ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI BOLOGNA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria Informatica

Enhancing Symbolic AI Ecosystems with Probabilistic Logic Programming: a Kotlin Multi-Platform Case Study

Tesi di laurea in
Linguaggi e Modelli Computazionali M

Relatore
Prof.ssa Roberta Calegari

Candidato
Jason Dellaluce

Correlatori
Dott. Giovanni Ciatto
Prof. Enrico Denti

Prima Sessione di Laurea
Anno Accademico 2020-2021

Abstract

As Artificial Intelligence (AI) progressively conquers the software industry at a fast pace, the demand for more transparent and pervasive technologies increases accordingly. In this scenario, novel approaches to Logic Programming (LP) and symbolic AI have the potential to satisfy the requirements of modern software environments. However, traditional logic-based approaches often fail to match present-day planning and learning workflows, which natively deal with uncertainty. Accordingly, Probabilistic Logic Programming (PLP) is emerging as a modern research field that investigates the combination of LP with the probability theory. Although research efforts at the state of the art demonstrate encouraging results, they are usually either developed as proof of concepts or bound to specific platforms, often having inconvenient constraints. In this dissertation, we introduce an elastic and platform-agnostic approach to PLP aimed to surpass the usability and portability limitations of current proposals. We design our solution as an extension of the 2P-Kt symbolic AI ecosystem, thus endorsing the mission of the project and inheriting its multi-platform and multi-paradigm nature. Additionally, our proposal comprehends an object-oriented and pure-Kotlin library for manipulating Binary Decision Diagrams (BDDs), which are notoriously relevant in the context of probabilistic computation. As a Kotlin multi-platform architecture, our BDD module aims to surpass the usability constraints of existing packages, which typically rely on low level C/C++ bindings for performance reasons. Overall, our project explores novel directions towards more usable, portable, and accessible PLP technologies, which we expect to grow in popularity both in the research community and in the software industry over the next few years.

Acknowledgements

This thesis marks the conclusion of a long phase of my life path. Of this little milestone of mine, I owe the accomplishment to many.

First, my thoughts go to my family and closest ones. To my parents Benedetta and Luigi, who believed in me from day one, way before I did. To my sister Gaia, who I deeply love and admire. To my girlfriend Laura, for having been resilient and supportive specially during difficult times.

Too many to be singularly mentioned, I'm grateful for all my close friends that I had the pleasure of encountering along the way. Each one of them has been a building block of the person I have become.

I'd also like to express my gratitude to all the precious mentors and role models that I had the privilege to learn from, and which had faith in my potential. Among the others, special thanks go to Vittorio Viarengo, Ned Rockson, and Nicolas Salhuana, for walking me through the first steps of my professional career.

Lastly, I'm thankful to Prof.ssa Roberta Calegari, Dott. Giovanni Ciatto, and Prof. Enrico Denti for their trust and support throughout the years. They pushed me to go the extra mile, more than once.

Contents

List of Figures

Listings

Chapter 1

Introduction

Artificial Intelligence (AI) is progressively conquering the software industry to become one of the most pivotal fields, with a fast-paced evolution of challenges and requirements that existing technologies often fail to match. Accordingly, the increasing demand for transparent and pervasive intelligence is opening new horizons for Logic Programming and Symbolic AI approaches. In this scenario, technologies such as *2p-kt* propose innovative solutions for multi-paradigm symbolic manipulation aimed to fit the requirements imposed by modern software environments. However, logic-based approaches alone are often not suitable to be integrated with present-day planning and learning workflows, which natively deal with uncertainty and probabilistic decision-making.

Probabilistic Logic Programming (PLP) is a modern research field that investigates the combination of Logic Programming with the probability theory. Although state of the art proposals demonstrate significant results, most solutions currently implement either proofs of concept or monolithic runtimes, often targeting single platforms or having inconvenient constraints and dependencies.

The goal of this dissertation is to propose an elastic and platform-agnostic inference engine for Probabilistic Logic Programming aimed to surpass the usability constraints of the current proposals. Our contribution is highly accessible in heterogeneous software environments and harmoniously integrates with existing logic programming workflows by being backwards-compatible with traditional *Prolog*.

We accomplish our purpose by proposing an extension of *2p-kt* for supporting PLP in the form of new micro-modules, thus inheriting the multi-paradigm and multi-platform nature of the project. Additionally, our proposal comprehends an object-oriented and *pure-Kotlin* library implementation for *Binary Decision Diagrams* (BDDs), a data structure commonly leveraged to perform probabilistic computation. Our BDD library acts as a multi-platform lean proposal that leaves space for PLP-specific optimizations, and that goes beyond some limitations of existing packages that rely on legacy low-level *C* or *C++* bindings for performance reasons. Finally, a well-covering test suite is provided to assert the validity of our PLP implementation and its backward compatibility with Prolog.

Accordingly, the reminder of this thesis is structured as follows. In Chapter 2, we provide a technical background for what concerns Probabilistic Logic Programming, Binary Decision Diagrams, and the *2p-kt* project. In Chapter 3, we analyze requirements and needs we acknowledged for our proposal. Chapter 4 discusses the design we envisioned for the project by also inspecting the architecture in its software modules. In Chapter 5, we disclose details about our *Kotlin* multi-paradigm implementation, the optimizations we applied, and some preliminary attempts. Chapter 6 articulates our assessment methodology. Finally, Chapter 7 concludes this thesis by summarising its main contribution.

Chapter 2

State of the Art

This chapter introduces notions and state of the art contributions that we reference in the following chapters of this dissertation. In Section 2.1, we provide background knowledge regarding Probabilistic Logic Programming. Section 2.2 is dedicated to Binary Decision Diagrams as a means to solving the Knowledge Compilation problem. Finally, in Section 2.3 we introduce the *2p-kt* project on which this contribution is based on.

2.1 Probabilistic Logic Programming

This section provides founding knowledge for the research field of Probabilistic Logic Programming. In Section 2.1.1, we give a broad introduction of Logic Programming approaches. Then, in Section 2.1.2 we present the Distribution Semantics with a simplified set of formal definitions. Section 2.1.3 describes Annotated Disjunctions and their impact on languages based on Distribution Semantics. In Section 2.1.4 we delineate the problem of Knowledge Compilation and state of the art solutions for it. Finally, in Section 2.1.5 we discuss the most common tasks achievable with Probabilistic Logic Programming.

2.1.1 Logic Programming

Symbolic approaches in artificial intelligence are based on the idea of representing a problem with high-level descriptive forms, formally referred to as *symbols*. Symbols are generally bound to specific problem domains and humans are capable of easily reading and understanding their meaning. Although the notion of *meaning* remains alien to software-based agents, they are still able to manipulate symbols according to mathematical and logical rules to perform reasoning tasks. Inherently, goals and behaviors determined by symbolic reasoning are intelligible and transparent to human observers.

The term *Logic Programming* refers to a declarative programming paradigm that uses formal logic to perform computing tasks and to represent knowledge. Logic programs define symbols and logical relationships between them, without disclosing any detail about the computational flows that might be involved. Approaches based on Logic Programming have been subject to considerable attention when associated with Artificial Intelligence in both the literature and the industry. Notorious applications include Knowledge Management, Robotics, Natural Language Processing, Bioinformatics, and Agent-Based Systems. Among many proposals, the language family based on *Prolog* is one of the most adopted implementations of Logic Programming.

Prolog is a programming language based on first-order logic. Its syntax is purposely meant to be easily understandable, and encodes logical propositions in the form:

$$Head :- Body$$

Clauses like the one above are formally known as *Horn Clauses*[15] and can be interpreted as *"Head is true if Body is true"*, where *Head* and *Body* are *terms*. As in first-order logic, terms can either be *atoms* or *predicates* with a given number of arguments. Formally, the number of arguments of a predicate is referred to as *arity*. Prolog also supports *Variables* terms, allowing the representation of both ground and non-ground formulae. Through the use of terms, the language defines strings, numbers, variables, lists, compound terms, and operators. Negated literals are supported through a mechanism commonly known as "*Negation as Failure*".

Listing 2.1: Prolog - Family Relationships Example

```
person(john).
person(anna).
person(mike).
person(jane).
male(john).
male(mike).
female(anna).
female(jane).
parent(mike, john).
parent(mike, anna).
parent(jane, anna).

father(X, Y) :- male(X), parent(X, Y).
mother(X, Y) :- female(X), parent(X, Y).
sibling(X, Y) :- parent(Z, X), parent(Z, Y).
```

An example of a term is *john*, that is a constant, or *human(john)*, a compound term where the predicate *human* is a function symbol with arity 1 and has *john* as its one argument. Conjunctions and disjunctions of terms are represented as terms as well. For reference, a simple Prolog theory representing family relations can be modeled as in Listing 2.1 Notably, that example shows that logic relationships are expressed in Prolog in the form of *Rules* that respect the semantics of Horn Clauses. Rules with no body are formally named *Facts* and implicitly have a *true* body.

Furthermore, the Prolog language allows for solving queries over knowledge bases, which is analogous to proving a theorem by resolution (Proof by Resolution [15]). Queries can have zero or many solutions, each one constituting a logic program that proves the query terms to be true over the given knowledge base. Prolog solvers usually rely on *Selective Linear Definite* (SLD) Resolution to solve queries by exploring solution spaces in a depth-first fashion.

With reference to Listing 2.1, an example of query might be `father(mike, X)`. Prolog solvers would find two solutions for such a query, that are `father(mike, john)` and `father(mike, anna)`, where the variable `X` is substituted with terms `john` and `anna`. A more generic query could be `father(X, Y)` aiming to find every father-child pair in the theory, which in this case would give the same results as the previous example. It is worth noting that queries such as `father(john, X)` would have no solution because they would have no proof nor substitution over the provided theory, making it a false statement. That is because Prolog is based on a *Closed World Assumption*, namely considering every term a false statement until proven otherwise.

Although Logic Programming provides means for performing symbolic reasoning tasks and representing knowledge, it fails to satisfy numerous challenges of Artificial Intelligence. Logic-based approaches are in fact constrained to a vision of reality based on pure-logic, so that terms and solutions can exclusively be either true or false statements. However, the majority of real-life situations deal with degrees of uncertainty, for which a binary approximation is clearly inadequate. Evident examples of those use cases are autonomous planning and automatic learning, which are probabilistic by nature.

2.1.2 Distribution Semantics

Various approaches have been proposed for combining logic programming with probability theory to attain robust reasoning capabilities in uncertainty-first scenarios. Among the others, we put our attention on methods based on *Distribution Semantics (DS)*[16]. Languages of this category assign probabilistic distributions over logic programs by encoding random choices to each clause. So, the probability of a query is defined as the joint distribution of all the random choices used to find a solution. The process gets more complicated in languages supporting function symbols. Languages based on DS define the probability over clause choices with different syntax, but they all have the same expressive power[13]. Notable mentions include *PRISM*[17], *Logic Programs with Annotated Disjunctions (LPADs)*[20], *ProbLog*[11], *Probabilistic Horn Abduction (PHA)*[10]. Of those lan-

guages, we exclusively focus on *ProbLog* because it has the simplest syntax.

ProbLog has been designed as a simple probabilistic extension of Prolog by expressing probability distributions over facts. With this idea, facts become *probabilistic facts* and respect the following notation:

$$\Pi_i :: f_i$$

Where $\Pi_i \in [0, 1]$ and f_i is an atom, implying that each ground instantiation of the term f_i is true with probability Π_i and false with probability $1 - \Pi_i$. As such, grounding a probabilistic fact creates two scenarios with different probabilities: one for which the fact is true, and one for which the fact is false. The concept can easily be extended to clauses, thus attaining *probabilistic clauses*. We proceed by providing some formal definitions, and purposely avoiding excessive detail to not diverge from the scope of this dissertation. Detailed definitions can be found in [13], that we use as reference in the following paragraphs.

An *atomic choice* is a triple (f, θ, k) indicating whether a grounding $f\theta$ of a probabilistic fact $F = p :: f$ is selected or not, and $k \in \{0, 1\}$ with $k = 1$ when the fact is selected and $k = 0$ when it is not. A set of atomic choice is *consistent* if only one alternative is selected for a given probabilistic fact. Formally, a consistent atomic choice does not contain both $(f, \theta, 0)$ and $(f, \theta, 1)$. A consistent set of atomic choice is referred to as a *composite choice*, for which the probability $P(\kappa)$ is defined as:

$$P(\kappa) = \prod_{(f_i,\theta,1)\in\kappa} \Pi_i \prod_{(f_i,\theta,0)\in\kappa} (1 - \Pi_i)$$

With Π_i being the probability that the fact f_i is true.

A selection σ is a composite choice that contains one atomic choice for *every* grounding of *every* probabilistic fact of a knowledge base. The logic program defined by a selection σ is referred to as a *probabilistic world* w_σ, and its probability is equivalent to the one of σ.

In languages that do not support function symbols and variables, the set of possible groundings of a probabilistic fact is finite, and the set of possible probabilistic worlds is finite as well. For sake of simplicity, from this point on we

provide definitions that stand by assuming the language does not support function symbols. If that's the case, we know that the probability of a probabilistic program depends on the set of all its probabilistic worlds $W = \{w_1, ..., w_n\}$, and that $\sum_{w \in W} P(w) = 1$.

Let q be a ground term representing a query, we define its conditional probability given a world w as $P(q \mid w)$, which is 1 if q in w and 0 otherwise. Then, the probability of a query q can be defined as following:

$$P(q) = \sum_{w} P(q, w) = \sum_{w} P(q \mid w) P(w) = \sum_{w \models q} P(w)$$

Briefly, the probability of a query can be obtained by summing the probabilities of all the possible worlds where the query is true.

This definition allows to compute the probability of a conjunction of ground terms $q_1, ..., q_n$ as well. As such, we are able to model problems involving conditional probability, in which we aim to calculate the probability of a query q given some evidence represented as a conjunction of ground terms $e = e_1 \wedge ... \wedge e_m$:

$$P(q \mid e) = \frac{P(q, e)}{P(e)}$$

Given the above, we proceed by providing a simple example. Consider the following program composed by two probabilistic facts that models an hypothetical problem of launching two coins.

$$0.5 :: heads1.$$
$$0.6 :: heads2.$$

The probabilistic facts *heads1* and *heads2* define the probability of landing on heads for the first and the second coin respectively. This program has four possible worlds:

$$w_1 = \{heads1, heads2\} \quad P(w_1) = 0.5 \times 0.6 = 0.3$$
$$w_2 = \{heads1, \neg heads2\} \quad P(w_2) = 0.5 \times (1 - 0.6) = 0.2$$
$$w_3 = \{\neg heads1, heads2\} \quad P(w_3) = (1 - 0.5) \times 0.6 = 0.3$$
$$w_4 = \{\neg heads1, \neg heads2\} \quad P(w_4) = (1 - 0.5) \times (1 - 0.6) = 0.2$$

Considering a query $q_1 = heads1 \wedge heads2$, for which both coins land on heads, the set of possible worlds W_{q_1} in which q_1 is true contains only w_1, so calculating the probability is straightforward.

$$P(q_1) = \sum_{w \in W_{q_1}} P(w) = P(w_1) = 0.3$$

Things get slightly more complex if we take into consideration a query $q_2 = heads1 \vee heads2$, in which either coin lands on heads. In this scenario, q_2 is true in three worlds: w_1, w_2, and w_3.

$$P(q_2) = \sum_{w \in W_{q_2}} P(w) = P(w_1) + P(w_2) + P(w_3) = 0.3 + 0.2 + 0.3 = 0.8$$

Note that we calculate probabilities of queries as simple summations of the probability of all the worlds in which they are true. This is not guaranteed to be generally safe in terms of correctness, because contributions from each world are combined disjunctively. As such, this situation can be compared to a disjunction of two independent Boolean random variables:

$$P(a \vee b) = P(a) + P(b) - P(a)P(b) = 1 - (1 - P(a))(1 - P(b))$$

As clearly visible, with a growing number of worlds and facts involved, calculating the probability of a query becomes a considerably complex problem, if worlds distribution are not disjoint from one another. This topic is discussed more in detail in Section 2.1.4.

2.1.3 Annotated Disjunctions

In the previous section we presented a first way for defining a probability distribution over logic facts. However, the definitions provided are not sufficient in the general case. By defining a probability value over head of clauses, we implicitly have two disjunctive logic programs: one in which the head term is true, and one in which it is false. The concept can be further extended to have multiple disjunctive logic programs from the same clause, by defining a formalism known as *Annotated*

Disjunctions (LPADs)[20]. These kind of programs consist of a set of rules of the following form:

$$h_1 :: \Pi_1 \vee ... \vee h_1 :: \Pi_n \leftarrow b_1, ..., b_m$$

Namely, clauses can have multiple head terms, each one with a probability value. In this form, each probabilistic clause represents a choice among n normal clauses with one head h_i and probability Π_i, that are disjoint from one another. Moreover, an additional clause is added implicitly:

$$null :: \Pi_0 \leftarrow b_1, ..., b_m$$

This clause represents the case in which none of the n heads is true, and inherently has probability $\Pi_0 = 1 - \sum_{k=1}^{n} \Pi_k$. Accordingly, the concept of *atomic choice* is extended as well. Given a clause C_i in an LPAD, an atomic choice can be defined by the triple (C_i, θ_j, k), where θ_j is a ground substitution for C_i and $k \in \{0, 1, ..., n\}$. A *selection* σ identifies a probabilistic world w_σ that contains all normal clauses obtained by selecting a single head term for each atomic choice (C_i, θ_j, k). As such, the probability of w_σ is formalized as follows:

$$P(w_\sigma) = P(\sigma) = \prod_{(C_i, \theta_j, k) \in \sigma} \Pi_{ik}$$

The rest of the definitions provided in the previous section, with the context of ProbLog, remain valid.

We proceed by providing a simple example of LPAD. Consider having a box containing 10 colored balls, of which 6 are known to be red and 3 to be green. In the following example, we model this problem with a single clause leveraging Annotated Disjunctions.

$$0.6 :: red\_ball \ ; \ 0.3 :: green\_ball.$$

The only clause of this program can be interpreted as three disjoint facts: one with 60% probability of being true if the ball is red, another with 30% probability if the ball is green, and a third one with 10% probability if the ball is neither red nor green. As these facts are disjoint, they can never appear together in the same *selection* σ. This concept is relevant while enumerating the set of probabilistic

worlds, which in this simple case is composed of only three elements (one for each disjoint head). Note, a similar ProbLog program with three non-disjoint probabilistic facts would have $2^3 = 8$ worlds instead.

2.1.4 Knowledge Compilation

As we mentioned in Section 2.1.2, providing formal definitions of Distribution Semantics for logic programs supporting function symbols is beyond the scope of this dissertation. As such, we only formalize the concepts that we reference in the rest of the narration.

We define a set of probabilistic worlds w_κ that is *compatible* with a composite choice κ as $w_\kappa = \{w_\sigma \in W \mid \kappa \subseteq \sigma\}$, with W being a set containing all worlds of a given program. As such, a composite choice identifies a set of probabilistic worlds. Note that, in programs with function symbols, w_κ may be uncountable and $P(w_\kappa)$ can potentially converge to 0. Given a query q, a composite choice κ is an *explanation* for q if $\forall w \in w_\kappa : w \vDash q$. A set K of composite choices is *covering* with respect to q if every world in which q is true belongs to w_K. Considering a program and a query q, we define the function:

$$Q(w) = \begin{cases} 1 & \text{if } w \vDash q \\ 0 & \text{otherwise} \end{cases}$$

We know that Q is measurable if q has a finite set K of finite explanations so that K is covering[13]. With that, we indicate that $P(q) = P(Q = 1)$.

Given the above, calculating the probability of a query over a probabilistic program with function symbols requires finding a covering set of *explanations*. However, as mentioned in Section 2.1.2, calculating the probability of a set of explanations brings back the formula for computing the probability of a disjunction:

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b)$$

In this context, the a and b terms of the formula represent explanations. We know that $P(a \wedge b) = P(a)P(b)$ if a and b are independent, however this is not usually

the case. Assuming a large set of explanations with no independence guarantee, computing the probability of a query with the formula above is a complex and intractable problem.

Observe that the probability of a query q is based on conjunctions and disjunctions of random variables defined by atomic choices over a probabilistic logic program. Inherently, a covering set of explanations can be encoded in a *Boolean formula*. Such a formula is a function of Boolean random variables, each encoding a choice, that has value 1 for assignments representing worlds in which q is true. Since we know the probability distribution of each Boolean variable, we can also compute the probability of the formula assuming value 1. However, to do so we must guarantee all Boolean variables to be *pair-wise independent*.

Knowledge Compilation is an approach based on the idea of compiling a Boolean formula in a strategic form that makes some specific computation task easier to perform. In our case, this happens in two steps: first, we compose a Boolean formula that represents a covering set of explanations for a query q, then, we transform the formula in a form that makes probability computation easier. For inference tasks, Probability is usually calculated over the new form through *Weighted Model Counting* (WMC), that can be computed in polynomial time with some specific forms.

In this domain, various approaches for Knowledge Compilation are present in the literature. State of the art proposals include *Binary Decision Diagrams* (*ProbLog1*[11], *PITA*[14]), *Multi-Valued Decision Diagrams* (*cplint*[12]), *Deterministic Decomposable Negation Normal Forms* (*ProbLog2*[5]), and *Sentential Decision Diagrams* (*ProbLog2*[5]). The choice of any of those options is clearly not neutral, as they show different tractability towards specific kinds of operations. In this dissertation, we focus exclusively on *Binary Decision Diagrams* (BDDs) that are defined in detail in Section 2.2.

2.1.5 Reasoning Tasks

Various approaches related to reasoning have been proposed in the literature with the scope of Probabilistic Logic Programming. Although this dissertation focuses only in a specific subset of those, we broadly describe the main ones to provide context to our narration:

Inference — We want to compute the probability of a query q over a logic program, eventually with some evidence e. This implies investigating the probability of evidence $P(e)$, the unconditional probability $P(q)$, and the conditional probability $P(q \mid e)$ as well. Other notable inference tasks include finding a *most probable explanation* and detecting the probability density of q. Generally, these approaches fall into two categories of *Exact Inference* or *Approximated Inference*. Exact Inference is meant to solve problems with exactness by strictly following theoretic definitions for calculations. Although this approach ensures correctness, it can easily become considerably resource expensive. Approximated Inference takes the opposite route, and attempts to perform computations by applying simplifications. This causes computation results to be formally incorrect but with acceptable thresholds, and resource usage to be reduced as well.

Weight Learning — We want to learn the parameters of a probabilistic logic program with a given structure and a set of data samples. Data is provided in the form of ground logical terms, and the system attempts to infer parameters that assign maximum probability to the samples over the given program. In this case, by *parameter* we refer to the probability values for each clause present in the program, that may not be known in certain cases.

Structure Learning — We want to learn whole probabilistic logic programs from data samples, including both their structure and parameters. Those approaches are mostly based on *Inductive Logic Programming* (ILP)[15], with specific adaptations to include the notion of probability.

As visible, reasoning tasks in Probabilistic Logic Programming are either concerned about inference or learning. In this dissertation, we focus on tasks regarding

Exact Inference only.

2.2 Binary Decision Diagrams

This section discusses Binary Decision Diagrams in the context of their combination with Probabilistic Logic programming. In Section 2.2.1, we introduce the data structure and provide fundamental definitions. Then, in Section 2.2.2 we inspect some main of the main algorithms. Finally, in Section 2.2.3 we investigate how Binary Decision Diagrams can be used in Probabilistic Logic Programming to support Knowledge Compilation, and how the probability of a logic program can be computed through Weighted Model Counting.

2.2.1 Definitions

Binary Decision Diagrams (BDDs) are data structures capable of representing Boolean functions. They gained considerable popularity in the design and verification of digital systems, which require the manipulation of large propositional formulae. BDDs are rooted graphs that have one level for each Boolean variable of the represented function. Nodes in the graph represent Boolean variables and have two outgoing arcs: one corresponding to the *true* value of the variable, and one corresponding to the *false* value of the variable. *Terminals* are an edge case of nodes that have no outgoing arcs and have a constant value of *true* or *false* (also called, respectively, *1-terminals* and *0-terminals*). More practically, each node of a BDD indicates a *choice* over a Boolean random variable and has one arc for each possible outcome.

Figure 2.1 shows a simple example of BDD with three Boolean variables X_1, X_2, and X_3. In that, variable nodes are represented as ellipsis boxes and terminal nodes with square boxes that have a value of 0 or 1. In each node, a solid outgoing arc represents the path for the *true* choice of the variable, and the dashed arc indicates the *false* path. The Boolean function can be evaluated by following all paths from the root node to the terminals, with the root node being the first

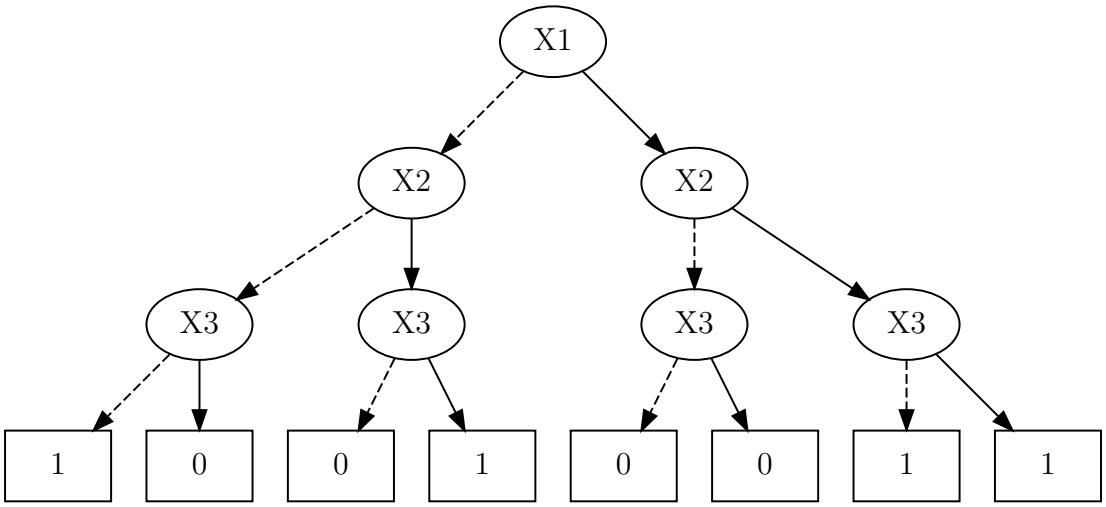


Figure 2.1: Example of Non-Reduced Binary Decision Diagram

Boolean variable. Furthermore, the order in which variables appear is consistent over the whole graph, and delineates a specific variable ordering constraint (in this case, $X_1 < X_2 < X_3$). In this form, the given Binary Decision Diagram is not distinguishable from a simple *Binary Decision Tree*[15]. This is because, in this example, the data structure has not been *reduced*. Details regarding a reduction algorithm for Binary Decision Diagrams are provided in Section 2.2.2.

Consider the reduced and ordered BDD in Figure 2.2, representing the function:

$$f = X_1 \cdot X_2 + X_3$$

This diagram shows relevant properties unseen in the previous example. Work from Bryant[2] demonstrated that reduced and ordered Binary Decision Diagrams are a *canonical* representation of Boolean functions. Besides uniquely representing a function, the canonicity property implies additional benefits such as making equivalence checking inexpensive and allowing the usage of memoization techniques on complex operations. Note that imposing a certain variable ordering constraint over a BDD is not a neutral choice, because it strongly affects the size of the diagram itself. We do not explore this topic in detail, but it's relevant to understand that an optimal variable ordering can potentially produce significantly

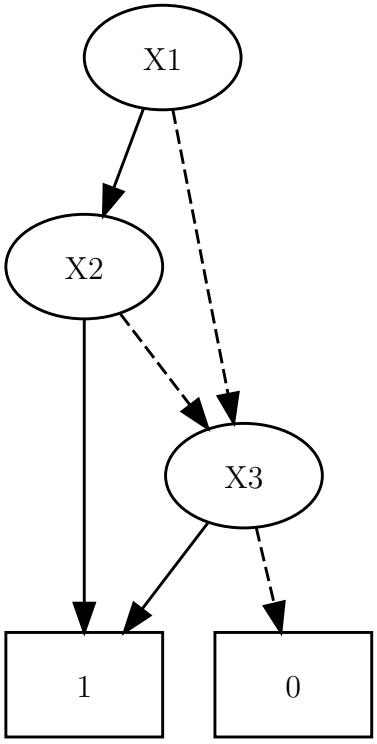


Figure 2.2: Example of Reduced Binary Decision Diagram

smaller BDDs than non-optimal orderings. However, the problem of finding the best variable ordering is known to be *NP-hard*. Lastly, note that the canonicity property is granted for a particular Boolean function with a one specific variable ordering.

Generally, reduction and ordering are taken for granted, and so informal references to BDDs implicitly indicate *Reduced Ordered Binary Decision Diagrams* (ROBDDs).

2.2.2 Algorithms

Given their importance in certain application domains, there has been considerable research interest of operations and algorithms over Binary Decision Diagrams. In this section, we inspect the operations involved in our project proposal, along with their related founding concepts.

First, we introduce some additional notation. Given a generic BDD node n:

- $id(n)$ is a unique identifying label for the node n
- $var(n)$ indicates the Boolean variable label represented by node n
- $lo(n)$ indicates the node reached through the *false* outgoing arc (*low* arc)
- $hi(n)$ indicates the node reached through the *true* outgoing arc (*high* arc)

For terminal nodes, which are leaves in the BDD graph structure, the notions of $var(n)$, $lo(n)$, and $hi(n)$, are not meaningful. Instead, for terminals we indicate with $val(n)$ the truth value of the node.

Considering any Boolean formula f, and a variable x inside that formula, it can be rewritten as:

$$f = (\neg x \wedge f[0/x]) \vee (x \wedge f[1/x])$$

Where $f[t/x]$ indicates the function f with the Boolean variable x substituted by the truth value t. This is known as the *Shannon Expansion*[1] of the Boolean formula f. If a Binary Decision Diagram represents a Boolean function f we

observe that for each node n, $lo(n)$ represents $f[0/var(n)]$ and $hi(n)$ represents $f[1/var(n)]$. As such, BDDs represent Boolean functions in Shannon normal form. This notion is fundamental to define construction algorithms for BDDs by means of Shannon Expansion.

With our notation, a BDD for a certain Boolean variable x can be simply be defined as a node n so that $var(n) = x$, $lo(n) = 0$ and $lo(n) = 1$. However, we are interested in constructing BDDs by means of Boolean operations between other BDDs. In that perspective, we start by observing that a generic binary operator $\otimes$ between two Boolean formulae f and g can be defined through Shannon Expansion as:

$$f \otimes g = (\neg x \wedge (f[0/x] \otimes g[0/x])) \vee (x \wedge (f[1/x] \otimes g[1/x]))$$

The `apply` algorithm revolves around this concept. Given two Boolean formulae f and g represented by BDD_f and BDD_g, and a binary Boolean operator $\otimes$, `apply` takes BDD_f and BDD_g as operands and produces a $BDD_{f \otimes g}$ representing the formula $f \otimes g$. Using this algorithm, any binary Boolean operation can be defined for a pair of BDDs. This concept can be extended to support unary operators as well. For instance, we can define the Boolean negation of a variable as $\neg x = x \otimes 1$.

The `apply` algorithm is defined as a recursive application of the binary operator over the nodes of the two operand BDDs, that builds the result by proceeding bottom-up. Usually, *Dynamic Programming* is used to optimize the procedure so that the complexity can remain bounded. The procedure is defined as in Algorithm 1. For reference, consider G being an *hash table* that supports Dynamic Programming, and `Mk` a procedure that constructs a new node from a triple (var, lo, hi). Given two operands B_f and B_g, `apply` has a time complexity of $O(\|B_f\| \cdot \|B_g\|)$, with $\|B_f\|$ and $\|B_g\|$ being the number of nodes in B_f and B_g.

By using `apply`, we are able to support all the basic Boolean operations (and, or, not) in Binary Decision Diagrams. However, the algorithm itself fails to construct reduced diagrams. So, we introduce another procedure: `reduce`, which constructs a ROBDD from an Ordered Binary Decision Diagrams. `apply` and `reduce` are orthogonal, and it is easy to adapt the construction algorithm to produce reduced BDDs directly. `reduce` proceeds bottom-up and follows a set of rules

Algorithm 1 Binary Decision Diagrams - `apply` Algorithm

```
1: function APPLY(op, a, b)
2:     init(G)
3:     function APP(u_1, u_2)
4:         if G(u_1, u_2) ≠ empty then return G(u_1, u_2)
5:         else if u_1 ∈ {0, 1} and u_2 ∈ {0, 1} then
6:             u ← op(u_1, u_2)
7:         else if var(u_1) = var(u_2) then
8:             u ← Mk(var(u_1), App(lo(u_1), lo(u_2)), App(hi(u_1), hi(u_2)))
9:         else if var(u_1) < var(u_2) then
10:            u ← Mk(var(u_1), App(lo(u_1), u_2), App(hi(u_1), u_2))
11:        else
12:            u ← Mk(var(u_2), App(u_1, lo(u_2)), App(u_1, hi(u_2)))
13:        end if
14:        G(u_1, u_2) ← u
15:        return u
16:    end function
17:
18:    return App(a, b)
19: end function
```

for performing reductions over the nodes of a BDD, which define a way of labelling nodes with their identifier $id(n)$.

- **Remove duplicate terminals**: If n is a terminal node, then set $id(n)$ to be $val)n)$
- **Remove redundant tests**: If $id(lo(n)) = id(hi(n))$ then set $id(n)$ to be $id(lo(n))$
- **Remove duplicate nodes**: If there exist a node m that has already been labelled so that $var(m) = var(n)$, $lo(m) = lo(n)$, and $hi(m) = hi(n)$, then set $id(n)$ to be equal to $id(m)$

This procedure makes use of a *hash table* to keep track of labelled nodes and have $O(1)$ lookup time. Given an input B, `reduce` has a time complexity of $O(\|B\| \cdot \log \|B\|)$, with $\|B\|$ being the number of nodes in B.

2.2.3 Knowledge Compilation

As we introduced in Section 2.1.4, Binary Decision Diagrams are a popular choice in Probabilistic Logic Programming as a solution for Knowledge Compilation. Again, in PLP we want to calculate the probability of a disjunction over a covering set of explanations. Recall that *explanations* are *composite choices* for a given query q, and that we can compare them to Boolean formulae where each variable is characterized by an *atomic choice*. As such, Binary Decision Diagrams are suitable for encoding explanations because they allow to represent large Boolean formulae, to reduce them, and to apply Boolean operations. Moreover, Boolean variables in a BDD are *pair-wise independent*.

So, the Knowledge Compilation task consists in encoding *explanations* into BDDs, and then manipulating them to compute the probability of queries. Consider an *atomic choice* (C, θ, k) representing a grounding of a *probabilistic clause*. We recognize that a Binary Decision Diagram node can easily be constructed as follows:

- Define a binary random variable $X_{C\theta}$ representing the probabilistic clause C with grounding θ, with probability π of being true equal to the probability of C.
- Ensure that $X_{C\theta}$ is somehow comparable to other random variables so that a certain variable ordering is respected
- Assign the random variable to node n so that $var(n) = X_{C\theta}$
- Set $lo(n) = 0$ and $hi(n) = 1$, so that n has a outgoing arc for each value of k

Then, consider a set of explanations K for the given query q. We want to represent each explanation as a BDD as well. So, Observe that explanations are defined as a *conjunction* over all the contained atomic choices. We can reproduce the conjunction by using the `apply` operation with *and* Boolean operator over all the BDD nodes encoding the atomic choices.

Finally, remember that the probability of a query q is defined over a disjunction of all its explanations. We obtain the disjunctive formula by using the `apply` operation with *or* Boolean operator over all BDDs representing the explanations of K. The final result is one Binary Decision Diagram that represents the following formula:

$$f_K = \bigvee_{\kappa \in K} \bigwedge_{(C_i, \theta_j, I) \in \kappa} (X_{C_i \theta_j} = I)$$

Given that we are able to compile the solving formula f_K of the query q into a BDD, we want to compute the probability of q. *Weighted Model Counting* over Binary Decision Diagrams is based on the Shannon Expansion, and can be used to compute the overall probability by considering the random variables in each node. There is a well-known algorithm that achieves this goal, as shown in Algorithm 2, that leverages Dynamic Programming to optimize the computation[11].

Note, that *Annotated Disjunctions* violate some assumptions of the Knowledge Compilation method proposed in this section. In LPADs, atomic choices are not binary, which would lead to Decision Diagrams where each node has more than two outgoing arcs. In fact, in previous works[12] considered the adoption of *Multi-Valued Decision Diagrams* (MDDs) to solve this issue. Namely, those are Decision

Algorithm 2 Binary Decision Diagrams - Probability Calculation

```
1: function Prob(node)
2:     if node is a terminal then return 1.0 · val(node)
3:     else
4:         if TableProb(node) ≠ null then return TableProb(node)
5:         else
6:             p0 ← Prob(lo(node))
7:             p1 ← Prob(hi(node))
8:             Let π be the probability of being true of var(node)
9:             Res ← p1 · π + p0 · (1 − π)
10:            Table(node) ← Res
11:            return Res
12:        end if
13:    end if
14: end function
```

Diagrams with an arbitrary count of outgoing arcs. However, given that BDDs software packages are more developed and efficient, a second conversion is often required from MDDs to BDDs in real applications. The most efficient method to implement such a conversion is by applying *binary splits* over the multi-valued random variables.

The open source community proposed various Binary Decision diagram manipulation packages. Considering that BDDs algorithms can reach considerable complexity with very large Boolean formulae, the most efficient packages rely on low-level implementations written in *C* or *C++*. Also, most implementations in other programming languages act as simple wrappers of those low-level packages. Most popular solutions include *CUDD*[18], *BuDDy*[3], *BeeDeeDee*[9], *JDD*[19].

2.3 2P-Kt

2p-kt is born as a modern and evolved reboot of *tuProlog*, a Prolog implementation proposed two decades ago. In Section 2.3.1, we give a broad introduction of

tuProlog, along with its merits and limitations. Section 2.3.2 discusses the Kotlin language focusing on its multi-platform capabilities. In Section 2.3.3, we inspect the core points of the *2p-kt* project and its architecture. Finally, in Section 2.3.4, we show the mechanism that enables multi-paradigm programming.

2.3.1 tuProlog

tuProlog is a logic programming framework supporting multi-paradigm programming via a clean, seamless, and bidirectional integration between the logic and object-oriented paradigms.[4]. The project was proposed in the late 90s and is structured as a Java implementation of a Prolog inference core. Its architecture is compact and follows sound engineering practices acting as a lean, modular, and configurable code base.

Indeed, tuProlog introduces relevant interesting characteristics. First, its inferential core is modeled as a *Finite State Machine* (FSM), thus making it easily understandable, verifiable, and virtually extensible. Second, the framework is lightweight and provides a strict amount of built-ins, thus optimizing resource expenses and making it suitable for devices with modest computation capabilities. Third, the project is highly extensible and customizable through the usage of *libraries*. Finally, it offers non-intrusive vectors that run Prolog inferences from Java, and to leverage Java objects during logic reasoning as well. The two worlds maintain a transparent separation but are empowered of full bidirectional cooperation opportunities that enables coherent multi-paradigm programming. Those characteristics, coupled with the inherited multi-platform nature of the *Java Virtual Machine* (JVM), make *tuProlog* suitable for being used in distributed environments and for ubiquitous computing.

On the other hand, tuProlog suffers from some structural defects and constraints that were mostly imposed by the technology limitations at the time of release. Among these problems, a relevant one is the intensive usage of *reflection* in multi-paradigm and internal invocations that deteriorates code readability and reduces the overall performance. Furthermore, classes for model and business logic are designed for being *mutable*, which occasionally leads to an ambiguous usage of

those and an overall poorly testable design.

2.3.2 Kotlin and Multi-Platform Support

Kotlin[8] is a statically typed programming language proposed by *JetBrains*, that mainly targets JVM , Android, JavaScript and Native platforms. The language is *blended* with a solid object oriented base and powerful functional traits. Mixing the two programming paradigms is considered the norm. Kotlin is inspired by other popular languages (Java, C#, JavaScript, Scala and Groovy), trying to inherit and mix the best characteristics of all of them.

Among the many others, one of the most interesting features of Kotlin is multi-platform support. The main goal is being able to share common code between multiple platforms, so that it has to be only written once and maintained in a single place. Currently supported platforms include: JVM, Android, JavaScript, iOS, Linux, Windows, Mac, and WebAssembly. This trait allows for valuable business interest due to the possibility of high code reuse and maintainability.

Kotlin multi-platform projects have different code source sets: one for each targeted platform, and one *common* source set that is shared between all of them. A visual representation of this architecture is represented in Figure 2.3.

The common source set is suitable to encode the business logic of an application, whereas each platform-specific codebase is responsible for more peculiar implementations required by their target system. This architecture is supported through the *expected/actual mechanism*. With that, common code declares to *expect* some functionality to be implemented externally, and seamlessly make use of them in the business logic by taking for granted that a real implementation will be provided. Then, platform-specific code fills the gaps by providing *actual* implementations of those functionalities. Usage of interfaces, information hiding, and other solid engineering principles is definitely required for a good code scalability. This design is resilient and suitable for an heterogeneous spectrum of usage, spacing from full-stack web applications to mobile development.

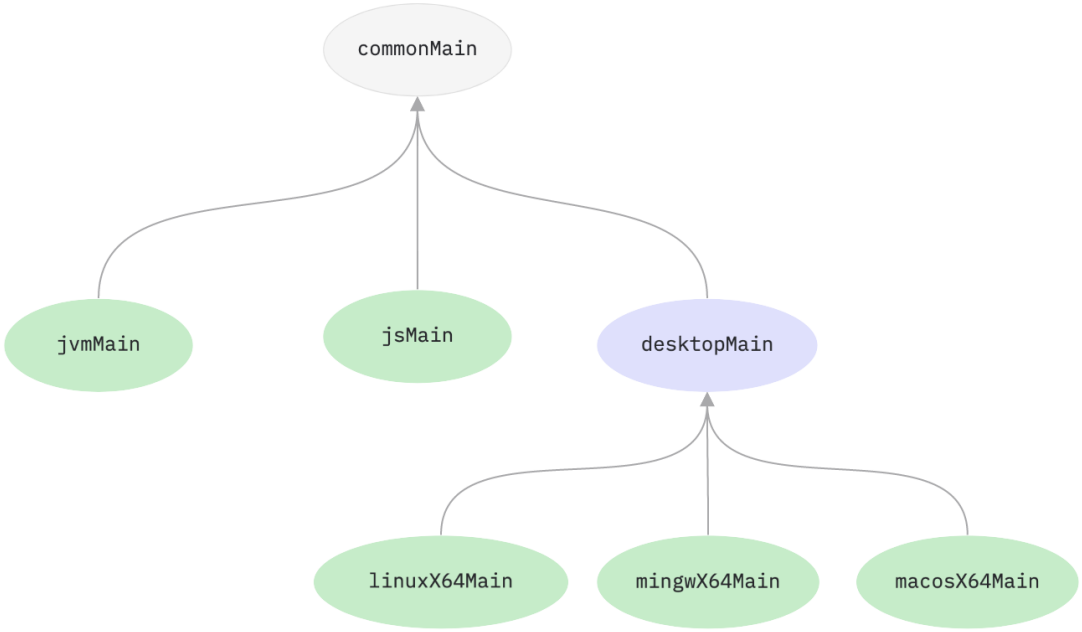


Figure 2.3: Structure of Kotlin Multi-Platform Projects. Source: `https://kotlinlang.org/docs/mpp-intro.html`

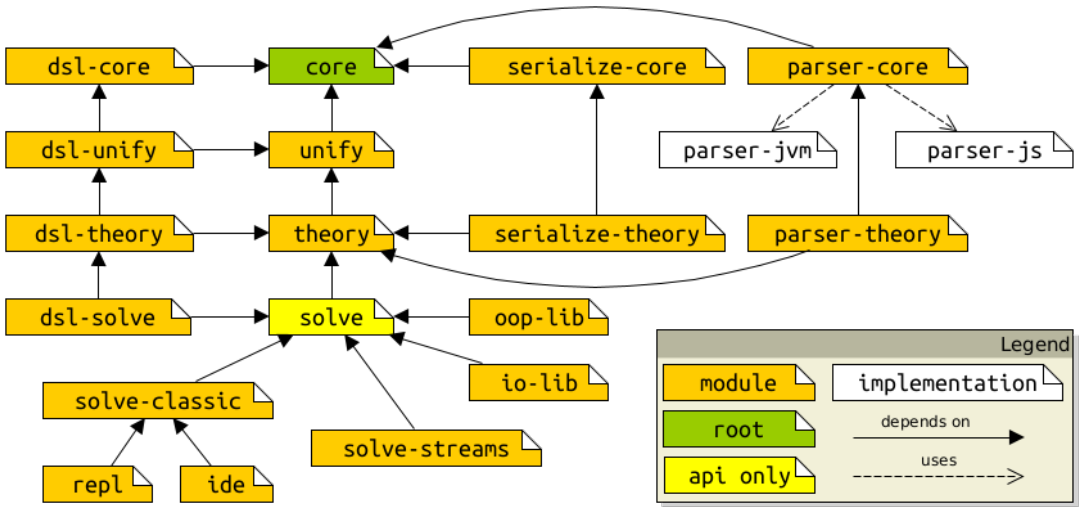


Figure 2.4: Overview of *2P-Kt*: Architectural Modules and Their Dependencies

2.3.3 The 2P-Kt Project

2p-kt is the natural evolution and modernisation of *tuProlog*, that aims to accomplish a broader view of becoming a multi-platform ecosystem for symbolic Artificial Intelligence. The Prolog framework has totally been rebooted and re-engineered as a multi-platform codebase written in Kotlin. In present day, 2p-kt fully supports JVM-based and JavaScript-based targets, and all core features of tuProlog have been re-implemented. The is subject of a fast-paced development, and keeps the doors open for novel extensions.

The project is split in several modules, each one providing a single self-contained feature, enforcing the *Single Responsibility Principle* (SRP) over the whole architecture. Each module represents a core functionality of the Prolog engine, that can even be used singularly, and defines a lean dependency graph. Modules also represent the unit of change for potential extensions, like the one we propose in this thesis. The whole architecture is shown in the diagram of Figure 2.4. We proceed by providing a brief description for each module.

Core — It is the basic code base on which most other modules rely on. It contains

model classes and it provides the basics for symbolic manipulation.

Unify — It contains model classes to support *unification*, and it provides the basic unification mechanism.

Theory —It provides the concept of Prolog *theory*, namely a set of logic clauses. Manipulation constructs are provided, and optimized data structures implement fast indexed clause retrieval.

Solver — It models basic interfaces and mechanisms for Prolog query resolution and inference. Solvers and their queries are configurable for easy extensibility. As such, other resolution or computation mechanics (such as the one of PLP), can somehow depend on interfaces of this model.

DSL-Core, DSL-Unify, and DSL-Theory — These modules define Domain Specific Language (DSL) powered by Kotlin native features in the field. The DSL provided are aimed to make it seamless for developers to define Prolog structures, use unification, or manipulate theories, all within the imperative Kotlin world.

Parser-Core and Parser-Theory — Parser modules implement the mechanic of the lexers that help reading core entities and theories, along with their formal grammars.

Serialize-Core and Serialize-Theory — These packages offer a way to serialize core entities and theories, namely translating them in a format that can be easily stored or externally transferred.

Solve-Classic — It provides implementations for solve module. It contains an enhanced implementation of tuProlog inferential core. The engine is re-engineered in a novel Finite State Machine design, that improves performance and fully leverages data immutability.

Solve-Streams — This is an alternative implementation of entities from the Solve module. It contains an experimental inferential core implementation, based on the concept of minimizing the FSM of tuProlog and enhancing it by stressing the usage of the functional paradigm.

IO-Lib — This library defines Prolog primitives that enable working with input-output streams and channels from the logic programming world.

OOP-Lib — This library enables multi-paradigm programming from the logic programming world, namely using object oriented entities while solving Prolog queries. This is possible through the implementation of some ad-hoc Prolog primitives that make the process totally transparent to developers.

Repl and IDE — Repl and IDE are two distributions of 2p-kt that bundle the engine, through a command line interface and in a graphical JavaFX desktop application respectively.

2.3.4 Multi-Paradigm Mechanic

2p-kt proposes a multi-paradigm programming model that enables a bidirectional connection between the two logic-based and object-oriented programming worlds. The core of this mechanic is defined inside the *solve* module and relies on the two concepts of *solver* and *library*.

Solvers are abstract entities capable of solving some logic query according to some logic, implementing one or more inference rule, via some resolution strategy. In other words, solvers act as a facade for some underlying implementation of a logic-based inference engine. Note, no particular logic paradigm is enforced at this level. In fact, solver APIs are open to many symbolic inference archetypes and developers are encouraged to experiment approaches that go beyond traditional logic programming. Those APIs are defined in the `Solver` Kotlin interface. More practically, solvers are the entrypoint through which developers can execute logic-based inference from the object-oriented world.

Listing 2.2 shows the signature of the `solve` method, which is the fundamental operation in a Solver. In the method, *goal* represents the logic query submitted to the underlying inference engine, and *option* is the hinge of customizability that regulates the behavior of solvers during resolutions. For instance, options can define a timeout that stops the solver after a maximum time interval and can impose a limit to the number of solutions to be found. Additionally, a key-value

Listing 2.2: Kotlin - Signature of the `solve` Method in *2P-Kt*

```kotlin
fun solve(
    goal: Struct,
    options: SolveOptions
): Sequence<Solution>
```

storage is provided for specifying custom options that are not available at default. Finally, note that solutions are returned through the *Sequence* construct of Kotlin, which leaves space for lazy resolution strategies where applicable.

However, solvers only encapsulate the business logic for symbolic inference, and do not include any knowledge base by default. Instead, solvers can be customized through the usage of *libraries*. A library is a collection of logic clauses and operators that define a specific functionality or capability to be included in a solver and that is applied during resolutions. For example, built-ins for Prolog ISO standards are implemented and bundled as a library. Note, components of a library are implemented as objects, thus allowing a first connection to the object-oriented world from the logic-programming realm. Not only this makes solvers and libraries fairly easy to test and debug, but it also enables the injection of imperative business logic inside symbolic resolutions. Accordingly, *Primitives* are special logic clauses that are solved during resolutions by invoking a method of a Kotlin object. Most Prolog built-ins, such as `findall` or `bagof`, are implemented as primitives. For clarity, Listing 2.3 shows how the `nl` primitive, that simply prints a line break on the standard output, is defined as an object in the *classic* solver of 2p-kt.

The *Object-Oriented Library* (OOP-Library) is another example of library that bundles primitives for object manipulation to be used in logic programs, thus solidifying the multi-paradigm proposal of the 2p-kt framework. The built-ins include primitives for creating objects, invoking methods, and casting primitive types with full transparency. In this way, open-minded developers can conceive algorithms that both leverage imperative programming and logic-based resolution to achieve optimal results.

Listing 2.3: Kotlin - Implementation of the `nl` Primitive in *2P-Kt*

```kotlin
object NewLine: PredicateWithoutArguments
        .NonBacktrackable<ExecutionContext>("nl") {
    override fun Solve
        .Request<ExecutionContext>
        .computeOne(): Solve.Response {
            return context.outputChannels.current.let {
                if (it == null) {
                    replyFail()
                } else {
                    it.write("\n")
                    replySuccess()
                }
            }
    }
}
```

In this dissertation, *solvers* and *libraries* (and their combination) represent the mechanism of our choice to implement probabilistic reasoning.

Chapter 3

Analysis

In this chapter we document and motivate the high level choices, the requirements and the constraints, that define and shape of our project. In Section 3.1 we debate the requirements of our Probabilistic Logic Programming proposition. Section 3.2 motivates the usage of *2P-Kt* in our work, and how our contribution coherently fits in the project. Finally, in Section 3.3 we discuss our needs of developing a new package for Binary Decision Diagrams and the requirements we imposed to it.

3.1 Probabilistic Logic Programming Engine

In this section we disclose the requirements that define the main properties and characteristics of our Probabilistic Logic Programming proposition. Most of our requirements are *non-functional*, and indicate high-level architectural principles. Generally, our concern is to go beyond some of the limitations of current state of the art proposals.

Lightweightness — Our probabilistic reasoning engine must have a minor footprint in terms of memory, CPU, and code size, namely attempting to use only resources that are strictly necessary. We assume that our solution could be executed in constrained environments or in devices with minimal computation capabilities. At the same time, we don't want to add unnecessary

heaviness on other projects that include our software as a dependency.

Minimality of dependencies — We intend to include quality open-source dependencies in our project wherever necessary, by also paying attention to avoid overly large imports that could threat the lightweightness requirement. This is a common issue in software development, where only a small fraction of the functionalities offered by the imported dependencies is effectively used. In those scenarios, we consider developing simple home-made implementations where it does not imply an excessive "reinvention of the wheel".

Portability — Our proposal must be usable in many environments and adaptable to multiple platforms. We intend to go beyond the platform constraints present in some state of the art proposals, and increase the accessibility of PLP inference.

Simplicity — Although the inner workings of a probabilistic-logic engine are not trivial by nature, we aim to provide a well-documented and understandable codebase. Our solution must be simple to use and have a minimal learning threshold.

Backwards compatibility with Prolog — Considering the background provided in Chapter 2, we observe that there are many similarities between traditional LP and PLP query resolution. Without considering the probability computation mechanic, LP and PLP only differ for the policy on which solutions are grouped or selected, and so they can share a common resolution strategy. In fact, probabilistic solvers produce solutions that are valid for traditional logic queries by just discarding their probability. However, it would be improper to use a probabilistic engine with the purpose of solving LP problems, such as Prolog queries. Indeed, probabilistic computation is fairly more expensive, and it produces solutions outputs that differ in grouping, order, and count. We aim to provide a unified solution that is capable of solving both LP and PLP problems and makes the transition between those two modes frictionless. We want to enable or disable probabilistic computation by simply toggling the engine configuration, to use PLP

theories for LP query resolution and viceversa, and to not compromise the native optimizations of each methods.

Flexibility — Our solution must not constrain developers to a specific use case. Instead, we aim for our solution to be open to updates, domain-specific optimizations, or tasks that go beyond simple probabilistic inference. Moreover, we envision our system to be seamlessly integrated with other technologies in the AI field. For instance, probabilistic reasoning well fits problems that deal with uncertainty such as Machine Learning, Deep Learning, and Data Mining, opening new horizons for adding *expert knowledge* to improve precision and control of automatic predictors.

3.2 Use of 2P-Kt

This section motivates the selection of *2P-Kt* as the platform of choice for our Probabilistic Logic Programming proposal towards an optimal satisfaction of the requirements listed in Section 3.1.

Having the goal of developing a PLP inference engine, we must choose between building a new solution from scratch or utilizing some existing infrastructure to apply our extension on. Undoubtedly, we prefer the second option. In this context, we acknowledge that *2P-Kt* is an optimal fit for our needs. The project consists of a logic-based ecosystem for symbolic AI, designed and implemented by taking openness, modularity, extensibility, and interoperability into account. Indeed, requirements such as *lightweightness* and *portability* are natively matched by the design principles of *2P-Kt*. Moreover, the *micro-module* oriented architecture of the project allows a minimization of external dependencies and makes it frictionless to create new extensions. Also, the fastly growing documentation of the project enables a favorable learning curve. Furthermore, multi-platform support is provided out of the box by relying on a Kotlin cross-platform setup.

Accordingly, choosing a base platform for our project also introduces new requirements and constraints. In the case of *2p-Kt*, this entails:

- Breaking down our solution into multiple micro-modules and be compliant with the internal dependencies of *2p-Kt*.
- Following the code of conduct of the project and matching the code style guidelines suggested by the authors to individual contributors.
- Avoiding tampering the existing codebase if not strictly necessary, by iteratively discussing any new change with the authors.
- Maximising the re-usage of existing modules and constructs in order to avoid code repetitions and overlappings.
- Respecting the provided *object-oriented* interfaces to create our probabilistic logic solver.

On the other hand, we also find some additional desirable possibilities:

- We can leverage the existing Prolog engine to perform reasoning tasks related to logic resolution. Then, by assuming that a valid logic solver is provided out of the box, we would only need to strategically pilot the resolution process and then perform the probability computation.
- Backwards compatibility with traditional Prolog can be accomplished by strategically delegating some work to the Prolog solvers of *2P-Kt*. The complexity of switching from a probabilistic solver to a simple logic one, and viceversa, can be hidden behind a configurable interface.
- The rich collection of unit tests asserting Prolog ISO standards can be included and reused in our test suite.

Finally, we advocate that our extension is *coherent* from the standpoint of *2P-Kt*'s ambitions. Although such a platform is designed as an ecosystem open to multiple kids of logic, only Prolog is supported so far. Given the lack of differentiated logic systems, our proposition of the probabilistic paradigm fits the growth intentions of the project and influences the future of its development.

3.3 Binary Decision Diagrams Library

This section discusses the requirements of Binary Decision Diagrams package we develop for addressing the *Knowledge Compilation* problem, which is necessary for implementing probabilistic logic engines as discussed in Section 2.1.4.

Knowledge Compilation in Probabilistic Logic Programming reasoning can be addressed through the usage of a variety of data structures, of which strengths and weaknesses have been richly analyzed in the literature. Among all the possibilities, we believe that *Binary Decision Diagrams* are the most suitable solution for our needs. First, BDDs are a well acknowledged choice in the field, so we have plenty of well documented research attempts to learn from and to use as a reference. Second, all the Boolean operations required for probabilistic computation are reproducible through the `apply` algorithm with fairly acceptable time and space complexities. Third, we can rely on a wide range of open source software packages implementing BDDs and their algorithms, that we can either use or learn from. Lastly, the data structure is relatively simple and the algorithms involved are generally understandable.

However, we find that no existing open source BDD package that we inspected is capable of fitting our use case. In fact, most packages rely on low-level implementations written in *C* or *C++* language in order to achieve good performance. Accordingly, many other packages are implemented on top of these low-level bindings as wrappers to support other programming languages. As such, we recognize these packages as defective value propositions for our project. First, implementations of these packages are scattered across many code repositories and there is a lack of a unified, coherent, multi-language solution. Second, most packages are not accessible through some of the most notorious public dependency registries, such as *NPM* or *Maven Central*. Third, the reliance on low-level bindings often limits the portability of the packages and defeats our multi-platform ambitions. Forth, most solutions usually implement more data structures or algorithms than the ones we concretely need. Lastly, we acknowledged a lack of good *object-oriented* proposals, whereas developers are constrained to using low-level constructs that limit the spectrum of domain-specific optimizations.

Accordingly, we advocate that developing a Binary Decision Diagram library from scratch is our preferable option. Our goal is to provide a lightweight, flexible, portable and simple *Kotlin-native* package. By writing our solution in Kotlin, we open novel horizons for multi-platform availability of BDDs, and we provide a set of open object-oriented constructs that can be adapted to many use cases. Nonetheless, such a code package would kindly fit in *2P-Kt* as a micro-module. We start by implementing only the algorithms we need without deeply diving into performance optimizations. However, we envision our solution to be open to other use cases and, potentially, to furtherly grow in the open source community. We impose the following requirements for our Binary Decision Diagrams package:

Lightweightness — Our solution should be minimal in terms of code size and resource usage. Considering that we develop the software package from scratch, we also plan to minimize external dependencies, if not completely avoiding them where not necessary. This principles would lead to a consistently lightweight solution that is easy to integrate in many workflow and platforms.

Object-Oriented Design — In contrast with most existing solutions in the field, our proposal must follow a well engineered object-oriented design. We want to benefits from the full potential of the design patterns and design principles that are at the state of the art of object-oriented systems. This implies factorizing the codebase in interfaces, micro-packages, and using the *factory* pattern. We plan to provide a rich set of abstractions so that custom operations, optimizations, and data representations can be integrated with ease.

Kotlin-Nativeness — The proposed software package must be written in *pure-Kotlin* and must fit in Kotlin multi-platform projects. This requirement provide some relevant advantages. First, the proposal would be one of the first of its kind, as we recognized a lack of consistent multi-language and multi-platform solutions in the field. Second, the Kotlin language brings the benefits of many state or the art object-oriented and functional programming constructs. Third, the architecture of Kotlin multi-platform projects allows for integrating platform-specific code, so that we can define the basic

abstractions and all the algorithms in a shared codebase and then leverage all native capabilities of each platform to optimize critical code parts.

Expressiveness — Our solution must be expressive, in the sense of not limiting developers in the adoption high-level constructs and operations. For instance, most proposals in the field force the representation of Binary Decision Diagram nodes through numeric labels, and the Boolean variables ordering is defined as the natural integer order of those labels. Instead, we advocate the representation of BDD nodes as generic *objects*, thus elevating their informative value and enabling the development of domain-specific operations and optimizations.

2P-Kt Integrability — Coherently with our PLP proposition, we plan to fit our solution in the *2P-Kt* project as a new micro-module. As such, this furtherly enriches *2P-Kt* by offering a data structure that can be used at will in logic workflows, such as we do in our probabilistic logic engine. Also, the newly introduced module can grow to host new data structures and algorithms. At the same time, we envision the BDD module to be standalone, namely not depending on other modules in *2P-Kt*, so that it can be used in external projects without the burden of including unwanted logic-related code.

Chapter 4

Design

In this chapter we examine the design and the architecture of our project in detail. In Section 4.1 we provide an overview of the architecture of our proposal, showing how it fits in the modular structure of *2P-Kt*. In Section 4.2 we present the design of the module for Binary Decision Diagrams. In Section 4.3 we explore the core traits of the abstract module for PLP support. Finally, in Section 4.4 we discuss the design of the module for our ProbLog inference engine.

4.1 Architectural Design

As we motivated in Section 3.2, one of the requirements is adopting *2P-Kt* as a base on which constructing our project. Accordingly, our project is compliant to the architecture of *2P-Kt* and is divided into multiple self-contained micro-modules, each one representing a contribution of our proposal. More specifically, we designed our solution to be divided in three modules. In Figure 4.1 we show how our projects fits in the *2P-Kt* architecture and dependency graph: the white icons represent the newly introduced modules, while the darker ones indicate modules already existing in *2P-Kt*. Each arrow indicates a directed dependency from one module to another.

The `bdd` module represents our proposal for the Binary Decision Diagram ma-

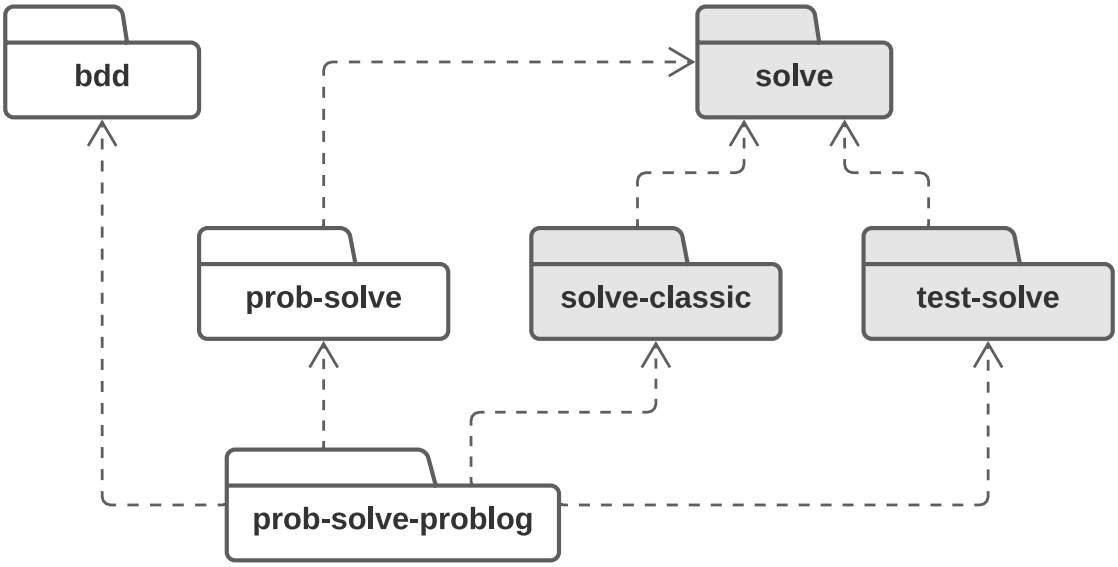


Figure 4.1: Modules and Architectural Dependencies of the Project

nipulation library. This module is purely self-contained, in the sense of not having any dependency over other modules of *2P-Kt*. This aspect is crucial for our value propositions, as we intend to promote the usage of the library as a lean external dependency on other projects as well.

The `prob-solve` module is meant to bundle all the entities and traits that are common to any potential implementation of solvers for the Probabilistic Logic Programming paradigm. This module is purely abstract, and only provides APIs or interfaces on which multiple PLP solver implementations can rely on. Considering the nature of PLP, this module depends on the `solve` module of *2P-Kt*, which provides basic abstractions for logic solvers. In other words, we model probabilistic-logic solvers as a direct subset of logic solvers.

The `prob-solve-problog` module contains the actual implementation of our proposed PLP solver supporting the ProbLog language. Coherently, it depends onto the abstractions of `prob-solve` and is compliant to them. The other fundamental dependency is the `bdd` module, which is used for manipulating Binary Decision Diagrams during probabilistic logic goal resolution. Additionally, it also

depends on `solve-classic` and `test-solve` modules of *2P-Kt*. Detailed motivations of these last two dependencies are provided in Section 4.4 and Chapter 6.

Given that our project adheres to the architecture *2P-Kt*, our modules are inherently managed with *Gradle*. As such, minimizing the number of dependencies of each module has been a priority in our architecture. We aim to enable other developers to use parts of our project without depending on any unused or superfluous code. Moreover, our micro-modules are intrinsically published and available on some the most notorious source repositories, such as *NPM*, *Maven Central*, and *Bintray*.

4.2 BDD Module

This section analyzes our design choices for the `bdd` module, which represents our proposal for the novel Binary Decision Diagrams manipulation library. In Section 4.2.1 we present the internal architecture of our library, presenting all its core components. In Section 4.2.2 we expose all the operations and manipulation tasks that we support in our proposal for the first release. Finally, in Section 4.2.3 we discuss how we attain optimal flexibility over platform-specific optimization and custom data representations.

4.2.1 Internal Architecture

As discussed in Section 3.3, we recognize the absence of a public, unified, coherent, portable solution for manipulating and representing Binary Decision Diagrams. As such, we propose a library that attempts to fill these gaps without sacrificing performance optimizations. We prioritize usability, extensibility, and portability over multiple languages and platforms. Principles of *information hiding* are widely applied over the architecture of the whole module. The core entities are all defined by abstract components that are accessible and manipulable from the external world. Every detail regarding representation strategies, algorithms implementation, and multi-platform support and optimizations, is segregated behind the abstract inter-

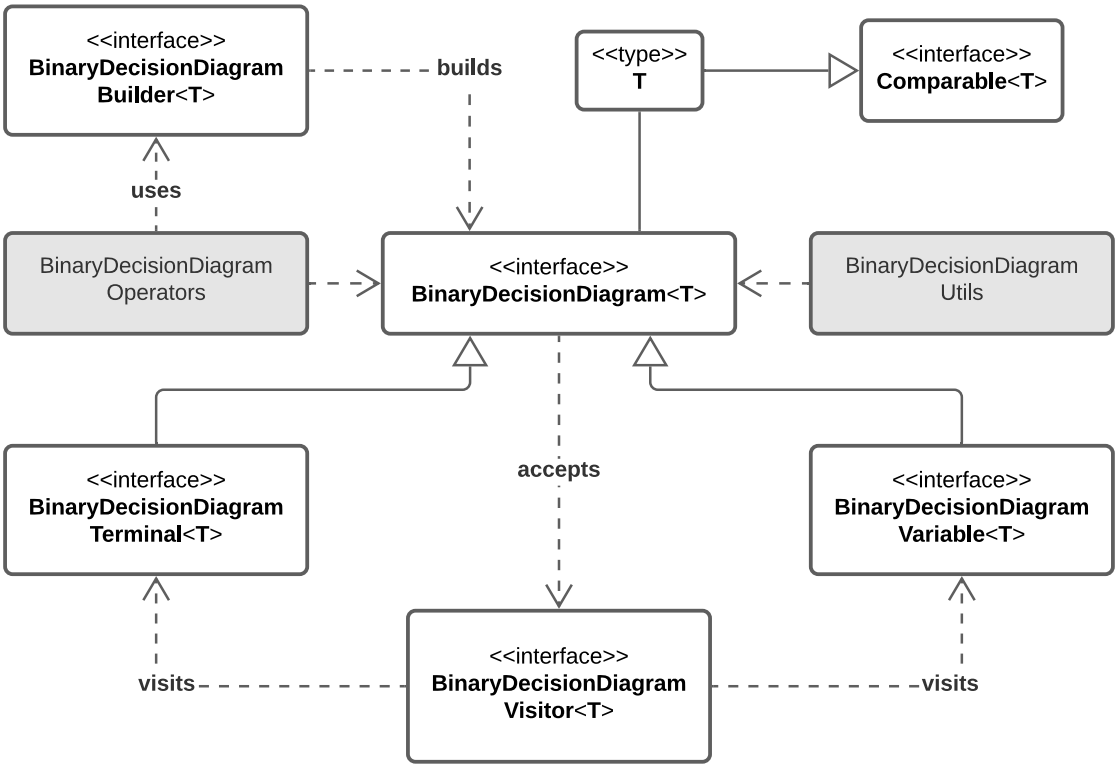


Figure 4.2: Binary Decision Diagram Module: Internal Architecture

faces of the module. Then, the *factory* pattern is extensively used to access specific implementations of each core abstraction. The diagram shown in Figure 4.2 describes the high level architecture of the module. To shape the set of core entities of the library, we took inspiration from some of the existing state of the art proposals. The core of our module is defined by a minimal set of components, that can satisfy a variety of use cases.

`BinaryDecisionDiagram` — This interface abstractly represents any Binary Decision Diagram. The generic type `T` dictates the value representing each `Variable` node inside the diagram. Since the representation strategy for variable nodes does not affect the manipulation business logic of BDDs, developers can leverage the generic type for an optimal *object-oriented* design. The only constraint over the generic type `T` is that it has to be *comparable*

to others of its kind. Moreover, the *visitor* pattern is used to flexibly explore the structure of Binary Decision Diagrams.

`BinaryDecisionDiagramVariable` — This interface represents variable nodes inside Binary Decision Diagrams. The entity is composed of three attributes:

`value` — The actual value representing the Boolean variable. By being represented with the generic type `T`, developers can represent Boolean variables as they prefer.

`low` — A reference to the *low* sub-diagram.

`high` — A reference to the *high* sub-diagram.

Given the presence of *low* and *high*, this interface indicates a *recursive* data structure. However, there is no constraint over how the referenced sub-BDDs should be retrieved or accessed. The variable ordering is determined by the *comparable* nature of `value`. In this way, developers are not constrained on some specific variable ordering method either, whereas most other proposals rigidly solve this task by using numerical identifiers.

`BinaryDecisionDiagramTerminal` — This interface represents terminal nodes inside Binary Decision Diagrams. The entity is composed by just one Boolean value, that represents the *truth* state of the terminal.

`BinaryDecisionDiagramVisitor` — This interface enables the *visitor* pattern over the recursive structure of Binary Decision Diagrams. It provides a type-safe and and fast way to explore the structure of diagrams, by also hiding their representation strategy.

`BinaryDecisionDiagramBuilder` — This interface represents abstract builders that are responsible for creating Binary Decision Diagrams.

`BinaryDecisionDiagramOperators` — This is a collection of all the operations for Binary Decision Diagrams supported by the library.

`BinaryDecisionDiagramUtils` — This is a collection of helper and utility functions that support the manipulation of Binary Decision Diagrams.

4.2.2 Supported Operations

With the term *operation*, we imply algorithms and operators aimed to the synthesis of new diagrams starting from one or more Binary Decision Diagram operands. Indeed, unary and binary Boolean operations on BDDs fall in this category. The proposed architecture is open to support any kind of operator. In the first version of the library, we commit to provide an implementation of the `apply` algorithm in both its unary and binary versions. We also support the *and*, *or*, and *not* binary operations, which can easily be attained by relying on the `apply` algorithm.

Besides construction operators, there are a variety of tasks that can be executed over Binary Decision Diagrams. These all fall in the collection that we defined *utils*. In its first proposal, our library supports the following set of operations:

`any` — This is the task of finding at least one node inside a BDD that satisfies a certain Boolean predicate. The simplest of those is the *existential* predicate, which aims to be true if the diagram contains at least one node. This can flexibly host more sophisticated logic that asserts the inner properties of diagram nodes.

`expansion` — This task is a generalization of the *Shannon Expansion* that we discussed in Section 2.2.2. We model this operation as a recursive reduction that computes a result value by evaluating each node inside a BDD in bottom-up order. The result is defined with a *generic* type as well. This operation opens a wide spectrum of possibilities. For instance, a possible simple application could be counting the number of nodes inside a BDD. Another example could also be creating a deep clone of the diagram itself, by using a *builder* to construct the result during the recursive reduction.

`map` — This task creates a deep clone of a Binary Decision Diagram by applying a certain transformation function over each of its nodes. Note, this operation can easily be defined as a sub-case of the `expansion` operator.

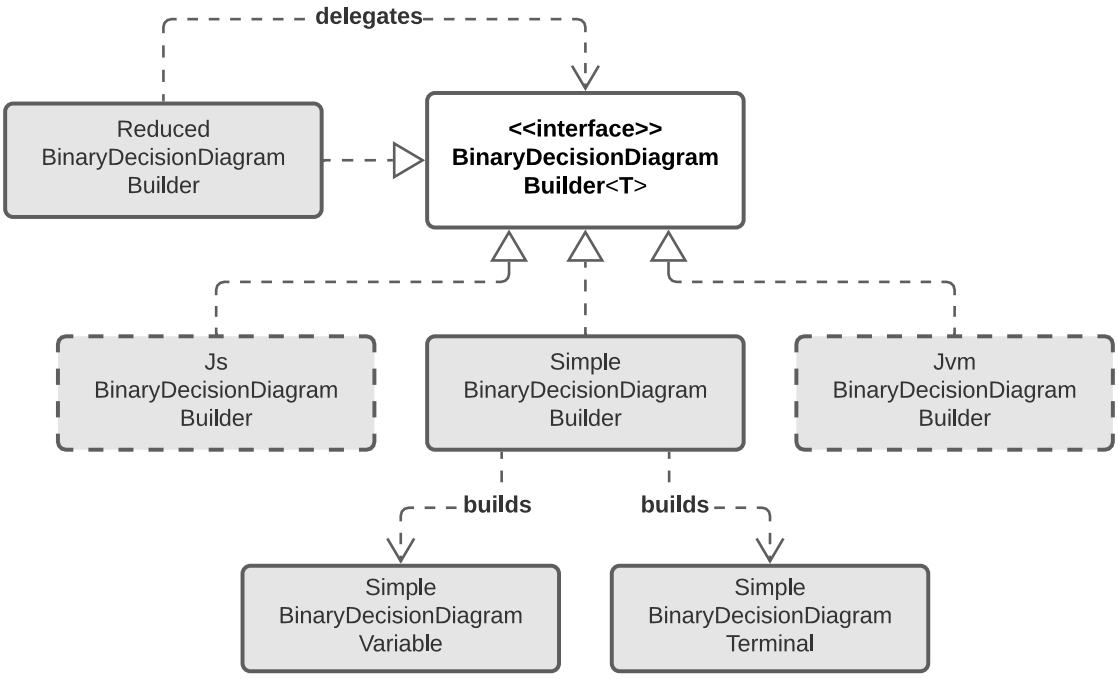


Figure 4.3: Binary Decision Diagram Builder Hierarchy

4.2.3 Optimizations and Multi-Platform Support

Moreover, the proposed design is strongly open to multi-platform support and domain-specific optimizations through the `BinaryDecisionDiagramBuilder` abstraction. In fact, the majority of library components do not take in consideration the way BDDs are created and represented, because *builders* are the central unique responsible of that task. As such, the library is mostly platform-agnostic by nature. In the perspective of Kotlin multi-platform projects, most of the code would reside in the *common* shared codebase. Then, each target platform can provide an optimized implementation of *builder* through the *expect/actual* mechanism, in order to satisfy specific constraints imposed by the application domain or the execution environment. For instance, certain platforms could benefits from array-based representations of BDDs to better leveraging memory locality, whereas other systems could be achieve better performance by manipulating dynamic pointers or by swapping memory with some persistent storage. As such, our library can easily be extended to host customized optimizations and representation strategies. Coherently, builders are injected inside each operation and construction algorithm so that their business logic can ignore the details of node construction. In our proposal, we commit to provide a simple initial representation strategy that makes BDD manipulation feasible across all the supported platforms, with some basic optimization to reach acceptable performance. An high level overview of hierarchy related to the *builder* abstraction is provided in Figure 4.3. In that, the darker components indicate the concrete implementations of *builder* that we include in our design. The ones delimited by a dashed line represent hypothetical implementations specific to one or more platforms.

Additionally, the *builder* abstraction enables the implementation of the `reduce` algorithm as well. This can be simply achieved through the *decorator* pattern. The idea would be to delegate the actual construction of BDDs to another instance of *builder*, and then leveraging some table-based approach to memoize the results in memory and then apply the node reduction criteria.

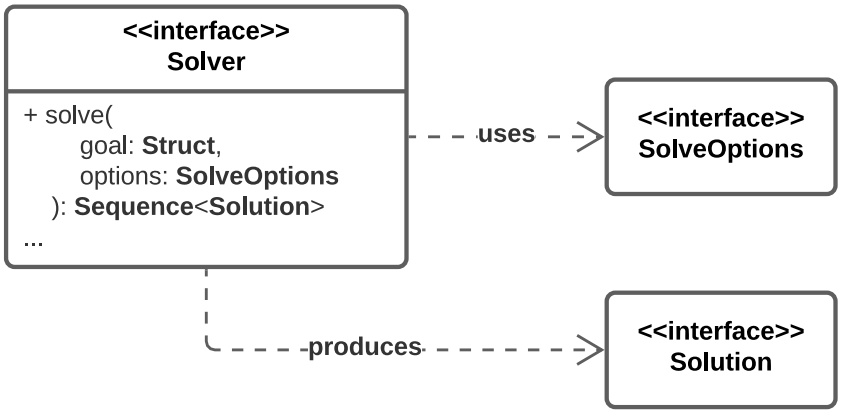


Figure 4.4: Design of Solvers in *2P-Kt*

4.3 Prob-Solve Module

The purpose of the `prob-solve` module is to provide a common base that could be shared with any potential implementation of PLP inference solver. For reference, the `solve` package of *2P-Kt* has an analogous responsibility towards simple LP solvers. As such, this module is meant to contain only interfaces and base types to be used in other modules. In Section 4.3.1 we show the constraints and guidelines imposed by the components on which this module depends on. Then, in Section 4.3.2 we discuss how we shape our extensions to fit such criteria.

4.3.1 Overview of Dependencies

As hinted in Section 4.1, we model probabilistic solvers to be a subset of logic solvers, at least for the scope of inference tasks. As such, we design PLP inference engines to be compliant with the core components that reside in the `solve` module on *2P-Kt*. Among the numerous entities provided inside that module, we are exclusively interested in the `Solver` interface and its semantics. This entity is responsible for defining the core traits of logic-based solvers. In Figure 4.4 we briefly illustrate how `Solver`, and its main dependencies are designed. The `solve` method, along with its variants, takes a logic query and a given set of options as input, and produces a sequence of solutions. Then, `SolveOptions` is a container

for a set of options that can influence the behaviour of a solver. Examples of options include limiting the number of solutions, implementing lazy resolution, and imposing a timeout to bound long executions. Finally, `Solution` abstractly represents the solution for a given query and its grounding. These three entities are the main ones on which Prolog solvers implemented in *2P-Kt* rely on.

Assuming that the probabilistic logic solvers shaped by our proposition have to fit the criteria of the `Solver` object hierarchy, we must decide a way to strategically extend the entities involved.

4.3.2 Design of the Extension

Given the context described in Section 4.3.1, we proceed by detailing *how* we plan to insert the traits for probabilistic resolution inside the components offered by *2P-Kt*.

First, we recognize that the only additional information that differentiates a regular logic solution from a probabilistic one is the notion of *probability*. Namely, we can potentially accomplish our purpose by simply adding a new field inside the `Solution` interface to represent the probability of a solution as a floating point number. However, it would be inconvenient to tamper the `Solution` interface because it is deeply involved with parts of the *2P-Kt* project. Instead, we propose to add an *extension* for such interface, that would be visible only in the scope of probabilistic solvers.

Then, assuming that we have a reliable way to distinguish probabilistic solution from regular logic ones, we also have to design a way to instruct a `Solver` whether to perform probabilistic resolution or not. Reminding the points of Section 3.1, we imposed as a requirement that probabilistic solver must be versatile enough to perform both probabilistic and regular logic resolution. As such, we advocate that modifying or extending the `Solver` extension would not be an optimal design to attain our goal. In that case, clients would face the ambiguity of using two different methods or interface to perform the different kinds of resolutions, which is inconvenient from the standpoint of both the existing codebase and its future

additions. Given this observation, we realize that the optimal solution would be to intervene on the `SolveOptions` entity instead. Considering these choices, the resulting workflow would be fairly simple. We briefly describe it in the following steps:

1. A specific instance of `Solver` would be obtained by a factory method.
2. The client customizes the options inside `SolveOptions` as preferred. Eventually, the *probabilistic* option can be toggled to specify whether probabilistic resolution is requested or not.
3. The customized instance of `SolveOptions` is provided to the instance of `Solver` along with the query goal.
4. Query is resolved. If requested, the solver will take care of performing probabilistic computation. If such a feature is not supported by the chosen solver instance, then falling back to regular logic resolution is consented by stubbing the probability result with a default value. This makes probabilistic resolution an *optional* and *best-effort* feature.
5. Instances of `Solution` will be produced in sequence, each one eventually carrying out the computed probability.

Note, the described process is totally transparent to all the clients that already rely on the semantics of `Solver` and his hierarchy. Moreover, this allows probabilistic solvers to be used as regular logic solvers with ease, by just unselecting probabilistic resolution in the `SolveOptions` configuration. Additionally, what described is not bound to any specific PLP language or resolution strategy.

Additionally, the manipulation of `Solution` and `SolveOptions` can be leveraged to attain an arbitrary number of additional features. In our proposal, we want solvers to produce a representation of the underlying data structure that determined the probability result. In the case of this project, the data structure would be a Binary Decision Diagram. We envision this feature to be optional as well, so that clients can request it at will depending on their use cases. For instance, one may want to visualize in some GUI a graphical representation of the

Binary Decision Diagram that is behind a given probabilistic solution, to better inspect and motivate its meaning.

Summarizing, we want `Solver`s to be extended in order to produce solutions enriched with the notion of *probability*. Disabling the probabilistic option must make the solver fall back to regular logic resolution. From the outside, `Solver`s can be used as usual without needing to be aware of the probabilistic extension.

4.4 Prob-Solve-Problog Module

In this section we present design and architecture of our Probabilistic Logic Programming solver for inference tasks, which represents the most remarkable contribution of this dissertation. In the previous chapters we occasionally leaked fragments of information regarding our PLP proposal, so in this section we proceed by providing a vivid and complete description of all the design choices involved. Accordingly, the remainder of this section is structured as follows. In Section 4.4.1 we provide a summary of the PLP features that we designed our engine to support. In Section 4.4.2 we discuss the crucial points of the resolution mechanics by also providing supporting examples. In Section 4.4.3 we analyze the main details behind the design of our PLP solution. Finally, in Section 4.4.4 we summarize the accomplishments achieved with our design.

4.4.1 Core Traits and Supported Features

As discussed in Section 2.1, there is a variety of research contributions that explored the field of Probabilistic Logic Programming. In the literature, proposals often differ for language syntax, features, semantics, and supported reasoning tasks. As such, contributing in this context demands for clarity in terms of *where* to fit in the novelty spectrum.

In our proposal, we attempt to further explore the directions of *usability* and *portability*, which have been a secondary concern in the other proposals we inspected. An additional value proposition is also represented by the enablement

and usage multi-paradigm programming. The latter opens the horizons for better reasoning optimizations, and favors portability by making it easier to integrate other technologies distant from the LP realm. From the standpoint of the supported features, we attempt to match a subset of the ones available at the state of the art. Our plan is to provide an usable and functioning PLP code base, initially supporting only the fundamental features, and aiming to be flexible for future growth. The core features and traits of our proposal can be summarized in the following points.

ProbLog syntax — Among the various languages and syntaxes perceptible in the literature, we opt for supporting ProbLog. We appreciate the simplicity of the language and the high compatibility with traditional Prolog. Considering the instruments provided in *2P-Kt*, supporting the probability operator on facts and clauses is a simple task. We do not aim to accomplish a faithful reproduction of the language, but we are interested to leverage it for its simplicity instead.

Compatibility with Prolog — As mentioned in Section 3.1, we want our PLP engine to accept both ProbLog and Prolog Knowledge bases interchangeably. That way, we are able to use libraries and legacy code written in Prolog to perform probabilistic reasoning.

Use of Binary Decision Diagrams — To solve the problem of Knowledge Compilation, we select Binary Decision Diagrams as our data structure of reference. This is also the choice of many state of the art PLP proposals, not to mention that BDD packages are considerably more evolved than the other alternatives. As such, we have a rich set of examples on which inspire our solution. Additionally, by developing a library for Binary Decision Diagrams of our own, we can accomplish a deeper control of the data structure and apply optimizations customized to our use case.

Probabilistic clauses — Traditionally, ProbLog supports probability distribution only defined over logic facts. Distributing probability over clauses can be accomplished by reformatting them and by strategically adding probabilistic

facts to the given theory. However, this transformation is often costly in terms of memory utilization and computation, as the logic resolution would face additional steps and depth. As such, most ProbLog proposals support probabilistic clauses natively, allowing annotating probability distributions on clauses similarly to how it is done on facts. Accordingly, we decide to support this trait in our PLP solution as well.

Probability with evidence — As discussed in Section 2.1.2, one common use case for probabilistic reasoning is the ability to solve queries given a certain evidence. In this context, by *evidence* we imply a set of logic facts that are known to be true, even though they may be defined over some probability distribution. Clearly, this influences the resolution process in most reasoning tasks. Coherently to most state of the art proposals, we fully support reasoning with evidence.

Annotated disjunctions — Recalling the contents of Section 2.1.3, Annotated Disjunctions are a special notation that supports the definition of non-binary probabilistic distributions over clauses and facts. This is not trivial to accomplish, and other state of the art proposals either support Annotated Disjunctions natively or perform strategic operations at the Knowledge Compilation level. Given their expressive power, we decide to support this feature in our engine.

Exact inference — Probabilistic Logic Programming allows for various different reasoning tasks, and each of them has been richly documented in the literature. As supporting many different tasks would add excessive complexity to this project, we decide to restrict our focus just on one. More specifically, in the first version of our proposal we opt for supporting *exact inference* exclusively. Again, this task concerns solving logic queries and estimating the probability of each solution without performing any sort of approximation. Although this approach is limiting in terms of performance, it is definitely easier to develop.

Listing 4.1: Prolog - Family Relationships Example (Version 2)

```
male(john).
male(mike).
female(anna).
female(jane).
parent(mike, john).
parent(mike, anna).
parent(mike, anna).
parent(jane, anna).
father(X, Y) :- male(X), parent(X, Y).
```

4.4.2 Behavioral Analysis of the Probabilistic Solver

In this section we explain the expected behavior of our PLP solver, comparing it to the mechanics of traditional Prolog solvers. First, we provide an example of query resolution for a generic Prolog solver over a simple logic program. Subsequently, we adapt the example so that it is suitable for PLP queries. Then, we demonstrate the mechanic behind query probability computation, and how it is related to Binary Decision Diagrams. Lastly, we inspect specific characteristics of probabilistic solvers including solving goals with negation, computing the probability of a query with *evidence*, and dealing with *Annotated Disjunctions*.

Prolog Query Resolution

First, we start by inspecting the resolution process that a generic Prolog engine is supposed to undertake in order to solve logic queries. For reference, consider the logic program in Listing 4.1, that is an adaptation of Listing 2.1 aimed to better support this analysis. Again, the program describes a basic logic model for family relationships. In this example, we assume that the Prolog engine searches solutions in depth-first order, following the mechanic of SLD resolution. With these premises, we expect the query `father(X, Y)` to produce a *Search Tree* similar to the one showed in Figure 4.5. For sake of simplicity, remind that a *Prolog Search*

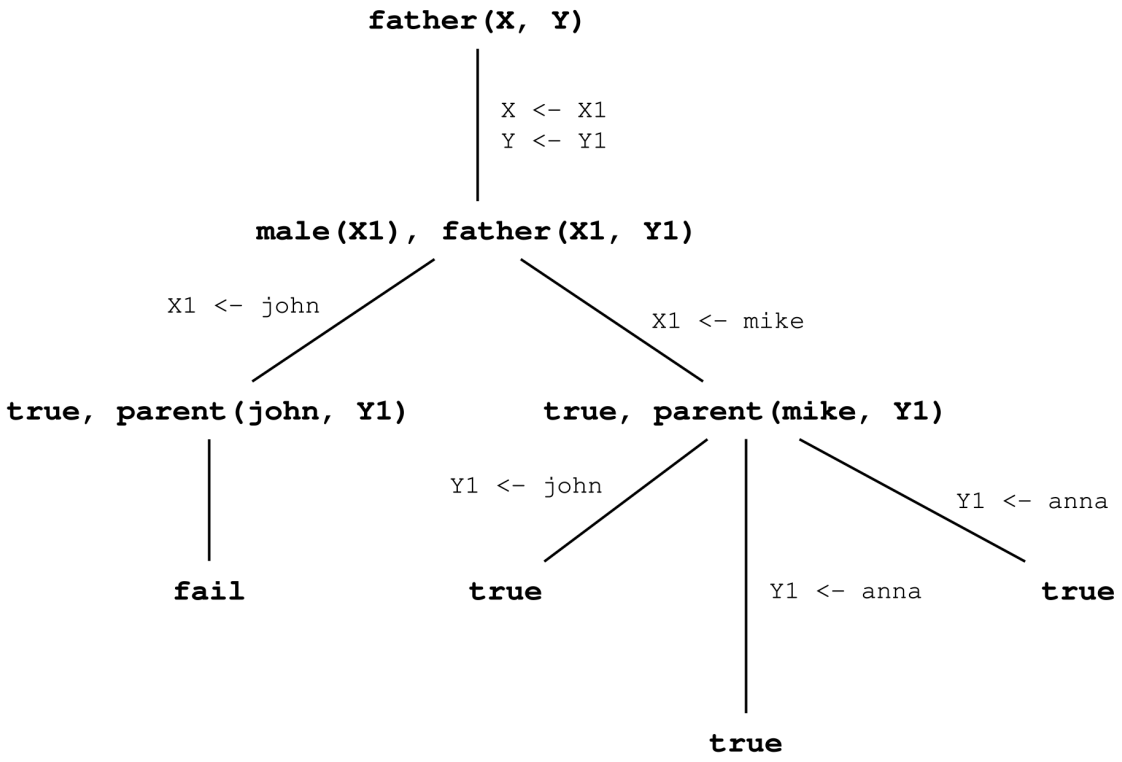


Figure 4.5: Example of Prolog Search Tree

Tree is an abstract representation of the resolution process that Prolog engines carry out for solving goals. The proof of a solved logic goal may be represented by a depth-first, left-to-right visit of the search-tree. In the tree, each node indicates an intermediate goal with a local substitution. Also, leaf nodes marked as `fail` indicate unfeasible goals on which the solver performs backtracking, whereas `true` leaves indicate success nodes that potentially lead to a solution.

By observing Figure 4.5, we conclude that the query produces three solutions: `father(mike, john)`, `father(mike, anna)`, and `father(mike, anna)`. Each solution is produced in sequence according to the depth-first visit of the search-tree. The left-most branch does not provide any solution and is pruned, as the engine fails to solve the `parent(john, Y1)` goal with the provided Knowledge Base. Note, the solution `father(mike, anna)` appears twice in the list. This happens because the `parent(mike, anna)` fact is repeated twice in the program of Listing 4.1,

Listing 4.2: ProbLog - Family Relationships Example

```
1 male(john).
2 0.80::male(mike).
3 0.65::female(anna).
4 0.90::female(jane).
5 0.60::parent(mike, john).
6 0.65::parent(mike, anna).
7 0.85::parent(mike, anna).
8 0.75::parent(jane, anna).
9 0.95::father(X, Y) :- male(X), parent(X, Y).
```

at lines 6 and 7 respectively. Although this does not make sense from a modeling standpoint, it a totally legit case from the perspective of a Prolog program. However, this detail assumes a different meaning in the context of probabilistic resolution.

Probabilistic Logic Query Resolution

We proceed in our example by considering a probabilistic logic program obtained by adding probability distributions over each clause of the program in Listing 4.1 using the ProbLog syntax. The result of this adaptation is shown in Listing 4.2. The probabilities assigned to each clause are fictional and not representative of reality, and are only supposed to be functional to this narrative. First, observe that the logic structure of the program has not been altered by any means. However, facts and clauses of this program are not guaranteed to be *true* anymore. Instead, they can either be *true* or *false*, depending on their assigned probability distribution. Note that the fact `male(john)` does not have any annotated probability, in which case we can assume it is guaranteed to be *true* with an implicit probability equal to 1. Also, observe that the facts at lines 6 and 7 still denote the same logic structure, but that now differ in probability distribution. Differently from Prolog, this case makes totally sense in the setting of probabilistic logic. In fact,

the two facts could represent probabilistic estimates derived from two different observations of the same phenomena.

For our purposes we are again interested in computing the probability of each solution of the query `father(X, Y)`, this time using the probabilistic logic Knowledge Base of Listing 4.2. Remind that in Section 3.1 we hinted that resolution strategies of Prolog solvers are also valid in the context of PLP inference. As such, we can solve the probabilistic query by following the search-tree of Figure 4.5 based on SLD resolution, and by altering its semantics to take in consideration the newly introduced probabilistic distributions. First, observe that each logic solution is obtained by following a path from the root of the search-tree to one of the *true* leaf nodes. The *Explanation* of each solution can be constructed by considering the probability distribution of all the probabilistic clauses selected in the solution proof.

From Section 2.1.2, recall that the selection of probabilistic clauses in each solution is related to the concept of *composite choice*. As such, consider a generic probabilistic fact in ProbLog notation $C = p :: f$. Such a fact can be seen as a Boolean variable X_C that is *true* with probability p and *false* with probability $1 - p$. The same concept applies to clauses as well, since the probability distribution is defined over the head terms. As a solution is defined over the logic conjunction of all the clauses selected to find it, an Explanation can be represented as the conjunction of all the Boolean variables X_{C_i} involved. Accordingly, we use the notation $EXPL(C_i)$ to indicate the contribution of a clause C_i in a certain explanation through its Boolean variable X_{C_i}. Also, to can specify that an Explanation is represented using Binary Decision Diagrams, we use the $EXPL_{BDD}(C_i)$ notation. So, given a solution s to a certain probabilistic query q, we define its Explanation as:

$$EXPL(s) = \bigwedge_i EXPL(C_i)$$

Then, the probability of each solution s can be computed by calculating the probability of $EXPL(s)$. For *Explanations* represented as Binary Decision Diagrams, this is feasible through *Weighted Model Counting* with the procedure shown in Algorithm 2. However, this is not sufficient for finding the whole solution set of a

probabilistic query. In traditional Prolog, solutions can appear with multiplicity higher than one for a given query, which is totally acceptable but does not provide any semantic contribution. Differently, multiple repetitions of the same solution represent different *probabilistic worlds* in PLP. From Section 2.1.2, recall that the probability of a probabilistic query is defined over the disjunction of all its *probabilistic worlds*. Considering that a given solution s can be found n times by a solver, we indicate its probability as:

$$p_s = PROB(\bigvee_{i \in (1,n)} EXPL(s_i))$$

With $PROB$ being the algorithm that computes the probability of an Explanation, depending on the data structures with which is represented. Note, that the number of repetitions n is inside the interval $(1, +\infty)$, and as such the solver can potentially never terminate computing the probability. This problem is mitigated by techniques of *Approximated Inference*, which we do not cover in this project.

We continue with our example by applying the knowledge provided in the previous paragraph. Considering a PLP solver configured to include Listing 4.2 in its Knowledge Base, we wish to find all the probabilistic solutions through *Exact Inference* for the query `father(X, Y)`. Observing, Figure 4.5, we conclude that we can find the same three solutions through SLD resolution: `father(mike, john)`, `father(mike, anna)`, and `father(mike, anna)`. Then, we label each clause as C_i, with i being their respective line number as in Listing 4.2. We proceed by enumerating the clauses selected by the solver to find each of the solutions.

$$\texttt{father(mike, john)} \rightarrow C_2, C_5, C_9$$
$$\texttt{father(mike, anna)} \rightarrow C_2, C_6, C_9$$
$$\texttt{father(mike, anna)} \rightarrow C_2, C_7, C_9$$

Then, the head of each clause is used to construct a Boolean variable with its defined probability, that is represented by a certain Explanation. Assuming that we represent Explanations as Binary Decision Diagrams, we define an Explanation

$EXPL_{BDD}(C_i)$ for each of the selected clauses, such that:

$$E_2 = EXPL_{BDD}(C_2),\ E_5 = EXPL_{BDD}(C_5),\ E_6 = EXPL_{BDD}(C_6),$$
$$E_7 = EXPL_{BDD}(C_7),\ E_9 = EXPL_{BDD}(C_9),$$

$$PROB(E_2) = 0.8,\ PROB(E_5) = 0.6,\ PROB(E_6) = 0.65,$$
$$PROB(E_7) = 0.7,\ PROB(E_9) = 0.95$$

Now, the actual solutions for the query, along with their probability, can be defined as follows:

$$s_1 = \texttt{father(mike, john)}$$
$$p_{s_1} = PROB(E_2 \wedge E_5 \wedge E_9) = 0.456$$
$$s_2 = \texttt{father(mike, anna)}$$
$$p_{s_2} = PROB((E_2 \wedge E_6 \wedge E_9) \vee (E_2 \wedge E_7 \wedge E_9)) = 0.7201$$

For performing *and-or* operations between the Explanations ($\vee$ and $\wedge$), we leverage the `apply` algorithm for Binary Decision Diagrams presented in Algorithm 1. Then, the probability of the resulting Explanation of each solution is computed Algorithm 2. In this simple example, the same results can also be achieved by manually manipulating the Boolean formula. For reference, the probability of s_2 can also be calculated as:

$$p_{s_2} = 0.8 \times (0.65 \times 0.85 + 0.65 \times (1 - 0.85) + (1 - 0.65) \times 0.85) \times 0.95 = 0.7201$$

We only provided the calculation for s_2 as it is the only non-trivial case of the example due to the presence of the disjunction between two repeated solutions. Instead, the probability of s_1 is the simple product of the probability of all the clauses selected to find it.

The example is supposed to show in detail the expected behavior of a generic PLP solver for performing Exact Inference tasks. We proceed by briefly summarizing the procedure point by point. First, we start by using a Prolog resolution strategy to search and collect all the solutions to a certain query. While collecting the solutions, we keep track of all the probabilistic clauses selected by the solver.

For each solution, those clauses are gathered in conjunction to build a single Explanation, in our case represented as a Binary Decision Diagram. Then, Explanations of identical solutions are reduced together disjunctively. At this point, we obtain a set of unique solutions, each one with its characteristic Explanation. Finally, the probability of each solution can be computed by manipulating their Explanation, in our case using Algorithm 2 for Binary Decision Diagrams.

Negation as Failure

In the previous section we highlight multiple similarities between query resolution in LP and PLP. Now, we put our focus on the additional peculiarities of PLP inference. The first difference we discuss is how *negated* goals are differently treated in probabilistic query resolution.

First, recall how negated goals are dealt with in traditional LP. A negated goal is the attempt of proving a term in the form $\neg q$ over a given Knowledge Base. A negated goal $\neg q$ is proven to be true if the solver fails to prove the term q to be true. This non-monotonic inference rule is known as *Negation as Failure*, and is implemented in Prolog through predicates like `\+(q)` and `not(q)`. This inference task is non-trivial if the negated goal is not ground. Note, in order to refute a negated goal $\neg q$ it is sufficient to prove q at most once, in traditional LP.

However, in the case of probabilistic logic, things are considerably different. First, Negation as Failure is not sufficient to cover negation in a probabilistic setting because goal resolution is not binary by nature. We can distinguish two different cases. In a simple scenario, if a solver fails to prove a certain goal, we could say that the negation of that goal is proven with probability of 1. However, the solver could be able to proven a certain goal, but still have no guarantee that its negation is failed. In fact, by definition, the probability of a negated goal is the complement of the probability of the relative non-negated goal. Given this fact, in order to solve a negated goal $\neg q$ a probabilistic solver must first try to solve the goal q compute its probability p_q. The probability of $\neg q$ is then defined as $p_{\neg q} = 1 - p_q$, which can potentially be 0. However, proving q only once does not suffice anymore. In fact, in the previous section we shown how the probability of

a certain goal is strongly defined by the disjunctions of all the repeated solutions found by the solver.

For clarity, consider again the logic program defined in Listing 4.2. As an example, solving a query such as `not(male(mike))` is straightforward. First, the probabilistic solver would first attempt to prove the goal `male(mike)`, for which it would find a probability of 0.8. Then, the probability of the query would be computed as $1 - 0.8 = 0.2$. However, solving a query such as `not(parent(mike, anna))` is certainly not trivial. Again, the solver would attempt to solve the goal `parent(mike, anna)`, for which it would find two solutions, with Explanations of probability 0.65 and 0.86 respectively. Then, it must obtain the overall probability of such goal as the disjunction of all the probabilities obtained, which would have value $0.65 \times 0.85 + (1 - 0.65) \times 0.85 + 0.65 \times (1 - 0.85) = 0.9475$. The probability of the negated query would then be $1 - 0.9475 = 0.0525$.

As we demonstrated, solving negated goals is quite expensive in the context of probabilistic solvers. In fact, the solver would first need to compute all the solutions of the non-negated goal in order to compute its overall probability. The procedure relies on probabilistic query resolution, which is inherently costly due to the necessity of finding and disjoining all the repeated solutions. As such, negation should be used carefully in probabilistic environments. Moreover, solving non-ground negated goals is certainly non-trivial, and in most cases not feasible at all. As such, we endorse the decision of some other state of the art proposals, that do not support the resolution of non-ground negated goals.

Probability with Evidence

Inference tasks in PLP solvers include solving queries given a set of known facts and clauses, which constitutes what is referred to as *evidence*. This resembles the topic of *conditional probability* well known in the probability theory. This notion is not present in traditional LP as it would not be influential during query resolution. On the other hand, evidence information is relevant in PLP as it changes the inferred probability of queries. Discussing the inner details of this kind of tasks is non trivial, and our of the scope of our narrative. From a more practical standpoint,

given a conjunction of ground literals e representing the evidence, we can identify two core tasks: inferring the probability of the evidence $P(e)$, and inferring the conditional probability $P(q \mid e)$ for a certain query q.

Solving the first tasks requires two steps. First, we need to find e over the provided Knowledge Base through simple LP inference. This is needed in order to ensure that e is a ground term, and that every evidence term inside the Knowledge Base gets considered. Then, the second step involves solving $P(e)$ through PLP inference, so that $P(e) = PROB(EXPL(e))$. Choosing between exact and approximated inference approaches is not relevant here.

The second task involves calculating $P(q \mid e)$. As introduced in Section 2.1.2, this can be accomplished through the formula:

$$P(q \mid e) = \frac{P(q, e)}{P(e)}$$

From this formula, we observe that in order to solve the second task we first need to compute $P(e)$. Then, using the same notation used in the previous section, the probability of an atomic query q given evidence e can be computed as:

$$P(q \mid e) = \frac{PROB(EXPL(q) \wedge EXPL(e))}{PROB(EXPL(e))}$$

Inherently, we need a way to distinguish evidence terms from the rest of the literals contained in the Knowledge Base. In our design, we endorse the choice we found in other proposals of using the `evidence/1` meta-predicate to indicate those terms. The purpose of the `evidence` predicate is to be used as the head term of facts inside the Knowledge Base. Given a fact `evidence(a)`, we know that the ground literal `a` is part of the evidence. Note, *evidence* can also be the head term of a clause, and can potentially contain a non-ground argument. In both cases, finding the ground conjunction of literals e is responsibility of the initial LP inference task.

Intuitively, solvers should be resilient to three evident edge cases: e can either be impossible to be solved, result in an empty conjunction, or have probability equal to 0. In those cases, we impose $P(e) = 1$ and $P(q \mid e) = P(q)$.

Annotated Disjunctions

In Section 2.1.3, we presented the concept of Annotated Disjunctions as a mean to represent non-binary probability distributions over facts and clauses. Existing PLP proposals support this feature either by natively dealing with multi-headed clauses in the solver, or by re-compiling the Knowledge Base so that Annotated Disjunctions are represented by an equivalent set of single-headed clauses. Given our choice of relying on the existing LP solvers of *2P-Kt*, our proposal adopts the second option by design.

In order to correctly map an Annotated Disjunction in a set of single-headed clauses, we need to fully respect the expected semantics. To better understand our approach, consider a fact f_0 with an Annotated Disjunction:

```
0.4::green; 0.6::red :- true.
```

During query resolution, we want to ignore f_0 and consider the two equivalent facts f_1 and f_2 instead:

```
P1::green :- true.
P2::red :- true.
```

Given this idea, we need to ensure that only one between f_1 and f_2 gets selected in a given logic solution during query resolution, along with its relative probability distribution. Considering two feasible explanations E_1 and E_2, selecting f_1 and f_2 respectively, we want to enforce $P(E_1 \wedge E_2) = 0$. This constraint is sufficient to respect the semantics of Annotated Disjunctions. The same principle applies to Annotated Disjunctions containing non-ground terms. In that case, the solver would need to find each grounding before enforcing the constraint. Furthermore, this approach also works for Annotated Disjunctions of clauses.

After ensuring that the semantics of an Annotated Disjunction is respected after compiling it in a set of equivalent clauses, we still need to adapt the probability computation procedure. Accordingly, recall that the probability distribution of an Annotated Distunction defines a random variable X having $n > 2$ values. As such,

we want to use $n-1$ Boolean variables $X_1, ..., X_{n-1}$ resembling the multi-valued variable X. There are multiple approaches to accomplish this goal, and in our proposal we adopt a technique based on *binary-splits*. The approach is based on the idea that the Boolean variables X_i are mutually exclusive, namely that one is *true* only if all the others are false. Inherently, the following can be observed:

$$\begin{aligned} X = i \quad &\rightarrow \quad \overline{X_1} \wedge \overline{X_2} \wedge ... \wedge \overline{X_{i-1}} \wedge X_i \qquad i = 1, ..., n-1 \\ X = n \quad &\rightarrow \quad \overline{X_1} \wedge \overline{X_2} \wedge ... \wedge \overline{X_{n-1}} \end{aligned}$$

Given this premise, the probability distribution of each Boolean variable X_i can be computed as following:

$$P(X_i) = \frac{P(X = i)}{\prod_{j=1}^{i-1}(1 - P(X_j))}$$

Note that $P(X = i)$ is known from the Annotated Disjunction as the probability of the head at index i.

Finally, implementing this technique in a PLP solver demands for a strategy to encode the binary split. Solvers needs to keep track of the Annotated Disjunction that originated a certain artificial clause when selecting it. For instance, it would be incorrect to implement the mutual exclusion of Boolean variables through *Negation as failure*, because that would impact the resolution process by potentially negating other clauses in the Knowledge Base as well. At the same time, the solver must be capable of maintain in memory information regarding the binary split until all the involved terms are grounded. Our implementation choices for this problems are discussed in Chapter 5.

4.4.3 Architectural Design of the Solver

In this section we show how we designed the internal architecture of our proposal to fit into *2P-Kt* as an extension module. As described in Section 4.3, our purpose is to develop a PLP solver for *Exact Inference* tasks by respecting both semantics and interfaces of Prolog solvers in *2P-Kt*. Moreover, we are interested in working on top of already existing LP solvers in order to maximize code reusage and to

optimally separate the responsibilities of each module. We proceed by discussing the design of our solution point by point.

Overview

Intuitively, solving PLP inference tasks by only using a LP-based solver is a non-trivial task. In our research, we acknowledge two potential approaches for addressing this problem:

1. Operating at the low level by rewriting or upgrading the underlying interpreter of a LP solver, so that it becomes capable of solving both LP and PLP tasks. With this approach, it's easier to add low-level optimizations for complex computations by tampering the interpreter's code directly. However, there also are some inconvenient downsides. First, the interpreter code would become more fragile, as future code updates in the many parts of *2P-Kt* could invalidate the expected behavior of the solver for PLP tasks. Second, it becomes progressively harder to upgrade the interpreter to support additional features, as the code would be optimized and minimized. Third, a general purpose solver supporting both LP and PLP could perform poorly while performing traditional LP tasks.

2. Designing and creating a library of LP-compliant meta-predicates implementing the additional PLP features. Meta-level interpretation has the benefits of being easier to develop and maintain, and adding new functionalities is frictionless. Moreover, this approach avoids the need of modifying the code of LP solvers, as the PLP-specific computations are implemented by the library predicates, which are written to be LP-compliant. This results in a more robust solution, as future updates in *2P-Kt* would not affect any of the code related to the PLP library. However, many downsides are present in this approach as well. First, the logic paradigm is not suitable to implement many routines involved in PLP inference, such as Knowledge Compilation and Weighted Model Counting. As such, the whole solution would be fairly less efficient than any other alternative. Second, the whole Knowledge Base

would need to be recompiled in order to be compliant with the new set of meta-predicates. This detail would not be transparent to clients and would compromise usability.

During the development of our proposal, we experimented the usage of both these two approaches. However, we did not find comfort in adopting any of them, as we found their weaknesses and downsides excessively limiting for our purposes. As such, we decided to introduce an alternative and unconventional third solution. Our idea is to adopt an hybrid approach based on multi-paradigm programming, by writing both object-oriented code and meta-predicates to inherit the benefits of both worlds. This is feasible in a multi-paradigm ecosystem such as the one of *2P-Kt*. We still introduce a library of meta-predicates covering probabilistic-related computations, which in this case only act as a facade. Through the *primitives* mechanism of *2P-Kt*, we are able to implement the behavior of each meta-predicate with object-oriented Kotlin code. This preserves the opportunity to optimize tasks such as Knowledge Compilation and Weighted Model Counting, and maintains the flexibility of delegating the whole logic-based reasoning to an underlying solver. Finally, a middleware component is added to perform Knowledge Base re-compilation in automatic, so that users can be totally unaware of the underlying meta-predicate system.

Given this overview, the design of our PLP solver is built on top of three interconnected components: a *Library of Meta-Predicates*, a *Knowledge Recompilation Engine* and a *Solver Piloting Engine*. A high-level representation of the whole architecture with its components is shown in Figure 4.6.

Library of Meta-Predicates

It's worth mentioning that we find the need of representing Explanations at the logic level, so that they can be manipulable and propagable by the meta-predicates. This demands for being able to attach an Explanation object to a logic term, which are the only entity known by LP solvers. Luckily, this is easily feasible in *2P-Kt*, which provides handy means to attach additional information from the object-oriented realm to logic terms used by solvers. Further details of how this

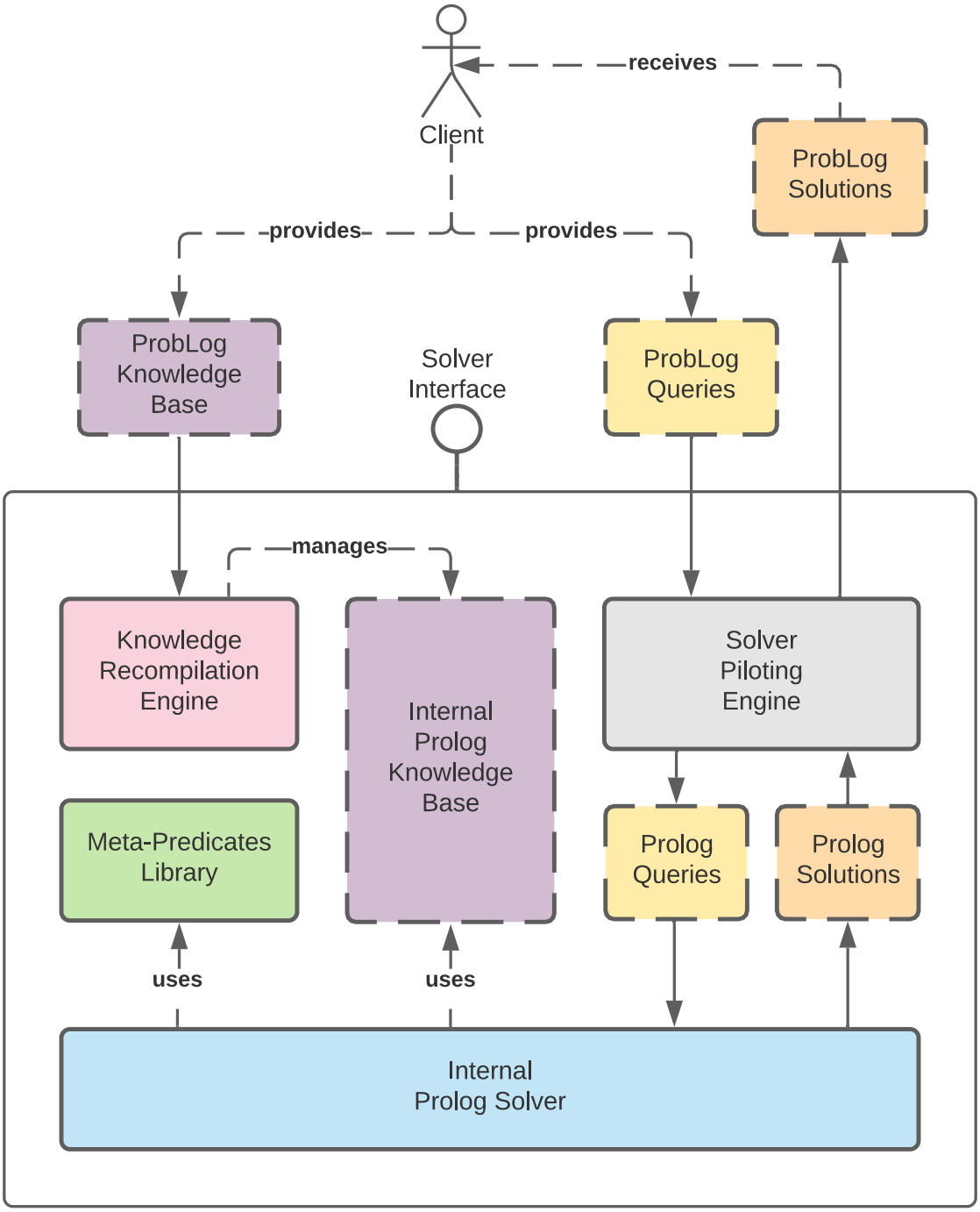


Figure 4.6: Architectural Overview of the PLP solver

is implemented in our proposal are provided in Section 5.2.1. We decided to manipulate Explanations instead of probabilities, as they provide a more generic and complete source of truth during goal resolution. Moreover, the probability of an Explanation can easily be computed anytime through Weighted Model Counting.

We introduced a collection of Prolog-compliant meta-predicates, each one implementing a specific task required for PLP query inference. Some meta-predicates can be implemented as logic rules, whereas others require imperative business logic that is insertable through the *primitive* mechanic of *2P-Kt*. The predicates are collected inside a Prolog-compliant *library*, so that it can be inserted in the configuration of the Prolog solvers of *2P-Kt*. We proceed by listing each meta-predicate and by describing its semantics.

`prob_query(?Prob, :Goal)` — This is the main entrypoint for probabilistic logic goal resolution. The first argument represents the numeric probability of the goal being true, and the second argument is the query goal. The probability argument can either be an input number or an output variable. If the goal argument is a non ground term, its variable are substituted for each solution found by the solver, as to be expected by traditional Prolog solvers.

`prob(-Expl, :Goal)` — This acts as a simple wrapper to attach an Explanation to a logic term. The purpose is using this as a meta-predicates both in the head and in the body of clauses to enable the propagation, manipulation, and substitution of Explanations along with the logic term. The first argument is a term representing the Explanation, whereas the second argument is the logic term itself.

`prob_solve(-Expl, :Goal)` — This meta-predicate solves a logic goal and computes its Explanation. The purpose was to create a counterpart of the `call` meta-predicate of traditional Prolog capable of operating with Explanations. Moreover, this implements the solution grouping semantics of PLP solvers. As such, each substitution of `Goal` is unique in the solutions produced by this meta-predicate, and its attached Explanation is computed to represent the disjunction of all the alternatives.

`prob_expl_build(-Expl, +Prob)` — This is a primitive to manipulate Explanations right at the logic level. More specifically, this creates an Explanation term, represented by the first argument, which encodes the numeric probability specified in the second argument. The resulting Explanation represents a Boolean random variable that has probability *Prob* of being true.

`prob_expl_and(-ResExpl, +Expl1, +Expl2)` — This primitive allows to compute the *and* operation between the two Explanation terms `Expl1` and `Expl2`. The result is an Explanation term substituted to the first argument `ResExpl`.

`prob_negation_as_failure(-Expl, +Goal)` — As already mentioned, the *Negation as Failure* semantics in PLP differs from the one of LP. As such, we introduced this meta-predicate as a surrogate of the `\+` and `not` Prolog predicates, which inherently also has an argument for the computed Explanation. Due to the motivations discussed in Section 4.4.2, passing a non ground term as a goal of this meta-predicate results in a failure.

`prob_solve_evidence(-Expl)` — From the perspective of the analysis of Section 4.4.2 regarding solving goals with a given evidence, this meta-predicate solves the task of computing *P(e)*. However, we respect the format of the other meta-predicates of computing an Explanation instead of the numeric probability. More correctly, this meta-predicate executes the business logic of solving the evidence e over the given Knowledge Base, and computes its Explanation *EXPL(e)*. Since we find no use in the final solved ground term e, only the Explanation is substituted and accepted as argument. Failure in finding or grouding the evidence e results in an Explanation encoding a probability 1 of being true.

`prob_solve_with_evidence(-GoalEvExpl, EvExpl, :Goal)` — This meta-predicate is complementary to `prob_solve_evidence`, and is responsible for the business logic required to solve $P(q \mid e)$ given a query q and an evidence e. More specifically, this solves `Goal` and substitutes `GoalEvExpl` with the conditional Explanation $EXPL(q, e) = EXPL(q) \wedge EXPL(e)$. Instead, `EvExpl` is substituted with the evidence Explanation *EXPL(e)* as in

`prob_solve_evidence`. This meta-predicate is used internally by `prob_query` to solve conditional queries.

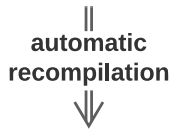
As already mentioned, the library of meta-predicates is useless if not coupled with an adaptation, or recompilation, of the whole Knowledge Base. This is required for piloting the LP solver to select the right meta-predicates and correctly implement the semantics of probabilistic reasoning.

Knowledge Recompilation Engine

The meta-predicates approach requires the Knowledge Bases to be formatted in an ad-hoc way, so that the LP solver is forced to select the meta-predicates during goal resolution. Clearly, this leads to a terrible user experience from the perspective of clients, which would instead be interested in writing their probabilistic-logic Knowledge Base in their preferred language (in our case, ProbLog). Moreover, the meta-predicates make the Knowledge Base more verbose and not intuitive. As such, the system demands for a middleware component that automates the recompilation process.

We name the component responsible for the automated recompilation *Knowledge Recompilation Engine*. As shown in Figure 4.6, the engine accepts probabilistic-logic Knowledge Bases provided by clients written in ProbLog syntax. Then, clauses contained in the Knowledge base are processed one by one, and recompiled by adding the required meta-predicates of our library and wrapping terms with the `prob` predicate where necessary. The logical semantic of each clause is not altered, and the program is ensured to work as intended. Note, one clause can be recompiled as one or more clauses. For instance, Annotated Disjunctions can be split in multiple single-headed clauses, as we mentioned before. Subsequently, the clauses resulting from the recompilation are stored in an internal Knowledge Base that is written in pure-Prolog syntax. This internal accumulator will act as the actual Prolog Knowledge Base used by the underlying LP solver for solving goals. Observe that consistency between the internal and external Knowledge Bases is guaranteed by the *Knowledge Recompilation Engine*, which intercepts all the probabilistic-logic clauses provided by clients. An example of recompilation is

```
male(john).
0.80::male(mike).
0.65::female(anna).
0.60::parent(mike, john).
0.95::father(X, Y) :- male(X), parent(X, Y).
```




```
prob(E, male(john)) :- prob_expl_build(E, 1.0).
prob(E, male(mike)) :- prob_expl_build(E, 0.8).
prob(E, female(anna)) :- prob_expl_build(E, 0.65).
prob(E, parent(mike, john)) :- prob_expl_build(E, 0.6).
prob(E, father(X, Y)) :-
    prob_expl_build(EH, 0.95),
    prob(EB1, male(X)),
    prob(EB2, parent(X, Y)),
    prob_expl_and(EB, EB1, EB2),
    prob_expl_and(E, EH, EB).
```

Figure 4.7: Example of ProbLog Knowledge Base Recompilation

represented in Figure 4.7, for which we took some of the clauses from Listing 4.2. Furthermore, the example show a practical correct usage of the meta-predicates in our library.

Note that we the whole recompilation system is transparent and invisible to clients, which may not even be aware of it. Furthermore, clauses are processed only once, at the moment of insertion in the Knowledge Base. However, we did not design a component that takes care of reversing the compilation transformation. We motivate this choice through the lack of use cases of an inverse transformation in our project, which however could potentially be attained with little effort. One downside is that the recompilation can be visible to clients if an inspection of the internal Knowledge Base is requested. However, we found this issue acceptable for the purposes of our value proposition, as it does not limit the usability of the solver since clients would still be able to consult the external Problog Knowledge Base in their possession.

Solver Piloting Engine

The last piece needed by our system to fully implement a PLP inference solver is a component that hides the presence of an underlying LP solver. We name this portion of software *Solver Piloting Engine*. This component is responsible for accepting LP and PLP queries from clients, properly configuring the underlying LP solver, piloting it to infer the solutions, extracting the probability values and present the results. As a matter of fact, this represents the presentation layer of our system. Note, solver configurations are handled at this level, and the component is capable of passing both LP and PLP queries to the inner solver on demand.

Also, the engine recompiles queries and goals bidirectionally to be compliant with the meta-predicates semantics of our library. For instance, our library assumes that each query is expressed through the `prob_query` predicate. Considering the example in Listing 4.2, a query such as `father(X, Y)` would be transformed in `prob_query(Prob, father(X, Y))`. Once solutions are found, the *Solver Piloting Engine* extracts the two terms `Prob` and `father(X, Y)`, and present their values to the clients in the correct format.

4.4.4 Accomplishments

Considering all the abstractions we introduced, we are able to develop a system capable of solving inference tasks for both probabilistic and traditional logic, supporting ProbLog and Prolog languages respectively. Our extension maximizes the usage of what is already existing inside *2P-Kt*, and is capable of benefiting of future updates of other modules with robustness. In fact, logic resolution is totally delegated to a Prolog solver from *2P-Kt*, and the extension only integrates the support for probabilistic computation. Furthermore, we are able to inherit the benefits from both the logic-based and object-oriented worlds with our hybrid approach based on multi-paradigm programming with Kotlin. In fact, the meta-predicates system allows fan expressive and flexible integration of the features related to probabilistic reasoning, whereas the *primitives* mechanic allow us to complement with imperative programming the more computation intensive

part. Also, semantics and programming interfaces of *2P-Kt* solvers are respected, so that our solver could be used as a simple LP solver with ease.

This section purposely discussed details of design and architecture of our project, without disclosing facts about *how* we were able to developing it. In fact, the design leaves abundant space for implementation choices and specific optimizations. We dedicate Chapter 5 to the discussion of our most relevant implementation choices.

Chapter 5

Implementation

In this chapter we present how we developed our project on top of design details discussed in Chapter 4, putting our focus only on the most relevant implementation choices and optimizations. Each section of this chapter dives into a specific portion of the codebase of our proposal.

5.1 Binary Decision Diagrams Library

In this section we discuss the most significant development choices and optimizations that we introduced in our codebase related to the Binary Decision Diagrams library. In Section 5.1.1, we describe the structure of our default implementation for the `BinaryDecisionDiagramBuilder` interface. In Section 5.1.2, we demonstrate how we widely leveraged the *visitor* pattern to implement the supported operators. Section 5.1.3 presents an optimization that uses the *visitor* pattern to avoid performance degradation due to type checking. In Section 5.1.5 we discuss an optimization that executes the `apply` algorithm and Shannon Expansion operations altogether to achieve better performance over the most recurrent computations. Finally, in Section 5.1.4 we explain how the immutability of the core data structures consented the use of lazy evaluation to cache the results of computation-heavy operations.

5.1.1 Default Builder

Reminding what explained in Section 4.2.1, multi-platform support of the library is enabled through the `BinaryDecisionDiagramBuilder` interface, which is responsible of the creation of Binary Decision Diagram nodes. Our choice is to develop a single implementation that could be used as the default options for all the supported platforms. Accordingly, we believe that providing platform-specific optimization is out of the scope of this contribution, which can be subject for future developments and updates instead.

Our default builder is a simple in-memory heap-based data structure with mono-directional pointers from parent to child nodes. Naturally, the data structure is simple enough to be represented through *data classes* available in Kotlin. Binary Decision Diagrams can largely grow in size, and so memory consumption is a serious concern that must be addressed in production environments. However, we decide not to solve this problem in our first release of the project, and so every node is maintained in memory with no swapping optimizations. Overall, this provides an acceptable solution that can be used across the platforms supported by *2P-Kt*

A key point of our implementation is that we opted to represent all data structures as *immutable* objects. This choice has the downsides of a general increase in memory consumption and additional computations of construction operations. However, we accept those defects as a tradeoff to benefit of the valuable guarantees provided by the immutability property. Among the others, one of the most valuable consequence is that each diagram node becomes a *canonical* representation of its content. As such, we can optimize resources usage by maintaining in memory a cached version of the whole node and all unary operations applied to it, which are guaranteed to remain constant during the lifetime of a program execution. Moreover, this provides strong guarantees in concurrent environments as well.

Finally, the value representing Variable nodes of Binary Decision Diagrams are supported through the *generic typing* available in Kotlin. As such, we are able to grant maximum reusage of our implementation of both the builder and the data structures produced by it, by also benefiting from type safety guarantees. As for design, we constrained the generic type to extend the *Comparable* interface, so

that variable ordering logic is explicitly implemented through type comparison.

5.1.2 Visitor Pattern for Operators

With reference to Section 4.2.2, we committed to provide support to three core operations for Binary Decision Diagrams in our first release: `any`, `expansion`, and `map`. These operations have a bunch of properties in common. First, they are all unary, in the sense that they operate on a single Binary Decision Diagram. This concept makes it safe to maintain their results in memory once computed, as we designed our diagrams to be *immutable*. Second, all those operations require the recursive exploration of the nodes of a diagram, with the purpose of accumulating information and synthesize a result of a certain kind. In the case of `any`, the result is a Boolean, whereas `expansion` and `map` can produce objects of generic type.

Given these observations, we acknowledge that the *visitor* pattern, well supported in our library, could be leveraged to implement all three operators. The idea is to provide one implementation of the `BinaryDecisionDiagramVisitor` interface for each operator, separating the business logic to be applied to Terminal and Variable nodes. Moreover, the usage of *generic types* available in the Kotlin language allows for a single implementation to be suitable for every coherent data type. For clarity, Listing 5.1 shows how we implement the `any` operator following this pattern. Additionally, observe that instances of visitor implemented with this semantic are inherently *stateless*. Namely, it is safe to use a single instance to execute its operator on more than one diagram, even concurrently and in parallel.

Implementing the `apply` algorithm through the *visitor* pattern makes things slightly more difficult. In fact, the algorithm is a first example of binary operation between Binary Decision Diagrams. As such, using a visitor to explore the structure of only one diagram is not sufficient. We solved this issue by adding an attribute to the visitor's class representing the second operand diagram. That way, the first operand is represented by the visited diagram, whereas the second one is referenced by the state attribute. Then, we reproduce the recursive call of the algorithm by either passing the visitor to another sub-diagram, or chang-

Listing 5.1: Kotlin - Implementation of the any Visitor for BDDs

```kotlin
internal class AnyVisitor<T: Comparable<T>>(
    private val predicate: (T) -> Boolean,
) : BinaryDecisionDiagramVisitor<T, Boolean> {

    override fun visit(
        node: BinaryDecisionDiagram.Terminal<T>
    ): Boolean = false

    override fun visit(
        node: BinaryDecisionDiagram.Variable<T>
    ): Boolean {
        var result = predicate(node.value)
        if (!result) result = node.low.accept(this)
        if (!result) result = node.high.accept(this)
        return result
    }
}
```

Listing 5.2: Kotlin - Implementation of the `any` Method for BDDs

```kotlin
fun <T : Comparable<T>> BinaryDecisionDiagram<T>.any(
    predicate: (T) -> Boolean
): Boolean {
    val visitor = AnyVisitor(predicate)
    return this.accept(visitor)
}
```

ing the referenced instance of the second operand. This multi-dispatching system practically reproduces the recursive flow of the algorithm defined in Algorithm 1. During the execution, a table is maintained as a state attribute inside the visitor to enable *dynamic programming*. This choices make the `apply` visitor a *stateful* object, which is not safe to be used in concurrent or parallel environments.

Overall, we hid our development choices under simple method signatures. That way, clients are not aware of the internal constraints, such as the mono-thread requirement of our implementation `apply`. Each method creates, or reuses, a visitor specific for its own operation and then submits it to the diagram to proceed with the exploration. Then, the computation carried out by the visitor is returned to the client as result of the method. Under the hood, each method is implemented by leveraging the *extension methods* feature offered in Kotlin, which provides an handy way of modularly adding functionalities to a class or interface without requiring the inheritance mechanism. So, the `BinaryDecisionDiagram` interface explicits information regarding the data structure itself only, whereas methods for the various operations are collected inside `BinaryDecisionDiagramUtils` and `BinaryDecisionDiagramOperators`. For reference, consult the code in Listing 5.2, which shows our implementation of the `BinaryDecisionDiagram.any` extension method by relying on the visitor shown in Listing 5.1.

5.1.3 Visitor Pattern for Type Checking

One of the most obvious problems of multi-platform programming is that performance discrepancy can happen between different compilation targets and execution environments. This is the case for the type-checking system of Kotlin multi-platform architectures, which are capable of targeting platforms where the concept of *type* is absent. In our case, we observed a noticeable performance decrease in the JavaScript platform, which is one of the targets supported by the *2P-Kt* project. According to our experience, *KotlinJS* seems to be inefficient in translating dynamic casting and type checking into JavaScript code, which causes a loss of performance when relying on Kotlin constructs such as `when`, `is`, or `as`. In this context, we propose an optimization that would mitigate the problem without tampering the underlying Kotlin framework.

As such, our approach leverages functional programming and the *double-dispatching* mechanic of the *visitor pattern* to replace type casting and `when` statements wherever possible in our codebase. The idea is simple, and consists in configuring a visitor with many function object, each one consuming one type case of a hierarchy. In the case of Binary Decision Diagrams, the visitor would accept two function objects, one consuming Terminals, and the other consuming Variable nodes. After the configuration, the visitor has to be passed to an object to be visited, such as a Binary Decision Diagram. During the visit, the natural double-dispatching of the visitor triggers one of its methods, which is responsible of delegating the visit to the correct consumer function object. Overall, we are able to specify one procedure for each case of a type hierarchy by leveraging *polymorphism* instead of *runtime type-checking*. The first is generally more performant, as it does not rely on any system or introspection mechanism, with the type cast being substituted by a chain of function calls. We name this class `CastVisitor`, of which our simple implementation is observable in Listing 5.3.

Listing 5.3: Kotlin - Implementation of `CastVisitor` for BDDs

```kotlin
/**
 * Example of usage:
 *
 * val castVisitor = CastVisitor<...>()
 * castVisitor.onTerminal = {
 *     it -> // Handle terminal node case
 * }
 * castVisitor.onVariable = {
 *     it -> // Handle variable node case
 * }
 * bdd.accept(castVisitor)
 * */
internal class CastVisitor<T: Comparable<T>, E> :
        BinaryDecisionDiagramVisitor<T, E> {
    var onTerminal: (
        (o: BinaryDecisionDiagram.Terminal<T>) -> E
    )? = null

    var onVariable: (
        (o: BinaryDecisionDiagram.Variable<T>) -> E
    )? = null

    override fun visit(
        node: BinaryDecisionDiagram.Terminal<T>
    ): E = onTerminal!!(node)

    override fun visit(
        node: BinaryDecisionDiagram.Variable<T>
    ): E = onVariable!!(node)
}
```

5.1.4 Lazy Evaluation

As hinted in Section 5.1.1, our choice of implementing Binary Decision Diagrams as an *immutable* data structure brings a wide range of benefits. In our perspective, the most interesting consequence is the opportunity of performing expensive computation *only once*, and save their results so that it can be returned to the caller in every subsequent request. This is possible as, by definition, the data structure does not change during its lifetime. In our use cases, a number of relevant operations can leverage this property. The simplest case is the one of the `toString` and `hashCode` methods. Both of those are unary operations that trigger a chain of recursive calls, as the data structure of BDDs is recursive by nature. Practically, this means that in order to compute the `hashCode` of a Binary Decision Diagram, the same computation is needed for all its sub-diagrams until a Terminal node is reached. The same principle applies for the `not` logic operator, which we implemented through the unary version of the `apply` algorithm. Diving into the subject, we observe that most operations attainable through Shannon Expansion could leverage this caching benefit. In our case, this applies to the *Weighted Model Counting* that computes the numeric probability of BDDs.

Under the hood, we leveraged the Kotlin *property delegation* mechanism by relying on the *Lazy* delegate. Briefly, the Kotlin language provides advanced tools for the *Delegation Pattern*. For reference, *property delegation* allows for delegating to an ad-hoc library object the computation or retrieval of the value for a certain property. The extensive usage of *generic typing*, grants maximum reusage of the delegate object across the Kotlin standard library. The delegate of our interest is `Lazy`, which accepts a lambda expression containing the computation of a value and triggers it only at the first request. Subsequently, the result of the first invocation is returned in order to avoid multiple computations. This is an excellent instrument for our caching purposes, and we widely used the `Lazy` construct across our codebase.

5.1.5 Apply-Then-Expansion Optimization

Indeed, the most frequent operations applied on Binary Decision Diagrams are the logic operations *and*, *or*, and *not*, which in our implementation all rely on the `apply` algorithm. However, we also acknowledge a frequent use of the Shannon Expansion operation in our project, that is leveraged to perform Weighted Model Counting with the purpose of computing numeric probabilities. Starting from this observation, we looked for an optimization that would increase the performance of our specific use case.

As such, we observe that the `apply` algorithm builds Binary Decision Diagrams bottom-up, in a similar way with which procedures based on Shannon Expansion perform their computation. Accordingly, we advocate that the two operations can be performed together in one single pass, so that Shannon Expansion computations are performed right away at the diagram construction. This is possible because the information required at each recursive level of the Shannon Expansion is the same used by the `apply` algorithm. The consequence of this approach is that we are able to pre-compute one or more values over a Binary Decision Diagram directly at during its construction, so that they are promptly available when requested. Obviously, calculating values eagerly makes the overall computation of the `apply` algorithm more expensive. However, this trade-off can become beneficial if the computed value is accessed frequently. Coupled with the reasoning discussed in Section 5.1.4, the values computed with this optimization are not re-computed in subsequent requests.

We named this new Binary Decision Diagram operation `applyThenExpansion`. Semantically, the operation is equivalent to invoking the two operations `apply` and `expansion` in sequence. However, the overall performance of the sequence is boosted due to the need of visiting the data structure only once, which is beneficial in case of very large Binary Decision Diagrams. This operation is implemented by leveraging the *visitor pattern* coherently to what we document in Section 5.1.2, and is based on a simple revision of the code that implemented the standalone `apply` operator.

5.2 Solver

In this section we discuss the most significant development choices and optimizations that characterize our implementation of the PLP inference solver. In Section 5.2.1 we show how we represent Explanations in our system. In Section 5.2.2 we explain an optimization that improves the indexing of the clauses inside the Knowledge Base of the solver. In Section 5.2.3, we disclose how we implement the automatic Knowledge Base recompilation. In Section 5.2.4 we show the strategy with which we implement Annotated Disjunctions. Finally, in Section 5.2.5 we present some optimizations that improve the performance for simple Prolog queries.

5.2.1 Representing Explanations

As hinted in Section 4.4.3, we want our PLP solver to be capable of manipulating Explanations right at the logic level. The advantage of this approach is that we can leverage the resolution strategy built inside the logic solver to iteratively construct the Explanation while solving goals. However, we also need to embed an Explanation object inside logic terms, so that the LP solver is capable of manipulating it like any other term. By manipulation, we intend the ability of selecting Explanation terms, using them for substitutions, and considering them as predicate arguments. Then, the business logic implementing the actual construction of the underlying Binary Decision Diagram data structures is handled by meta-predicates such as `prob_exp_and`.

First, we introduce an object interface for representing Explanations themselves, which we name `ProbExplanation`. By definition, an Explanation is an abstract collection containing one or more logic terms, that can also be manipulated and combined with other Explanations. The code in Listing 5.4 is a stripped version of the actual interface definition, containing the core methods and properties. As visible, we define the interface by the ability of applying the Boolean operators *and*, *or*, and *not*, and by the possibility of extracting a numeric probability value from its content. Also, the `containsAnyNotGroundTerm` property

Listing 5.4: Kotlin - `ProbExplanation` Object Interface

```
internal interface ProbExplanation {
    val probability: Double
    val containsAnyNotGroundTerm: Boolean
    fun not(): ProbExplanation
    infix fun and(
        that: ProbExplanation
    ): ProbExplanation
    infix fun or(
        that: ProbExplanation
    ): ProbExplanation
    fun apply(
        transformation: (ProbTerm) -> ProbTerm
    ): ProbExplanation
}
```

returns true if the Explanation contains at least one term that is not ground. This case is possible because the Explanation is supposedly constructed during goal resolution, which may require keeping track of terms yet to be grounded. Finally, we added the `apply` method, which is useful to apply a transformation to every logic term contained in the Explanation. Note, the definition is abstract and not bound to Binary Decision Diagrams at all. In fact, we want to leave the door open for future developments that could consider adopting a different data structure for representing Explanations and computing probabilities, such as one of the alternatives we reference in Section 2.1.4. Coherently, we provide an implementation of the `ProbExplanation` interface based on our Binary Decision Diagram library.

As visible, we also introduced the new class `ProbTerm`. This is a simple extension of the `Term` interface native in *2P-Kt* meant to represent logic terms that have an attached probability values. In ProbLog syntax, the probability value is expressed through the `::/2` operator. The class adds few data attributes to represent the probability, and a numeric identifier that is used to keep track of

the clause from which the term was selected from the Knowledge Base. Under the hood, the identifier is used for determining the variable ordering of Binary Decision Diagrams.

Furthermore, we extend the `Term` hierarchy of *2P-Kt* by introducting the `ProbExplanationTerm` class. Intuitively, this entity simply represents a logic term with an Explanation attached internally. This enables the logic manipulation capabilities mentioned at the beginning of this section. Logic terms of this class are meant to be subject to logic substitution and to be arguments of meta-predicates such as `prob_exp_and`. Note, we did not design this logic term to be *constant*. In fact, we want the solver to be capable of applying substitutions to the non-ground terms contained inside an Explanation. In fact, this is the primary use case we found for the `ProbExplanation.apply` method.

This implementation choices allow us to introduce some additional optimizations. First, we acknowledge that the same object instances can be reused at runtime for the Explanations representing the *true* and *false* constants. Practically, those Explanation are Binary Decision Diagrams containing a single variable node with probability value of either 1 or 0. Moreover, we recognized that the usefulness of `prob_expl_build` was marginal. In fact, we could decide to pre-compile an Explanation object inside clauses and facts of the Knowledge Base right away during the recompilation process. This is made possible by the `ProbExplanationTerm` class, which allows attaching Explanation objects direcly inside the terms in the Knowledge Base. As such, the `prob_expl_build` meta-predicate is not present in our implementation of the PLP library.

5.2.2 Clause Head Optimization

In the example we show in Figure 4.7, it is clearly visible that our automatic recompilation supposedly uses the `prob/2` predicate to *wrap* the head of every clause inside the Knowledge Base. In this way, the `prob/2` meta-predicate does not require any underlying implementation, as it serves as a simple reference for the body of other clauses to allow goal resolution. However, this causes an unexpected degradation of the performance.

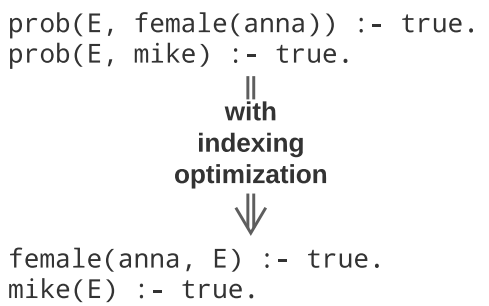


Figure 5.1: Indexing Workaround for the `prob` Meta-Predicate

Under the hood, *2P-Kt* provides a set of optimizations aimed to improve the performance of clause selection. The `Theory` interface, which represents abstract logic Knowledge Bases in *2P-Kt*, has an ad-hoc implementation that performs clause indexing. As modern databases normally do, indexing allows for a faster retrieval of content by constructing tree-like data structures meant to enable *Binary Search*. In that way, *2P-Kt* is capable to attain logarithmic time-complexity for fetching single clauses inside Knowledge Bases. However, this optimization is in conflict with the outputs of our recompilation. In fact, the information used to index clauses constitutes of the name of the head predicate and its first argument. By wrapping every head predicate with the `prob/2` meta-predicate, we practically neutralize the effect of indexing because every clause in the Knowledge base ends up having a head with the same predicate name.

We solve this issue by adopting two workarounds. First, we avoid wrapping the head predicates with the `prob` meta-predicate. The Explanation term is instead put as last argument of the head predicate. If the head of a clause is an atom, we transform the atom to a predicate by using the atom literal as the predicate name, and by making the Explanation term its only argument. As a matter of fact, we simply increase the arity of each head term by one, by making the Explanation term the last argument. For clarity, in Figure 5.1 we show an example of how we apply this transformation. However, we still reference the `prob` meta-predicate in the body of each rule in the recompiled Knowledge Base. As such, the second workaround consists in providing an imperative implementation of the `prob` meta-predicate as a *2P-Kt primitive*. The implementation of the meta-predicate

is responsible of hiding the complexity of the first workaround. As such, the code unwraps the inner term from the `prob` predicate, then adds the Explanation term as last argument so that it becomes coherent with the format of the head terms inside the Knowledge Base.

By adding our workarounds, we are able to fully benefit of the indexing optimizations of *2P-Kt*. The downside is that the solver will have additional level of depth in its resolution process due to the need of invoking the *primitive* code of `prob` every time a goal is solved. Further optimizations can be added to mitigate this defect, which we do not cover in this dissertation.

5.2.3 Knowledge Base Recompilation

In Section 4.4.3, we present the expected behavior of the *Knowledge Recompilation Engine*. Coherently with the design constraints, we opt for implementing the engine as an extension of the `Theory` interface of *2P-Kt*. As already mentioned, `Theory` is the abstraction responsible of representing clause databases and logic Knowledge Bases. Our idea is to provide an ad-hoc implementation of `Theory` that applies the recompilation transformation every time a clause is added. Also, we leverage the *class delegation* feature of Kotlin to delegate the real clause database management to an already existing implementation of `Theory` provided by *2P-Kt*. As such, our code acts as an interceptor to implement the *Decorator Pattern*. Every time a clause in ProbLog syntax is provided by clients, it gets transformed according to the recompilation specifics, and then accumulated in a *2P-Kt* Prolog theory through delegation.

The most relevant notion regarding the recompilation procedure itself, is that we adopt an approach based on *cascading* transformations. We advocate that the idea is optimal due to the high number of edge case that the recompilation engine is supposed to deal with. Accordingly, we implement the concept by introducing an internal object interface `ClauseMapper`, of which definition is visible in Listing 5.5. We created one concrete class implementation of such interface for each specific case of the recompilation. For reference, specific recompilation cases include mapping legacy Prolog clauses, ProbLog clauses, evidence-related predi-

Listing 5.5: Kotlin - `ClauseMapper` Object Interface

```
internal interface ClauseMapper {
    fun isCompatible(clause: Clause): Boolean
    fun apply(clause: Clause): List<Clause>
}
```

cates, and Annotated Disjunctions. Of those, the case of Annotated Disjunctions require specific reasoning, which we document in Section 5.2.4. The `ClauseMapper` interface is minimal and only exposes the two methods `isCompatible` and `apply`. The first returns true if the class is capable of recompiling the clause passed as argument, while the second applies the recompilation itself, accepting one clause as parameter and returning a list containing one or more clauses. In this scenario, we apply each mapping transformation in cascade with a given priority order, by invoking the `apply` method of the first case for which the `isCompatible` method returns true.

5.2.4 Annotated Disjunctions

As for the design presented in Section 4.4.2, we want to support Annotated Disjunctions by reproducing their semantics in an LP solver. To achieve this goal, we transform the multi-valued random variables defined by Annotated Disjunctions in a set of disjoint Boolean random variables through *binary-splits*. The disjunction semantics is guaranteed if the solver is capable of imposing a mutual exclusion constraint over the set of Boolean variables, so that only one of them is true in a given Explanation. We proceed to explain our implementative approach. First, the recompilation engine splits clauses with Annotated Disjunctions in a list of single-headed clauses so that the LP solver can accept them. Then, we need to ensure that mutual exclusion constraint gets respected by the solver during goal resolution. We achieve this by pre-constructing Explanations containing the binary splits right at the recompilation step, and by inserting them in the head of the splitted clauses.

Listing 5.6: ProbLog - Example of Annotated Disjunction for a Rolling Dice

```
1/6::dice(1); 1/6::dice(2); 1/6::dice(3); 1/6::dice(4);
   1/6::dice(5); 1/6::dice(6).
```

Listing 5.7: Prolog - Example of Recompilation of Annotated Disjunctions

```
dice(6, '<expl:bdd:1616357538>') :- true.
dice(5, '<expl:bdd:1544331835>') :- true.
dice(4, '<expl:bdd:-1230819326>') :- true.
dice(3, '<expl:bdd:2106363611>') :- true.
dice(2, '<expl:bdd:-707309726>') :- true.
dice(1, '<expl:bdd:1383151320>') :- true.
```

For example, consider the Annotated Disjunction of the fact shown in Listing 5.6, which describes the probability distribution of a rolling dice with six faces. The dice can roll on each face with the same probability, and the mutual exclusion property ensures that the dice rolls only on one face in the same solution. As discussed, our recompilation engine splits the annotation in six single-headed facts. Listing 5.7 shows the result of the recompilation for the annotation of Listing 5.6, which is written in Prolog. As visible, each head term contains an unusual literal as its last argument. Considering what described in Section 5.2.1 and Section 5.2.2, such a term represents a pre-constructed Explanation attached to each fact. The weird literal in each term is the result of a printout of the underlying instance of `ProbExplanationTerm`, that represents Explanations inside the Knowledge Base. Under the hood, each of those Explanations contain a Binary Decision Diagram constructed ad-hoc to preserve the mutual exclusion constraints of the binary splits. The recompilation engine generates the splitted Boolean random variables according to the definitions provided in Section 4.4.2. In Figure 5.2, we show an example of Binary Decision Diagram representing an Explanation with the pre-constructed binary split for one of the clauses of Listing 5.7.

The biggest takeaway of this approach is that the mutual exclusion semantics of

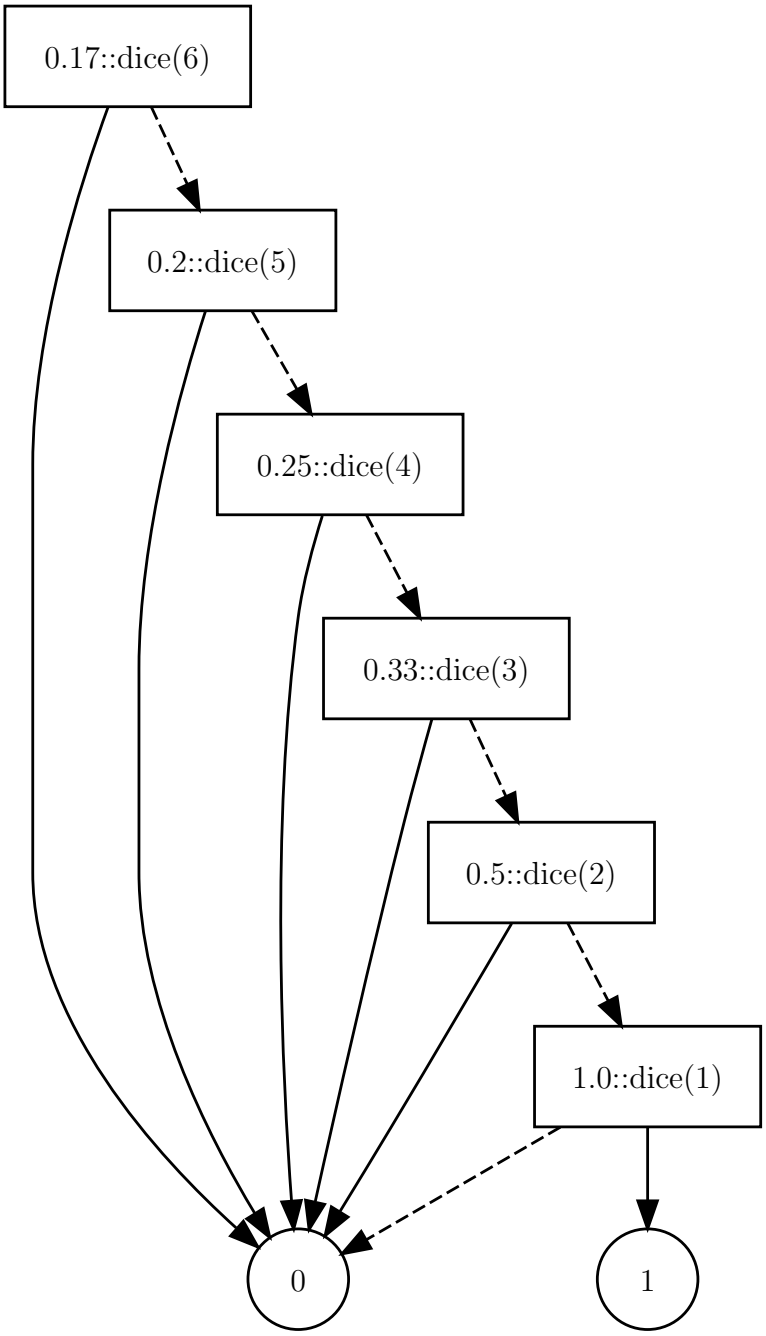


Figure 5.2: Binary Decision Diagram with Binary Splits

Annotated Disjunctions is implemented right away by the recompilation process, which is *static* and happens only once. Namely, the recompilation happens out of the scope of the goal resolution process of the solver, right when a clause is added to the Knowledge Base. As such, the LP solver is not responsible of any additional business logic at runtime, and the application of mutual exclusion constraints is out of its awareness. Note, the complexity of the problem is not avoided, but just delegated to the underlying Binary Decision Diagram data structures. By segregating the core responsibility inside the Binary Decision Diagram library, we are able to better control the hard computation to fine tune performance and to maintain the logic of the LP solver simple. Observe that this implementation has the defect of being inefficient for Annotated Disjunctions of large size. As the number of disjoint heads grow, the overhead of pre-constructing the binary splits and propagating them during goal resolution becomes more visible.

One of the ways with which we optimize this approach is by leveraging the `applyThenExpansion` operator to inspect the probability of a Binary Decision Diagram as soon as possible. During goal resolution, if the solver constructs a Binary Decision Diagram which is known having a probability of 0, then the solver discards the current solution by declaring a failure. This does not violate the PLP semantics, because the Explanation of solutions is constructed progressively as a conjunction. As such, we have a fast way to determine if a solution has a probability equal to 0, which makes it not influential from the standpoint of probabilistic computation. By definition, the mutual exclusion of the binary splits is meant to produce Explanations with null probability, so this optimization is helpful when dealing with Annotated Disjunctions implemented with our approach.

5.2.5 Prolog-Mode Optimizations

As stated in Section 3.1 and Section 4.4.1, we want to implement a logic solver that it is able of solving inference tasks for both PLP and LP, supporting the ProbLog and Prolog languages respectively. According to how we designed our project, this is easily attainable given the fact that a Prolog solver from *2P-Kt* is used under the hood to perform goal resolution in both cases. As such, full Prolog inference

support is achieved by simply ignoring the information related to Explanations and probability, and by disabling the solution grouping mechanic related to probabilistic computation. However, although we can obtain a functional Prolog solver in this way, it would become fairly less performant than any other pure-Prolog solution due to the additional complexity related to the meta-predicates and Binary Decision Diagram operations. As such, we adopted a set of workarounds aimed to improve the performance of goal resolution for pure-Prolog problems. Although this effort is not sufficient to match the performance of Prolog-only solvers, it is still enough to guarantee acceptable usability.

First, operations related to Binary Decision Diagrams are not ignored, but totally avoided instead. By checking a configuration argument, the `prob_expl_and` primitive recognizes if the current goal is being solved in Prolog-only mode, thus avoiding useless computations and substituting the output Explanation with a stub Binary Decision Diagram. The stub data structures is a simple terminal node encoding a probability equal to 1. This also means that Prolog and ProbLog Knowledge Bases can coexists in the same solver, and can be used interchangeably. In ProbLog mode, Prolog clauses are considered having probability distribution of 1, whereas in Prolog mode the distribution is ignored if the probability is not null.

A second optimization consists in reverting the semantics of *Negation as failure* in case of Prolog-only inference, so that at most one goal needs to be solved for the failure condition to be checked. This is fairly more efficient than iterating over all the solutions for a negated goal, like the solver is supposed to do in PLP-mode.

Finally, the solution grouping mechanic of probabilistic computation is avoided by short-circuiting the `prob_solve` meta-predicate to become equivalent to `prob`. The advantage of segregating the mechanic inside a single meta-predicate, is that it is easier to disable it at runtime by simply checking the solver configuration.

Chapter 6

Validation

In this chapter we provide an evaluation our project and some validation metrics. In Section 6.1 we present how we test our implementation. In Section 6.2 we provide a brief hands-on demonstration of how our solver can be used. Finally, in Section 6.3 we provide performance benchmarks of our solver for a set of PLP examples.

6.1 Testing Setup

In Chapter 5 we discuss some of the most relevant implementation choices that characterize the development of our project. Considering the first version of our PLP solver, we are interested in asserting if the requirements are satisfied. As the proposed solver is meant to deal with both LP and PLP tasks, the test suite we provide in the `prob-solve-problog` module is subdivided in two parts.

First, we want to assess that the solver maintains full support to traditional Prolog problems. This is not guaranteed, as supporting probabilistic computation introduces additional complexity that could affect the expected behaviour for LP tasks. As such, we insert the `test-solve` module of *2P-Kt* as an additional dependency in our codebase. `test-solve` contains a well-covering suite of fine-grain unit tests for Prolog solvers, asserting that solvers behave as expected with all the

features of standard LP. Given the above, our PLP solver successfully passes 327 out of the 361 unit tests in the suite, thus achieving an estimated success rate of 90.58%. However, the 34 failed tests are caused by inappropriate assertions, that are not meaningful in our implementation. As such, we adapted the suite to ignore the remaining 34 unit tests, by documenting our reasons in each one of them. Broadly, we recognize three main reasons for which we opt for ignoring unit tests. First, some tests expect a specific stack trace during goal resolution, which is altered in our solution due to the presence of the meta-predicates levels. Second, some tests assess the internal morphology of the Knowledge Base, which is meaningless in our case due to the recompilation step. Lastly, we ignore that fail due to a mismatch between error messages produced by our solver and the expected ones, as that is an implementation detail that does not affect the semantics. Overall, we advocate that the 90.58% success rate is not a meaningful measure in our case, as every test asserting the actual Prolog resolution semantics are passed.

For testing the behavior of our solver with PLP problems, we use a more coarse-grain approach. More specifically, we use a set of examples and experiments for the ProbLog language that we adapted from the website[7] of another state of the art proposal. In our case, those represents examples of acceptance tests assessing that our implementation computes results as expected. Each test exemplifies one or more features of PLP, thus effectively verifying the correct behavior of our solver. Although we acknowledge that this testing strategy is not as effective as a fine-grain unit test suite applied to every component of the solver, this still provides an acceptable feedback for our first version of the project. Moreover, this provided a fast spot-checking strategy that guided the development phase. Our solver successfully reproduces all the expected results of the examples in the suite, thus passing all the tests we adapted. Details regarding the performance achieved in each example are provided in Section 6.3.

Finally, we also introduced a minimal testing suite for the `bdd` module as well. Since the tests in `prob-solve-problog` already assess the correct functioning of the probabilistic solver, we have a good estimate of the correctness of the `bdd` module out of the box. In fact, considering how much Binary Decision Diagrams are correlated to probabilistic computation, our solver would not pass the accep-

tance test suite if the BDD library was faulty. On top of that, we also introduced few coarse-grain unit tests in the library module, aimed to verify the correctness of the three fundamental operators *and*, *or*, and *not*. Overall, the Binary Decision Diagram library successfully passes all our testing efforts.

6.2 Proof of Concept Demonstration

According to the design and implementation choices, which we discuss in Chapter 4 and Chapter 5 respectively, we shaped the software modules of our contribution to be easily adaptable in various application use-cases. In this section, we provide an brief demonstration to show the potential of our PLP solver in real-world scenarios.

In *2P-Kt*, the `ide` module bundles a minimal graphical user interface application conceived as a playground meant for users to experiment with the tools of the logic ecosystem. The IDE application executes on the JVM and its presentation layer is implemented with *JavaFX*, thus granting cross-platform support and high extensibility. As such, we implement a simple adaptation of the IDE that enables the usage of our solver, thus allowing users to experiment Probabilistic Logic Programming problems. This is a proof of concept of how our proposition can be integrated in various application domains with ease.

For our demonstration, we select one of the examples with use for testing purposes in Section 6.1. The problem consists in modeling a *probabilistic graph*, which are types of graph where the existence of some edges is uncertain. In this use case, PLP can be leveraged to estimate with which probability a path between two nodes exists. In the example, the Knowledge Base of Listing 6.1 models the probabilistic graph represented in Figure 6.1. The IDE accepts Knowledge Bases both from files and from text input, and is designed to resemble a simple text editor. Also, control panel for logic resolution tasks is shown on the bottom. We can specify a query and submit it to an underlying logic solver. Once found, the query solutions are shown in a list view. Also, a tab view allow the inspection of the internal state of the solver. In this example, we want to find all the feasible path in the graph, and calculate their probability. A screenshot of the IDE solving

Listing 6.1: ProbLog - Example of Probabilistic Graph Modeling

```
0.6::edge(1,2).
0.1::edge(1,3).
0.4::edge(2,5).
0.3::edge(2,6).
0.3::edge(3,4).
0.8::edge(4,5).
0.2::edge(5,6).

path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), Y \== Z, path(Z,Y).
```

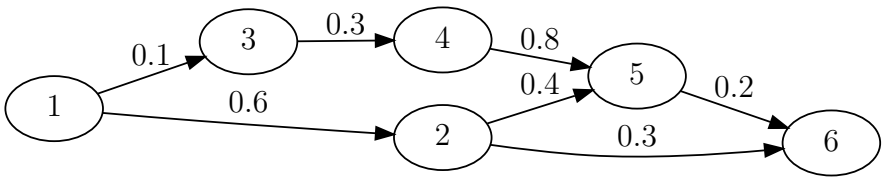


Figure 6.1: Example of Probabilistic Graph

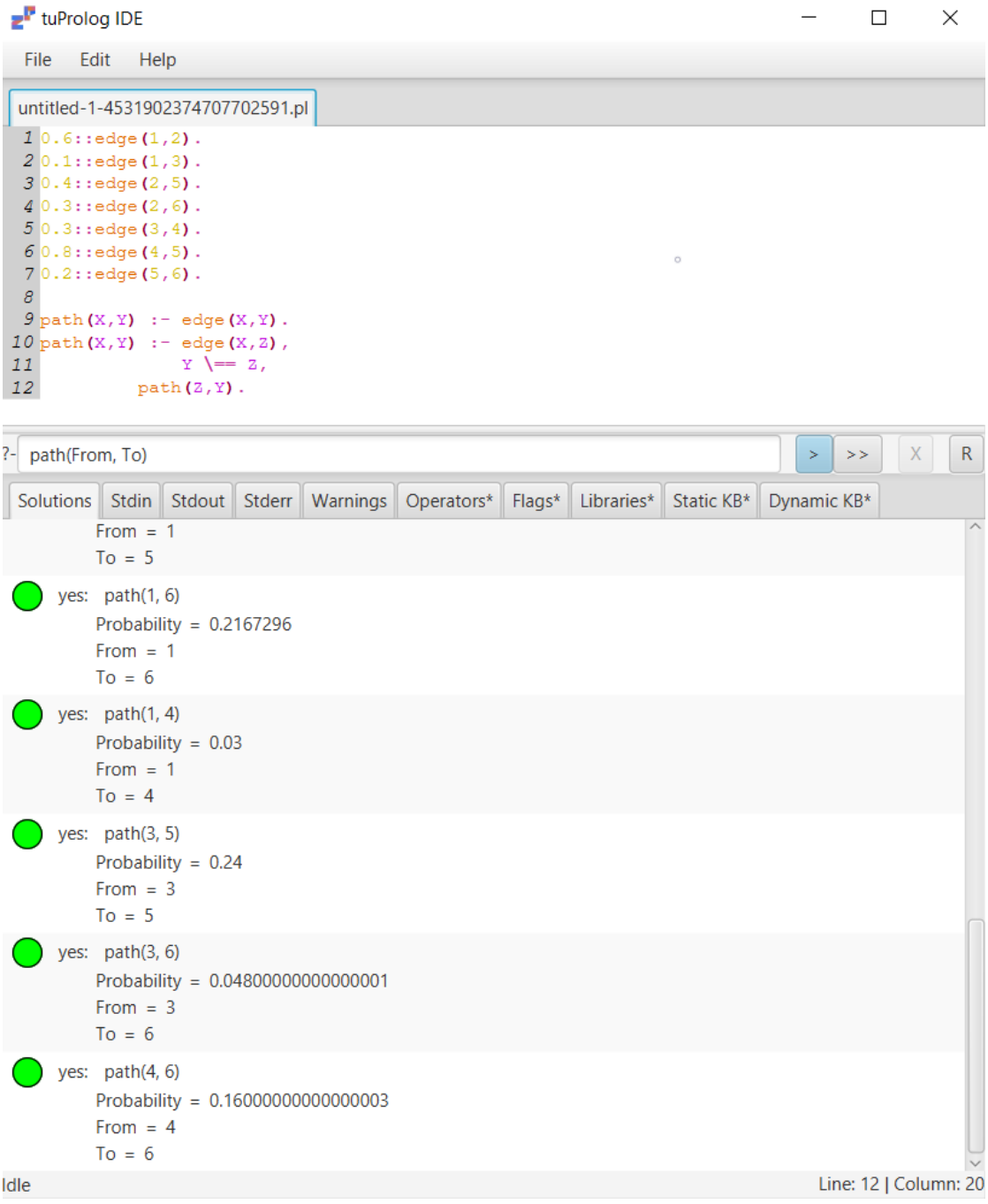


Figure 6.2: *2P-Kt* IDE Solving a PLP Query

our query is shown in Figure 6.2. As visible, the probability values are presented in the list view attached to their relative solutions.

Furthermore, our proposal allows a deep manipulation of the Explanation behind each solution. Most proposals at the state of the art rely on external packages for this task, whereas we propose an internal implementation through our library for Binary Decision Diagrams. As such, we can achieve more expressive graphical representations in the presentation layer. For instance, we can attach a printable view of the Binary Decision Diagram that originated a given solution, with direct reference to the probabilistic clauses contained in the Knowledge Base. For that, we implement an additional operator in our library that serializes Binary Decision Diagrams in a graph-like image representation format. Under the hood, we implemented the new operator by leveraging the *visitor pattern* to serialize the diagram following the specific of the DOT[6] language. Accordingly, users are able to inspect and render the Binary Decision Diagram behind each solution. For instance, Figure 6.3 show a Binary Decision Diagram generated by our solver that characterizes the solution `path(1,6)` of the sample query. Figure 5.2, shown in the previous chapter, is rendered in the same way. Note, the diagrams we render are optimized for better computation performance. For debugging purposes, certain optimizations can be disabled in order to obtain more expressive diagram picture, which can help understand the rationale of specific solutions.

6.3 Performance Benchmarks

In this last section we want to provide some information regarding the performance and execution time of our PLP solver in probabilistic logic inference tasks. Our experiment uses some of the ProbLog examples[7] that we adapted in our testing suite as we mention in Section 6.1. We run the experiments on both the target platforms supported by *2P-Kt*, which are the JVM and JavaScript.

The benchmarks have been measured on a local physical machine equipped with a *Intel Core i7-10750H* CPU with 6 cores and 12 threads, 12MB of L3 cache, and a clock frequency variable from 2.6GHz to 5GHz. Also, the setup has 32GB of

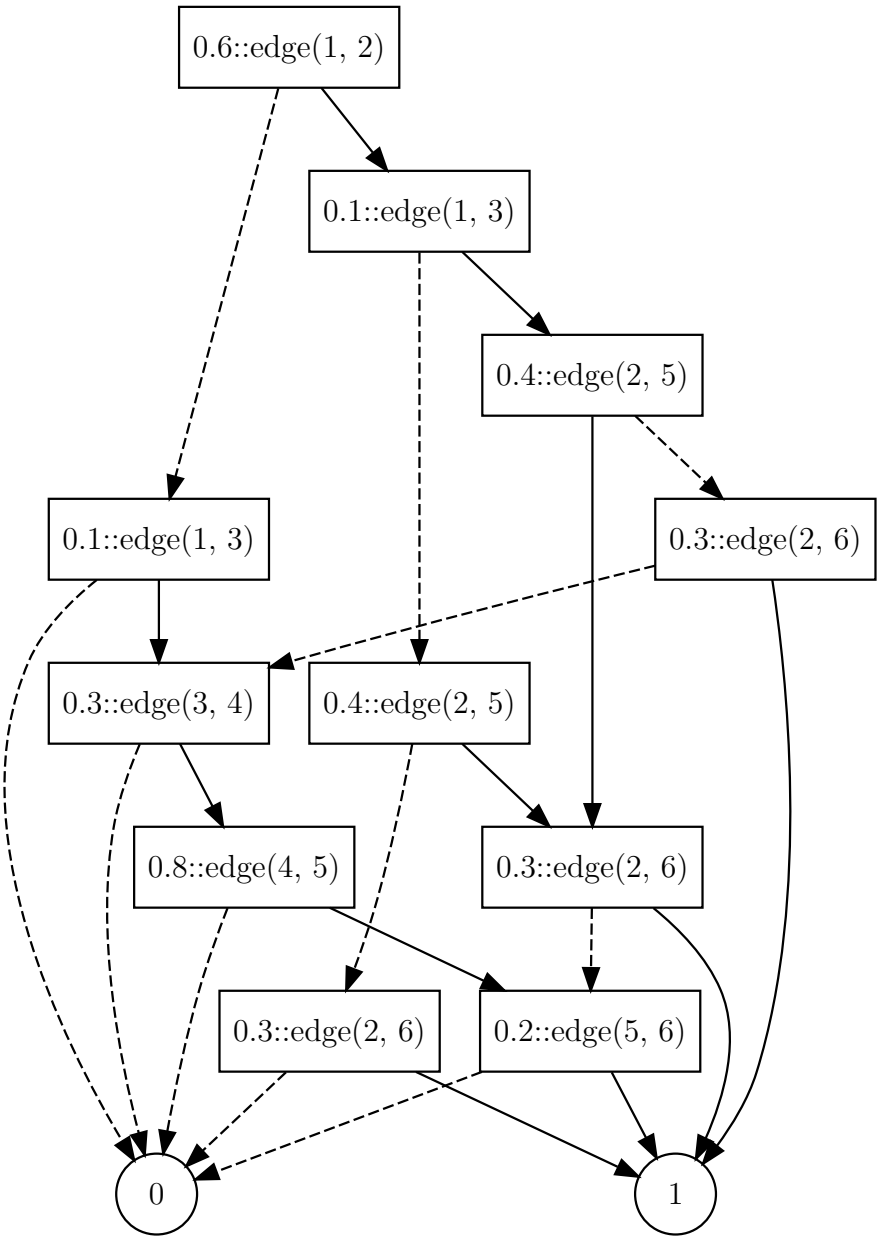


Figure 6.3: Binary Decision Diagram Constructed by our PLP solver

DDR4-2933MHz RAM divided in two modules of 16GB, and a *NVIDIA GeForce GTX 1650Ti* GPU with 4GB of memory GDDR6. Furthermore, the installed operating system is *Microsoft Windows 10 Pro* x64 at version 10.0.19042 and build 1083. The JVM binaries are executed on a Java 11 *OpenJDK* 64-Bit JVM, whereas the JavaScript target runs on *NodeJs* version 14.15.4. At the time of writing, Kotlin version 1.4.20 is used for the compilation.

The results of our experiment are summarized in Section 6.3. Execution times of both the JVM and JavaScript executables are reported for each example. The experiment shows interesting results. First, we have a first insight of the complexity of probabilistic computation, and how it varies depending on the specific tasks and its required resolution depth. Second, it is clearly visible that JavaScript artifacts are fairly less performant than their JVM counterpart. This discrepancy is determined by a multitude of factors, which are not totally bound to our specific solver implementations. Our observations highlight the need of further research in these topics, and we look forward to achieve better performance optimizations in future developments of this project.

Table 6.1: Execution times of the solver on PLP inference examples

Example	**Test Name**	**JVM**	**JS**
Tossing Coins	Basic ProbLog	29ms	40ms
	Noisy-or	2ms	18ms
	First-order	14ms	32ms
	Probabilistic clauses	5ms	14ms
Bayesian Networks	Probabilistic facts	13ms	120ms
	Probabilistic clauses	7ms	55ms
	First-order	95ms	212ms
	Annotated disjunctions	63ms	269ms
Rolling Dice	Annotated disjunctions	3ms	58ms
	Negation as failure	8ms	68ms
	Negation as failure (2)	3ms	20ms
	Arithmetic expressions	40ms	286ms
	Arithmetic expressions (2)	402ms	2669ms
Probabilistic Graphs	Probabilistic graph	62ms	170ms
Monty Hall	Monty Hall Puzzle	113ms	455ms
	Alternative representation	65ms	98ms

Chapter 7

Conclusions

In this thesis we explore innovative ideas in the scope of Probabilistic Logic Programming. Our project is in an early stage, and still not suitable to be compared with other proposals in the field. However, our contribution consists in the experimentation of alternative models and approaches, which we believe to be a solid value proposition towards the growth of this research field. As such, we advocate that the goals of this thesis have been accomplished.

By working on top of the *2P-Kt* symbolic AI ecosystem, we overcome the usability and portability constraints that affect most other state of the art proposals. In fact, our artifacts can be deployed in all the platforms supported by *2P-Kt*, which at the time of writing are JVM and JavaScript and are expected to grow in count in the near future. We design and implement a logic solver suitable for inference tasks for both LP and PLP problems. Our solution is flexible, and allows a frictionless transition from the traditional logic paradigm to the novel probabilistic one. Also, the proposed solver can handle both pure-logic and probabilistic Knowledge Bases interchangeably. Coherently, complete backward compatibility with Prolog inference is successfully accomplished and effectively verified through a well-covering suite of unit tests. We propose a unique multi-paradigm design that combines object-oriented modeling and logic programming, thus opening new horizons in terms of ad-hoc optimizations and codebase management. Accordingly, our solver is developed on top of an hybrid approach that

successfully combines the benefits of meta-interpreters and low-level engine implementations by also mitigating the downsides of both. Additionally, the proposed solver is compliant with the most common features at the state of the art of Probabilistic Logic Programming. Our software supports the ProbLog language, solves inference tasks without approximation, calculates probabilities with evidence, and accepts Annotated Disjunctions.

Additionally, we introduce a new library for the manipulation of Binary Decision Diagrams. Our solution is written in *pure-Kotlin* with no external dependencies, with the attempt of pioneering the support of this kind of data structures on multiple platforms. We also overcome the need of depending on an external package for Binary Decision Diagrams manipulation, which is a common constraint in most state of the art proposals. Accordingly, the increased control over the library codebase enable the introduction of ad-hoc optimizations for our use case. As our library is designed to be a minimal and standalone module, it can flexibly be included in other projects that require the manipulation of Binary Decision Diagrams. Coherently, our design is open to platform-specific implementations with the purpose of suiting a wide spectrum of future use cases.

Ultimately, one of the top priorities of this research effort is to leave the door open to future developments. Our design is purposely abstract, and we endorse the future exploration of alternative implementation ideas. Among the others, some of the future directions we envision include: supporting *Approximate Inference*, approaching additional *Knowledge Compilation* data structures, implementing the solver as a standalone optimized *Finite State Machine*, and adopting alternative resolution strategies such as *Tabled Execution*. Conclusively, additional work will be required for evolving this first version of the project to a more mature production-ready software. Moreover, we envision our Binary Decision Diagram library to grow in the open source community, and serve as a proof of concept for the potential benefits of object-oriented and platform-agnostic approaches to the subject.

Bibliography

[1] George Boole. *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. Walton and Maberly, London, 1854.

[2] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[3] Haim Cohen, John Whaley, Jorn Wildt, and Nikos Gorogiannis. BuDDy. `https://sourceforge.net/p/buddy/`.

[4] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin Heidelberg, 2001. 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

[5] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory Pract. Log. Program.*, 15(3):358–401, 2015.

[6] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SoftwareE - Practice and Experience*, 30(11):1203–1233, 2000.

[7] KU Leuven DTAI Research Group. ProbLog - probabilistic programming. `https://dtai.cs.kuleuven.be/problog/index.html`.

[8] JetBrains. Kotlin programming language. `https://kotlinlang.org/`.

[9] Alberto Lovato, Damiano Macedonio, and Fausto Spoto. A thread-safe library for binary decision diagrams. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, volume 8702 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2014.

[10] David Poole. Probabilistic horn abduction and bayesian networks. *Artif. Intell.*, 64(1):81–129, 1993.

[11] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2462–2467, 2007.

[12] Fabrizio Riguzzi. A top down interpreter for LPAD and cp-logic. In Roberto Basili and Maria Teresa Pazienza, editors, *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, 10th Congress of the Italian Association for Artificial Intelligence, Rome, Italy, September 10-13, 2007, Proceedings*, volume 4733 of *Lecture Notes in Computer Science*, pages 109–120. Springer, 2007.

[13] Fabrizio Riguzzi. *Foundations of Probabilistic Logic Programming*. River Publishers, Gistrup, Denmark, 2018.

[14] Fabrizio Riguzzi and Terrance Swift. The PITA system for logical-probabilistic inference. In Stephen H. Muggleton and Hiroaki Watanabe, editors, *Latest Advances in Inductive Logic Programming, ILP 2011, Late Breaking Papers, Windsor Great Park, UK, July 31 - August 3, 2011*, pages 79–86. Imperial College Press / World Scientific, 2011.

[15] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[16] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, pages 715–729. MIT Press, 1995.

[17] Taisuke Sato and Yoshitaka Kameya. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 1330–1339. Morgan Kaufmann, 1997.

[18] Fabio Somenzi. CUDD: Cu decision diagram package. `https://vlsi.colorado.edu/~fabio/`.

[19] Arash Vahidi. JDD: a pure java bdd and z-bdd library. `https://bitbucket.org/vahidi/jdd`, 2003.

[20] Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic programs with annotated disjunctions. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 431–445. Springer, 2004.