provide descriptions for some of the predicates that we will be referring to in later chapters.

- Predicate domain_cardinality($+PVariable$, $-Cardinality$), with *Cardinality* being the number of possible values for the probabilistic variable *PVariable*.
- With maximal($+PVariable$, $-MaxElem$, $-MaxPrb$) the most probable element (*MaxElem*) in *PVariable*'s static distribution can be accessed. Variable *MaxPrb* will hold the element's probability.
- Similarly, minimal($+PVariable$, $-MinElem$, $-MinPrb$) provides the least probable element (*MinElem*) in *PVariable*'s static distribution. Variable *MinPrb* will hold the element's probability.

3.9 Synopsis

As a postscript to this chapter we emphasise some of the key points of probabilistic finite domains.

- Support of local (conditional constraint) and global change (method declarations and composite event probability).
- Use of directionality in conditionals is a computational compromise similar to the directionality of Logic Programming clauses and their computational rationale.
- *Pfd* is extensional and provides no intentional or analytical means for calculating event probabilities.

Chapter 4

Semantics

This chapter exposes a semantic formalisation of *Pfd* . One of the main benefits of CLP is that it presents a generic means of providing formal semantics for *Pfd* . We will follow a slightly non traditional formalisation of the CLP scheme since it is a better match for the work at hand. Moreover the selected approach is more concisely presented in the literature.

4.1 CLP

Constraint enhancements within Logic Programming have been used to address two of its shortcomings. The most widely researched one is the ability to incorporate domain-wide efficient algorithms; such as consistency techniques and linear equation solving. Another shortcoming addressed is that of the collapse of the semantics with regard to real number arithmetics. The issues which *Pfd* attempts to address are different and twofold. Firstly, to facilitate probabilistic reasoning and secondly, to allow generic introspection of the proof procedure based on the probabilistic reasoning constructs.

In detailing the semantic properties of *Pfd*, computations we will follow the *value constraints* subscheme for CLP ([vE97]). The name reflects the subscheme's applicability to situations where possible values, of a constraint variable, can be efficiently represented as sets. This thesis argues that this is also true for *Pfd* . Moreover we show that this is an intuitive way for capturing the operational notion of the static distribution of probabilistic variables. Another reason for choosing this subscheme is the fact that it presents a mathematically cleaner framework, when compared with the general CLP scheme ([JL86, JM94]).

4.2 Value Constraints Subscheme

In this section we will review the value subscheme of CLP. We gather together the results which exist in the literature and are of relevance to the original work that follows in the succeeding sections. In our exposition of the value subscheme, we follow [vE97].

CLP replaces the Herbrand base of LP, with parametrised semantic domains which are characterised by tuples of the form, $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T} \rangle$. Where Σ is a signature, describing function and relation symbols that can occur in constraints. $\mathcal{D}$,$\mathcal{L}$ and $\mathcal{T}$ are the Σ specific structure, class and theory respectively. Computation with respect to program $\mathcal{P}$ for goal G is described in terms of derivations. The result of a successful derivation is constraint c for which $\mathcal{P}, \mathcal{T} \models \forall[c \rightarrow G]$.

A derivation, is in turn described by a sequence of transitions. Transitions are functions on states, each state characterised by a triplet of the form $\langle A, C, S \rangle$. Where A is a set of atoms, implicitly thought as the conjunction of these atoms and C with S being sets of constraints, also thought as implicitly conjoined. The constraint sets, named active (C) and passive (S), jointly form the constraint store. The initial state is $\langle G, \emptyset, \emptyset \rangle$ and for final state $\langle \emptyset, C, S \rangle$ we have

$$\mathcal{P}, \mathcal{T} \models \forall[(C \wedge S) \rightarrow Q] \ .$$

The transitions possible may be any of the following four :

Resolution.

$$\langle A \cup a, C, S \rangle \ \rightarrow_r \ \langle A \cup B, C, S \cup \{s_1 = t_1 \dots s_n = t_n\} \rangle$$

when there exists rule $pred(t_1, \dots, t_n) \ \leftarrow \ B$ in $\mathcal{P}$ for selected $a = pred(s_1, \dots, s_n)$. This corresponds to the unfolding step of LP clauses. If no suitable rule exists in $\mathcal{P}$ then we have :

$$\langle A \cup a, C, S \rangle \ \rightarrow_r \ fail$$

Constraint transfer.

$$\langle A \cup c, C, S \rangle \ \rightarrow_c \ \langle A, C, S \cup \{c\} \rangle$$

for selected constraint c. This is where constraints that appear in A, are added to the store.

Store management.

$$\langle A, C, S \rangle \ \rightarrow_i \ \langle A, C', S' \rangle$$

with $(C', S') = infer(C, S)$. This transition allows generic store management.

Consistency test.

$$\langle A, C, S\rangle \rightarrow_s fail$$

if $\neg consistent(C)$, otherwise :

$$\langle A, C, S\rangle \rightarrow_s \langle A, C, S\rangle$$

In the general CLP scheme, passive constraints are normally those which can not be added to the active constraints (typically because no effective procedure for deriving information from them exists). The infer step ($\rightarrow_i$) in that case requires that

$$\mathcal{D} \models \forall[(C' \wedge S') \leftrightarrow (C \wedge S)] \ .$$

The existence of passive constraints accommodates situations such as the inability of the CLP(R) solver to cope with non linear constraints.

The value constraint subscheme, on the other hand, uses passive constraints as a canonical representation of constraint variables. In this light, S is a set of unary predicates of the form $v(V)$. Where v is a denotation for some subset of D. Moreover, the subscheme requires a less strict form for the infer step,

$$\mathcal{D} \models \forall[(C' \wedge S') \rightarrow (C \wedge S)] \ .$$

In what follows we will use $v_i(V_i)$ to refer to the denotation of variable V_i in a particular store.

For a complete treatment and motivation for the value constraints subscheme the reader is encouraged to seek the original publication ([vE97]). Here we apply the minimum necessary material from the above publication that are need for developing the semantics of *Pfd.* The appeal of the subscheme to this end is twofold. Firstly, the redefinition of passive and active constraints is more natural in capturing the intuitions behind probabilistic variables, than the standard definition. Secondly, although we can treat probability numbers as quotients of very long integers, which allows for extremely accurate representation, still the subscheme provides an extra safety net, for the semantically clean incorporation of machine arithmetics as approximation to its mathematical counterpart.

4.3 The Parameters

Our first task is to instantiate the semantic domain $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T}\rangle$. In this section we describe a semantic domain suitable for *Pfd* .

The signature Σ specifies function symbols, $\{ \mathbf{I}\,, \mathbf{I\!\ast}, sd, l, \in_{\mathbf{p}}, p\}$. (We abbreviate *label* and *static_distribution* to l and sd respectively.) Each of the defined functions corresponds to a constraint described in the previous chapter. The arity and types of each function symbol are :

- $\mathbf{|\ddagger}\,(V_1, V_2) \rightarrow D$ where each V_i is a probabilistic variable.
- $\mathbf{|}\,(Pt_1, Pt_2) \rightarrow D$ where each Pt_i is of the form $p(V_j, \bar{s})$. With $\bar{s} = s_1, \ldots, s_n$ a number of constants and p is a predicate symbol ($p \in \Pi$).
- $\in_{\mathbf{P}f} : Fd \rightarrow D$. Where Fd is the powerset of constants.
- $p : Pt \rightarrow Q$. With Q being the rational fractions ≤ 1.
- $l : D \rightarrow \langle H, Q \rangle$. Where H are atoms in the Herbrand base.
- $sd : D \rightarrow S$. S being the powerset of constant-fraction pairs.

$\mathcal{D}$ has two parts. The first is the set of values D and the second defines the structures that hold over D (predicates and functions specified by Σ). In *Pfd* the domain D is the set described by, $\{\langle f, \{\langle s_i, q_i \rangle\}\rangle \cdot s_i \in H, q_i \in Q\}$. Where s is a constant, q is a rational number (represented by quotient of the form Int_1/Int_2 where Int_1 and Int_2 are positive integers, such that $Int_1 \leq Int_2$) and f is a function identifier also expressed in $\in_{\mathbf{P}f}$. The interpretations are :

- $X = sd(V) \leftrightarrow V = \langle f, X \rangle$
- $X = l(V) \leftrightarrow (V = \langle f, S \rangle \wedge X \in S)$
- let $Pt_i = pred(V_i, \bar{s})$ and $Pt_i(s_j) = pred(V_i/s_j, \bar{s})$

$$X = p(Pt_i) \leftrightarrow V_i = \langle f, S_i \rangle \wedge \sum_j q_j \cdot \langle s, q \rangle \in S_i \wedge \mathcal{P} \vdash Pt_i(s)$$

- $X = \in_{\mathbf{P}f}(S) \leftrightarrow X = \langle f, R \rangle \wedge (\forall s \in S \rightarrow \langle s, q \rangle \in R) \wedge f(S) = R$
- $X = Pt_1 \mathbf{|}\ \ Pt_2 \leftrightarrow L = \{L' \cdot \forall s_2 \wedge \langle s_2, q_2 \rangle \in R_2 \wedge$

$$L' = \begin{cases} \in_{\mathbf{P1}}(\{s_i \cdot \forall \langle s'_i, q'_i \rangle \in R_1 \wedge \mathcal{P} \vdash Pt_1(s'_i)\}) *' q_2 & \text{, if } \mathcal{P} \vdash Pt_2 s_2 \\ R_1 *' q_2 & \text{, otherwise} \end{cases}$$

$$\}$$

$$\wedge X = \{< s, q > \cdot \forall l \in L \wedge q = \sum_{\forall \langle s, q_i \rangle \in l} q_i\}$$

where $*' : D \text{ x } Q \rightarrow D$

such that $X = R *' q \leftrightarrow \{\langle s_i, r_i \rangle \cdot \forall i \langle s_i, q_i \rangle \in R \wedge r_i = q_i * q\}$

- $X = Pt_1 \mid\ Pt_2 \leftrightarrow$ as above by replacing $\mathcal{P} \vdash Pt_1(s'_i)$ with $s'_i \neq s_2$

The class of Σ-formulas $\mathcal{L}$, describes all valid constraints. These are formed from the variables and function symbols in Σ and the logical connectives $\{\wedge, \vee, \neg\}$. The axioms of theory $\mathcal{T}$ used here is based on Probability Theory and is explicated by the infer derivations.

4.4 Transitions

The store in the CLP scheme is split into active and passive constraints sets (C and S). The intuition to this approach is that C will hold all the constraints over which an efficient algorithm operates. Constraints in S are candidates for transfer to C if certain conditions are met at some point of the computation. A familiar example of this behaviour is the delay of non-linear constraints in CLP(R) until they become linear.

The value constraints subscheme relays different roles for these sets. The form of active constraints is, $C = \{v_1(V_1), \dots v_n(V_n)\}$ where each v_i is a value predicate of Σ depending on C. For each V_i in S corresponds a single $v_i(V_i)$ in C. The subscheme leaves it to the constraint management to decide what happens to constraints in S once they are added to the store via a constraint transfer ($\rightarrow_c$). This approach is motivated by the existence of a *canonical* representation for each variable (its value at each stage of the computation). One of the immediate results is that there exists a generic characterisation for the notions of local and global consistency.

The appeal of value constraints for *Pfd* is the notion of canonical representation. Local and global consistencies are not important here, since they are not used for deriving probabilistic information from the store. In *Pfd* each v_i maps to an elements in D.

4.4.1 Consistency

We start detailing the remaining two transitions from the simpler one. Intuitively speaking the store may become inconsistent in any of the following three ways: (a) there are no possible values left for some variable, (b) there exist a cyclic conditional dependency, or (c) redefinition of a probabilistic variable. These three cases are formalised by the following statements.

- $\exists v_i(V_i) \in C \cdot v_i(V_i) = \langle f, \langle \emptyset, R \rangle \rangle \rightarrow \neg Consistent(C, S)$.

- $\exists V_i V_j \cdot cond(V_i, V_j) \in S \wedge V_j \in dep(V_i, S) \rightarrow \neg Consistent(C, S)$

  where $cond(V_i, V_j) = Pt_i \mid\!\neq\ Pt_j$ or $cond(V_i, V_j) = Pt_i \mid\ Pt_j$

$$\text{and } dep(V,S) = \bigcup_j \{V_j\} \cup dep(V_j, S) \cdot cond(V, V_j) \in S \ .$$

- $\exists V_i \cdot S = \{\in_{\mathbf{P}_f}(V_i)\} \cup S' \wedge \in_{\mathbf{P}_g}(V_i) \in S' \rightarrow \neg Consistent(C,S)$.

4.4.2 Infer

The store management transition ($\rightarrow_i$) is where additions to the store (in the passive store S) are reflected to the denotations of the variables involved (kept in the active store C). Here we will give a characterisation of how given a consistent S we can evaluate its unique set of variable denotations (C'). Each denotation in C' reflects a static distribution.

$$infer(C,S) = \begin{cases} (C, \{X = l(v_i(V_i))\} \cup S') & \text{, if} \quad v_i(V_i) \in C \wedge \\ & \qquad S = \{X = l(V_i)\} \cup S' \\ (C, \{X = sd(v_i(V_i))\} \cup S') & \text{, if} \quad v_i(V_i) \in C \wedge \\ & \qquad S = \{X = sd(V_i)\} \cup S' \\ (C, \{X = p(v_i(V_i))\} \cup S') & \text{, if} \quad v_i(V_i) \in C \wedge \\ & \qquad S = \{X = p(V_i)\} \cup S' \\ (\phi(\emptyset, \epsilon(S), S), S) & \text{, otherwise} \end{cases}$$

In the above definition of infer(C,S) the first three cases deal with the situations that require no change in C. These constraints require information about a particular variable, but do not change the probabilistic information held in the store. In each of this cases, the syntactic form of the constraint (in S) triggers a value to be assigned to its logical variable according to the corresponding function in $\mathcal{D}$. The fourth case deals with the remaining possibilities, which leave S unchanged. At the same time it replaces C with the new variable denotations that distill the probabilistic knowledge accumulated in S. This is achieved by the following functions.

$$\epsilon(S) = \{v_i(V_i) \mid \forall i \ V_i = \in_{\mathbf{P}_f}(Fd) \in S \text{ where } v_i(V_i) \text{ denotes } \in_{\mathbf{P}_f}(Fd)\}$$

$$\phi(V,L,S) = \begin{cases} \phi(\{\alpha(v_i(V_i), V, S)) \cup V\}, L', S) & \text{, if} \quad L = \{v_i(V_i)\} \cup L' \\ & \qquad \wedge\ dep(V_i, S) \subseteq V \\ V & \text{, otherwise } (L = \emptyset) \end{cases}$$

$$\alpha(v_i(V_i), V, S) = \begin{cases} \alpha(cond(v_i(V_i), v_j(V_j)), V, S') & \text{, if } \; v_j(V_j) \in V \wedge \\ & S = \{cond(V_i, V_j) \cup S'\} \\ v_i(V_i) & \text{, otherwise} \end{cases}$$

The intuition behind $\epsilon(S)$ is that it is the set of definitional denotations for all probabilistic variables in S. Function $\phi(V, L, S)$ constructs the desired C by depleting L and generating the conditional denotation that is incrementally added to V. Finally $\alpha(v_i(V_i), V, S)$ constructs the conditional denotation for a variable from its definitional one, according to the conditional constraints in S and the variable denotations in V.

Function ϵ is trivially finitely computable. Similarly for α although for the correct result it is required that

$$\forall j \;\cdot\; cond(V_i, V_j) \in S \rightarrow V_j \in V \;\; .$$

This follows immediately from the definition of *dep* and the condition $dep(V_i, V) \subseteq V$ in ϕ. In turn the correctness of ϕ depends on the truth of

$$L \neq \emptyset \rightarrow v_i(V_i) \in L \wedge \; dep(V_i, S) \subseteq V \;\; .$$

This statement is true whenever $consistent(C, S)$ holds (second case in the definition of consistency).

4.5 Transitions Strategy

The purpose of the transitions is to transform, preferably in a finite number of steps, the initial state $\langle G, \emptyset, \emptyset \rangle$ to the final store $\langle \emptyset, C, S \rangle$. To achieve this we present some rules and restrictions for the order of their application. These rules and restrictions collectively formulate the transitions strategy.

A common approach to defining such a strategy is via notions such as fair application of transitions and applications until an invariant state is reached. In *Pfd* there are simpler means of defining such a strategy. This is mostly due to: (a) the importance of static distributions as a value representation, and (b) the ability to characterise D solely in terms of functions.

The importance of controlling the application of transitions has already been highlighted in the previous section; where the correctness of $\rightarrow_r$ require a consistent store (ensured by a preceding $\rightarrow_c$). In full the rules governing transition application are:

- $\rightarrow_r$ can only be followed by $\rightarrow_r$ or $\rightarrow_c$.

- $\rightarrow_c$ is always followed by $\rightarrow_s$.
- $\rightarrow_i$ can only be followed by $\rightarrow_r$ or $\rightarrow_c$.
- $\rightarrow_s$ is always followed by $\rightarrow_s$.

Finally, we note that although the characterisation of $\rightarrow_r$ involves a re-evaluation for all the variables from their definitional distribution, it is very often the case that this can be done incrementally. This depends on knowledge of the form for the preceding $\rightarrow_c$ transition.

Chapter 5

Probabilistic Programs

In this chapter we show how some well known problems can be modelled in *Pfd* . These problems are mainly drawn from the area of statistics. The key questions for this chapter include the following. How can intuitive causal probabilistic declarations be part of a sound model ? Is it possible to model analytical reasoning with the extensional tools of *Pfd* ?

5.1 The Three Curtains

One of the staple examples that lecturers use for cautioning freshers with, is the Monty Hall problem. This problem is an example of case where even in seemingly simple situations, *intuition* may lead to erroneous conclusions. Although people can readily deal with the probabilities on the constituent parts, still they have difficulty in combining these probabilities even in this limited example.

The general form of this game involves a contestant (also referred to as the player, P) being offered a choice of three curtains (A, B and C). One of the curtains veils a car while the other two hide a goat, each. Once the contestant expresses his choice the host, reveals a goat from behind one of the remaining two curtains, which has not been chosen by the player. At this stage of the game there are two closed curtains remaining. The host then asks the contestant whether he wishes to change his mind or to stay with the original choice. The player at this stage has to make his final decision. If the player chooses the curtain veiling the car, then we say the player won the car or simply that he won.

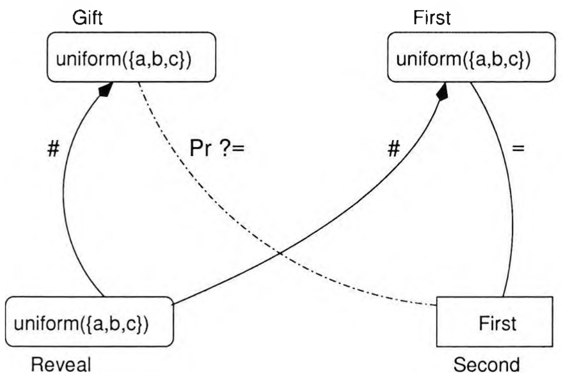
Given this description the students are then asked to form an opinion on whether defining a strategy for the player's final choice has an influence to player's chances of winning the car.

Usually the are two candidate strategies offered to them: (α) player continues with original choice, (β) player swaps the original choice. Most students seem to agree in that there is no difference in the chances of these two strategies.

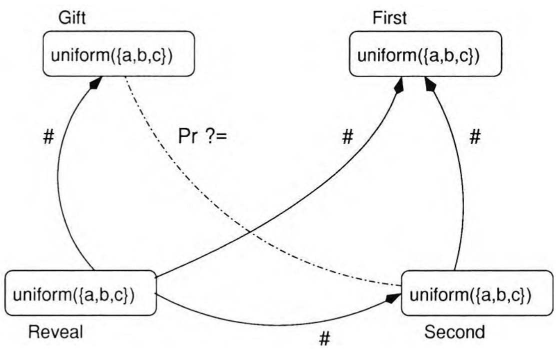
Statistical analysis reveals that strategy β leads to a better chance of success $p(\beta) = \frac{2}{3}$ (we denote the success of strategy β by $p(\beta)$ and similarly for any remaining strategies). Whereas, the probability of a win under strategy α is $p(\alpha) = \frac{1}{3}$. In the later case the derivation is straight forward since the player's second choice carries the same probability as that of the first choice; which, being an uneducated guess between three indistinguishable objects, is equal to $\frac{1}{3}$. On the other hand, for any original choice in strategy β, the chances of the car being behind the not-chosen curtains is $\frac{2}{3}$. This is so, because the prior probability of the car being behind the curtain, which was opened by the host is also incorporated. An alternative derivation of $p(\beta)$ when we have $p(\alpha)$ is by using the fact that the two strategies are complimentary and exhaustive, thus

$$1 = p(\alpha) + p(\beta) \Rightarrow p(\beta) = \frac{2}{3}$$

In general terms the player's dilemma can be captured in a single strategy (γ) where the choice of swapping the original curtain occurs with a probability p_s. Strategy α is the special case when $p_s = 0$ and β is equivalent to $p_s = 1$. Based on γ the chances given a particular p_s are, $p(\gamma) = \frac{1+p_s}{3}$.



(a) α Strategy



(b) β Strategy

Figure 5.1: Graphs for strategies α and β.

We start modelling the problem with a schematic representation for strategies α and β in Figure 5.1. In these graphs, nodes represent probabilistic variables and solidly drawn edges mark constraints between variables (labelled and directed accordingly). Dashed edges labelled with a question mark indicate that the probability of the event on the right side of the question mark is assigned to the variable on the left side. The predicate is applied on the two connected variables and direction is not important in this case.

Our main reason for including these graphs is to illustrate the level at which the programmer is required to think about probabilities in *Pfd*. At the global level he needs to decide which objects of the real world become variables in the program and for each variable to define its overall probabilistic behaviour. At the local level, connections between probabilistic objects are confined to a minimum by constraints involving exactly two variables.

curtains(α, Pr) :-
  $Gift \in_{\mathbf{p}} uniform([a,b,c])$,
  $First \in_{\mathbf{p}} uniform([a,b,c])$,
  $Reveal \in_{\mathbf{p}} uniform([a,b,c])$,
  $Reveal \mid\!\neq Gift$,
  $Reveal \mid\!\neq First$,
  Second $=$ $First$,
  $Pr\ is\ \mathbf{p}(Second{=}Gift)$.

Figure 5.2: Clause for strategy α.

curtains(β, Pr) :-
  $Gift \in_{\mathbf{p}} uniform([a,b,c])$,
  $First \in_{\mathbf{p}} uniform([a,b,c])$,
  $Reveal \in_{\mathbf{p}} uniform([a,b,c])$,
  $Second \in_{\mathbf{p}} uniform([a,b,c])$,
  $Reveal \mid\!\neq Gift$,
  $Reveal \mid\!\neq First$,
  $Second \mid\!\neq First$,
  $Second \mid\!\neq Reveal$,
  $Pr\ is\ \mathbf{p}(Second{=}Gift)$.

Figure 5.3: Clause for strategy β.

In Figures 5.2, 5.3 and 5.4 we give clauses for the three strategies. The first argument of clauses *curtains/2,3* is simply a strategy identifying atom, while the last one is the probability of winning the car when following the respective strategy. The clause for strategy γ has an extra argument which should be instantiated to p_s. In all three cases $Gift, First$ and $Reveal$

curtains(γ, SwapWith, Pr) :-
    $Gift \in_{\mathbf{p}} uniform([a,b,c])$,
    $First \in_{\mathbf{p}} uniform([a,b,c])$,
    $Reveal \in_{\mathbf{p}} uniform([a,b,c])$,
    $Second \in_{\mathbf{p}} uniform([a,b,c])$ **|#** $First$ *with* $SwapWith$,
    $Reveal$ **|#** $Gift$,
    $Reveal$ **|#** $First$,
    $Second$ **|#** $Reveal$,
    Pr *is* $\mathbf{p}(Second{=}Gift)$.

Figure 5.4: Clause for γ strategy.

are mapped to uniformly distributed probabilistic variables, each having three possible values. $Gift$ is a straightforward representation of the curtain concealing the gift. In the absence of other evidence we assume that the gift is placed randomly. In a similar way, $First$ models the player's first choice. Variable $Reveal$ is more interesting, since one could argue that the host has at most a choice between two rather than three curtains. Some times the host does not have a choice at all, since the car cannot be revealed. By using the complete finite domain we believe that the process of problem solving is better facilitated. When an object is declared, the programmer is encouraged to think about the generic (probabilistic) behaviour of the object. This is very much followed in our clauses.

Furthermore, there are some common constraints. The fact that the revealed curtain cannot be the concealing one is captured by $Reveal$ **|#** $Gift$. Similarly, it cannot be the player's first choice, thus we have $Reveal$ **|#** $First$. These are directional constraints (as described in 3.4) that work in very similar way to Logic Programming clauses. Ideally, in both cases is that we only need to express the logical relationship between the variables, but in practice the choice of a particular expression may lead to more efficient computations. The final goal, in all three clauses, assigns the probability of the win event to the logic variable Pr. A win is achieved when the values for $Second$ and $Gift$ coincide, this being tested with $Second = Gift$.

In strategy α (clause in Fig 5.2) the player's decision to proceed with the first choice made, is elegantly captured by the unification of a logical variable, $Second$ to the probabilistic variable $First$. In strategy β, $Second$ becomes a probabilistic variable, with the constraint $Second$ **|#** $First$, capturing the always-swap strategy. Finally, for strategy γ we need an extra parameter (SwapWith) which will control the preference between the two choices, at the player's second decision. With this at hand, we use the conditional probability definition to capture the definitional dependency of *Second* to *First*, by

$$Second \in_{\mathbf{p}} uniform([a, b, c]) \mid\!\ddagger\; First \;\; with \;\; SwapWith$$

This constraint states that the variable, *Second*, assumes the same value as that of *First* (this need be a scalar value and not a set) with probability $1 - SwapWith$ and the rest of the values in its finite domain ($\{a, b, c\}$ here) with a probability of SwapWith (uniformly distributed between them).

5.2 Enumerations

The first step in trying to determine probabilities of events is to count the number of different possibilities. In statistics the different methods devised for assisting with such tasks are collectively known as methods of enumeration. For the significance of these methods in statistics and an excellent introduction to the field the reader is advised to follow [Fel59]. Here we will focus on the four methods of enumeration which are most well known. The objective is not that of competing in terms of efficiency to any other methodology, but rather to show that the extensional mechanism of *Pfd* can yield correct results for such analytical methods.

The enumeration methods we will be looking at, deal with cases where a number of objects are selected from a collection of alternatives. In particular, they can deal with two orthogonal qualities of such tasks. The first is whether one draws the objects with replacement. While the second is whether the order in which objects are drawn is significant. For each case we give, a brief example, the analytical formula, and a *Pfd* clause that models this method. For further mathematical analysis, of the methods the reader is referred to any standard textbook dealing with statistics and Probability Theory. Here, we will refer to the process of choosing (r) objects from a collection of (n) alternatives, as *sampling*.

5.2.1 Sampling with replacement, when order matters

As an example of such sampling, consider a digi-code that is formed from four digits each drawn from ($\{0, \ldots, 9\}$) when digits are taken one at the time, noted down, and replaced before the next digit is drawn. We are interested in counting how many different ordered combinations exist. (Order matters, means that code 0123 is distinguishable from code 3012.) The analytical formula for this is n^r, where n is the number of alternatives (here n=10) and r the number of objects drawn (r=4).

In the clause presented in Figure 5.5 the digi-code is represented by a list of probabilistic variables all having uniformly distributed domains. In addition, each variable ranges over the

```
ordered_with_replacemement( R, Fd, Pr ) :-
    n_pfd_vars( R, uniform(Fd), Code ),
    n_random_selection( R, Fd, Assigned ).
    Pr is p(Code=Assigned) .
```

Figure 5.5: Clause for ordered sampling with replacement.

same finite domain, which is the list of all digits. Predicate *n_pfd_vars/3* creates the list of R variables (Code), each defined with probabilistic method $uniform$ over the finite domain Fd. (For a complete definition of the *n_pfd_vars/3* predicate, see Appendix B.) The rest of the predicate helps us test the validity of our claim. (The predicate *n_random_selection/3* creates a random list, Assigned, of R elements chosen from the Fd list of elements.) Thus for the particular four digit code example a query of the form

$$? - ordered\_with\_replacemement(4, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], Pr).$$

should instantiate Pr to the chances of a randomly single combination which is $\frac{1}{10^4}$.

5.2.2 Sampling without replacement, when order matters

If in the previous code example, we are not allowed to replace digits after each draw, then all the dig-codes that include repetitions of digits are automatically barred. This constitutes sampling without replacement. The analytical formula giving the number of arrangements in the unordered case is,

$$\frac{n!}{(n-r)!} = P_r^n$$

These are known as the permutations of n objects taken r at a time. In the particular example presented above (n=10,r=4) we have,

$$P_r^n = \frac{10!}{6!} = 10 * 9 * 8 * 7 = 5040$$

The clause in Figure 5.6 defines a digi-code by means of the *n_pfd_vars/3* predicate as done for the replacing case. The new element is that all variables in the list *Code*, are now required to hold a distinct value. This is achieved by using the predicate (see appendix B) *distinct/1*, which simply posts a number of pairwise conditional differences constraints. Due to the symmetry of the problem, directionality is of no importance in this example. Again we test by mean of a random list. The extra twist is that a query to this predicate may return a $Pr = 0$ since Assigned may include repetitions of digits. In all other cases $Pr = \frac{1}{5020}$.

```
ordered_without_replacemement( R, Fd, Pr ) :-
    n_pfd_vars( R, uniform(Fd), Code ),
    distinct( Code ),
    n_random_selection( R, Fd, Assigned ).
    Pr is p(Code=Assigned) .
```

Figure 5.6: Clause for ordered sampling without replacement.

5.2.3 Sampling without replacement when order does not matter

In many situations the elements from the sampling space may fall in categories that make them indistinguishable for the purposes of an enumeration. This scenario is often met in situations of repeated experiments. As an example, consider the number of the different signals one can send when using 3 yellow and 4 red flags on a single post. The distinguishable arrangements are often called, the combinations of n objects taken r at a time. The number of combinations is calculated by :

$$C_r^n = \frac{7!}{4!3!} = 35$$

```
un_ordered_without_replacemement( R, Fd1, Fd2, Pr ) :-
    append( Fd1, Fd2, Fd ),
    length( Fd1, L1 ),
    n_pfd_vars( R, uniform(Fd), Code ),
    split_on( L1, Code, Vs1, Vs2 ),
    map_constr_pwise( Vs1, X1 @< Y1 | Y1, X1, Y1 ),
    map_constr_pwise( Vs2, X2 @< Y2 | Y2, X1, Y1 ),
    n_random_selections( R, Fd, Assigned ),
    Pr is p(Code=Assigned) .

map_constr_pwise( [], _Constraint, _Lv, _Rv ).
map_constr_pwise( [_F], _Constraint, _Lv, _Rv ).
map_constr_pwise( [F,S||T], Constraint, Lv, Rv ) :-
    copy_term( (Constraint)-Lv-Rv, (Instance)-F-S ),
    call( Instance ),
    map_constr_pwise( [S||T], Constraint, Lv, Rv ).
```

Figure 5.7: Predicates for unordered sampling without replacement.

In Figure 5.7 we give the two sets of values (Fd1 and Fd2) that correspond to the collections of objects in the indistinguishable pools. The main novel predicate is the mapping of constraint $X1@ < Y1$ I $Y1$ in a pairwise fashion down the elements, of the two lists. This is an extensional means for capturing the fact that within each list, elements are indistinguishable and thus we impose an order on them as to avoid undesirable repetitions. For instance if y_n stands for the nth yellow flag, in essence we do not allow the arrangement $y_2y_1y_3r_1r_2r_3r_4$, because the position of y_2 violates the positioning constraint. In the analytical description that combination is indistinguishable from $y_1y_2y_3r_1r_2r_3r_4$. Similar to the other clauses a call to the procedure will instantiate Pr to the probability of a single arrangement, which in this case is $\frac{1}{35}$.

5.2.4 Sampling with replacement when order does not matter

This can be viewed as a special case of sampling without replacement when order does matter, since it can be reduced to listing r 0's for the objects to be selected and n-1 —'s for the number of different groups available. If we then extend the example with the flags, to the case where there is a sufficiently large number of flags for six different colours from which we wish to make signals of 10 flags, one particular listing would be :

00|0|00|0000||0

Which reads as selecting 2 flags from colour one, 1 from second colour, 2 from third, 4 from the fourth, none from the fifth and 1 from the sixth colour. The formula is

$$C_r^{n-1+r} = \frac{(6-1+10)!}{6!(10-1)!} = \frac{15!}{6!9!} = 30030$$

Since this case is a specialisation of the non-replacement one, it is obvious that the clause in Figure 5.7 can be used.

5.3 Caesar Code

The curtains problem (Section 5.1) illustrated the expression of probabilistic causation and how *Pfd* can be used to compute over such relations. In this Section we will concentrate on how *Pfd* can be used to reason with statistical information.

5.3.1 Problem Definition

The Caesar encoding is one of the earliest encoding schemes. The basic idea is that each letter of the alphabet is encoded by another (unique) letter and that each letter in the encoding correspond to a single decoded character (see Table 5.1 for an example of a valid encoding).

dict. chars	encoded chars
a	t
b	s
c	b
d	a
e	o
h	h
i	c
l	p

dict. chars	encoded chars
m	w
o	f
p	r
r	y
s	m
t	v
u	n
v	l

dictionary words	amicable	cohered	euphoria	mutant	shame	verb
Caesar encoding	twcbtspo	bfhoyoa	onrhfyct	wnvtdv	mhtwo	loys

Table 5.1: A Caesar encoding example.

The specifics of an encoding include information such as whether one is pooling from lower and upper case letters, or where space and other special characters are encoded. In the remaining of this section, we will be looking at the encoding of letters that are encoded in lower case character codes. More specifically, we show how a number of encoded words drawn randomly from a *dictionary* can be decoded back to their original character codes. The dictionary used is the one found in Unix operating systems (*/etc/share/words*) which contains more than 20000 words. From this dictionary we sampled words of at least four characters (no uniqueness constraint was imposed on this rule). As a final assumption we state that all legal decodings appear in the dictionary.

Note that this formulation of the problem is in effect harder to solve than the case where words are picked from written text. This is because in the text based version we have extra information on word frequencies and about the frequency of two and three letter. With this information the size of the space is reduced while the statistics become more reliable.

5.3.2 Modelling the Problem

In *Pfd* terms, the first important question in modelling a problem is "Which objects in the problem will be the variables?". Here the obvious candidate is one variable per encoded letter. It thus follows that each variable takes values over all the decoded characters. The more interesting part is the probabilities attached to each possible value.

Before determining the probabilities, we give an example of how variables can be used to model the Caesar coding and what a solution might look like. As an example, consider an encoded word such as *'bfhoyoa'* from the words in Table 5.1. The desired outcome is a variable assignment that reflects the original word, which in this case is the dictionary word *'cohered'.*

We choose to represent each input character code by a single logical variable X_i. The subscript of each such variable, is a reference which relates the variable to the ith lower case encoded letter. So for instance, the encoded letter 'b' is represented by X_2. The domain of each of these variables are the letters of the alphabet under decoding. This in our discussions, will be the set of lower case character codes. Thus 'bfhoyoa' will be represented by

$$< X_2, X_6, X_8, X_{15}, X_{25}, X_{15}, X_1 > \tag{5.1}$$

with

$$X_i \in \{a, \ldots, z\}$$

The desired solution for this example reads :

$$< X_2 = c, X_6 = o, X_8 = h, X_{15} = e, X_{25} = r, X_1 = d >$$

Trying to solve the full problem as modelled so far, can be computationally very expensive. For instance the possible combinations for a four letter word are :

$$26 * 25 * 24 * 23 = 358800$$

In addition to this, consider the fact that any four letter word (in the dictionary) is as likely to appear as a combination as any other one. Also, we note that the consistency techniques come into play after we start instantiating some variables.

The classical approach in minimising the number of combinations which one will have to consider, before finding the solution, is to superimpose the two frequencies as to obtain likelihood measures that can guide searching. The first frequency we refer to, is the relative number of occurrences of a single character in the dictionary. While the second is that of characters in the encoded message. The intuition behind this approach is that letters which appear often in the dictionary are likely to appear equally often in the message.

In the approach we follow, the frequency of a letter in the encoded message (also referred to as *encoded character code*) is used as a marker. The frequencies of letters in the dictionary (*dictionary character codes*) are then compared to this marker. These comparisons, produce a number of proximity metrics. Each metric is indicative of the likelihood of the encoded character code to be an encoding of a particular dictionary character code. These proximity metrics are then normalised to produce the probability distribution over the finite domain ($\{a, ..., z\}$).

For instance consider variable X_2 of our previous example (5.1). This variable corresponds to the encoded character 'b'. We will denote the encoded characters with E_i, with the subscript relating each encoded character to a single variable. The particular code has a frequency (according to the data in 5.2) of $\frac{5}{38}$ derived by :

$$freq(E_2) = \frac{number\ of\ times\ E_2\ occurs}{total\ number\ of\ characters}$$

This frequency is used as a marker. From this a number of proximity measures are obtained, indicating the fitness of the dictionary character frequencies to that of the marker. To do this, we use the absolute value of the difference between the two numbers,

$$prox(E_i, C_j) = \mid freq(E_i) - freq(C_j) \mid$$

Where $freq(C_i)$ are frequencies appearing in Table 5.2 (These are simplified numbers, approximating frequencies from the full dictionary.)

C_i	freq(C_i)
a	24
b	8
c	14
d	12
e	31
f	5
g	8
h	6
i	23

C_i	freq(C_i)
j	0
k	1
l	14
m	6
n	20
o	16
p	4
q	0
r	18

C_i	freq(C_i)
s	18
t	16
u	7
v	2
w	1
x	1
y	4
z	1
$\sum C_i$	260

Table 5.2: Approximate dictionary frequencies (unit $\sim$ 260).

The final step in obtaining a distribution is to normalise the proximity figures. This process is often used for the derivation of a probability function from an arbitrary one. In so far as it concerns this example, our main task is to ensure that the values of the proximity numbers correspond to fractions that sum to one. Following well known practice in normalisation techniques we define :

$$p(X_i, C_j) = \frac{prox(E_i, C_j)}{\sum_k prox(E_i, C_k)}$$

Where $p(X_i, C_j)$ reads as the probability of variable X_i having the value C_j. Table 5.3 shows a number of variables, their corresponding encoded characters and parts of each variable's derived

X_i	E_i	$f(E_i)$	$p(X_i,a)$	$p(X_i,b)$	$p(X_i,c)$	$p(X_i,d)$	*out of*	...
X_1	a	5/38	5883	5731	5788	5769	/149500	...
X_2	b	2/38	1898	1942	1993	1980	/49900	...
X_3	c	2/38	1898	1942	1993	1980	/49900	...
X_4	d	1/38	1588	1740	1683	1702	/43775	...
X_5	e	6/38	7508	7356	7413	7394	/191750	...
...	...			...				...

Table 5.3: Proximity based probabilities.

probability. The collated formula derived from the above is :

$$p(X_i, C_j) = \frac{\mid \frac{freq(E_i)}{\sum_l freq(E_l)} - freq(C_i) \mid}{\sum_k \mid freq(E_i) - freq(C_k) \mid} \tag{5.2}$$

5.3.3 *Pfd* Specifics

Having defined the finite domains involved and the associated probability distribution, it is now straightforward to capture the variable definition with a user-defined function as described in Section 3.3. The definition appear in Figure 5.8. Its constituents refer to the *Pfd* rationals library (Appendix A) and it implements Equation 5.2.

```
probabilistic_method( proximity(Fd,Marker,Frqs,Probs) ) :-
    rationals_subtract_list_from( Frqs, Marker, Subtr ),
    map_list( rationals_abs, Subtr, Absl),
    rationals_add_list( Absl, Sum ),
    rationals_subtract_list_from( Absl, Sum, Diffs ),
    rationals_to_probabilities( Diffs, Probs ).
```

Figure 5.8: Clause for proximity method definition

The intuitive reading, is that the *proximity probabilistic method* constructs a list of probabilities *Probs* each corresponding to a single domain value in *Fd* (*Fd* is the finite domain of the variable). To construct these probabilities the input frequencies (in *Freqs*) are subtracted from a (Pivot) rational value, followed by the normalisation of the results from subtractions. This example illustrates the use of extra arguments in probabilistic method constructions (Section 3.3). An example of variable definition using this method is :

$$Letter \in_{\mathrm{p}} proximity([a,b,c], 1/2, [1/3, 1/4, 2/3])$$

Apart from the definition of probabilistic finite domains, we need a method for searching through the search space for candidate solutions. We achieve this through the general purpose labelling introduced in Section 3.7. In particular, we use :

$$label(Vars, max\_unique\_alt, Vals, ValPrbs, BranchPrb, CurrPrb).$$

The unique qualifier helps to guide the generation of values and according to our modelling is correct. The heuristic selection is used since minor re-orderings to the order of valuations are of no particular importance. This is due to the fact that we are looking for few solutions over a multitude of valuations (the sieving is through checks to the dictionary). On the other hand, the heuristic selection provides superior execution behaviour.

5.3.4 The Program

After placing the foundations in the preceding sections, we now proceed in presenting the top-most predicates that compose our *Pfd* solution to the Caesar encoding problem. For a complete listing of all the predicates the reader is referred to Appendix B.

```
% caesar( +EncodedWs, +Dict, -DecodedWords ) :-
caesar( EncodedWs, Dict, DecodedWs ) :-
    dictionary_info( Dict, DictWs, Freqs, AlphaBeto ),
    count_occurrances( EncodedWs, Codes, Counts, Sum ),
    proximity_vars( Codes, Counts, Sum, AlphaBeto, Freqs, Vars ),
    substitute_with_vars( EncodedWs, Codes, Vars, WordsVs ),
    decode_words( WordsVs, Vars, DictWs, Codes, DecodedWs ).
```

Figure 5.9: The *caesar/3* predicate.

The *caesar/3* predicate (5.9) provides an entry point to the decoding procedure. Its declarative reading is that *DecodedWs* are a valid decoding of the words in *EncodedWs* which are words drawn from *Dict*. Note that by words we will refer to a list of words, each represented, in the usual Prolog style, as a list of characters. The validity of words is expressed against the internal dictionary. Further to the procedural reading of *caesar/3*, the analysis of the input words to find the encoding characters and their frequencies, is achieved with *count_occurrences/4*. The necessary probabilistic variables are then defined in *proximity_vars/6*. The most important information needed therein, are the two frequencies to be super-imposed (*Counts* and *Freqs*). After being declared, the variables are formed into words by means of replacing the encoded character codes in *substitute_with_vars/4*. The final goal (*decode_words/5*) starts the recursive part of the solution.

```
% decode_words( +WordsVs, +Vars, +DictWords, +ChrCodes, -DecodedWs ).
decode_words( [], _Vars, _DictWs, _CharCodes, [] ).
decode_words( WordsVs, Vars, DictWs, CharCodes, DecodedWs ) :-
      min_cardinality_word( WordsVs, BestWordVs, UnqVsWord, RestWs ),
      label(UnqVsWord, max_unique_alt, Val, _VlProbs, _Prob, _AccProb)
      wordcodes_to_guess( UnqVsWord, WordVs, Vals, Guess ),
      word_in_dict_words( Guess, DictWs ),
      decode_words( RestWs, Vars, DictWs, CharCodes, DecodeWs ).
```

Figure 5.10: The *decode_words/5* predicate.

Predicate *decode_words/5* (Fig.5.10) is where the management of guessing words takes place. The overall task of decoding the code is broken down into: (a) by finding a good first word to decode, (b) have a guess on this and (c) recurse to acquire guesses for any remaining words. More specifically *min_cardinality_word/4* provides a means for selecting the next candidate word, its main intuition based on the number of uninstantiated variables within each word and moreover on the population of the domain for each such variable. The variables in the chosen word are then passed to the *label/6 Pfd* primitive, which provides combinations of values for them. Combinations are then reformulated into a single word (*wordcodes_to_guess/4*) that correspond to the candidate word (we refer to these re-formulations as *guesses*). Each guess is then checked against the dictionary (*word_in_dict_words/2*) with the goals being backtracked upon, until a guess is found in the dictionary words. Upon having a valid guess, the predicate recurses to complete the easier task of decoding the remaining words.

```
% proximity_vars( +CharCodes, +Occurances, +Sum, +Fd, +Frqs, -Vars ).
proximity_vars( [], [], _Sum, _Dom, _Frqs, [] ).
proximity_vars( [Code|Codes], [Occ|Occs], Sum, Dom, Frqs, [V|Vs] ).
      V ∈p proximity(Dom, Occ/Sum, Frqs) ,
      proximity_vars( Codes, Occs, Sum, Dom, Prbs, Vs ).
```

Figure 5.11: The *proximity_vars/6* predicate.

Predicate *proximity_vars/6* implements a straight forward list traversal procedure, but includes a good example of a real world use for the probabilistic variable declaration operator

($\in_p$). The arguments to the *proximity* term, match the first three arguments to the probabilistic method of Figure 5.8.

```
% min_cardinality_word( +WordsVs, -BestWord, -UnqVsWord, -RestWs ) :-
min_cardinality_word( [H|T], BestWord, UnqVsWord, RestWs ) :-
    remove_duplicates( H, NoDplH ),
    word_cardinality( NoDplH, 0, HCard ),
    min_cardinality_word( T, NoDplH, H, HCard, WordReps ),
    WordReps = UnqVsWord-Word,
    selects( Word, [H|T], Rest ).

% word_cardinality( +Vars, +AccCard, -FinCard ).
word_cardinality( [], Card, Card ).
word_cardinality( [H|T], AccCard, WCard ) :-
    domain_cardinality( H, HCard ),
    NxAcc is HCard + AccCard,
    word_cardinality( T, NxAcc, WCard ).
```

Figure 5.12: The *min_cardinality/4* and *word_cardinality/3* predicates.

The final predicates presented here (the full program is in Appendix B) are the ones which deal with the selection of a good candidate word for decoding. The interesting work is happens within *word_cardinality/3*, where the cardinality of a word is equal to the summation of the cardinalities of its constituent variables. (The cardinality of a single variable is supplied by the *Pfd* predicate *domain_cardinality/2*, as defined in Section 3.8.) Minimisation over word cardinality means we concentrate on the word that has minimum degrees of freedom to become a guess. This provides a simple yet highly effective (in this case) selection criterion amongst different words. While the computation is at its first stages (when all variables are uninstantiated), the selection procedure will pick one of the words with the minimum number of letters. As the computation proceeds, the words with the minimum number of uninstantiated variables will be easily identified and selected.

5.4 Other Problems

As a final remark to this chapter we list some areas in which *Pfd* might be applied successfully. We also present marginal argumentation on how and why the techniques introduced in this chapter can be used to solve problems in these areas.

5.4.1 Biology

In certain applications of Nuclear Magnetic Resonance (NMR), spectroscopy biologists need to match experimental data to a best fitting protein structure. The probabilities involved are of two kinds. On one hand, there are probabilities on the observed values, while on the other hand, they are chain continuation likelihoods. The nature of the problem is very similar to the codes example we have presented. The main difference is that the two probabilities are not so much superimposing, but rather interlocking to each other.

Another problem comes from the area of molecular biology and in particular from protein packing. The goal is to predict the folding of side-chains of a protein. The current approaches use specialised algorithms that may or may-not take into account that certain foldings are more likely than others. The main drawback of these algorithms is that they are pieces of codes, which are only good in performing this particular task. But protein folding is only a means in itself thus more compositional tools would be of help.

One way to model this problem within *Pfd* is to use the likelihood of foldings to drive the search of solutions. (Via a multi-variable labelling.) The challenging programming required, will be to identify less significant (peripheral) chains which do not alter the topology of the protein.

Furthermore, we believe that biologists will benefit and be able to better utilise computing resources with the abstract notion of probability presented by *Pfd* and the declarative style of programming in which it is so naturally embedded.

5.4.2 Dynamic Scheduling

Dynamic scheduling refers to scheduling in environments where one is trying to schedule unforeseeable or uncertain tasks or events that affect availability of tasks. (Similarly one may consider dynamic planning.) Most current approaches deal with these problems at the rescheduling level, where usually one is concerned with keeping information about previous decisions at hand, in the event that rescheduling might occur.

Pfd can be used in these areas to unify uncertainty about task availability, as well as about uncertainty in the domain of discourse. Thus, all reasoning and meta-reasoning can be considered at a single level.

Chapter 6

Implementation

In this chapter we present a prototype system implementing most of the *Pfd* constructs. Firstly we discuss some of the syntactical issues and the overall architecture of the system. This is followed by a brief description of some techniques employed. We have tried to describe the system implementing *Pfd* in such a fashion as to provide insight to the task of implementing *Pfd* under any programming language.

6.1 Overall

We have built an evaluation system that implements the majority of the *Pfd* constructs. For ease of reference we will refer to the code implementing *Pfd* as the *system* or *meta-interpreter*.

Important interactions between *Pfd* and the meta-interpreter :

- *Pfd* and the system had a gradual and interleaved development,
- feedback from the system was paramount since our research hopes to produce a practical programming language subset,
- the system proves that *Pfd* is highly integrative.

The system has been implemented as a meta-interpreter in Prolog. Its two main differences from standard Logic Programming meta-interpreters are the use of a store to carry the constraints information (which is a technique from CLP) and the use of built-in transformations to map *Pfd* to Prolog clauses. The result is a robust shell based on high level programming constructs. This has been an invaluable help to our research since fundamental changes in *Pfd* itself were possible to be integrated within minimal amount of time and programming effort.

The first part of the meta-interpreter is in essence performing a consultation/compilation step. During this phase, programs are read-in and clauses are asserted in the internal database. These clauses are of a syntactic form that facilitate execution during the next step. Execution refers to the phase where queries are answered against the internal database in the normal Prolog interaction. A query is in general, a conjunction of goals. In this system, each of the goals belong to one of two main categories. A goal can either be a Prolog clause, in which case normal unfolding takes place, or it might be a constraint, in which case some interaction with the store will take place. (Unless otherwise stated or easily inferred from the context, we will use the term *constraint* to refer to *Pfd* constraints.)

Constraints are the means for updating and interrogating a global (i.e. query-wide) data structure. In accordance with established constraint research we refer to this structure as the *store*. Operations on the store must perform two main duties. Firstly, that the addition of new conditional constraints does not introduce cyclic (conditional) dependencies and secondly, that when calculating the probability of events, all current information are taken into consideration. It is important to note, that unlike other constraint systems, the *Pfd* store does a minimal amount of pro-active consistency checking.

From a probabilistic perspective pro-active consistency is not appealing; particularly when considering the objectives and restrictions of *Pfd*. From a finite domains perspective, the story of course is a different one, since consistency checking is a major area of research with important results and well established techniques. In this regard, we have introduced the novel notion of decoupling the domain elements from particular probability values. This in effect, ensures that *Pfd* will be able to work along side existing finite domain solvers, thus compositionally incorporating the best consistency techniques.

The meta-interpreter does not depend on any Prolog system specifics since it is largely written in pure Prolog and when this was not possible ISO standard compliant code had been used. The original platform was SICStus Prolog, which provided a reliable system that is widely used in academic institutions. One particularly useful feature of the platform, is its arbitrarily long integer arithmetic capabilities. On the other hand, the meta-interpreter has already been ported (minimal changes) to a freely available Prolog system (SWI Prolog), thus making it easier for people to evaluate and use it.

The first step in realising *Pfd* is to approximate the introduced operators by more machine friendly character combinations. The correspondence of operators is given in Table 6.1. (Note that entries appearing on the fourth column are the actual declarations of the operators, defining precedence and association.) The compilation of the meta-interpreter's operators, is achieved with a number of predicates that are used by the Prolog engine while reading program clauses.

Description	*Pfd*	meta	declaration
Var Defn.	$\in_{\mathbf{p}}$	pin	op(950, xfy, (pin))
Conditional	$\mathbf{I}$	/	op(900, xfy, (/))
Cond. Different	$\mathbf{I}_{\neq}$	/#	op(900, xfy, (/#))
Qualifier Eq.	$=$	$=$	-
Qualifier Diff.	$\neq$	#	-

Table 6.1: Operators Correspondence

Some of these predicates are presented in Fig. 6.1. The entry point for the compilation is via *term_expansion/2*. The order is important since these clauses change the behaviour of the program that reads them in. Transformation of each term culminates in the assertion of a *pfd_clause/2*, the first argument being the head, while the second is the body of the clause.

The answering of queries is handled by the customary *demo* predicate. Within each iteration of the *demo/4* predicate (Fig 6.2) the meta-interpreter tries to reduce a *selected goal.* There are three possibilities to the identity of the selected goal. Two of the alternatives deal with Prolog predicates. Prolog predicates are either user defined ones, in which case they are unfolded and their body is added to the goals to be proven, or they are built-ins, in which case predicates are funnelled through to the underlying Prolog system. Both of the preceding cases make no use of the information held in the store. Store interactions are solely managed when the selected goal is a constraint. For the sake of brevity, we refrain from presenting more code here, but the interested reader can find the top-level predicates concerned with store interaction in Appendix C.

Here we give a description of store interactions. This is achieved by splitting the constraints into three categories and providing accounts for each category.

Additions to the store. When adding conditional constraints we need to check for cycles. Thus, we need to make sure that in when the selected goal is a conditional constraint then the *Qualifier* variable (Section 3.4) does not dependent on *Dependent* variable. Whereas, variable definitions have to make sure that the variable in question is of the correct form.

Derivation of information from the store. The common characteristic in this case is that the store remains unaffected, while some information is extracted from it. In the case of information about a specific variable, this is a straightforward matter.The more interesting case is when we enquire about the probability of a predicate. The benefit of using Prolog for our implementation, is that probabilistic variables within the predicate can be substituted with (eventually all) alternative values and then test the validity of the pred-

```
expand_pfd_goal( (Left/Right), pfd_con(cond(ExpL,ExpR)) ) :-
    expand_constraint( Left, ExpL ),
    expand_constraint( Right, ExpR ).

expand_pfd_goal( (VarL/#VarR), pfd_con(diff(VarL,VarR)) ).

expand_pfd_goal( (Var pin Method), pfd_con(prin(Var,MethodStr)) ) :-
    expand_method( Method, MethodStr ).

expand_pfd_goal( Var is p(Predicate), pfd_con(prob(Predicate,Var)) ).

expand_pfd_goal( AnythingElse, AnythingElse ).

term_expansion( Clause, (:- true) ) :-
    ( Clause = (Head:-Goals) - >
        expand_body( Goals, ExpGoals ),
        ensure_list( ExpGoals, ListOfGoals ),
        assertz( pfd_clause(Head,ListOfGoals) )
        ;
        assertz( pfd_clause(Clause,[]) )
    ).
```

Figure 6.1: Clausal transformations with *expand_term/2*.

icate (call execution). The information derived, in that case, is the sum of products of probabilities of all variable valuations for which the predicate is true (execution succeeds).

Constriction of variable information. One or more variables, which are present in the store valuate to single values. Upon backtracking all possible valuations occur. In the sake of uniformity these constrictions in the meta-interpreter are implemented as singleton sets. A more performance-aware approach might allow atom entities in the domains.

As a final, but important, note to the discussion about the overall architecture of our system, we comment on the internal representation of probabilistic variables. Our choice has been to represent probabilistic variables as atoms[1]. In this respect we lose some of the power Prolog variables posses. What is gained is the ability to have an extra syntactic handle for manipulating internal objects. This was an important feature while developing a prototype, which may be altered in improved implementations of *Pfd* within Prolog.

```
demo( [], Store, Store ).
demo( Goals, InStore, OutStore ) :-
    goal_select( Goals, Selected, Rest ),
    demo( Selected, Rest, InStore, OutStore ).

demo( PfdClause, Goals, InStore, OutStore ) :-
    pfd_clause( PfdClause, Body ),
    body_and_goals( Body, Goals, NxGoals ),
    demo( NxGoals, InStore, OutStore ).

demo( pfd_con(Constraint), Goals, InStore, OutStore ) :-
    constraint_to_store( Constraint, InStore, NxStore ),
    demo( Goals, NxStore, OutStore ).

demo( Goal, Goals, InStore, OutStore ) :-
    prolog_predicate( Goal, InStore, MoreGoals ),
    body_and_goals( MoreGoals, Goals, NxGoals ),
    demo( NxGoals, InStore, OutStore ).
```

Figure 6.2: The demo predicates.

[1]e.g. of the internal representation of a variable, 1224_

6.2 Techniques

In this section we take a closer look to some lower level algorithmic issues of the system. Where possible we will present them in as abstract means as possible, in order to make the technical argumentation applicable to the general task of implementing *Pfd* in any programming language.

6.2.1 Rationals

Probabilities throughout the system are represented as rationals. In particular, in the form of quotients, which have an integer numerator and an integer denominator (with the restriction, denominator $\neq 0$). Our task has been greatly assisted by the presence of fast arithmetic operations for arbitrary long integers, in the implementation platform and an ancient algorithm for finding the greatest common divisor (Gcd) of two integers. (The algorithm is attributed to Euclid.)

```
gcd( M, N, C ) :-
    ( N =:= 0 ->
        C is M
        ;
        NewN is M mod N,
        gcd( N, NewN, C )
    ).
```

Figure 6.3: Euclid's Gcd Algorithm.

The Prolog code for Gcd is given in Fig 6.3. This code is used in operation between rationals (notably addition and subtraction) and for removing common factors present in both the numerator and the denominator of a single rational. The algorithmic analysis of these techniques is outside the scope of this thesis since they have been researched and analysed widely in the literature. What is important from our standpoint, is that we have used this algorithm as a successful means for implementing the arithmetics of probabilities.

6.2.2 Detection of Cycles

When adding constraints to the store, the system checks for cyclic dependencies which may later lead to inaccuracies in the evaluation of probabilities for some event. As noted in the previous Section, additions to the store occur when dealing with variable definitions and conditional constraints. Our discussion here is pertinent to conditional constraints and to conditional variable definition (in as far as this is a shorthand for a variable definition and a single conditional constraint).

The main reason why cycles are disallowed has to do with the directionality of the conditional constraint. For example, the constraint $X = 2 \mid Y\#3$ should be read as, variable X takes the value two when the value of variable Y is other than three. From an operational perspective, whenever X appears in a predicate for which we seek a probability, its value will be conditioned over the possible values of Y. Which in turn, requires the distribution of Y. If now, Y depends on X, we arrive at the point where the determination of the distribution of X, depends on having a distribution for the same variable. Thus, what we need to ensure is that Y does not, in any number of steps, conditionally depend on X.

An example of a cyclic dependency occurs in the following :

$$X = 2 \mid Y\#3, \ldots, Y = 4 \mid Z = 1, \ldots, Z\#3 \mid X\#1$$

The behaviour of the system upon encountering this cycle is that of printing an error message and terminating execution.

In order to facilitate the detection of cycles, a graph of dependencies is kept in the store. This is a directed acyclic graph. The nodes of the graph are the probabilistic variables in the store and edges represent a *depends_on* relationship. A consistent graph for part of our example above is shown in Fig. 6.4. When the final constraint ($Z\#3 \mid X\#1$) appears, the graph is checked for the existence of a path from X to Z. Since such a path exists, the addition of (Z depends_on X) would introduce a cycle to the graph and is therefore barred.

$$\begin{tikzcd}
 & X \arrow[r] & Y \arrow[r] \arrow[dl] & V \\
W & Z \arrow[l] & &
\end{tikzcd}$$

Figure 6.4: Consistency example graph.

The operation for finding a path between two given nodes in a graph, can be implemented in fairly inexpensive fashion[2]. The standard algorithm runs in $O(n \, log \, n)$ time, where n is the number of nodes in the graph (here this is the number of probabilistic variables, occurring in conditional constraints). The operation will occur m times, where m is the number of conditional constraints.

[2]particularly so, for directed acyclic graphs

6.2.3 Heuristic Labelling

One of the reasons why probability has not been used in many practical systems is because of efficiency considerations. Here we will show how in some cases we can employ generic heuristic to overcome this drawback. By heuristics, we refer to rules that operate on some locality of a particular area and which are used to provide, either an approximation to a globally optimal value, or a globally suboptimal solution. The need for such an approximation, is normally due to the fact that the calculation of the global value requires computations that are far more expensive than the heuristic ones. The drawback in many approaches which use heuristics, is that the heuristics are treated as first class objects, in which case the semantics of the computation are tainted by the semantics of the heuristic, which in more cases than not, are not sound.

Pfd takes a different approach, in that it always maintains the soundness of the computation, but allows for the use of heuristics to alter the order in which results are found. This is achieved by allowing, in controlled situations, to trade efficiency for accuracy in the order in which valuations occur.

One way of defining the notion of order accuracy more precisely is by introducing the sequence e, of the form

$$e = \langle 1, \ldots, n \rangle$$

Let e' then denote a permutation of the original order (e) of elements ($\langle 1, \ldots n \rangle$). We quantify *how* accurate a valuation is by means of M, where :

$$M(e') = \sum_{i=1}^{n} \frac{\mid e'_i - i \mid}{n^2}$$

M takes values in the interval $[0,1]$ with $M = 0$ iff $e = e'$.

The heuristic handle provided by *Pfd* is the label constraint (in particular the labelling of n variables). Here we will concentrate on the max_unique_alt selection method (Section 3.7) although similar techniques can be applied to alternative heuristic labelling schemes. This constraint has the form,

$$label(Variables, max\_unique\_alt, Values, VlPrbs, BrPrb, TotPrb)$$

The objective is to provide one Value for each variable in Variables. Each $Value$ has an associated probability (in $VlPrbs$). $BrPrb$ is the product of all the $Value$ probabilities, denoting the current branch's overall probability. $TotPrb$ is the summation of all the branch probabilities for the valuations considered so far. Furthermore max_unique_alt prompts for all $Values$ to be unique. The desired behaviour is that the constraint will succeed a number of times (upon backtracking) where each subsequent success bounds *BrPrb* to a value smaller than any of the preceding ones. So for instance, the first time the constraint succeeds, we want the most probable combination of *Values* (marked by the maximum *BrPrb*).

Finding the most probable combination is an expensive operation that in the average case has $O(n!)$ behaviour. In addressing this problem, we devised an algorithm that is polynomial in finding a good candidate, but which does not guarantee that the found candidate is the valuation with the next best *BrPrb* value.

This is a search algorithm that traverses the space of possible valuations according to an approximation heuristic. Each partially instantiated branch[3] upon introduction to the tree gets assigned a value which approximates the branch's final probability. At each stage the children of the most promising branch are added to the tree (in the case of a leaf, its *Values* are returned to the top level). Thus, at a random time the tree is expected to have branches at various depths (essentially a form of best-first search).

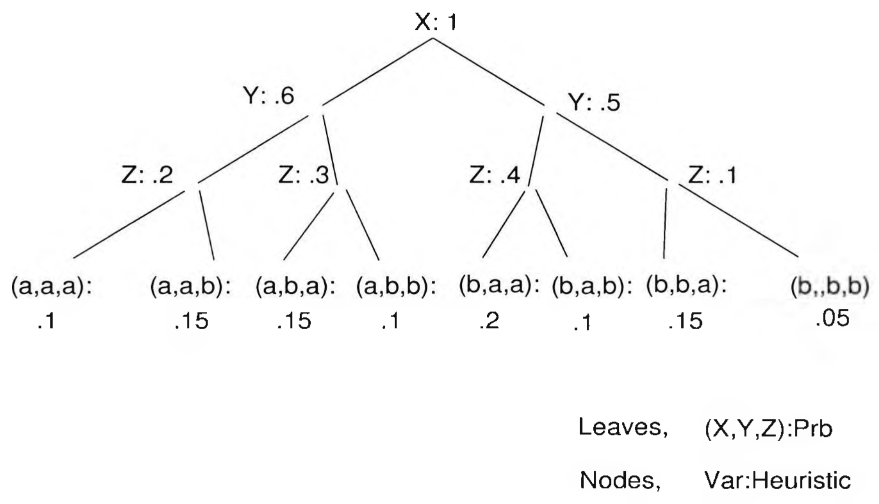


Figure 6.5: Tree illustrating probabilistic approximation.

A very simple tree, exemplifying the approximation process, is drawn in Fig. 6.5. There are three variables involved, each with two possible values. Non-leaf nodes are labelled by a variable (getting instantiated at that level) and a heuristic metric. Leaves are labelled by the list of values assigned to the variables and their corresponding probabilities. Note that the particular numbers are only used for the purposes of illustration and do not have any other meaning attached to them. The important notion is the difference between optimal and approximated

[3]partially instantiated branch: having a mix of instantiated and uninstantiated variables

order. These are juxtaposed in Table 6.2.

Optimal	Heauristic
$(b,a,a):.2$	$(b,a,a):.2$
$(a,a,b):.15$	$(b,a,b):.1$
$(a,b,a):.15$	$(a,b,a):.15$
$(b,b,a):.15$	$(a,b,b):.1$
$(a,a,a):.1$	$(a,a,a):.1$
$(a,b,b):.1$	$(a,a,b):.15$
$(b,a,b):.1$	$(b,b,a):.15$
$(b,b,b):.05$	$(b,b,b):.05$

Table 6.2: Optimal versus approximated order.

Children of a selected branch are found by instantiating one of the still free Variables to one of its possible Values (starting with the most probable one). The particular metric serving *max_unique_alt* is

$$Metric = ParentMetric + f(MarginalPrb - MaxAlternativePrb)$$

where *Metric* and *ParentMetric* are the obvious variables. *MarginalPrb* is the probability of the instantiated variable to assume the particular value, which differentiates this child from its siblings. *MaxAlternativePrb* is the max probability assigned to this value in any of the non selected variables. Function f ensures that the resulting number is an integer value within certain limits. These limits are evaluated when setting up the labelling and depend on the number of variables and the number of alternative values for each variable.

In the meta-interpreter, the search tree has been implemented as a binary tree, with each node being an autonomous representation of the branch from root to itself. When considering *Pfd* within alternative programming paradigms, it will be more appropriate to use arrays (indexed by each branch's Metric) and/or link lists. The experimental performance of the algorithm has been very encouraging and has shown that the main resource that is used extensively is memory. We do not consider this to be a characteristic of the algorithm itself, but rather the representation of the tree within Prolog. In an imperative language this can be implemented as an array (the higher the index the most promising the branches) or a linked list. Although arrays can be implemented in Prolog, the drawback in doing so, is that in order to have a handle to the structure, the structure itself has to be passed as argument. Thus collapsing the array manipulation to that of trees. Our experiments have shown that the array version is slightly slower than the tree-based one.

Our final note for this algorithm, is that we view it as an illustration of gaining performance within a sound semantics environment. The importance of the particular metric used, or the particular implementation are thus secondary.

6.3 Alternative Propagation Techniques

The methods for effecting probabilistic propagation we presented so far are not very elaborate in the way they try to achieve their aims. The reason for this is twofold. Firstly, because in this thesis we are primarily concerned with other aspects of *Pfd*. And secondly, because its likely that resarch from sister areas might be able to be adopted.

Probabilistic propagation in *Pfd* is akin to propagation in graphical models, which is a fruitful research area, and in particular to propagation in Bayesian Networks. More specifically, the task of calculating the probability of an event (as defined in 3.6) can be seen as analogous to that of calculating the posterior probability of a variable in a Bayesian Network. This Bayesian Network can be constructed on demand, from the *Pfd* constrains in the store. The main difference is that the conditional constraints cannot be mapped statically to Bayesian Network edges. This mapping can only happen as part of the *Pfd* propagation procedure.

Still, research in Bayesian Networks, have valuable results that might be beneficial for *Pfd*. These include: better propagation properties in Networks with particular characteristics, such as singly connected graphs (polytrees, [Pea88]) and approximating techniques for propagation in Bayesian Networks ([Nea90]) and graphical models, such as the iterative decoding algorithm ([WLK95]).

Chapter 7

Experiments

In this chapter we present some experimental results. The meta-interpreter of Chapter 6 is used to answer queries against examples taken from Chapter 5. We use the three curtains example to illustrate the logical derivations possible with *Pfd*, while the Caesar coding example is used to show some of the statistical inferences possible. The main motivation for this juxtaposition is to strengthen the argument that *Pfd* can deal with problems from both interpretations of probability (Section 2.1.4).

7.1 Logical

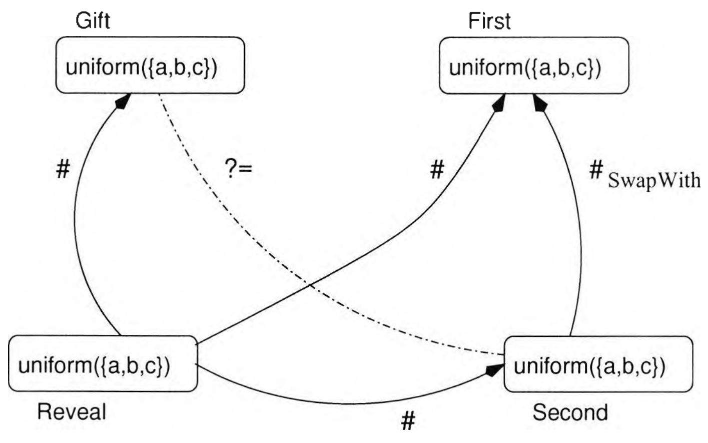
The machine readable program for the Curtains example (Section 5.1) is derived from the presented code by a straightforward application of the translations in Table 6.1. (For completeness all the example programs are included in Appendix B.)

In this section we will look at strategy γ in greater detail (its graphical representation and code are given in Fig. 7.1). We choose to concentrate on this strategy, since it is the most general one. Note that similar and somewhat simpler analyses hold for the other two strategies.

In Fig. 7.2, we show four queries and their results when executed within the meta-interpreter. In the first query we ask for all the *simple*[1] strategies and their probability of success. In the second one, we ask for the probability of success when players take a fifty-fifty approach in choosing their final curtain (strategy γ). The remaining two queries demonstrate that strategy γ (Fig. 7.1) at its extremities (zero and one) collapses into strategies α and β, respectively.

Figure 7.3 shows a tree-like structure which illustrates how the result of the second query

[1]these are strategies α and β



```
curtains( gamma, SwapWith, Pr ) :-
    Gift pin uniform([a,b,c]),
    First pin uniform([a,b,c]),
    Reveal pin uniform([a,b,c]),
    Second uniform([a,b,c]) /# First with SwapWith,
    Reveal /# Gift,
    Reveal /# First,
    Second /# Reveal,
    Pr is p(Second=Gift).
```

Figure 7.1: Graph and clause for γ Strategy.

```
?- curtains( Strategy, Probability ).
     Strategy = alpha
     Probability = 1/3;
     Strategy = beta
     Probability = 2/3;
no
?- curtains( gamma, 1/2, Probability ).
     Probability = 1/2;
no
?- curtains( gamma, 0/1, Probability ).
     Probability = 1/3;
no
?- curtains( gamma, 1/1, Probability ).
     Probability = 2/3;
no
```

Figure 7.2: Curtains queries.

is derived. The four leftmost labels at the top of the figure, correspond to the variables in the program. Aligned underneath each variable, are values the variable assume at particular stages of the computation. Edges are labelled with the probability with which variables assume the value at the rightmost end of the edge. (The tree should be read from left to right.) The two labels at the top right hand side of Figure 7.3, mark columns that hold product values. The first product column labelled Probabilities, holds the probability attached to each branch; while the second one, only records the probability of branches, for which the strategy was successful ($Second = Gift$). These two columns are also totalled vertically to provide the total probability of the traversal (equal to one) and the probability of success (one-half).

The importance of the presented tree is that it has a dual reading. In one reading it can be used to map the computation of the *Pfd* program. The second more common reading, is provided by probabilistic texts when attempting to illustrate the validity of the result. This illustration is often necessary since the result differs from the one people assume when asked to consider this example. (Commonly people claim that strategies α and β lead to identical likelihood of success.) This tree, in probabilistic circles, is known as the game tree. The traversal of the tree, within the meta-interpreter in this example, is triggered by the constraint $Pr\ is\ \mathbf{p}(Second{=}Gift)$.

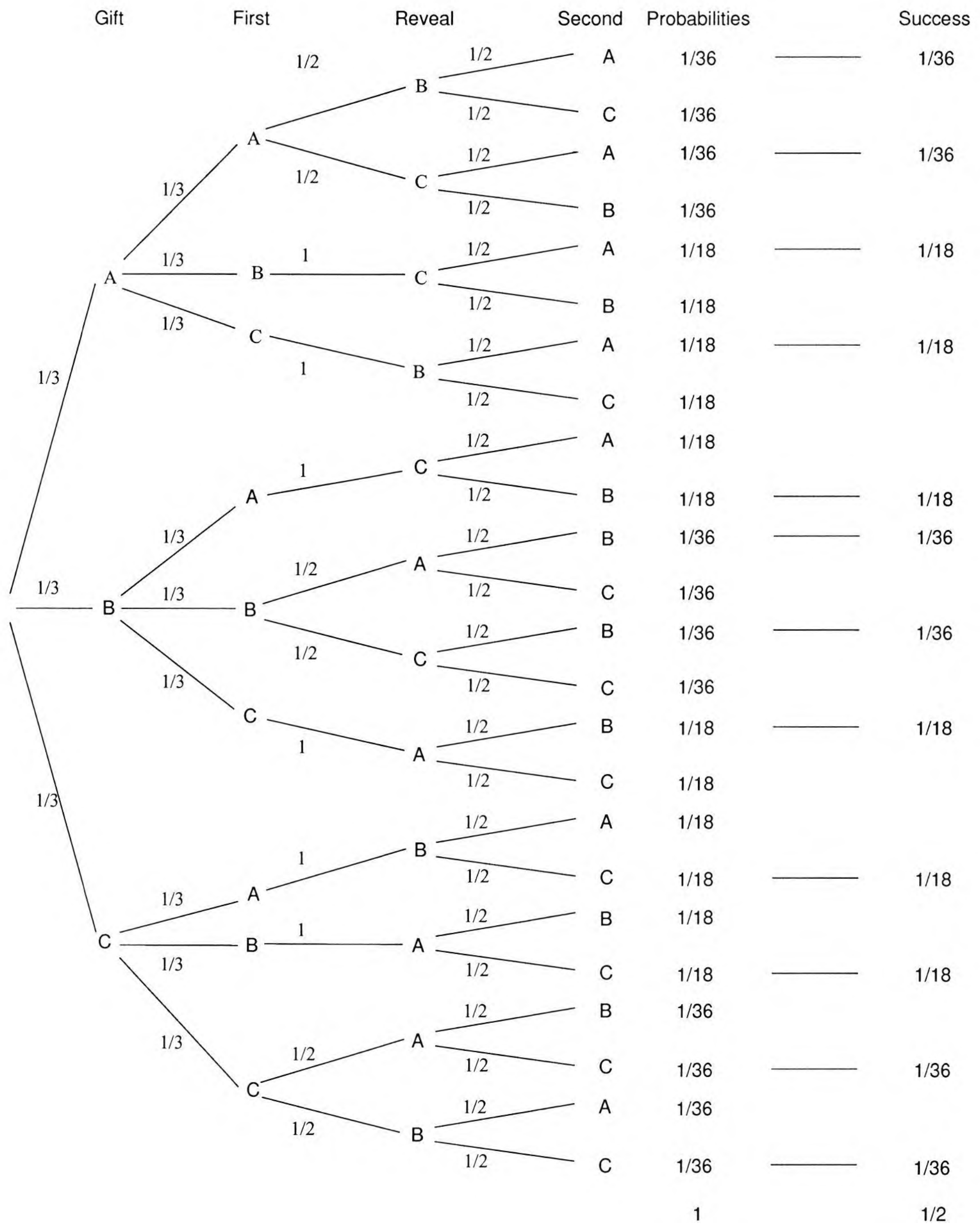


Figure 7.3: Probability Tree for strategy γ ; with $ChooseWith = 1/2$.

The final probability is computed in time proportional to

$$O(\prod_i V_i) = O(\mid First \mid * \mid Gift \mid * \mid Reveal \mid) \tag{7.1}$$

where $\mid V_i \mid$ is the cardinality[2] of variable V_i . In general, the variables which participate in 7.1 are those that appear in *Predicate*; where (Predicate) is the one for which we seek a probability of success, by a constraint of the form

$$Pr\ is\ \mathbf{p}(Predicate)$$

Also included to this calculation are all variables to which the predicate's variables depend upon. Thus in this example, the predicate variables are *Gift* and *Reveal* while *First* is included because of the *Reveal/#First* constraint. Substituting in 7.1 for the cardinalities of the three variables, we get :

$$O(3 * 3 * 3) = O(27) \tag{7.2}$$

The main two tasks executed twenty seven times are an arc consistency algorithm and a meta-call. Arc consistency is achieved in polynomial time and the meta-call incurs a constant time cost.

7.2 Statistical

The Caesar encodings program can be naturally split into two parts. In the first part, we set up the variables to model the problem. Here we will see how a domain specific probabilistic method is used. In the second part, we analyse the behaviour of the generic approximated algorithm, which provides the computational horsepower for solving the problem.

In this section we repeat some of the code presented in Section 5.3. This is done for ease of reference. Here we are interested in analysing the execution behaviour of the code, rather than its modelling properties (as it was the case in Section 5.3). Our starting point in this analysis is the *caesar/7* predicate (Fig.7.4). Its arguments read :

EncodedWs the encoded words (represented as a list of lists of character codes).

Dict identifies the dictionary from which *EncodedWs* were drawn and the dictionary words against which any decoding guesses, will be tested against.

DecodedWs are the decoded words. The form is that of EncodedWs but the place of each of its variables, is occupied by the corresponding decoded letter.

```
% caesar( +EncodedWs, +Dict, ?DecodedWs ) :-
caesar( EncodedWs, Dict, DecodedWs ) :-
    dictionary_info( Dict, DictWs, Freqs, AlphaBeto ),
    count_occurrances( EncodedWs, Codes, Counts, Sum ),
    proximity_vars( Counts, Total, Alphabet, Freqs, Vars, Codes ),
    word_codes_to_vars( EncodedWs, Codes, Vars, WordVars ),
    decode_words( WordVars, Codes, DictWs, Vars, DecodedWs ).

proximity_vars( [], _Total, _Dom, _Prbs, [], [] ).
proximity_vars( [Code-Occs|T], Total, Dom, Prbs, [Hv|Tv], [Code|Tc] ) :-
    Hv pin proximity(Dom,Prbs,Occs/Total),
    proximity_vars( T, Total, Dom, Prbs, Tv, Tc ).

probabilistic_method( proximity(Fd,Marker,Frqs,Prbs) ) :-
    rationals_subtract_list_from( Frqs, Marker, Subtr ),
    maplist( rationals_abs, Subtr, Abstr ),
    rationals_add_list( Abstr, Total ),
    rationals_subtract_list_from( Abstr, Total, Diffs ),
    rationals_to_probabilities( Diffs, Prbs ).
```

Figure 7.4: Top level predicate for *Pfd* solution to Caesar encodings problem.

The predicate, firstly gets all the necessary information pertaining dictionary *Dict* with *dictionary_info/4*, and then it derives similar information for *EncodedWs* with *count_occurances/4*, so it will be able to set up one probabilistic variable for every encoded letter (*proximity_vars/6*). The input letters are then replaced in the words, by the corresponding variables. Having set up all the necessary structures, the predicate calls *decode_words/5* which will try and solve the problem. In the code shown in Fig.7.4, we also give the definition of the probabilistic method, used in declaring the probabilistic variables in our Caesar program. The predicates within the proximity method, refer to the rationals library which are part of the meta-interpreter (Appendix A). The declarative reading for all the rational predicates used here are tabulated in Table 7.1.

The approach we have taken in decoding a given set of words, is to select one word at each stage, guess a combination of values for its free variables, and finally check whether the combination of values renders the word as a valid dictionary entry. The remaining words are then decoded. Some important points of our approach :

- Selection of words is of paramount importance to the overall efficiency.
- The decoding of one word instantiates a number of variables; this in turn, reduces the choices available within the remaining words. Thus, each decoding makes the guessing of consecutive words a much easier task. In effect the crux of the problem is to correctly guess the first selected word.
- Alternative guesses are selected upon failure.

rationals predicate	declarative reading
subtract_list_from($[R_1, R_2, \ldots]$, R_m, L_s)	$L_s = [R_m - R_1, R_m - R_2, \ldots]$
abs(R_1, R_2)	$R_2 =\mid R_1 \mid$
add_list($[R_1, R_2, \ldots]$, R_{sum})	$R_{sum} = \sum_i R_i$
to_probabilities($[R_1, R_2, \ldots]$, R_{sum}, $[R_{p1}, R_{p2}, \ldots]$)	$R_{pi} = R_i / R_{sum}$

Table 7.1: Rational arithmetics for proximity method.

The decoding of words is implemented in the predicate *decode_words/5*, of Figure 7.5. The selection of the encoded word to be decrypted at any stage, is achieved by a simple yet effective means. The word selected is the one with the least degrees of freedom. By degrees of freedom,

[2]number of elements in the variable's domain

we refer to the product of (unique) letters in a word, multiplied by the elements in each variable's finite domain[3]. This is implemented in a straightforward manner by *min_cardinality_word/5*, which employs the *Pfd* constraint *domain_cardinality/2* (Fig. 7.5).

```
% decode_words( +WordsVs, +Vars, +DictWords, +ChrCodes, -DecodedWs ).
decode_words( [], _Vars, _DictWs, _CharCodes, [] ).
decode_words( WordsVs, Vars, DictWs, CharCodes, DecodedWs ) :-
    min_cardinality_word( WordsVs, BestWordVs, UnqVsWord, RestWs ),
    label( UnqVsWord, max_unique_alt , Vals, _VlProbs, _Prob, _AccProb ),
    wordcodes_to_guess( UnqVsWord, WordVs, Vals, Guess ),
    word_in_dict_words( Guess, DictWs ),
    decode_words( RestWs, Vars, DictWs, CharCodes, DecodeWs ).

% min_cardinality_word( +WordsVs, -BestWord, -UnqVsWord, -RestWs ) :-
min_cardinality_word( [H|T], BestWord, UnqVsWord, RestWs ) :-
    remove_duplicates( H, NoDplH ),
    word_cardinality( NoDplH, 0, HCard ),
    min_cardinality_word( T, NoDplH, H, HCard, WordReps ),
    WordReps = UnqVsWord-Word,
    selects( Word, [H|T], Rest ).

% word_cardinality( +Vars, +AccCard, -FinCard ).
word_cardinality( [], Card, Card ).
word_cardinality( [H|T], AccCard, WCard ) :-
    domain_cardinality( H, HCard ),
    NxAcc is HCard + AccCard,
    word_cardinality( T, NxAcc, WCard ).
```

Figure 7.5: Decoding words predicates.

In the core of the above predicates, resides the *label/6* constraint. Its declarative reading is that a number of variables are instantiated to their most likely combination. The misfortune is that searching for the most likely combination requires a breadth-first search; since we are unable to order the combinations until we have seen all of them. This would deem our solution to the Caesar encodings impossible by todays computational powers.

[3]in the interest of uniformity we will consider instantiated variables as having an one element finite domain

In order to avoid this limitation, we use the generic[4] heuristic labelling method of Section 6.2.3, which trades accuracy of order for usage of computational resources. Note that, it is the order that is compromised, (i.e. the first set of values assigned by the procedure might not be the most likely one) and not the underlying semantics (i.e. incorrect probability attached to branches or to domain elements).

By using the approximate labelling we have achieved some encouraging results. Obviously these cannot compete with specialised imperative programs in solving the particular problem. Our main objective is to show that the declarative formalism of *Pfd*, even via a meta-interpretation, can address some non-trivial problems.

In the Caesar encoding problem, the hardest word to decode is the first one; very often by a great difference from the second. For the solution presented here, as well as most of the solutions in the literature, the ease by which the solution is found is determined by the closeness of the two letter frequencies (dictionary versus encoded). This in turn is inversely dependent on the number of words to decode. The more words to decode, the more likely it is that we have a better match between the frequencies, thus increasing the chances to find the solution within a certain number of steps.

In Figure 7.6 we give the comparative timings for a *Pfd* and a CLP(FD) solution [5] to a Caesar decoding experiment. The x-axis enumerates number of words chosen from the dictionary, while on the y-axis are the execution times in seconds. In these experiments each of the decoded word has at least four distinct letters and the guessed words are tested against the restricted set of the input words rather than the whole dictionary. The difference in absolute execution time in favour of *Pfd* is expected to be larger than what is suggested by the graph in Fig. 7.6, since CLP(FD) is implemented natively, whereas *Pfd* relies on an extra layer of meta-interpretation.

In the curve corresponding to the *Pfd* solution (explicated with error-bars equal to the population's standard deviation, in Fig. 7.7) the near constant execution time for the range of thirty to one hundred words indicates that the benefits of having better precision (at the one hundred end) is negated by the extra time taken to guess and search through more words. The same argument holds for the corresponding deviations. The most important cut-off point is at the fifty words mark. At this point the statistical information becomes very unreliable, although it still performing better than the uneducated-guess case. A main factor seems to be that the most commonly letters are being selected in better than random order. In Fig. 7.8 we give the complete execution graph for the CLP(FD) case (the predicates implementing our

[4]in the sense that it does not depend on the particular problem

[5]the CLP(FD) solution is identical to the *Pfd* program apart from variable definitions, these are given in Appendix B.4, and labelling, where we have used the standard left-to-right labelling provided in CLP(FD) systems

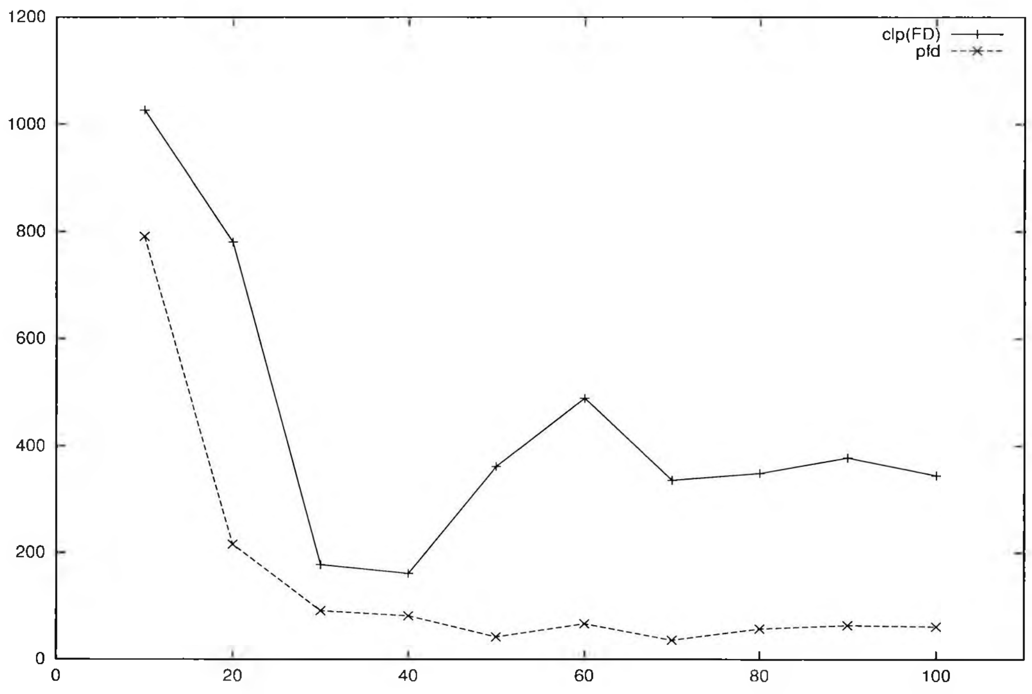


Figure 7.6: *Pfd* versus CLP(FD) timings comparison.

approach can be found in Appendix B.4).

One encouraging factor in the analysis of performance, for the *Pfd* program, is that approximately half of the execution time is consumed in garbage collection. The main data-structure that contributes to this is the *max_unique_alt* labelling procedure's tree structure. This will be substantially reduced in an OO or imperative implementation of *Pfd* or even in Prolog systems that implement *Pfd* labelling natively.

In concluding this section it is worth noting that by approximating the order of results we: (a) do not compromise the semantics of *Pfd* and (b) are able to deal with global dependencies, of probabilistic execution order, in a transparent manner.

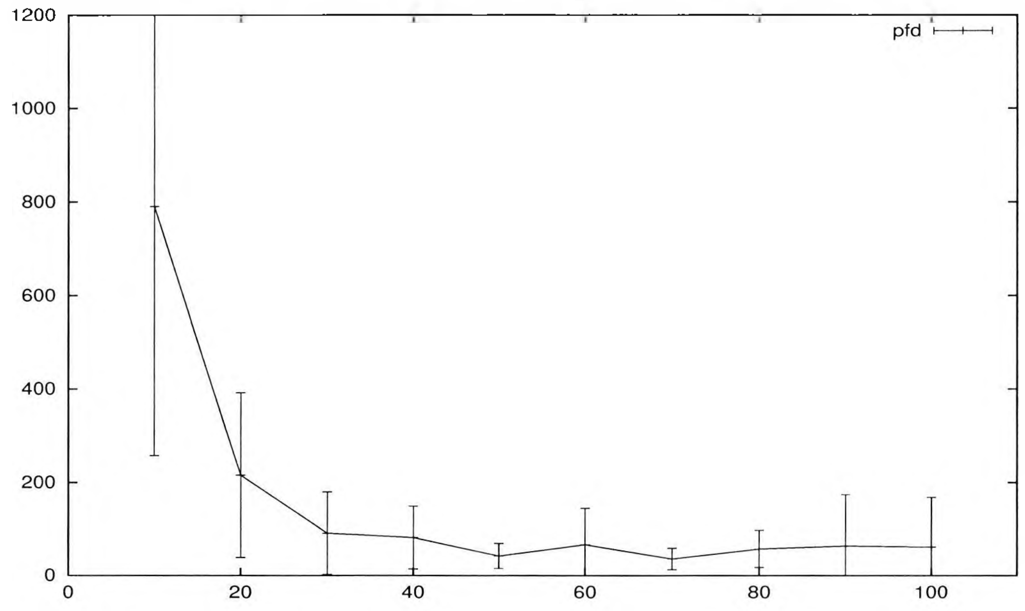


Figure 7.7: *Pfd* timings graph.

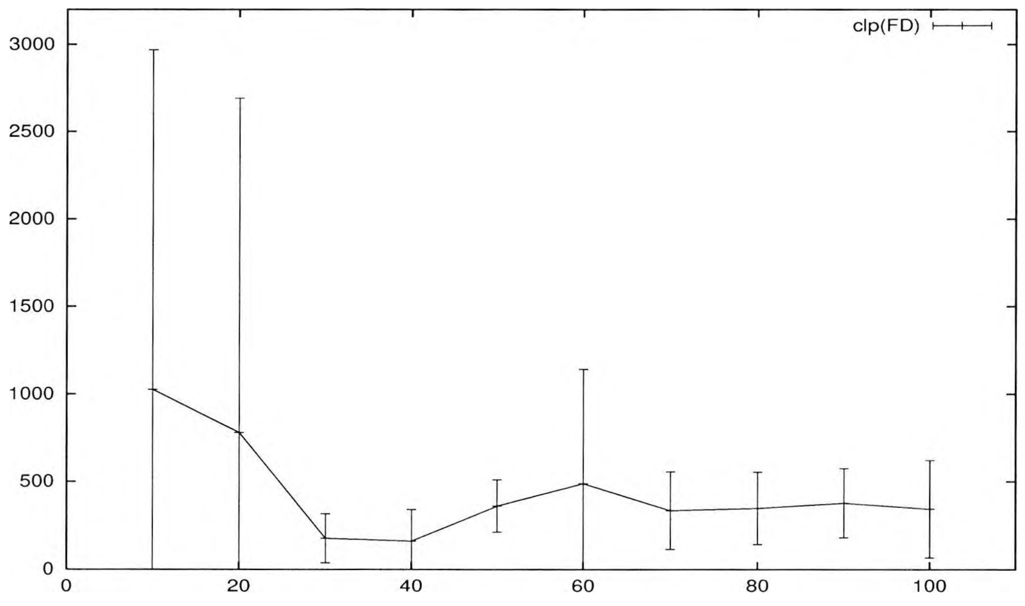


Figure 7.8: CLP(FD) timings graph.

Chapter 8

Conclusions

In this final chapter we present some concluding remarks. In particular, we comment on *Pfd* 's positive aspects, as well as on its limitations. Also we include some directions for future work.

8.1 General

In this thesis we have presented a novel set of constructs that allow programming with probabilistic concepts. Probabilistic Finite Domains are intended to be a practical means for general purpose programming with such concepts. Towards this we have employed and extended a number of different technologies.

In particular :

Logic Programming. Although *Pfd* can be annexed to a variety of programming language, we have shown how Logic Programming leads to a formalism with greater expressive power.

Finite Domains. As a convenient abstraction for the ascription of elementary probabilities to computational units.

Constraint (Logic) Programming. A clean means for attaching probabilistic finite domains to Logic Programming.

Our approach sheds some new light into the ideas of constraint solving, in that the techniques and algorithms presented here, are not so much concerned with constraint solving per se. Instead, we progress to the idea of constraints as a special knowledge base, which can assist the process of problem solving. On the other hand, we are able to take advantage of Finite Domains constraint solving, via the separation of a variable's domain and its probabilistic behaviour (captured via method functions). Through this separation, probabilistic behaviour is captivated at a level which is intuitive to a human reasoner.

Based on the elementary probabilities, we provide means of calculating probabilities of composite events; in accordance with the axioms of Probability Theory. As a result, we have shown that *Pfd* is capable of dealing with probabilistic problems, irrespective of the interpretation (belief versus frequencies) imposed when modelling particular problems.

8.2 Positive Points

In the light of *Pfd* as an extension to Logic Programming, for general problem solving, we are able to claim an array of positive points. These include:

- Probabilistic constructs are compositional. This is an important property for any approach that seeks to improve the programming capabilities of probabilistic languages.
- Ability to take advantage of constraint propagation in Finite Domains.
- Integration of transparent probabilistic function that capture human intuitions at a local level.
- Provision of conditional constraints which differentiate probabilistic causation from the purely logical one, while it allows accountability of resource usage.
- Use of a generic labelling algorithm for semantically consistent approximation.
- Prototype implementation, which has promoted experimentation and initial evaluation.

8.3 Limitations

On the other hand, we have identified a number of points, which, either limit the applicability of *Pfd*or need further investigation. These include:

- The calculation of composite event probabilities, involves computations that are time proportional to the factorial of number of variables involved.
- Evaluation of *Pfd* programs' behaviour for problems which use composite event probabilities for driving computational decisions.
- The current arsenal of probabilistic methods is limited to the ones needed in our evaluating examples.

- The restriction against mutual dependencies, in conditional constraints, might be overly restraining.

8.4 Future Work

In addressing some of the identified limitations and in further evaluation of the possible benefits of *Pfd*, we propose a number of key issues on which future work should be directed. We group these in three categories: technical improvements, further evaluation, and extensions.

Technical Improvements.

- Investigate algorithms and the different semantics of allowing cyclic dependencies. In our current work, we chose to concentrate in disallowing all such dependencies, since there is no universal way of interpreting the probabilistic meaning of cyclic dependencies. Further research may provide, either special cases where this is possible, or a new generic way for calculating such probabilities.
- Construct more probabilistic methods, to make the system more readily useful to a wider audience and suits of problems. These methods might be both generic ones such as *normal* or embrace particular areas such as *card_deck*.

Further Evaluation.

- Continue to apply *Pfd* to different problems. These should be drawn both from belief based and from frequent based interpretations of probability. In particular, seek problems that take advantage of probabilities for directing the computation.
- Use experience from the application to different problem areas, to realise approximations which are pertinent to particular kinds of probabilistic settings. An example of such an approximation is the labelling method presented in this thesis.

Extension.

- A far more challenging question is how to enhance *Pfd* with analytical, intentional, formulas. For example γ strategy in the curtains example, has a simple analytical formulation,
$$p(W) = \frac{1+W}{3} \ .$$
The reason that this was not included in our analysis is because such formulations normally require expert analysis and do not give compositional handles. On the

other hand, in most cases they provide enormous computational savings. Thus, it would be worthwhile to investigate whether they can be integrated in *Pfd* along with the extensional constraints.

8.5 Epilogue

Probabilistic concepts, along with randomised and other non-deterministic ones, have been attracting a lot of attention in recent years. This is as a result of the maturity of deterministic approaches, for which we have started to get a better understanding of their capabilities and limitations. As a direct consequence of this, researchers are attempting to refine the abstractions available by turning to more general concepts. In this light, we see Probability Theory as a definite candidate abstraction.

In this work, we concentrated in programming languages research. Programming languages are one of the most valuable assets to humanity. Their contribution to society in their short, yet so influential, life is manifested in our every day lives. It is thus important to induce their amalgamation with the more general abstractions of Probability. When this is done in a successful manner the benefits to and quality of programs will increase dramatically.

The formalism we have proposed here (*Pfd*) raises some interesting question in these areas. We also put forward some innovative answers. We hope that these help clarify some of the issues involved and promotes discussion at a higher level.

We view programmers as creators of abstractions and hope that concepts such as those introduced by our work, assist them in producing more effective abstractions.

Appendix A

The rationals predicates

```
% rationals_addition( +R1, +R2, -R3 ) :- R3 = R1 + R2.
rationals_addition( LNom/LDnm, RNom/RDnm, Simplified ) :-
   rationals_lcd_factors( LDnm, RDnm, ResDnm, LFctr, RFctr ),
   ResNom is LNom * LFctr + RNom * RFctr,
   rationals_simplify( ResNom/ResDnm, Simplified ).

% rationals_subtraction( +R1, +R2, -R3 ) :- R3 = R1 - R2.
rationals_subtraction( LNom/LDnm, RNom/RDnm,  ResNom / ResDnm ) :-
   rationals_lcd_factors( LDnm, RDnm, ResDnm, LFctr, RFctr ),
   ResNom is LNom * LFctr - RNom * RFctr.

% rationals_multiplication( +R1, +R2, -R3 ) :- R3 = R1 * R2.
rationals_multiplication( Nom1/Dnm1, Nom2/Dnm2, Simplified ) :-
   ResNom is Nom1 * Nom2,
   ResDnm is Dnm1 * Dnm2,
   rationals_simplify( ResNom/ResDnm, Simplified ).

% rationals_division( +R1, +R2, -R3 ) :- R3 = R1 / R2.
rationals_division( NomNom/NomDnm, DnmNom/DnmDnm, Simplified ) :-
   ResNom is NomNom * DnmDnm,
   ResDnm is NomDnm * DnmNom,
   rationals_simplify( ResNom/ResDnm, Simplified ).
```

```
% rationals_simplify( +R, -SimpleR ).
rationals_simplify( Nom/Dnm, SNom/SDnm ) :-
   gcd( Nom, Dnm, Gcd ),
   SNom is Nom // Gcd,
   SDnm is Dnm // Gcd.

% rationals_lcd_factors( +Nat1, +Nat1, -LCD, -Fact1, -Fact2 ).
rationals_lcd_factors( NatOne, NatTwo, LCD, OneFact, TwoFact ) :-
   gcd( NatOne, NatTwo, GCD ),
   OneFact is NatTwo // GCD,
   TwoFact is NatOne // GCD,
   LCD is OneFact * TwoFact * GCD.

% gcd( +Int1, +Int2, -Gcd ).
gcd( M, N, C ) :-
   ( N =:= 0 -> C is M
        ;  NewN is M mod N,
           gcd( N, NewN, C )
   ).

% rationals_abs( +R, -AbsR  ).
rationals_abs( Nom/Dnm, AbsNom/Dnm  ) :-
   ( Nom =< 0 ->
      AbsNom is abs(Nom)
      ;
      AbsNom is Nom
   ).

% rationals_to_probabilities( +Rats, +Sum, -Probs ) :-
rationals_to_probabilities( Rats, Sum, Probs ) :-
   rationals_invert( Sum, MUs ),
   rationals_all_multiply_list( Rats, MUs, Probs ).

% subtract_list_from( +Rats, +Omni, -SubRats ).
```

```
subtract_list_from( [], _Omni, [] ).
subtract_list_from( [H|T], Omni, [DiffH|DiffT] ) :-
   DiffH is Omni - H,
   subtract_list_from( T, Omni, DiffT ).

% rationals_add_list( +Rats, -Sum ) :-
rationals_add_list( [H|T], Res ) :-
   rationals_add_list( T, H, Res ).

% rationals_add_list( +Rats, +Acc, -Sum ).
rationals_add_list( [], Ans, Ans ).
rationals_add_list( [H|T], Acc, Res ) :-
   rationals_addition( H, Acc, NewAcc ),
   rationals_add_list( T, NewAcc, Res ).

% rationals_all_multiply_list( +Rats, +Factor, -MultiRats ).
rationals_all_multiply_list( [], _R, [] ).
rationals_all_multiply_list( [H|T], Multi, [MTH|MsTT] ) :-
   rationals_multiplication( H, Multi, MTH ),
   rationals_all_multiply_list( T, Multi, MsTT ).

% rationals_invert( +R, -OneOverR ).
rationals_invert( Nom/Dnm, NNm/NDm ) :-
   ( Nom > 0 ->
      NNm = Dnm,
      NDm = Nom
      ;
      ( Nom = 0 ->
         write( error(division_by_zero,rationals_invert/2) ),
         nl, abort
         ;
         NNm = - Dnm,
         NDm = Nom
      ) ).
```

Appendix B

The examples predicates.

B.1 Curtains

```
% curtains( ?Strategy, -Prob ) :-
curtains( alpha, Pr ) :-
   Gift pin uniform([1,2,3]),
   First pin uniform([1,2,3]),
   Reveal pin uniform([1,2,3]),
   Reveal /# Gift,
   Reveal /# First,
   Second /# Reveal,
   Second = First,
   Pr is p(Second=Gift).

curtains( beta, Pr ) :-
   Gift pin uniform([1,2,3]),
   First pin uniform([1,2,3]),
   Reveal pin uniform([1,2,3]),
   Reveal /# Gift,
   Reveal /# First,
   Second /# First,
   Second /# Reveal,
   Pr is p(Second=Gift).
```

```
% curtains( +Strategy, +ChooseOtherWith, -Prob ) :-
curtains( gamma, ChooseOtherWith, Pr ) :-
   Gift pin uniform([1,2,3]),
   First pin uniform([1,2,3]),
   Reveal pin uniform([1,2,3]),
   Reveal /# Gift,
   Reveal /# First,
   Second pin uniform([1,2,3]) /# First with ChooseOtherWith,
   Second /# Reveal,
   Pr is p(Second=Gift).
```

B.2 Enumeration

```
% ordered_with_replacement( +Robjects, +FromFd, -CodeProb ) :-
ordered_with_replacement( R, Fd, Pr ) :-
   n_pfd_vars( R, uniform(Fd), Code ),
   n_random_selections( R, Fd, Assign ),
   Pr is p( Code=Assign ).

% ordered_without_replacement( +Robjects, +FromFd, -CodeProb ) :-
ordered_without_replacement( R, Fd, Pr ) :-
   n_pfd_vars( R, uniform(Fd), Code ),
   distinct( Code ),
   n_random_selections( R, Fd, Assign ),
   Pr is p( Code=Assign ).

% n_pfd_vars_1( +N, +Pfd, -TheNPfdVars ).
n_pfd_vars_1( 0, _Pfd, [] ).
n_pfd_vars_1( N, Pfd, [Var|Vars] ) :-
   N > 0,
   Var pin Pfd,
   NxN is N - 1,
   n_pfd_vars_1( NxN, Pfd, Vars ).
```

```
% n_random_selections( +N, +Set, -Selected ) :-
n_random_selections( N, Fd, Assign ) :-
   length( Fd, Limit ),
   n_random_selections( N, Limit, Fd, Assign ).

% n_random_selections( +N, +Limit, +Set, -Selected  ).
n_random_selections( 0, _Limit, _List, [] ).
n_random_selections( N, Limit, List, [H|T] ) :-
   N > 0,
   random( 1, Limit, Rnd ),
   nth1( Rnd, List, H ),
   NxN is N - 1,
   n_random_selections( NxN, Limit, List, T ).

% distinct( +PfdVars ).
distinct( [] ).
distinct( [H|T] ) :-
   distinct( T, H ),
   distinct( T ).

% distinct( +PfdVars, +PfdVar ).
distinct( [], _Inv ).
distinct( [H|T], Inv ) :-
   H /# Inv,
   distinct( T, Inv ).
```

B.3 Caesar Encondings

```
% caesar( +EncodedWs, +Dict, -DecodedWs ).
caesar( EncodedWs, Dict, DecodedWs ) :-
   dictionary_info( Dict, DictWs, Freqs, AlphaBeto ),
   count_occurrances( EncodedWs, Codes, Counts, Sum ),
   proximity_vars( Counts, Sum, AB, Freqs, Vars, Codes ),
   word_codes_to_vars( EncodedWs, Codes, Vars, WordVars ),
   decode_words( WordVars, Codes, DictWs, Vars, DecodedWs ).

%  decode_words( +WordsVs, +Vars, +DictWords,  +ChrCodes, -DecodedWs ).
decode_words( [], _Vars, _DictWs, _CharCodes, [] ).
decode_words( WordsVs, Vars, DictWs, CharCodes, DecodedWs ) :-
   min_cardinality_word( WordsVs, BestWordVs, UnqVsWord, RestWs ),
   label( UnqVsWord, unique_heuristic, Val, _VlProbs, _Prob, _AccProb ),
   wordcodes_to_guess( UnqVsWord, WordVs, Vals, Guess ),
   word_in_dict_words( Guess, DictWs ),
   decode_words( RestWs, Vars, DictWs, CharCodes, DecodeWs ).

% proximity_vars( +Counts, +Sum, +Dom, +Prbs, -Pvars, -Codes ).
proximity_vars( [], _Sum, _Dom, _Prbs, [], [] ).
proximity_vars( [C-Os|T], Sum, Dom, Prbs, [Hv|Tv], [C|Tc] ) :-
   Hv pin proximity(Dom,Prbs,Os/Sum),
   proximity_vars( T, Sum, Dom, Prbs, Tv, Tc ).

% word_codes_to_vars( +Words, +Codes, +PVars, +Codes ).
word_codes_to_vars( [], _Codes, _Vars, [] ) :-
   !.
word_codes_to_vars( [H|T], Codes, Vars, [HCode|TCode] ) :-
   !,
   word_codes_to_vars( H, Codes, Vars, HCode ),
   word_codes_to_vars( T, Codes, Vars, TCode ).
word_codes_to_vars( Code, Codes, Vars, Var ) :-
   nth1( Nth1, Codes, Code ),
   nth1( Nth1, Vars, Var ).

% min_cardinality_word( +Words, -Word, -UnqVsWord, -RestWords ).
min_cardinality_word( [H|T], Word, UnqVsWord, Rest ) :-
   remove_duplicates( H, NoDplH ),
   word_cardinality( NoDplH, 0, HCard ),
   min_cardinality_word( T, NoDplH, H, HCard, UnqVsWord, Word ),
   select_once( Word, [H|T], Rest ).
```

```
% min_cardinality_word( +Words, +NoDpWrd, +Word, +Min, -NoDpWrd, -Word ).
min_cardinality_word( [], NoDpWrd, Word, _CurrMin, NoDpWrd-Word ).
min_cardinality_word( [H|T], CrNoDpWrd, CurrW, CurrMin, WordReps  ) :-
   remove_duplicates( H, NoDplH ),
   word_cardinality( NoDplH, 0, HCard ),
   ( HCard =< CurrMin ->
      NxMin is HCard,
      NxNdpWrd = NoDplH,
      NxW = H
      ;
      NxMin is CurrMin,
      NxNdpWrd = CrNoDpWrd,
      NxW = CurrW
   ),
   min_cardinality_word( T, NxNdpWrd, NxW, NxMin, WordReps ).

% word_cardinality( +Word, +AccCard, -Card ).
word_cardinality( [], Card, Card ).
word_cardinality( [H|T], AccCard, WCard ) :-
   domain_cardinality( H, HCard ),
   NxAcc is HCard + AccCard,
   word_cardinality( T, NxAcc, WCard ).

% word_vars_to_codes( +Word, +Vars, +Codes, -Codes ).
word_vars_to_codes( [], _Vars, _Codes, [] ).
word_vars_to_codes( [HV|TVs], Vars, Codes, [HC|TCs] ) :-
   nth1_vars( Vars, HV, 1, Nth1 ),
   nth1( Nth1, Codes, HC ),
   word_vars_to_codes( TVs, Vars, Codes, TCs ).

% nth1_vars( Vars, Var, Acc, Nth1 ).
nth1_vars( [H|T], V, Acc, Nth1 ) :-
   ( H==V ->
      Nth1 is Acc
      ;
```

```
        NxAcc is Acc + 1,
        nth1_vars( T, V, NxAcc, Nth1 )
    ).

% wordcodes_to_guess( +WordUnqVs, +WordVls, +UnqVals, -WordVls ).
wordcodes_to_guess( [], WordVls, _Vals, WordVls ).
wordcodes_to_guess( [Hvr|Tvrs], WordVrs, [Hvl|Tvls],  WordVls ) :-
    substitute( Hvr, WordVrs, Hvl, NxWordVrs ),
    wordcodes_to_guess( Tvrs, NxWordVrs, Tvls, WordVls ).
```

B.4 Caesar clp(FD) predicates.

```
% Needs SICStus 3.7
:- ensure_loaded( library(clpfd) ).

% Adopted from pfd solution to deal with clp(FD) solution
% of the caesar decoding.
caesar( EncodedWs, Counts, _Sum, AlphaBeto, Freqs, Vars, DecodedWs ) :-
    Alphabeto = [First|Rest],
    last( Rest, Last ),
    proximity_vars( Counts, First, Last, Freqs, Vars, Codes ),
    all_distinct( Vars ),
    word_codes_to_vars( EncodedWs, Codes, Vars, WordVars ),
    decode_words( WordVars, Codes, Vars, DecodedWs ).

proximity_vars( [], _First, _Last, _Prbs, [], [] ).
proximity_vars( [Code-_Occs|T], First, Last, _Prbs, [Hv|Tv], [Code|Tc] ) :-
    Hv in First..Last,
    proximity_vars( T, First, Last, Prbs, Tv, Tc ).

word_cardinality( [], Card, Card ).
word_cardinality( [H|T], AccCard, WCard ) :-
    fd_size( H, HCard ),
    NxAcc is HCard + AccCard,
   word_cardinality( T, NxAcc, WCard ).

label_n( Vars, Vals ) :-
   labeling( [leftmost], Vars ),
   Vals = Vars.
```

Appendix C

Constraint manipulating predicates.

```
% constraint_to_store( +Constraint, +InStore, -OutStore ) :-
% Case 1: Label N variables.
constraint_to_store( label_n(Method,Vars,Vals,Prob,AccPr), In, Out ) :-
   store_enquire( In, [act(Active)] ),
   active_selects_del( Vars, Active, VarToDataList, RemAct ),
   data_choices_for_label_n( VarToDataList, Meths, Fds, Nds ),
   method_pfd_constract_act_many( Meths, Fds, OrderOp, RemAct, Pairs ),
   retractall( (probability_sum(Vars,_AnyOldAccPr)) ),
   assert( probability_sum(Vars,0/1) ),
   probe_n( Method, Pairs, Vals, _ValPrbs, Prob ),
   data_update_dmns( Vals, VarToDataList, NewVarToData ),
   active_additions( NewVarToData, RemAct, NewAct ),
   dependent_values_satisfies_all( Nds, Vals, NewAct ),
   probability_sum( Vars, OldAccPr ),
   retractall( (probability_sum(Var,_VOA)) ),
   rationals_addition( OldAccPr, Prob, CurrPrb ),
   assert( probability_sum(Var,CurrPrb) ),
   single_update_store( act(NewAct), In, Out ).
```

```
% Case 2: Label a single variable.
constraint_to_store( label(Var,LMeth,Val,Prob,AccPr), In, Out ) :-
   store_enquire( In, [act(Active)] ),
   active_selects_del( [Var], Active, [Var-VarData], RemAct ),
   retractall( (probability_sum(Var,_AnyOldAccPr)) ),
   assert( probability_sum(Var,0/1) ),
   data_choices( [mtd(Method),dmn(Fd),nds(Nd)], VarData ),
   method_pfd_constract_act( Method,  Fd, RemAct, CnDom, CnProbs ),
   probe( LMeth, CnDom, CnProbs, Val, Prob ),
   dependent_value_satisfies_all( Nd, Val, Active ),
   probability_sum( Var, OldAccPr ),
   retractall( (probability_sum(Var,_VOA)) ),
   rationals_addition( OldAccPr, Prob, AccPr ),
   assert( probability_sum(Var,AccPr) ),
   data_update( dmn(Val), VarData, NewData ),
   active_additions( [Var-NewData], RemAct, NewAct ),
   single_update_store( act(NewAct), In, Out ).

% Case 3: Probababilistic in operator.
constraint_to_store( prin(Var,Method), In, Out ) :-
   store_enquire( In, [cnt(Count),act(Active),rts(Roots),gr(Graph)] ),
   must_be( var(Var) ),
   number_chars( Count, NumbCs ),
   append( NumbCs, [0'_], EscCs ),
   atom_chars( Var, EscCs ),
   NxtCount is Count + 1,
   ( MethodStr = (Method,RefVar,Cnstr,Prob) ->
      rationals_is_rat( Prob, RatPrb ),
      OutMethodStr = (NormMethod,RefVar,Cnstr,RatPrb),
      graph_add_constraint( RefVar, Cnstr, Var, Graph, NewGr ),
      NewRts = Roots,
      Nds = [RefVar-Cnstr],
      active_selects_del( [RefVar], Active, [RefVar-RefData], MidAct ),
      data_swap( rqr(ReqBy), RefData, NewReqBy, NewVDt ),
      ord_add_element( ReqBy, Var, NewReqBy ),
        active_additions( [RefVar-NewVDt], MidAct, SecAct )
        ;
        Method = MethodStr,
        OutMethodStr = NormMethod,
        ord_add_element( Roots, Var, NewRts ),
        NewGr = Graph,
        Nds = [],
        SecAct = Active
     ),
     method_normalise( Method, NormMethod, Fd ),
     ord_add_element( SecAct, Var-[OutMethodStr,Fd,Nds,[]], NewAct ),
     store_update( In,
        [rts(NewRts),act(NewAct),cnt(NxtCount),gr(NewGr)],
              Out ).

% Case 4: Conditional.
constraint_to_store( conditional(DepVarCn,RefVarCn), In, Out ) :-
     bi_unconditional_constraint( DepVarCn, DepWch, DepVar, DepVal ),
     bi_unconditional_constraint( RefVarCn, RefWch, RefVar, RefVal ),
     Reqs = (RefWch-RefVal,DepWch-DepVal),
     constraint_to_store_conditional( DepVar, RefVar, Reqs, In, Out ).

% Case 5: Conditionaly-different constraint.
constraint_to_store( cond_diff(DepVar,Var), In, Out ) :-
     constraint_to_store_conditional( DepVar, Var, cond_diff, In, Out ).

% Case 6: Probababilistic in operator.
constraint_to_store( prob(Constr,Prob), Store, Store ) :-
     pfd_predicate( Constr, PrlgCnstr, PfdVs, PrlgVs ),
     variables_to_roots_active( PfdVs, Store, Roots, Active ),
     findall( BranchPr,
         ( probe_variables( Roots, Active, [], Probed, 1/1, BranchPr ),
           probed_predicate_is_in_event( PrlgCnstr, PfdVs, PrlgVs, Probed )
         ),
           Probabilities
```

```
      ),
   rationals_add_list( Probabilities, Prob ).

% Case 7: Domain cardinality of a variable.
constraint_to_store( domain_cardinality(Var,Card), Store, Store ) :-
   must_be( pfd_var(Var) ),
   single_enquiry_for_store( act(Active), Store ),
   active_selects( [Var], Active, [Var-VarData] ),
   data_choose( dmn(Fd), VarData ),
   length( Fd, Card ).

% variables_to_roots_active( +Vars, +Store, -ResRoots, -ResActive ) :-
variables_to_roots_active( Vars, Store, ResRoots, ResActive ) :-
   store_enquire( Store, [act(Active),rts(Roots),gr(Graph)] ),
   graph_variables_reach( Vars, Graph, [], Reach ),
   active_split_roots( Active, Roots, Reach, ResRoots, ResActive ).

% constraint_to_store_conditional( +DepVar, +Var, +Reqs, +In, -Out ) :-
constraint_to_store_conditional( DepVar, Var, Reqs, In, Out ) :-
   store_enquire( In, [act(Act),gr(Graph),rts(Roots)] ),
   active_selects_del( [Var,DepVar], Act, [Var-VDt,DepVar-DpVDt], MidAct ),
   data_add_needs( Reqs, Var, DpVDt, NewDpVDt ),
   data_add_required( DepVar, diff, VDt, Roots, NewVDt, NewRoots ),
   active_additions( [Var-NewVDt,DepVar-NewDpVDt], MidAct, NewAct ),
   graph_add_constraint( Var, diff, DepVar, Graph, NewGraph ),
   store_update( In, [act(NewAct),gr(NewGraph),rts(NewRoots)], Out ).
```

▯

Bibliography

[Abr96] Samson Abramsky. Semantics of interaction. In *Proc. of 21st Int. Coll. on Trees in Algebra and Programming – CAAP'96, Linkoping*, volume 1059 of *Lecture Notes in Computer Science*, page 1. Springer-Verlag, 1996.

[AH94] M. Abadi and J. Y. Halpern. Decidability and expressiveness for first-order logics of probability. *Information and Computation*, 112(1):1–36, 1994. A preliminary version appears in Proceedings of the 30th Annual Conference on Foundations of Computer Science, 1989, pp. 148-153.

[Ale88] Romas Aleliunas. A new normative theory of probabilistic logic. In *Proceedings of the Canadian AI Conference*, pages 67–74. Morgan Kaufman, 1988.

[Bac90a] Fahiem Bacchus. Lp, a logic for representing and reasoning with statistical knowledge. *Computational Intelligence*, 6:209–231, 1990.

[Bac90b] Fahiem Bacchus. *Representing and Reasoning With Probabilistic Knowledge. A Logical Approach to Probability.* Artificila Intelligence. MIT Press, 1990.

[Bac96] Rolf Backofen. Controlling functional uncertainty. In Wolfgang Wahlster, editor, *Proceedings of* 12^{th} *European Conference on Artificial Intelligence*, pages 557–561. John Wiley & Sons, Ltd, 1996.

[Bal87] J. F. Baldwin. Evidential support logic programming. *Journal of Fuzzy Sets and Systems*, 24:1–26, 1987.

[Bay63] T. R. Bayes. An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society*, 53:370–418, 1763.

[BFGK96] Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. Context-specific independence in bayesian networks. In *Proceedings of the* 12^{th} *Annual*

Conference on Uncertainty in AI (UAI), pages 115–123, Portland, Oregon, August 1996.

[BGHK93] Fahiem Bacchus, Adam J. Grove, Joseph Y. Halpern, and Daphne Koller. Statistical foundations for default reasoning. In *Proceedings of the* 13^{th} *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 563–569, Chambery, France, August 1993.

[BK97] Christel Baier and Marta Kwiatkowska. Domain equations for probabilistic processes. *Electronic Notes in Theoretical Computer Science*, 7:20, 1997.

[Bor70] L. Borkowski, editor. *Logical Foundations of Probability Theory. Jan Lukasiewicz.* North-Holland, 1970.

[Bre94] Richard P. Brent. Uses of randomness in computation. Technical Report TR-CS-94-06, Australian National University, June 1994.

[Car62] Rudolf Carnap. *Logical Foundations of Probability.* The University of Chicago Press, 1962.

[CD96] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 27, 1996.

[CFS93] Philippe Codognet, Francois Fages, and Thierry Sola. A metalevel compiler of clp(fd), and its combination with intelligent backtracking. In Alain Colmerauer and Frédéric Benhamou, editors, *Constraint Logic Programming: Selected Research*, chapter 23, pages 437–456. The MIT Press, Cambridge, Mass., 1993.

[CG98] Veerle M. H. Coupe and Linda C. van der Gaag. Practicable sensitivity analysis of bayesian belief networks. Technical Report UU-CS-1998-10, University of Utrecht, 1998.

[CMG97] J. Laurie Snell Charles M. Grinstead. *Introduction to Probability.* American Mathematical Society, 1997.

[Coo90] Gregory F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42(2-3):393–405, 1990.

[Daw] A. P. Dawid. Conditional independence for statistics and AI. Lecture Notes, Cambringe, 1997.

[Daw79] A. P. Dawid. Conditional independence in statistical theory (with discussion). *J of R Stat. Soc. B*, 41:1–31, 1979.

[DBH$^+$99] B. Demoen, M. Garcia de la Banda, W. Harvey, K. Marriott, and P. Stuckey. An overview of HAL. In *In Proceedings of Principles and Practice of Constraint Programming*, pages 174–188, October 1999.

[DC93] Daniel Diaz and Philippe Codognet. A minimal extension of the wam for clp(fd). In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, June 1993. MIT Press.

[dF31] Bruno de Finetti. Sul significato soggettivo della probabilita. *Fundamenta Mathematicae*, 17:298–329, 1931.

[DGH81] Richard O. Duda, John Gascnig, and Peter Hart. Model design in the prospector consultant system for mineral exploration. In Bonnie Lynn Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 334–348. tioga, 1981.

[dH98] J.I. den Hartog. Comparative semantics for a process language with probabilistic choice and non-determinisim. Technical Report IR-445, Vrije Universiteit Amsterdam, http://www.cs.vu.nl/ tcs, February 1998.

[DHN81] Richard O. Duda, Peter E. Hart, and Nils J. Nilsson. Subjective bayesian methods fo rule-based inference systems. In Bonnie Lynn Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 192–199. Tioga, 1981.

[DL93] Paul Dagum and Michael Luby. Approximating probabilistic inference in bayesian belief networks is np-hard. *Artificial Intelligence*, 60(1):141–153, 1993.

[DP80] Didler Dubois and Henri Prade. *Fuzzy Sets and Systems. Theory and Applications*, volume 144 of *Mahematics in Science and Engineering.* Academic Press, 1980.

[DP97] Adnan Darwiche and Judea Pearl. On the logic of iterated belief revision. *Artificial Intelligence*, 29(1-2):1–29, 1997.

[DS97] Alex Dekhtyar and V. S. Sabrahmanian. Hybrid probabilistic programs. In *Proceedings of the Fourteenth International Conference on Logic Programming*, 1997.

[DvdG95] Marek J. Druzdzel and Linda C. van der Gaag. Elicitation of probabilities for belieff networks: Combining qualitative and quaantitative information. In *Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pages 141–148, Montreal, Canada, August 1995.

[Elk93] Charles Elkan. The paradoxical success of fuzzy logic. In *National Conference on Artificial Intelligence (AAAI'93)*, July 1993.

[Eps77] Richard A. Epstein. *The Theory of Gambling and Statistical Logic.* Academic Press, 1977.

[Fab96a] Zdenek Fabian. Information and entropy of continuous random variables. Technical Report 694, Instute of Computer Science, Academy of Sciences of the Czech Republic, 1996.

[Fab96b] Zdenek Fabian. On the relation between gnostical and probability theories. Technical Report 671, Instute of Computer Science, Academy of Sciences of the Czech Republic, April 1996.

[Fel59] William Feller. *An Introduction to Probability Theory and Its Applications.* Mathematical Statistics. Willey, 2nd edition, 1959.

[FH94] Ronald Fagin and Joseph Y. Halpern. Reasoning about knowledge and probability. *Journal of the ACM*, 41(2):340–367, 1994. A preliminary version appears in Proceedings of the Second Conference on Theoretical Aspects of Reasoning About Knowledge, 1988, pp. 277-294.

[FHK96] Nir Friedman, Joseph Y. Halpern, and Daphne Koller. Context-specific independence in bayesian networks. In *Proceedings of the* 12^{th} *Annual Conference on Uncertainty in AI (UAI)*, pages 1305–1312, Portland, Oregon, August 1996.

[FHM90] R. Fagin, J.Y. Halpern, and N. Megiddo. A logic for reasoning about probabilities. *Information and Computation*, 87:78–128, 1990.

[Fit88] Melvin Fitting. Logic programming on a topological bilattice. *Fundamenta Informaticae*, 11:209–218, 1988.

[Fit91] Melvin Fitting. Bilattices and the semantics of logic programming. *Journal of Logic Programming*, 11(1 and 2):91–116, July 1991.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3), October 1998.

[Gam97] Dani Gamerman. *Marko Chain Monte Carlo. Stochastic Simulation for Bayesian Inference.* Texts in Statistical Science Series. Chapman & Hall, 1997.

[GKP88] George Georgakopoulos, Dimitris Kavvadias, and Christos H. Papadimitriou. Probabilistic satisfiability. *Journal of Complexity*, 4:1–11, 1988.

[GO94] Alan Gernham and Jane Oakhill. *Thinking and Reasoning.* Blackwell Publishers, 1994.

[GZ97] Joshua Grass and Shlomo Ziberstein. Planning information gathering under unceertainty. Technical Report CMPSCI 97-32, University of Massachusetts at Amherst, May 1997.

[Hal90] Joseph Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46:311–350, 1990. A preliminary version appears in Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI 89), 1989, pp. 1375-1381.

[Hal97] Joseph Y. Halpern. A logical approach to reasoning about uncertainty: a tutorial. In X. Arrazola, K. Korta, and F. J. Pelletier, editors, *Discourse, Interaction, and Communication*, pages 141–155. Kluwer, 1997.

[HG97] Petr Hajek and Lluis Godo. Deductive systems of fuzzy logic (a tutorial). Technical Report V-707, Institute of Computer Science, Academy of Sciences of the Czech Republic, Czech Republic, February 1997.

[Hod97] W. Hodges. Compositional semantics for a language of imperfect information. *Logic Journal of the IGPL*, 5(4):539–563, 1997.

[Hog90] Christopher John Hogger. *Essentials of Logic Programming.* Graduate texts in Computer Science. Oxford University Press, Oxford, 1990.

[Jae97] Manfred Jaeger. Relational Bayesian Networks. In *Proceedings of UAI-97*, San Francisco, CA, 1997. Morgan Kaufmann.

[JL86] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. Technical Report 86/74, Monash University, Victoria, Australia, june 1986.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *POPL'87: Proceedings 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, 1987. ACM.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. An abstract machine for CLP($\mathcal{R}$). In *Proceedings ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI), San Francisco*, pages 128–139, June 1992.

[JMSY94] Joxan Jaffar, Michael Maher, Peter Stuckey, and Roland Yap. Beyond finite domains. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer, may 1994. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).

[Joh93] C. W. Johnson. A probabilistic logic for the development of safety-critical, interactive systems. *International Journal Of Man-Machine Studies*, 1993.

[Jon89] Claire Jones. *Probabilistic Non-determinism.* PhD thesis, University of Edinburgh, Edinburgh, Great Britain, August 1989.

[Jos92] Cliff Joslyn. Possibilistic semantics and measurement methods in complex systems. In Bilal Ayyub, editor, *In Proceedings of the 2nd Int Symposium on Uncertainty Modeling and Analysis (ISUMA'92)*, pages 208–215, IEEE Computer Society, 1992.

[JPM98] Cristina Sernadas Javier Pinto, Amilcar Sernadas and Paulo Mateus. Non-determinism and uncertainty in the situation calculus. Technical Report 98-PSSM-probsc, Instituto Superior Tecnico, Lisboa, Portugal, April 1998.

[KH96] Daphne Koller and Joseph Y. Halpern. Irrelevance and conditioning in first-order probabilistic logic. In *Proceedings of the* 12^{th} *Annual Conference on Uncertain ty in AI (UAI)*, pages 569–576, Portland, Oregon, August 1996.

[KM92] D. Koller and N. Megiddo. A logic for approximate reasoning. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 153–164, Cambridge, Massachusetts, October 1992.

[Kol33] A. N. Kolmogorov. *Grundbegriffe der Wahrscheinlichkeitsrechnung.* Verlag von Julius Springer, Berlin, 1933.

[Kol50] A. N. Kolmogorov. *Foundations of the Theory of Probability.* Chelsea Publishing Co., 1950. Tranlation from the German original.

[Kow79] Robert Kowalski. *Logic for Problem Solving.* Artificial Intelligence. Elsevier, New York, 1979.

[Koz98] Dexter Kozen. Set constraints and logic programming. *Information and Computation*, 142(1):2–25, April 1998.

[Kra97a] Ivan Kramosil. Probabilistic analysis of Dempster-Shafer theory. Part one. Technical Report 716, Academy of Science of the Czech Republic, 1997.

[Kra97b] Ivan Kramosil. Probabilistic first-order predicate calculus with doubled nonstandard semantics. Technical Report 714, Academy of Science of the Czech Republic, 1997.

[Kra98] Ivan Kramosil. Probabilistic analysis of Dempster-Shafer theory. Part two. Technical Report 749, Academy of Science of the Czech Republic, 1998.

[KY95] George J. Klir and Bo Yuan. *Fuzzy Sets and Fuzzy Logic: theory and applications.* Prentice-Hall, 1995.

[Lev93] Leonid A. Levin. Randomness and non-determinism. *Journal of Symbolic Logic*, 58(3):1102–1103, April 1993.

[LKI99] Thomas Lukasiewicz and Gabriele Kern-Isberner. Probabilistic logic programming under maximum entropy. In *5th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, volume 1638 of *Lecture Notes in Artificial Intelligence*, pages 279–292, London, UK, 1999. Springer.

[LLW96] Jimmy H. M. Lee, Ho-fung Leung, and Hon-wing Won. Towards a more efficient stochastic constraint solver. In *Second International Conference on Principles and Practice of Constraint Programming (CP96)*, pages 338–352, Cambridge, Massachusetts, USA, August 19-22 1996.

[LMM95] E. Lamma, P. Mello, and M. Milano. A meta constraint logic programming architecture for qualitative and quantitative temporal reasoning. Technical Report DEIS-LIA-001-95, University of Bologna (Italy), 1995. LIA Series no. 5.

[LMS95] P. Lincoln, J.C. Mitchell, and A. Scedrov. Stochastic interaction and linear logic. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, volume 222 of *London Mathematical Society Lecture Notes Series*, pages 147–166. Cambridge University Press, 1995.

[lPW92] Thierry le Provost and Mark Wallace. Domain independent propagation. In *FGCS'92*, Tokyo, June 1992.

[LS94a] Laks V.S. Lakshmanan and Fereidoon Sadri. Modeling uncertainty in deductive databases. In *Proceedings of the Int. Conf. on Database Expert Systems and Applications*, number 856 in Lecture Notes in Computer Science, pages 724–733, Athens, Greece, September 1994. Springer.

[LS94b] Laks V.S. Lakshmanan and Fereidoon Sadri. Probabilistic deductive databases. In *Proceedings of the Int.Logic Programming Symp., (ILPS'94)*, Ithaca, NY, US, November 1994. MIT Press.

[Luk99] Thomas Lukasiewicz. Probabilistic logic programming. In *13th biennial European Conference on Artificial Intelligence*, pages 388–392, Brighton, Uk, August 1999.

[LY92] F.C. Lam and W.K. Yeap. Bayesian updating: on the interpretation of exhaustive and mutually exclusive assumptions. *Artificial Intelligence*, 53(2-3):245–254, 1992.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mat93] Ludek Matyska. Logic programming with fuzzy sets. Technical Report TCU/CS/1993/4, City University, London, Great Britain, December 1993.

[MR93] Ugo Montanari and Francesca Rossi. Finite domain constraint solving and constraint logic programming. In Alain Colmerauer and Fréderic Benhamou, editors, *Constraint Logic Programming: Selected Research*, chapter 11, pages 201–222. The MIT Press, Cambridge, Mass., 1993.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

[Nea90] Richard E. Neapolitan. *Probabilistic Reasoning in Expert Systems: Theory and Algorithms.* John Willey & Sons, 1990.

[Nea93] Radford M. Neal. Probabilistic inference using markov chain monte carlo methods. Technical Report CRG-TR93-1, Dept. of Computer Science, University of Toronto, September 1993.

[Nel87] Edward Nelson. *Radically Elementary Probability Theory.* Number 117 in Annals of Mathematics Studies. Princeton University Press, 1987.

[Nil86] Nils J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28:71–87, 1986.

[NM90] Ulf Nilsson and Jan Maluszynski. *Logic, Programming and Prolog.* John Wiley, Chichester, England, 1990.

[Pap91] Athanasios Papoulis. *Probability, Random Variables and Stochastic Processes.* McGraw-Hill, 3rd edition, 1991.

[Pea86] Judea Pearl. Fusion, propagation, and structuring in belief networks. *Artificial Intelligence*, 29:241–288, 1986.

[Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufman, San Mateo, 1988. Revised second printing.

[Pea94a] Judea Pearl. Bayesian networks. Technical Report R-217, University of California, 1994. In M. Arbib (Ed.), Handbook of Brain Theory and Neural Ne tworks, MIT Press, 149-153, 1995.

[Pea94b] Judea Pearl. Three statistical puzzles. Technical Report R-217, University of California, May 1994.

[Poo93a] David Poole. Logic programming, abduction and probability: a top-down anytime algorithm for estimating prior and posterior probabilities. *New Generation Computing*, 11(3-4):377–400, 1993.

[Poo93b] David Poole. Probabilistic horn abduction and bayesian networks. *Artificial Intelligence*, 64:81–129, 1993.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, June 1990.

[PvdG96] N. B. Peek and L. C. van der Gaag. A case-based filter for diagnostic belief networks, 1996.

[Rei71] Hans Reichenbach. *The Theory of Probability.* University of California Press, 1971. Translated from German Edition of 1949.

[Ros96] Sheldon M. Ross. *Stochastic Processes.* John Wiley & Sons, 2nd edition, 1996.

[Rub95] Ronitt Rubinfeld. Randomness and computation, 1995. Lecture Notes.

[Sha76] Glenn Shafer. *A Mathematical Theory of Evidence.* Princeton University Press, 1976.

[Sha83] Ehud Y. Shapiro. Logic programs with uncertainties: A tool for implementing rule-based systems. In *Proceedings of IJCAI'83*, pages 529–532. William Kauffman, 1983.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* Logic Programming. MIT Press, Cambridge, Mass, 1986.

[SS97] R. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.

[TN94] Ahmed Y. Tawfic and Eric Neufeld. Temporal bayesian networks. In *TIME-94, An International Workshop on Temporal Representation and Reasoning*, Pensacola Beach, Florida, May 1994.

[Van88] Pascal Van Hentenryck. A constraint approach to mastermind in logic programming. *SIGART Newsletter*, 103:31–35, January 1988.

[Van89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[vdGB97] Linda C. van der Gaag and Hans L. Bodlaender. Comparing loop cutsets and clique trees in probabilistic inference. Technical Report UU-CS-1997-42, University of Utrecht, 1997.

[vdGM96a] Linda C. van der Gaag and J.-J.Ch. Meyer. Characterizing normal forms for informational independence. Technical Report UU-CS-1996-21, University of Utrecht, 1996.

[vdGM96b] Linda C. van der Gaag and J.-J.Ch. Meyer. The dynamics of probabilistic structural relevance. Technical Report UU-CS-1996-47, University of Utrecht, 1996.

[vdGM97] Linda C. van der Gaag and J.-J.Ch. Meyer. Informational independence: Models and normal forms. Technical Report UU-CS-1997-17, University of Utrecht, 1997.

[vE86] Maarten H. van Emden. Quantitative deduction and its fixpoint theory. *Journal of Logic Programming*, 3(1):37–54, April 1986.

[vE97] Maarten H. van Emden. Value constraints in the CLP scheme. *Constraints*, 2(2):163–183, October 1997.

[vEK76] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:733–742, 1976.

[vP94] Jan von Plato. *Creating Modern Probability.* Cambridge Studies in Probability, Induction, and Decision Theory. Cambridge University Press, 1994.

[Wan96] Yongge Wang. *Randomness and Complexity.* PhD thesis, Universität Heidelberg, Germany, 1996.

[WCJL$^+$98] Mark Wallace, Yves Caseau, Eric Jacquet-Lagreze, Helmut Simonis, and Gilles Pesant. CP98 Workshop on Large Scale Combinatorial Optimisation and Constraints. *Electronic Notes in Discrete Mathematics*, 1, 1998.

[Whi89] Joe Whittaker. *Graphical Models in Applied Multivariate Statistics.* John Wiley & Sons Ltd, 1989.

[WLK95] N. Wiberg, H.-A. Loeliger, and R. Kötter. Codes and iterative decoding on general graphs. *European Trans. on Telecommun*, 6:513–525, 1995.

[Wut92] Beat Wuthrich. Towards probabilistic knowledge bases. Technical Report ECRC-92-09, ECRC, Munchen, Germany, 1992.

[Wut93] Beat Wuthrich. Learning probabilistic rules. Technical Report ECRC-93-03, ECRC, Munchen, Germany, January 1993.

[Zad65] Lofti A. Zadeh. Fuzzy sets. *Information and Computation*, 8(3):338–353, June 1965.