ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

2p-kt: A Kotlin-based, Multi-Platform Framework for Symbolic AI

Tesi di laurea magistrale

Relatore
Prof. ANDREA OMICINI

Co-relatore
Dott. GIOVANNI CIATTO

Presentata da
ENRICO SIBONI

Terza Sessione di Laurea
Anno Accademico 2018-2019

Abstract

Today complex software systems are typically built as aggregates of heterogeneous components, where symbolic AI may effectively help facing key issues such as intelligence of components and management of interaction. However, most solutions for symbolic manipulation are currently either proof of concept implementations or full-fledged monolithic runtimes, mostly targeting a single platform or a specific problem. Among the many, two decades ago, the *tuProlog* engine proposed a flexible and modular architecture on top of a portable platform – namely, the JVM – which should have overcome the aforementioned problems. Sadly, the technology of the time forced some design choices which are nowadays limiting its development and its porting on other platforms – critical for modern AI –, such as JavaScript (JS) or iOS. For such reasons in this thesis we propose (the design and development of) *2p-kt*, an object-oriented, modular engine for symbolic manipulation written in *pure Kotlin* – thus supporting compilation on several platforms, there including JVM, and JS –, coming with a natural support for Prolog-like SLD resolution but virtually open to other resolution strategies. *2p-kt* design is conceived to maximise components deployability, lightweightness, reusability and configurability, while minimising their mutability, and keeping its architecture open to the injection of new Prolog libraries or resolution strategies.

To my dear family

Acknowledgements

I would like to thank Prof. Andrea Omicini for offering me the opportunity to carry out this work and for his teachings throughout the last two years. Furthermore I would like to thank Dott. Giovanni Ciatto for supervising my work and for his patience, his great support, and his trust in me.

I would like to express my gratitude to my mother, Loretta, and my sister Federica, for their everlasting love and for always supporting me and my decisions. I will be grateful forever for your love.

Contents

Chapter 1

Introduction

Nowadays, the needs for intelligent pervasive systems are opening new perspectives for logic programming and symbolic Artificial Intelligence (AI). At the same time, they also impose novel requirements for logic-based development frameworks, which should allow logic programmers to easily integrate Prolog chunks with other programming languages, and to be possibly executed by a multiplicity of platforms, including web, mobile, and native ones. However, most solutions for symbolic manipulation are currently either proof of concept implementations or full-fledged monolithic runtimes, mostly targeting a single platform or a specific problem.

Among the many Prolog technologies proposed so far, the *tuProlog* [1] engine introduced an innovative, flexible and modular architecture on top of a portable platform – namely, the JVM – which should have overcome the aforementioned problems. However, the technology of the time (early 2000s) forced some design choices which proved themselves inadequate in the next years. To target such issue, in this thesis, we aim at rebooting the *tuProlog* project preserving good design choices made at the time while addressing some issues that we identify as critical. In doing so, we also increase the modularity of *tuProlog*, as well as its portability on other platforms.

Accordingly, in this thesis we discuss the design, architecture, and development of *2p-kt*, that is, an object-oriented, modular engine for symbolic manipulation written in *pure Kotlin*, featuring a native multi-platform support. *2p-kt* design is conceived to maximise components deployability, lightweightness, reusability and configurability, while minimising their mutability, and keeping its architecture open to the injection of new Prolog libraries or resolution strategies.

The *2p-kt* project is divided in several modules, each one featuring a Prolog macro-functionality. For instance, it comprehends *(i)* a module addressing knowledge representation, *(ii)* a module for Prolog unification, and *(iii)* a general module for Prolog resolution. In particular, the latter module comes in two specific implementations: a *classic* one – emulating the functioning of the original *tuProlog* engine – and an *experimental* one—in which we (successfully) assess the implementation of a Prolog engine in a functional fashion, where immutability reigns supreme.

Finally, we endow *2p-kt* of an extensive testing suite, through which its functioning is validated. This testing suite, if correctly maintained, will enable *2p-kt* project developers to possibly exploit some kind of agile development. Indeed, every principal module is finely tested, providing both a "living" documentation and a way to intercept possible regressions while modifying existing code.

Accordingly, the remainder of this thesis is structured as follows. In chapter 2, we provide a technical background for what concerns Prolog and logic programming. We also describe *Kotlin* in detail, being the programming language we rely on for the realisation of our project. In chapter 3, we describe how the *2p-kt* project came to life, which are its predecessor *tuProlog* merits and defects, defining a set of requirements for the system we wish to realise. In chapter 4, we model the system and its components, describing their structure, interaction, and behaviour. Then for each module we discuss its detailed design choices. In chapter 5, we provide an overview on the system implementation, describing all the elements of interest. In chapter 6, we describe the system validation carried out through automated tests, of which an implementation overview is also given.

Chapter 2

State of the Art

In this chapter, we provide a technical background briefing on the arguments of interest of this thesis. In section 2.1, we describe Prolog language summarising its basic concepts and the abstract computation model. Furthermore, a brief overview of Prolog implementations in provided in section 2.2. Finally, in section 2.3, we introduce *Kotlin* language and its main features.

2.1 The Prolog language

Prolog is a logic programming language leveraging on a small set of basic – yet powerful – mechanisms, including pattern-matching, tree-based data structures, unification, and automatic backtracking.

It has its roots in first-order logic, and unlike many other programming languages, Prolog is intended primarily as a *declarative* programming language.

In this section we provide an introduction to logic programming and Prolog.

2.1.1 Brief history

Logic programming has its root in automated deduction and first-order logic. The former was studied by Kurt Gödel and Jacques Herbrand in the 1930s [2], whose work can be seen as the origin of the "computation as deduction" paradigm. The latter was first introduced by Gottlob Frege and subsequently modified by

Giuseppe Peano and Bertrand Russell throughout the second half of the 19th century [3].

These works led Alan Robinson, in 1965, to the invention of a resolution principle based on the notion of *unification* [4], which makes it possible to prove theorems of first-order logic and thus compute with logic.

The final steps towards logic programming were made in the 1970s by *(i)* Robert Kowalski, who introduced the notion of logic programs with a restricted form of resolution, compared to the one proposed by Robinson, as a feasible proof search strategy [5]; *(ii)* Alain Colmerauer and its team, who worked on a practical realisation of the idea of logic programs, giving birth to Prolog.

Since then, for about ten years, there was an explosion of dialects of Prolog, implementing various different extension of it, even changing its semantics. The need for a standard was becoming increasingly important.

The ISO Prolog Standard is the result of another ten years of international discussion which began in 1985, which ended in 1995 [6].

The main difference between logic programming and the other programming paradigms is the way computation is conceived. In commonly used programming paradigms, given an input (i.e. expression), the computation is the application of fixed rules that produce an output. In logic programming, given an input (i.e. conjecture), the computation consists in searching a proof to that input in a solution space, following a predefined search strategy.

2.1.2 Concepts summary

The logic programming paradigm substantially differs from other programming paradigms. When stripped to the bare essentials it can be summarised by the following three features [7]:

- Computing takes place over the domain of all ***terms*** defined over a "universal" language.
- Values are assigned to variables by means of automatically generated ***substitutions***, called ***most general unifiers***. These values may contain vari-

ables, called *logical variables.*

- The control is provided by a single mechanism, called *automatic* ***backtracking***.

Terms. A term is defined recursively by applying a finite number of times the following syntax rules:

$$\begin{array}{rcl}\langle Term\rangle & := & \langle Variable\rangle \mid \langle Constant\rangle \mid \langle Compound\rangle \\ \langle Constant\rangle & := & \langle Number\rangle \mid \langle Atom\rangle \\ \langle Number\rangle & := & \langle Integer\rangle \mid \langle Float\rangle \\ \langle Compound\rangle & := & \langle Atom\rangle\texttt{(}\langle Args\rangle\texttt{)} \\ \langle Args\rangle & := & \langle Term\rangle \mid \langle Args\rangle\texttt{,}\ \langle Args\rangle\end{array}$$

So a term can be a *variable*, a *constant* (an *atom* or a *number*) or a *compound term*, i.e. a *functor*, whose name is an atom, together with its *arguments* (a non-empty sequence of comma-separated terms between parentheses) [6].

The objects used to build terms belong to the following disjoint sets:

- The *variables* denoted by a sequence of letters, digits or underscore characters, beginning with an upper case letter or by the underscore character. Variables are used to refer to any object.

  In the reminder of this thesis we denote by $\mathcal{L}(\langle Variable\rangle)$ the set of all *variable* symbols, and their instances will be written as mono-spaced strings.

  For instance, `X1`, `_y2`, `Var`, `Atom`, `A_variable` are examples of variables in $\mathcal{L}(\langle Variable\rangle)$.

- The *atoms*, denoted by a sequence of letters, digits or underscore characters beginning with a lower case letter or a sequence of arbitrary characters in single quotes. Atoms formed with any characters, may be unquoted if there is no ambiguity with other sets. Atoms are used in particular to denote names of predicates or functors.

  In the reminder of this thesis we denote by $\mathcal{L}(\langle Atom\rangle)$ the set of all *atom* symbols, and their instances will be written as mono-spaced strings.

For instance, `x1`, `atom`, `'1'`, `'This is a single atom'`, `this_too`, and `'    '` are examples of atoms in $\mathcal{L}(\langle Atom\rangle)$, whereas the *null atom* is denoted `''`.

- The *numbers* are partitioned into *integers* (negative and positive integers) and *floating-point numbers*, or in short *floats*. They will be denoted as usual.

  In the reminder of this thesis we denote by $\mathcal{L}(\langle Number\rangle)$ the set of all *number* symbols, and their instances will be written as mono-spaced strings.

  `33`, `-33`, `33.0`, `-0.33E+02` are examples of integer and float numbers in $\mathcal{L}(\langle Number\rangle)$.

- The *compound terms* are characterised by the name of their functor, called the *functor*, and the number of their arguments, called *arity*. The outermost functor of a compound term is called its *principal functor*.

  In the reminder of this thesis we denote by $\mathcal{L}(\langle Compound\rangle)$ the set of all *compound term* symbols, and their instances will be written as mono-spaced strings.

  For instance, `f(_var1, g(Var2, a), b_c)` is an example of a compound term in $\mathcal{L}(\langle Compound\rangle)$, where the functor `'f'` has arity 3 and `'g'` arity 2. `'f'` is the principal functor.

Atoms and numbers form the *constants*, also called *atomic terms*. `foo`, `'Bar'`, `1.2`, `-0.33E+02` are atomic terms. Moreover, atoms can be viewed as functors of arity 0.

A term is said to be *ground* if it has no variable in it. The (set of the) *variables of a term* is the set of all the variables occurring in a term. It is empty if the term is ground.

In general, terms have no *a priori* meaning: each one of them can be associated to a domain-specific entity by means of *pre-interpretation*.

In the reminder of this thesis we denote by $\mathcal{L}(\langle Term\rangle)$ the set of all *term* symbols, for which holds the relation $\mathcal{L}(\langle Term\rangle) = \mathcal{L}(\langle Variable\rangle) \cup \mathcal{L}(\langle Atom\rangle) \cup \mathcal{L}(\langle Number\rangle) \cup \mathcal{L}(\langle Compound\rangle)$

Substitution. A *substitution* is a mapping from variables to terms. It is assumed that a substitution is the identity mapping with the exception of a finite number of variables. The notation:

$$\{V_1/t_1,\ \ldots,\ V_n/t_n\}$$

denotes a substitution where all variables are mapped to themselves, with the exception of $V_1, \ldots, V_n$ where $V_i \in \mathcal{L}(\langle \mathit{Variable} \rangle)$ is mapped to a term $t_i \in \mathcal{L}(\langle \mathit{Term} \rangle)$, different from V_i, for $1 \leq i \leq n$.

The *identity substitution* corresponds to the identity mapping. Therefore its (finite) representation is the empty set and it is called the *empty substitution.*

Substitutions denote *bindings.* Given $V \in \mathcal{L}(\langle \mathit{Variable} \rangle)$ and $t \in \mathcal{L}(\langle \mathit{Term} \rangle)$, such that $\{V/t\}$ belongs to some substitution, the variable V is said to be *bound to* t by some substitution. If t is not a variable, V is said to be *instantiated* (by some substitution).

A substitution σ is naturally extended to a function on terms by defining:

$$\sigma(\mathtt{f}(t_1,\ \ldots,\ t_n)) = \mathtt{f}(\sigma(t_1),\ \ldots,\ \sigma(t_n))$$

In particular for any constant c, $\sigma(c)$ is defined to be c. Substitutions will be represented by Greek letters acting as postfix operators, hence the application of a substitution σ to a term t is denoted: $t\sigma$ (instead of $\sigma(t)$), and $t\sigma$ is called an instance of t.

A term $s \in \mathcal{L}(\langle \mathit{Term} \rangle)$ is an instance of the term t if there exists a substitution σ such that $s = t\sigma$. For example, applying substitution $\{$`X/a`, `Y/f(X, b)`, `Z/g(X, Y)`$\}$ to term `f(Y, Z)` gives the term `f(f(X, b), g(X, Y))` which is an instance of it.

As substitutions are mappings they can be composed. Let θ and σ be the substitutions represented, respectively, by $\{X_1/s_1,\ \ldots,\ X_n/s_n\}$ and by $\{Y_1/t_1,\ \ldots,\ Y_m/t_m\}$ where

$$(\{X_1,\ \ldots,\ X_n\} \cup \{Y_1,\ \ldots,\ Y_m\}) \subseteq \mathcal{L}(\langle \mathit{Variable} \rangle)$$

and

$$(\{s_1,\ \ldots,\ s_n\} \cup \{t_1,\ \ldots,\ t_m\}) \subseteq \mathcal{L}(\langle \mathit{Term} \rangle)$$

The representation of θ composed with σ can be obtained from the set

$$\{X_1/s_1\sigma,\ \ldots,\ X_n/s_n\sigma,\ Y_1/t_1,\ \ldots,\ Y_m/t_m\}$$

by removing all bindings $X_i/s_i\sigma$ for which $X_i = s_i\sigma$, with $1 \leq i \leq n$, and all bindings Y_j/t_j such that $Y_j \in \{X_1,\ \ldots,\ X_n\}$, with $1 \leq j \leq m$.

The notion of instance extends to substitutions. A substitution σ is an instance of the substitution θ if there exists a substitution μ such that $\sigma = \theta\mu$.

A substitution is *idempotent* if successive applications to itself yield the same substitution. Examples: a ground substitution is trivially idempotent; the following substitution is idempotent {`X/a`, `Y/f(T, b)`, `Z/g(a, U)`}.

A term s is a *renaming* of the term t, *with regard to* (*w.r.t.*) a set of variables $S \subseteq \mathcal{L}(\langle Variable\rangle)$, if s is obtained from t by mapping different variables into different variables and no variable in term s belongs to S. Example: the term `f(T, U, T)` is a renaming of the term `f(X, Y, X)` w.r.t. the set {`X`, `Y`, `Z`, `W`, `P`, `Q`}, but `f(Y, U, Y)` is not.

Most General Unifier. A substitution σ is a unifier of two terms if the instances of these terms by the substitution are identical. Formally, σ is a unifier of $t_1 \in \mathcal{L}(\langle Term\rangle)$ and $t_2 \in \mathcal{L}(\langle Term\rangle)$ if $t_1\sigma$ and $t_2\sigma$ are identical. It is also a *solution* of the equation $t_1 = t_2$, and, by analogy, it is called the unifier of the equation. The notion of unifier extends straightforwardly to several terms or equations (i.e. all terms or equation members become identical). Terms or equations are called *unifiable* if there exists a unifier for them. They are *not unifiable* otherwise.

A unifier is a *most general unifier* (*MGU*) of terms if any unifier of these terms is an instance of it. A most general unifier always exists for terms if they are unifiable. There are infinitely many equivalent unifiers through renaming.

The process of solving an equation between terms is called *unification.* This process adheres to a simple set of rules [6]:

i Given two non-initialised variables $\{X,\ Y\} \subset \mathcal{L}(\langle Variable\rangle)$, with $X \neq Y$, they unify with unifier $\{X/Y\}$

ii Given a non-initialised variable X and a non-variable term $u \in \mathcal{L}(\langle Term\rangle) \setminus \mathcal{L}(\langle Variable\rangle)$, they unify with $\{X/u\}$

iii Given two constants, they unify if and only if they are the same constant, with empty unifier

iv Given two terms, they unify if and only if they have the same functor and arity, and their arguments unify recursively.

For instance, given the expression

$$\texttt{g(a, Y)} = \texttt{g(X, Z)}$$

the substitution {`X/a`, `Y/Z`} represents the *MGU* and is less constraining than substitutions {`X/a`, `Y/b`, `Z/b`} and {`X/a`, `Y/a`, `Z/a`}, which are instances of the first. But the equation `X` = `f(X)` has no solution (no unifier for `X` and `f(X)`).

In logic programming, atomic actions are equations between terms, which are executed through the unification process. In other words, the unification process is the basic operation on which the proof-search strategy relies to check whether a proof to a given conjecture has been found or not. As stated before, the proof-search strategy employed in logic programming is the one proposed by Kowalski, known as *SLD-resolution* principle. So, in order to understand how computation works in its entirely, in the following section we briefly explain how the SLD-resolution principle works, after a short but necessary introduction to *Horn Clauses* [8].

2.1.3 Horn clauses and SLD-resolution principle

In logic, sentences are called *prepositions*, which can be written through *predicates*. Predicates are essentially structured terms and, in logic, are called atoms: if `p` is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms, then $\texttt{p(}t_1, \ldots, t_n\texttt{)}$ is an atom.

If the literal A is a logic atom, then the logical formula "A" states that A is true, while "$\neg A$" states that A is false. Literals can be combined through *logical* *connectives* to build more complex logic formulas. If A and B are literals, they can be combined through: *conjunction* (e.g. $A \wedge B$, both A and B are true); *disjunction* (e.g. $A \vee B$, either A or B is true); *implication* (e.g. $A \rightarrow B$, if A is true then B is true); *equivalence* (e.g. $A \leftrightarrow B$, A is true if and only if B is true).

A *logic clause* is a finite disjunction of literals. Given n literals $A_1, \ldots, A_n$ and m literals $B_1, \ldots, B_m$, the formula

$$A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_m$$

is a logic clause with n “positive” literals and m “negative” literals, which is often written as

$$A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$$

A conjunction of logic clauses is called a *Clausal Normal Form* (CNF).

Finally, *Horn clauses* are logic clauses with at most one positive literal. One example of them is represented by *definite clauses*, which are logic clauses with exactly one positive literal (e.g. $A \leftarrow B_1, \ldots, B_m$). Furthermore, there are two kinds of definite clauses: *unitary clauses*, which have no negative literals (e.g. $A \leftarrow$) and *definite goals*, which have no positive literals (e.g. $\leftarrow B_1, \ldots, B_m$). Both of them are Horn clauses too.

We are interested in Horn clauses since the SLD-resolution principle works as follows: given a logic program P written as a CNF of Horn clauses and a formula F, it shows that it is possible to compute (by contradiction) whether P logically entails F. So, in order to exploit it, logic programs are written as CNF of Horn clauses. In a logic program, definite clauses are called *rules*, unitary clauses are called *facts* and definite goals are simply called *goals*. Rules and facts make up the “source code” of a logic program, while goals represents the input that must be proven.

The SLD-resolution principle, as the one proposed by Robinson, proceed by *contradiction*: it negates the formula F and succeeds if it fails to prove it against the program P. To prove a goal G with respect to a program P, the principle works as follows:

1. It looks for a logic clause in P whose head (i.e. the positive literal) unifies with G (*clause-choice*).
2. There are three possible outcomes to this search:
   (a) no clause could be found: in this case the resolution fails.
   (b) a rule R of form $A \leftarrow B_1, \ldots, B_m$ is found: being θ the MGU of G and R, then the proof of G succeeds, and is represented by $G\theta$, if it is possible to further prove the sub-goals $B_1\theta, \ldots, B_m\theta$, where $B_i\theta$ is the application of θ to B_i, with $1 \leq i \leq m$. Thus, these sub-goals represent now the current goal.
   (c) a fact F is found: being θ the MGU of G and F, no sub-goals are added to the current goal and the solution is represented by $F\theta$.
3. If the current goal is empty, the resolution ends successfully (SLD refutation). Otherwise, a selection rule is adopted to choose the next sub-goal to prove (*predication-choice*), starting back from point 1. If the current goal never gets emptied, the resolution does not terminate.

Backtracking. Resolution relies on *automatic backtracking*: at a given point τ in the resolution process, for the current goal/sub-goal G there could be more than one clause in the program whose head matches with it; after choosing one of them, if the resolution of the related sub-goals fails, the process automatically backtracks to point τ , where another clause is chosen. Backtracking is performed until at least one "candidate" clause is still present.

The SLD-resolution principle is often called a "proof search" because of its non-deterministic nature: as just stated, while looking for a clause in the program P, there could be more than one whose head matches with the current goal (*or-nondeterminism* brought by *clause-choice*); furthermore, we have non-determinism even when the resolution process has to choose the next goal to prove among several sub-goals (*and-nondeterminism* brought by *predication-choice*). The way these forms of non-determinism are dealt with, represents how the proof-search proceeds in the solution space.

In Standard Prolog those selection algorithms are fixed as follows, representing the *standard computation rule* [6]:

- *predication-choice* always selects the leftmost predication in current goal G
- *clause-choice* selects “unifiable” clauses according to their sequential order in P

This makes Prolog computations deterministic, a basic feature for every programming language.

2.2 Prolog implementations

In the last decads, several Prolog implementations have been proposed. Some of them are radically different from others, with different syntax and different semantics (e.g. Visual Prolog) [9]. The code portability has always been a problem but, after the production of ISO Prolog standard, at least code that strictly conforms to ISO-Prolog is portable across ISO-compliant implementations.

In this section we will provide a brief overview of current Prolog implementations.

BProlog [10] is a Prolog engine available as a freeware for non commercial use at `http://www.picat-lang.org/bprolog/`. It consists of a versatile and efficient Constraint Logic Programming (CLP) system. It supports Unix and Windows platforms. The project reached version `8.1` and it has not been updated any further since 2014. It supports the Prolog ISO standard. It supports interoperability with the Java Virtual Machine (JVM) by means of an external Java interface, but it doesn’t support any JavaScript interoperability.

JIProlog is a Prolog engine available as an open source project `AGPLv3`-licensed at `https://github.com/jiprolog/jiprolog`. It consists of a Java Prolog interpreter and its main features are full Java/Prolog and Prolog/JDBC interactions support. It supports the JVM platform as it is a Java project. The project is alive and actively maintained. It supports the Prolog ISO standard. It natively supports interoperability with JVM, but it doesn’t support JavaScript interoperability.

Ciao! prolog [11] is a Prolog engine available as an open source project `LGPL`-licensed at `https://github.com/ciao-lang/ciao`. It consists of a Prolog-based logic programming system and its main features are CLP and multi-paradigm programming. It supports Unix and Windows platforms as it is a C project. The project is alive and actively maintained. It supports the Prolog ISO standard. It supports interoperability with JVM by means of a complex Java/Prolog interface using sockets, but it doesn't support any JavaScript interoperability.

ECLiPSe [12] is a Prolog engine available as an open source project `MPL`-licensed at `https://sourceforge.net/projects/eclipse-clp/`. It consists of a CLP system. It supports Unix and Windows platforms as it is a C project. The project is alive and actively maintained. It supports the Prolog ISO standard. It supports interoperability with JVM by means of an external library, but it doesn't support any JavaScript interoperability.

GNU Prolog [13] is a Prolog engine available as an open source project `GPLv2`-licensed at `https://sourceforge.net/projects/gprolog/`. It consists of a Prolog compiler and its main features are CLP and C interoperability. It supports Unix and Windows platforms as it is a C project. The project reached version `1.4.4` and it has not been updated any further since 2018. It supports the Prolog ISO standard. It doesn't support interoperability with both JVM and JavaScript.

Jekejeke Prolog [14] is a Prolog engine available as an open source project at `https://github.com/jburse/jekejeke-devel`, licensed under their own Evaluation/Distribution[1] license. It consists of a Prolog interpreter runtime library. It supports JVM and Android platforms as it is a Java project. The project is alive and actively maintained. It supports the Prolog ISO standard, but its developers documented some discrepancies[2] with it. It natively supports interoperability with JVM, but it doesn't support any JavaScript interoperability.

JLog is a Prolog engine available as an open source project `GPLv2`-licensed at `https://sourceforge.net/projects/jlogic/`. It consists of a Prolog interpreter and its main features are Java compatibility and suitability for educational

[1]`http://www.jekejeke.ch/idatab/doclet/cust/en/docs/recital/package.jsp`

[2]`http://www.jekejeke.ch/idatab/rsclet/prod/docs/10_dev/15_stdy/07_compliance/package.pdf`

purposes. It supports the JVM platform as it is a Java project. The project reached version `1.3.6` and it has not been updated any further since 2012. It supports the Prolog ISO standard. It natively supports interoperability with JVM, but it doesn't support JavaScript interoperability. It also supports programmatic usage of Logic Programming in Java.

JScriptLog is a Prolog engine available as an open source project `GPLv2`-licensed at `https://sourceforge.net/projects/jlogic/`. It consists of Prolog interpreter and its main feature are JavaScript compatibility and suitability for educational purposes. It supports JavaScript-enabled platforms as it is a JavaScript project. The project reached version `0.7.5.beta` and it has not been updated any further since 2012. It supports the Prolog ISO standard. It natively supports interoperability with JavaScript, but it doesn't support JVM interoperability.

LPA-Prolog [15] is a Prolog engine available as a freeware at `http://www.lpa.co.uk/sup_dow.htm`. It consists of a Prolog compiler for Windows and its main features are a user interface and the Windows support. It supports only the Windows platform. The project is alive and actively maintained. It does not supports the Prolog ISO standard. It supports interoperability with JVM by means of an external library, but doesn't support any JavaScript interoperability.

Open Prolog is a Prolog engine available as a freeware at `https://www.scss.tcd.ie/misc/open-prolog/`. It consists of a Prolog implementation whose main feature is Mac OS support. It supports the Mac OS platform. The project reached version `1.1b10` and it has not been updated any further since 2005. It partially supports the Prolog ISO standard. It doesn't support interoperability with both JVM and JavaScript.

SICStus [16] is a Prolog engine available as a commercial product at `https://sicstus.sics.se/`. It consists of a Prolog development system and its main features are performance and Prolog ISO standard support. It supports Unix and Windows platforms. The project is alive and actively maintained. It supports the Prolog ISO standard. It supports interoperability with JVM by means of external library, but it doesn't support any JavaScript interoperability.

Strawberry Prolog is a Prolog engine available as an open source project `AntiGNU-`

licensed at `http://www.dobrev.com/download.html`. It consists of a Prolog compiler and its main feature is the debugging facility. It supports Unix and Windows platforms as it is a C++ project. The project is alive and actively maintained. It doesn't support the Prolog ISO standard. It doesn't support interoperability with both JVM and JavaScript.

SWI-Prolog [17] is a Prolog engine available as an open source project `BSD`-licensed at `https://github.com/SWI-Prolog/swipl-devel`. It consists of a Prolog environment and its main feature is to be useful in building research prototypes and interactive systems. It supports Unix, Windows and JavaScript-enabled platforms. The project is alive and actively maintained. It supports the Prolog ISO standard, but there are some documented discrepancies. It supports interoperability with both JVM and JavaScript by means of, respectively, external or third parties libraries.

Tau prolog is a Prolog engine available as an open source project `3-Clause BSD`-licensed at `https://github.com/jariazavalverde/tau-prolog`. It consists of Prolog interpreter and its main feature is to be an in-browser full client-side Prolog implementation. It supports JavaScript-enabled platforms as it is a JavaScript project. The project is alive and actively maintained. It supports the Prolog ISO standard. It natively supports interoperability with JavaScript, but it doesn't support JVM interoperability.

tuProlog [1] is a Prolog engine available as an open source project `LGPL`-licensed at `https://gitlab.com/pika-lab/tuprolog/2p`. It consists of a lightweight Prolog system and its main feature is Java interoperability. It supports JVM and Android platforms as it is a Java project. The project is alive and actively maintained. It supports the Prolog ISO standard. It natively supports interoperability with JVM, but it doesn't support JavaScript interoperability. It also supports programmatic usage of Logic Programming in Java.

Visual Prolog is a Prolog engine available as a freeware at `https://www.visual-prolog.com/`. It consists of a Prolog-based multi-paradigm programming language and its main feature is the user interface. It supports Unix and Windows platforms. The project is alive and actively maintained. It doesn't support the Prolog ISO standard. It doesn't support interoperability with both JVM and JavaScript.

XSB [18] is a Prolog engine available as an open source project `LGPL`-licensed at `https://sourceforge.net/projects/xsb/`. It consists of a deductive database and logic programming system and its main feature is tabled resolution. It supports Unix and Windows platforms. The project is alive and actively maintained. It supports the Prolog ISO standard. It supports interoperability with JVM by means of an external library, but it doesn't support JavaScript interoperability.

YAP-Prolog [19] is a Prolog engine available as an open source project `GPL`-licensed at `https://github.com/vscosta/yap-6.3`. It consists of a Prolog compiler and its main feature is to be modular. It supports Unix and Windows platforms as it is a C project. The project is alive and actively maintained. It supports the Prolog ISO standard. It supports interoperability with JVM by means of an external library, but it doesn't support JavaScript interoperability.

Among the 18 Prolog implementations taken in consideration, only 13 are still actively developed. On these, we find out that:

- 10 advertise Prolog ISO standard support (*JIProlog*, *Ciao! prolog*, *ECLiPSe*, *Jekejeke Prolog*, *SICStus*, *SWI-Prolog*, *Tau prolog*, *tuProlog*, *XSB*, *YAP-Prolog*).

- 10 support some kind of interface to the JVM world (*JIProlog*, *Ciao! prolog*, *ECLiPSe*, *Jekejeke Prolog*, *LPA-Prolog*, *SICStus*, *SWI-Prolog*, *tuProlog*, *XSB*, *YAP-Prolog*).

- 2 feature some kind of JavaScript interface (*SWI-Prolog* and *Tau prolog*)

SWI-Prolog turns out to be the best implementation at matching our "goodness" criteria. But none of the reviewed implementations – not even *SWI-Prolog* – feature a native support to be used both in JVM and JavaScript-enabled platforms.

2.3 Kotlin and multi-platform support

Kotlin [20] is a statically typed programming language targeting JVM, Android, JavaScript and Native platforms. It has constructs that make it suitable either for *object-oriented* and *functional* programming styles; a mix of the two is also possible.

In this section we provide a description of the language after a brief historical introduction.

2.3.1 Brief history

In last fifteen years Java has always been one of the top-two most-used programming languages, hand in hand with C Language [21]. After the advent of Android development around 2010, *Java 6* use increased. Unfortunately, while Java updates (like *Java 7* and *Java 8* with lambda support) were released for the standard platform, the Android SDK platform struggled aligning, leaving programmers to use outdated and "uncomfortable" features.

JetBrains' *Kotlin* open-source project started in 2010 reaching the first official stable 1.0 release in February 2016. In 2017 at *Google IO* the official *Kotlin* support for Android development, was announced. This potentially brought all the new Java features (till *Java 13*), to the light, in Android development, maintaining compatibility with Java 6 bytecode.

Kotlin 1.1 introduced full stable support for compiling to JavaScript (*Kotlin/JS*), and the experimental support for *Coroutines*. *Kotlin 1.2* introduced the experimental support for multi-platform projects. *Kotlin 1.3* introduced full stable support for Coroutines, and reworked the multi-platform support to be more effective (in terms of project structure) but still leaving it experimental, and therefore subject to changes.

At time of writing *Kotlin* latest release version is 1.3.61 of November 2019.

2.3.2 Main features

Kotlin is inspired by some popular languages (*Java*, *C#*, *JavaScript*, *Scala* and *Groovy*), trying to concentrate best things of them in one language. It is designed with Java interoperability in mind, so Java-Kotlin interaction and vice-versa is painless.

To present its main features a comparison to Java can be more effective than a simple list of its features—most of which are standard among procedural languages.

Fixed Java Problems

Kotlin "fixes" some Java old well-known flaws. Here we briefly enumerate the most relevant ones, explaining how *Kotlin* addresses them.

Null references are controlled by the type-system. In *Kotlin* if a type should contain the `null` value, the programmer has to explicitly declare it. Then the type-system enforces the programmer to manage properly the situation, when the declared type is used. The result is that programmers are now aware of where nullability could cause problems, avoiding unexpected problematic `NullPointerExceptions`.

There are no *raw types*. In *Kotlin* if a class C has some type arguments, the programmer can't declare something of type C without explicitly declaring even its type arguments. Java had to maintain this capability to ensure backwards compatibility with Java code written before *Generics* introduction.

***Kotlin* arrays are invariant.** This prevents the possibility of run-time failure, unlike in Java. Arrays of primitive types are however compiled to Java primitive types arrays, to avoid boxing/unboxing overhead and maximise performances.

Functions are first-class citizens. *Kotlin* has proper functions types, because they are a first-class concept. In Java we had SAM-conversions (Single Abstract Method), i.e. interfaces with a single declared method (also known as Functional

Interfaces), that could be treated as lambda functions through syntactic sugar. The SAM-conversion created – behind the scenes – an anonymous object implementing that interface with the body of the lambda, hence maintaining backwards compatibility.

***Kotlin* does not have checked exceptions.** In *Kotlin* is not mandatory to catch exceptions. This makes declared APIs more comfortable to be used, rather than Java APIs with lots of `throws` clauses. Checked exception are theoretically a very good feature for code stability. Pragmatically, they do more harm than good, because library users will ignore all caught exceptions most of the time. Furthermore, checked exceptions introduce problems in software in terms of scalability and possibility of evolution [22] [23].

Kotlin Specific Features

Kotlin "inherits" from the languages inspiring it, some very handy features, which enhance the programming experience.

Extension functions. *Kotlin* provides the ability to extend a class with new functionality without having to inherit from the class or use design patterns such as *Decorator*. To declare an extension function, we need to prefix its name with the *receiver type*, i.e. the type being extended. This makes very easy the creation of *Domain Specific Languages (DSL)*.

It is worth pointing out that extensions are *resolved statically* and so the receiver type determining which extension will be called is not the run-time type, but the compile-time one.

Smart-casts. In many cases, explicit casts after a "type check" are not needed, because *Kotlin* compiler tracks these checks, and smart casts variables for you to the correct type.

String templates. String literals may contain template expressions, i.e. pieces of code that are evaluated and whose results are concatenated into the String.

With the usage of `$` symbol, the programmer can embed expression in Strings without manually concatenating the single pieces.

Delegation as first-class citizen. The *Delegation pattern* is a good alternative to implementation inheritance, and *Kotlin* supports it natively requiring zero boilerplate code. A class `Derived` can implement an interface `Base` by delegating all of its public members to another specific `Base` implementation.

Object Expression and Decalration. *Kotlin* provides a direct support to the widely used *Singleton pattern.* With the `object` keyword the programmer can declare a new type (ie. a class) which will have only one instance by design.

Declaration-site variance. Inherited from C#, declaration-site variance is added to already present use-site variance of Java. Thus developers can declare generic types variance once for all while designing a library. In this way library users can benefit from that design, without needing to declare complex use-site types.

Operator overloading. *Kotlin* allows us to provide implementations for a predefined set of operators on user declared types. These operators have fixed symbolic representation (like `+` or `*`) and fixed precedence. To implement an operator, we provide a *member function* or an *extension function* with a fixed/predefined name, for the corresponding type. Functions that overload operators need to be marked with the `operator` modifier. This feature is another building block for *DSL* construction.

Data classes. Frequently there's the need to create classes whose main purpose is to hold data. In such a class, some standard functionalities and utility functions are often mechanically derivable from held data, like `equals()`, `hashCode()`, `toString()`, and others. Indeed, most of IDEs provide automatic generation functionalities for some of these methods. In *Kotlin*, such data container class is called a *data class* and it's marked with the keyword `data` in its declaration. Classes marked with it, will receive an automatic implementation for the aforementioned methods.

Collection Standard Library with separated mutable and immutable classes. In *Kotlin* classes in the Collection package are separated by means of their mutability. This allows for different treatment of variance with different types (i.e. mutable collections parameters are correctly treated as invariant, while immutable ones can allow a more flexible usage).

Coroutines. *Kotlin* approaches asynchronous and non-blocking programming with *Coroutines*, supporting them at language level with minimal APIs in the standard library and delegating most of the functionality to the *Coroutines* specific libraries. Unlike many other languages with similar capabilities (e.g. JavaScript), `async` and `await` are not keywords in *Kotlin* and are not even part of its standard library, but are library functionalities of the external `coroutines` module. Moreover, *Kotlin*'s concept of *suspending function* provides a safer and less error-prone abstraction for asynchronous operations than futures and promises.

2.3.3 Multi-platform programming

One of the main enticing feature of *Kotlin*, among the others, is that of multi-platform support. Working on all platforms is an explicit goal for *Kotlin*, but this is a premise to a much more important goal: sharing code between platforms. Currently supported platforms include: JVM, Android, JavaScript, iOS, Linux, Windows, Mac and even embedded systems like STM32[3]. This brings the invaluable benefit of reuse for code and expertise, saving the effort for tasks more challenging than implementing everything twice or multiple times.

One of the key capabilities of *Kotlin* multi-platform code is a way for "common code" to depend on platform-specific declarations. In multi-platform projects we have different code source sets: one for each targeted platform, and one common source set that is shared for all platforms. In the common source set, programmers

[3]`https://en.wikipedia.org/wiki/STM32`

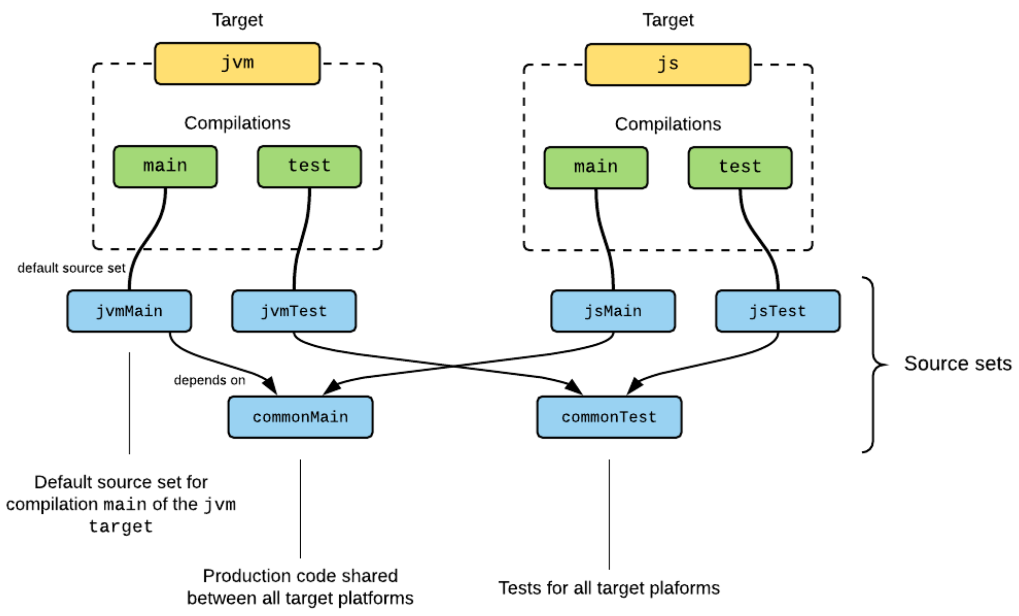

Figure 2.1: Structure of a *Kotlin* multi-platform project. Source: `https://kotlinlang.org/docs/reference/building-mpp-with-gradle.html`

should implement the overall software system logic, while platform specific source sets only contain those implementations accessing platform specific features. This architecture will work thanks to the **expected/actual mechanism**.

With this mechanism the "common code" declares to *expect* some functionality to be implemented somewhere else, and in the meanwhile it can use it to logically complete the system common implementation. The targeted platforms code can later implement the *actual* functionality, accessing platform specific characteristics, that were inaccessible in "common code". This resembles the well-known *Template Method pattern* but applied at the architectural level, between source sets.

This mechanism opens the possibility to widen the targeted platforms range, in different moments of a project life, needing only to implement platform specific code for the newly supported platform and automatically inheriting all the already implemented (and hopefully tested) system logic. Even though this feature is still in an experimental status, it seems very promising.

Chapter 3

2p-kt Project

2p-kt is the natural evolution and modernisation of *tuProlog* Prolog implementation. In this chapter we discuss how the idea of *2p-kt* project came to life.

In section 3.1 we provide some thoughts about or predecessor *tuProlog*, then in section 3.2 we provide our vision for the use of *2p-kt* and finally in section 3.3 we talk about its requirements that drove the design.

3.1 tuProlog

tuProlog is a Java object-oriented Prolog engine which has been designed to build intelligent components for dynamic Internet infrastructures [24]. The original purpose of that work was to develop a *malleable architecture* for *tuProlog* inference core, applying sound engineering practices such as object-oriented design, reuse of established community knowledge in the form of patterns, loose coupling of components, modularity, and a clear and clean separation of concerns.

tuProlog is still maintained to date, but unstructured interventions performed by many developers and an insufficient level of testing have led to the accumulation of a lot of technical debt, making changes and enhancements application much more difficult than necessary. Furthermore, some design choices were affected by the limits of the JVM technology of yore. In fact, the development of *tuProlog* started in the early 2000s, when JDK 5 had not yet been introduced.

This means that the core of *tuProlog* has been developed without relying on the many constructs which make Java such a modern and flexible OOP language—such as generics, for-each loops, enums, and many others[1]. Despite such issues never affected the many conceptual and fundamentals merits of *tuProlog*, they heavily affect the maintanability and evolvability of the *tuProlog* source code.

tuProlog merits. We list here the merits of *tuProlog*, that are, those features which have proven themselves to be good design decisions in hindsight, and will therefore be retained:

- *Inferential core modelled as a Finite State Machine.* The inferential core of *tuProlog* is built in the form of a *Finite State Machine* (*FSM*). This virtually enables developers to extend the inferential core with possibly new states, in order to affect its behaviour.
- *Hard-code as few as possible built-ins inside inference core.* Indeeds, if no libraries has been loaded, the *tuProlog* core *natively* supports only those libraries which:
  - directly in influence the resolution process, such as `'!'/0`;
  - are too basic to be defined elsewhere, such as `fail/0`;
  - need to be defined near the core for efficiency reasons, such as `','/2`.
- *Possibility to call Java from Prolog.* *tuProlog* enables users to call any Java class, library, object *directly from the Prolog code* with no need of pre-declarations, awkward syntax, etc., with full support of parameter passing from the two worlds, yet leaving the two languages and computational models totally separate so as to preserve *a priori* their own semantics — thus bringing the power of the object-oriented platform (e.g. Java Swing, JDBC, etc) to the Prolog world for free.
- *Capability to use Prolog programmatically in Java.* *tuProlog* users can exploit Prolog engines *directly from the Java code* as one would do with any

[1]`https://en.wikipedia.org/wiki/Java_version_history#Java_5_updates`

other Java library with its functionality, again with full support of parameter passing from the two worlds in a non-intrusive, simple way that does not alter any semantics. This virtually brings the power of logic programming into any Java application.

- *Notion of Library.* A library in *tuProlog* is a container for reusable code, written according to the best fit paradigm, chosen between LP and OOP. In libraries it is possible for example to implement a Prolog clause "body" using Java code, and this works smoothly with the rest of the architecture. They are the practical point of integration between the two paradigms.

- *The idea of a Term hierarchy.* The idea to have a hierarchy of types reflecting LP concepts makes possible their programmatic usage from Java.

- *The idea of being lightweight.* *tuProlog* was designed to be minimal and highly configurable (for example, through libraries). These characteristic go hand in hand with each other, since the more the core is minimal and more it must be configurable, to add needed functionalities. Therefore *tuProlog* core lightweightness makes it suitable to execute on resource constrained devices as well as common devices.

tuProlog defects. Here we point out defects of *tuProlog*, i.e. those features that in retrospect were bad design choices (although some of them were forced by the technology of the time) and will be addressed by *2p-kt*.

- *Model classes at wrong detail level.* In *tuProlog* we have a base class `Term` extended by three classes `Var`, `Number`, `Struct`. Then the hierarchy explores various types of numbers (with sub-classes `Float`, `Double`, `Int`, `Long`) but does not explore the variability of classes that cuold emerge under `Struct`. In fact the `Struct` class has a lot of methods to check if represented structured term is of some specific sub-type, i.e. `isList`, `isAtom`, etc.

  The lack of a first-class support to some basic Prolog structure types is felt when programmatic type enforcement is needed (e.g. in parameter types).

- *Model classes are mutable.* Currently model classes internals can be modified by means of method calls side effects. For example, `Var` class contains the binding to the bound term instance, if any, and this reference can be modified at any time.

  This makes code more prone to "external attacks" (maybe unwanted) coming from other code using references to mutable internals. It is now known that mutability should be avoided if not necessary to achieve requirements on performance of software [25].

  Furthermore, because the history of changes is important to support the *backtracking* mechanism, this pervasive mutability led to the creation of auxiliary data structures maintaining the history of changes made to mutable components.

- *Primitive, directives and function invocation uses reflection.* In *tuProlog* to define primitives, directives and functions a programmer should implement them as methods, in a library class, with a specific name depending on the desired predicate name. Then they will be searched at invocation time by the Java reflection mechanism.

  Although this approach seems to have no problems, it increases the code rigidity and forces the programmer to use *non-coded conventions*, possibly leading to errors. Furthermore, is now known that the reflection mechanism introduces some overhead, that could have been avoided with different engineering.

3.2 Use scenarios

2p-kt has been thought to explicitly address two main use scenarios: *(i)* its use by software programmers as a library to exploit the logic programming paradigm and *(ii)* its use as a basic component to be further extended by adding new Prolog libraries, primitives, and so on.

Logic programming library. The use as logic programming library is the primary usage. It is intended for “final users”, i.e. programmers that would like to inject in their software some logic programming, in a seamless way. A basic usage would be the exploitation of *symbolic manipulation* and the *unification mechanism*, but the main feature would be the ability to write Prolog programs and *solve goals* against them.

Prolog extensions base. The secondary usage is as a code base to implement Prolog extensions, deviating from the Prolog Standard functionalities. Programmers could create their extensions, writing new Prolog libraries, containing custom theories and primitives. Then these extensions could be loaded during Prolog engine creation, or later during computations, and hence be internally exploited.

3.3 Requirements

In this section we list requirements that will dictate the shape of the project. Some of them are specifically meant to address some previously identified defects.

Architectural requirements

Multi-platform support. *2p-kt* should be able to run on, at least, two different platforms (e.g. JVM and JavaScript), and support extendability to new ones.

Strong modularity. *2p-kt* should be split into modules in a way that allows to import separately parts of interest, hence avoiding to create a monolith library. For instance, it should be possible to import the symbolic manipulation functionality separately from Prolog solver one.

Functional requirements

Prolog Standard support. *2p-kt* should support the implementation, and then the execution, of standard built-ins [6], following their behaviour description.

Immutability by default. Core classes should be immutable, solving the corresponding defect in *tuProlog* (section 3.1). The overall implementation should follow the principle that, if mutability is not strictly necessary, it should be avoided.

Detailed Term hierarchy. A detailed hierarchy should be created to better represent terms and their variations. The term hierarchy should include all "types" introduced in Prolog Standard reference [6].

Term instances should be of most specific class. Correct class instances creation should be enforced. For instance, the creation of a Term which is, in particular, an Atom, has always to return an instance of the Atom class.

Solver computation max duration. It should be possible to specify a max duration for Solver computations, after which the goal solution search will be exceptionally interrupted.

Non-functional requirements

Maximise programmatic usability of concepts. *2p-kt* should provide the ability to use Prolog concepts with minimal effort and no additional "boiler-plate code". In past this was addressed with the usage of a Prolog parser which was the intermediary from the Prolog language and its programmatic usage in other languages. This should not be the way to proceed. We want, for example, a *Domain Specific Language* (DSL) enabling programmers to exploit the Prolog language concepts without the need of cumbersome intermediaries (like a parser).

Project code testing. *2p-kt* project code should be tested. More precisely, every public feature should be directly or indirectly tested.

Chapter 4

2p-kt Design

In this chapter we explore the architectural and detailed design of the project. In section 4.1 we provide an overview of the project architecture, then each further section provides a deep look in each project module design.

4.1 Architectural design

The project is split in several modules, each one providing a single self-contained feature, enforcing the *Single Responsibility Principle* (SRP) yet at architectural level. We have the following modules (Figure 4.1):

Core: It is the *basic building block* for others modules. It contains model classes used throughout the overall project. It provides the basics for symbolic manipulation.

Unify: It depends on *core* module. It contains model classes to support unification. It provides the basic unification mechanism.

Theory: It depends on *unify* module. It models the concept of Prolog theory (program). It provides theory manipulation constructs.

Solve: It depends on *theory* module. It models basic concepts useful for Prolog solver implementations. It provides all model classes needed implementing a Prolog solver.

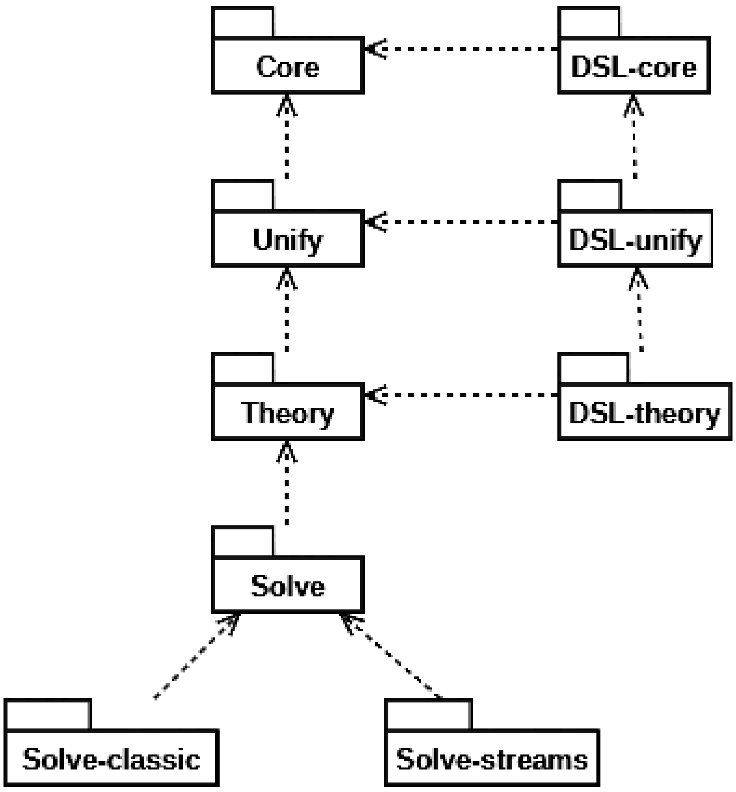

Figure 4.1: *2p-kt* modules architectural dependencies.

Solve-Classic: It depends on *solve* module. It contains an enhanced implementation of *tuProlog* inferential core. It provides the *tuProlog* enhanced re-implementation.

Solve-Streams: It depends on *solve* module. It contains an experimental inferential core implementation, based on *tuProlog*, with an implicit backtracking mechanism. It provides the newly proposed solver implementation.

DSL-Core: It depends on *core* module. It models a *Domain Specific Language* (DSL) inspired by Prolog itself. It provides simple and uncluttered DSL to enable programmers to create *core* data structures in a Prolog-like fashion.

DSL-Unify: It depends on *dsl-core* and *unify* modules. It adds unification constructs to *dsl-core*. It provides an enhanced DSL enabling programmers to exploit Prolog unification with no "boilerplate code".

DSL-Theory: It depends on *dsl-unify* and *theory* modules. It adds theory manipulation facilities to *dsl-unify*. It provides an enhanced DSL enabling programmers to write Prolog theories in a clear and compact way.

4.2 Core module

Core module contains model classes for Prolog domain (Figure 4.2). The modeling follows quiet closely the Term definition (section 2.1.2).

4.2.1 Main "core" types

At top of the hierarchy we have the `Term` type which has as sub-types `Constant`, `Struct` (alias for *Compound*) and `Var` (alias for *Variable*). `Constant` has as sub-types `Numeric` and `Atom`. `Numeric` is specialised in `Integer` and `Real`. Because atoms can be viewed as functors with zero arity, `Atom` is also a sub-type of `Struct`.

Other core classes are specialisations of `Struct` (or its sub-types) representing remarkable structured terms, with particular meanings. The proposed design addresses the defect of *tuProlog* of not being specific enough.

Indicator. It is the term describing the structure of compound terms.

In particular it is a *ground term* in the form *name*/*arity*, where *name* $\in \mathcal{L}(\langle Atom \rangle)$ denoting the name of a predicate or a functor and *arity* is a non-negative integer denoting the number of its arguments. For example the structure `a(b, c(d, e), f)` has `'a'/3`, as its indicator.

List. It denotes terms that in Prolog model "collections".

An *empty list* in Prolog is denoted by the atom `[]`. It is captured by `EmptyList`, a `List` sub-type.

A *not empty list* in Prolog is denoted by the functor `'.'/2`, captured by `Cons` (a `List` sub-type), where the first argument is a term and the second a list. A list of elements is the term `.(a1, .(a2, .(..., .(aN, []) ...)))`, but it may be written with a simpler notation: `[a1, a2, ..., aN]`.

One may also use the concatenation syntax. By definition, writing `[a1, ..., aN | [b1, ..., bM]]` is equivalent to writing `[a1, ..., aN, b1, ..., bM]`. But if last sub-term is not an empty list, the concatenation syntax becomes mandatory, and the whole term is called to be a *list-term.* For example, term `.(a1, .(a2, .(..., .(aN, t) ...)))`, where `t` is not an empty list, is represented as `[a1, ..., aN | t]`. Lists and list-terms are disjoint sets of terms. List-terms have at least one element. List-terms whose last sub-term is a variable, like `[a1, ..., aN | X]`, are called *partial lists.*

Tuple. It is the structure with indicator `','/2`. It has a left and a right element, and can be nested to contain more elements, as for List dot functor.

Set. It is a structured term somehow similar to List.

An *empty set* is denoted by the atom `{}`. It is captured in `EmptySet`, a `Set` sub-type.

A *not empty set* is denoted by the functor `'{}'/1`. The term `'{}'(','(a, ','(b, c)))` can be written, in a simpler notation, as `{a, b, c}`. If the single argument is a Tuple, then the Set contains each element part of Tuple recursive structure, otherwise the Set contains the single Term element. Examples: `{1}` contains `1`, `{a, b}` contains `a` and `b`, but `{[1, 2, c]}` contains `[1, 2, c]`.

Empty. It is a sub-type of `Atom`, and the common super-type for empty data structures (`EmptyList` and `EmptySet`).

Truth. It is a sub-type of `Atom` which models Prolog truth special atoms. In particular it has two possible instances: the atom `true` and the atom `fail`.

Clause. It denotes terms used in Prolog to write executable programs. Their principal functor is `':-'/2` and they can be rules, represented by `Rule` type, or directives, represented by `Directive` type.

A *rule* in Prolog has a particular (abstract) syntax [6]:

$$\begin{aligned}
\langle \mathit{Rule} \rangle &:= \langle \mathit{Head} \rangle \ \texttt{:-}\ \langle \mathit{Body} \rangle \mid \langle \mathit{Predication} \rangle \\
\langle \mathit{Head} \rangle &:= \langle \mathit{Predication} \rangle \\
\langle \mathit{Predication} \rangle &:= \langle \mathit{Atom} \rangle \mid \langle \mathit{Compound} \rangle
\end{aligned}$$

It is defined to be a term whose principal functor is `':-'/2`, with a first argument, called the *head* and a second, called the *body*, or a term which is a predication. A *predication* is defined as a term which is an atom or a compound term.

A *body* follows the syntax:

$$\begin{aligned}
\langle \mathit{Body} \rangle := \ & \texttt{(}\langle \mathit{Body} \rangle\texttt{,}\ \langle \mathit{Body} \rangle\texttt{)} \\
\mid \ & \texttt{(}\langle \mathit{Body} \rangle\texttt{;}\ \langle \mathit{Body} \rangle\texttt{)} \\
\mid \ & \texttt{(}\langle \mathit{Body} \rangle\ \texttt{->}\ \langle \mathit{Body} \rangle\texttt{)} \\
\mid \ & \langle \mathit{Variable} \rangle \\
\mid \ & \langle \mathit{Predication} \rangle
\end{aligned}$$

It is defined as a term, whose principal functor is `','/2` (a *conjunction* of bodies), `';'/2` (a *disjunction* of bodies), `'->'/2` (an *implication* of bodies), a variable or a predication whose principal functor is different from, `','/2`, `';'/2` or `'->'/2`. We will say that a body (of a clause) *contains* a given term, if this term occurs in the position of a predication using the rules defining a body.

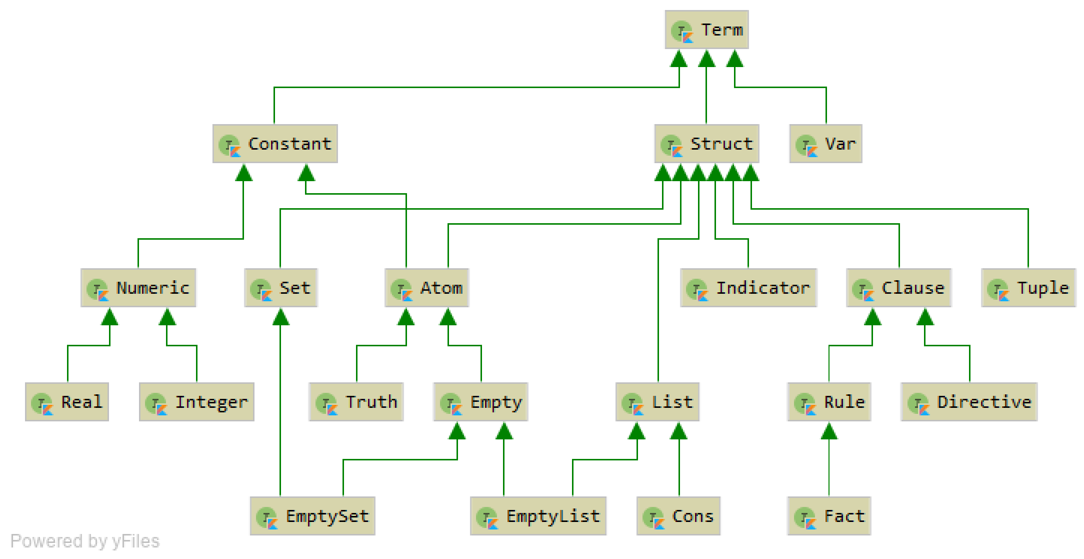

Figure 4.2: The hierarchy of classes at *core of 2p-kt*, modelling Prolog domain.

A *fact* in Prolog is a rule term whose body is the atom `true`. It is captured by `Fact`, a `Rule` sub-type. Because the body is fixed, it can be omitted in writing facts. Thus facts are (syntactically) predications, and vice-versa.

A *directive* in Prolog follows the (abstract) syntax:

$$\langle \mathit{Directive} \rangle \quad ::= \quad \texttt{:-}\ \langle \mathit{Body} \rangle$$

It is defined as a term whose principal functor is `':-'/2`, with first argument always `null` and second argument, the body, filled following previously mentioned body syntax rules. Directives can be thought to all effects as rule terms, with no head specified.

A clause term is said to be *well-formed* if and only if its head does not contain a number or a variable and its body is well-formed. A clause body is said to be well-formed if and only if it does not contain a number, following the definition of "contains" given before for clause bodies.

4.2.2 Other "core" features

Alongside the expected core model types, the core module provides other useful functionalities, general enough to gain a prominent place in such module.

Substitution. The homonym Prolog concept is captured in a `Substitution` type.

Unlike in *tuProlog*, where `Var` used to contain the binding to the bound term, in *2p-kt* we applied the SRP. `Var`s in fact have the sole task of representing Prolog variables. `Substitution`s in turn have the sole task to represent variable bindings.

Scope. Scope concept is strictly related to variables. Variable scope is captured in `Scope` type. The breadth of a variable scope is limited to single terms. Every time a new term is considered in Prolog, its variables are different from other terms ones.

For example if we consider the term `a(A) :- b(A)`, there are two `A` variables and they are the same (instance). Hence with same bindings, if any. Considering instead terms `a(A)` and `b(A)` separately, there are two `A` variables and they are different (instances). Hence with different bindings.

TermVisitor. The well-known *Visitor pattern* is provided over terms through `TermVisitor` type, to enable actions dispatch on terms components. Specific actions can be carried out for each specific `Term` sub-type, making very easy to think about structural visit of terms.

Operator, Specifier and OperatorSet.

Other basic concept in Prolog is the *operator*. An *operator* is a particular structured term, with arity 1 or 2, that can be written in a prefix, infix or postfix notation, without parenthesis. It is described with a triple formed by its name (an atom), specifier (one of the atoms: `xf`, `yf`, `xfx`, `xfy`, `yfx`, `fx`, `fy`) and priority (an integer). An `Operator` type will represent it.

Specifier	Class	Associativity
`fx`	prefix	non-associative
`fy`	prefix	right-associative
`xfx`	infix	non-associative
`xfy`	infix	right-associative
`yfx`	infix	left-associative
`xf`	postfix	non-associative
`yf`	postfix	left-associative

Table 4.1: Specifiers for operators.

The *priority of a term* is normally 0, when it is written in functional, list or curly notation, or if it is a bracketed expression or an atomic term. But if the term is written as an unbracketed expression in operator notation, its priority is the priority of its principal functor.

The *specifier* of an operator (Table 4.1) is a mnemonic that defines the *arity*, the *class* (prefix, infix or postfix) and the *associativity* (left-, right- or non-associative) of the operator. The arity is equal to the number of `x` and `y` in the specifier. The specifier `f` symbol is a placeholder indicating a specifier name. The specifier `y` and `x` defines how implicit associativity works with operand(s). Indeed, the former means "associative here", the latter means "non-associative here". The implicit associativity is particularly useful for writing subexpressions with the same operator without parenthesis. For example the expression `(1 + 2 + 3 + 4)` is the term `'+'('+'('+'(1, 2), 3), 4)))` if the specifier of `'+'` is `yfx`. A `Specifier` type will represent all available specifiers.

The `OperatorSet` type captures the concept of currently known set of operators.

4.3 Unify module

The unify module aims at describing the *unification process* and its basic components (such as equations of terms introduced in Section 2.1.2).

4.3.1 Abstract unification algorithm

The algorithm is described below by four equation transformation rules and by two failure conditions. At every step the computation state is characterised by a set of equations. The initial set is a set of one or more equations. The step consists of the application of a transformation rule to one of the equations or in checking that an equation satisfies a failure condition. The computation terminates if an equation, in the current set of equations, satisfies a failure condition or if none of the rules is applicable to any equation [6].

- Transformation rules:

  Splitting: Replace an equation of the form $\texttt{f}(s_1,\ \ldots,\ s_n) = \texttt{f}(t_1,\ \ldots,\ t_n)$, where $\{s_1,\ \ldots,\ s_n,\ t_1,\ \ldots,\ t_n\} \subseteq \mathcal{L}(\langle Term \rangle)$ and $n \geq 0$, by the equations $s_1 = t_1,\ \ldots,\ s_n = t_n$.

  Identity Removal: Remove an equation of the form $X = X$, where $X \in \mathcal{L}(\langle Variable \rangle)$.

  Swapping: Replace an equation of the form $t = X$, where $X \in \mathcal{L}(\langle Variable \rangle)$ and $t \in \mathcal{L}(\langle Term \rangle) \setminus \mathcal{L}(\langle Variable \rangle)$, by the equation $X = t$.

  Variable elimination: If there is an equation of the form $X = u$, where $X \in \mathcal{L}(\langle Variable \rangle)$ and $u \in \mathcal{L}(\langle Term \rangle)$, such that X does not appear in u (*negative occurs-check*) but X appears in some other equation, then replace any other equation $s = t$, where $s \in \mathcal{L}(\langle Term \rangle)$, by the equation $s\{X/u\} = t\{X/u\}$.

- The failure tests halt and report failure if the set includes an equation in one of the following forms:

  Disagreement: $\texttt{f}(s_1,\ \ldots,\ s_n) = \texttt{g}(t_1,\ \ldots,\ t_m)$, where $\texttt{'f'}/n \neq \texttt{'g'}/m$, with $n, m \geq 0$.

  Positive occurs-check: $X = u$, where $X \in \mathcal{L}(\langle Variable \rangle)$, $u \in \mathcal{L}(\langle Term \rangle) \setminus \mathcal{L}(\langle Variable \rangle)$, and u includes X.

This algorithm is called the *Herbrand algorithm* [6]. Given two terms it always terminates with success (the remaining set of equations defines an idempotent *MGU* of the two terms) or with failure (there is no unifier). The two actions corresponding to a negative or positive occurs-check correspond to the so called *occurs-check tests*.

For example starting with one equation `f(X, Y)` = `f(g(Y), h(T))`:

1. *Splitting* produces two equations {`X` = `g(Y)`, `Y` = `h(T)`}; the first equation does not correspond to any transformation, but
2. a *variable elimination* may be performed with the second, leading to the equations: {`X` = `g(h(T))`, `Y` = `h(T)`},
3. which corresponds to the substitution: {`X/g(h(T))`, `Y/h(T)`}, which is an idempotent *MGU*.

Notice that the equation `X` = `f(X)` corresponds to the positive occurs-check case and leads immediately to a failure.

The occurs-check problem. The abstract unification algorithm definition imposes implementing the occurs-check tests inside it. It is easy to observe that these tests, which are performed very frequently, influence the performances of the algorithm [6].

Now if one omits the occurs-check tests, we are faced with different problems:

1. the behaviour of the Herbrand unification algorithm *may be unsound*: it may succeed when it should fail. Example: {`X` = `f(X)`, `Y` = `a`} gives the (wrong) "solution" {`X/f(X)`, `Y/a`}.
2. the Herbrand unification algorithm *may not terminate*. Example: starting with the set of equations: {`X` = `f(Y, X)`, `Y` = `f(X, Y)`} the fourth transformation rule may be applied indefinitely.
3. the result of the Herbrand unification algorithm is *no longer independent of the way the transformation rules are applied*. Example: starting with the set

of equations: {`X` = `f(Y)`, `Y` = `f(X)`, `Z` = `a`, `Z` = `b`} the fourth transformation rule may be applied indefinitely on the first two equations instead of firstly considering the last two equations which immediately lead to a disagreement.

4.3.2 Main “unify” types

Equation The `Equation` type captures the concept of Prolog equation of terms, and it is specialised in four sub-types aimed at simplifying the unification algorithm writing (according to transformation rules). An equation, as a matter of fact, can be:

- `Identity`, an equation of identical terms.
- an `Assignment`, stating that $V = t$, where $V \in \mathcal{L}(\langle Variable \rangle)$ and $t \in \mathcal{L}(\langle Term \rangle)$.
- `Comparison`, hence an equation comparing two terms, possibly different (not already split by the transformation rule).
- `Contradiction`, that is an equation stating a contradiction, equating two different terms.

Unificator The `Unificator` type represents the concept of “unificator engine”. The unification process is executed within a “substitution context” set at start time, usually empty, represented by predefined substitutions.

Its main purpose is to compute an *MGU* between two terms, optionally disabling the occurs-check. Other derived features are *(i)* the ability to check if two terms match (i.e. if there is an *MGU*), *(ii)* the ability to get the unified term, hence with the computed *MGU* already applied.

4.4 Theory module

Theory module models the concept of “Prolog program”. A Prolog program (also called just theory) is basically a collection of `Clause`s.

A theory, during execution, can be read and written (enabling meta-programming) and because of these features it can be referred as a `Clause` *database*. When reading clauses from a clause database, they are retrieved in insertion order. Furthermore, in a database all the clauses are well-formed (Section 4.2.1). `ClauseDatabase` type captures this clause database concept.

Prolog ISO standard also tells that clauses in a clause database, which will be used during execution, should be *transformed*. This means that all variables $X \in \mathcal{L}(\langle Variable \rangle)$ in the position of a predication in clauses body, should be replaced by a term `call(`X`)`. For example, the clause `product(A) :- A, A` is stored in the database after preparation for execution as the term: `product(A) :- call(A), call(A)`.

In our design `ClauseDatabase` is not in charge of doing that "preparation for execution", because following a lazy approach we will do that transformation only for really needed clauses, "on the fly", during resolution. This reduces responsibilities of `ClauseDatabase` type to its core, and potentially avoids to transform a very big database of clauses just for using few of them. On the other hand, at design level, we do not predispose a way to prevent multiple transformations of the same clause. In a Prolog theory where a clause, needing transformation, is used a lot of times, a trivial implementation of the resolution process could lead to bad performances.

4.5 Solve module

The solve module contains basic concepts at the root of Prolog standard resolution process, like ExecutionContext, Libraries, Primitives. All these concepts are solver-implementation independent.

It is designed to be used at least in two different ways: *(i)* as a solid common base for Prolog libraries programmers that would like to enhance the language with new functionalities, working with solver-implementation independent abstractions or *(ii)* as a convenient Prolog solver interface, to programmatically solve goals.

Before we introduce the technical detailed design, we provide a description of the Prolog execution model.

4.5.1 Prolog execution model and the search-tree

The execution model is first presented for a subset of Standard Prolog called **definite Prolog** [6]. Then the model is extended to handle all the procedures.

In *definite Prolog* there are only user-defined procedures in the form of clauses in which the body is a sequence (denoted by conjunctions) of predications. So all the clauses have the form:

$$h(t_0) \;\texttt{:-}\; p_1(t_1),\ p_2(t_2),\ \ldots,\ p_n(t_n).$$
or
$$h(t_0) \;\texttt{:-}\; \mathit{true}.$$

where the t_i are (possibly empty) sequences of terms. Goals are definite bodies (i.e. a nonempty sequence of predications).

The execution model is defined on the principle of a *general resolution algorithm* whose input data are a single goal and a clause database. This algorithm corresponds to a proof procedure (a particular case of unit *resolution*), aimed at finding instances of the initial goal which are logical consequences of the definite program.

The general resolution algorithm. The general resolution of a goal G with a database P is defined by the following algorithm:

1. Start with a *current goal* which is the initial definite goal G and a *current substitution* which is the empty substitution.

2. If G is `true` then stop (*success*), otherwise

3. Choose a predication A in G (*predication-choice*)

4. If A is `true`, delete it, and proceed to step (2), otherwise

5. If no freshly renamed clause in P has a head which unifies with A then stop (*failure*), otherwise

6. Choose in P a freshly renamed clause H `:-` B whose head unifies with A by substitution σ which is the MGU of H and A (*clause-choice*), and
7. Replace in G the predication A by the body B, flatten and apply the substitution σ to obtain the new current goal, let the new current substitution be the current substitution composed with σ, and proceed to step (2).

A *freshly renamed clause* means a clause which is renamed w.r.t. all the variables which have occurred in all the previous resolution steps.

The steps (3), (6), and (7) are called *resolution step.* The substitution σ is called the *local substitution.*

In the case of success (step 2), the current substitution restricted to the variables of the initial goal is the *answer substitution.*

This algorithm defines in an indeterminate manner successful, failed and possibly infinite computations. It is “indeterminate” because the order in which the computations may be considered is not fixed. It depends on the predication-choice (step 3) and the clause-choice (step 6).

Prolog Standard, to attack this *non-determinism*, fixes those choices: predication-choice always chooses the first predication in the sequence G, while clause-choice always chooses the “unifiable” clauses according to their sequential order. With these fixed choices a new algorithm may be designed. It will use the notion of *search-tree.*

The Prolog search-tree. The different computations defined by the general resolution algorithm may be represented by a tree called the *Prolog search-tree* that we define as follows:

- Each node is labelled by the *local substitution* and the *current goal.*
- The labels of the root are the empty substitution an the initial goal to be executed.
- There are two kinds of leaf-nodes:
    - Nodes whose goal label is `true`, called *success nodes.*

  - Nodes with a goal label different from `true` such that there is no renamed clause whose head is unifiable with the chosen predication, called *failure nodes.*

- A non-leaf node has as many children as there are clauses whose head (with a suitable renaming) is unifiable with the chosen predication (the first predication in the current goal). If $B_1, \ldots, B_n$ is the goal associated with the node, B_1 being the chosen predication, and A `:-` $C_1, \ldots, C_m$ is a freshly renamed clause with B_1 and A unifiable, then the corresponding child is labelled with the local substitution which is the *MGU* σ of B_1 and A, and the current goal which is the sequence of predications obtained after flattening, $\sigma(C_1, \ldots, C_m, B_2, \ldots, B_n)$.

  The children are in the same order as the clauses in the database. A *left-to-right* order of the children will be assumed.

At every node a *current substitution* may be computed as the composition of all the local substitutions along the path starting from the root up to that node (inclusive). To every success node there corresponds an *answer substitution* which is the current substitution of that leaf, restricted to the variables of the initial goal.

Notice that the notion of search-tree depends only on the predication-choice. The clause-choice specifies the way it is visited. Given a search-tree, the execution of a goal in the context of a database, with the Prolog Standard fixed choices, may be represented by a **depth-first left-to-right** visit of the search-tree. This visit defines the visit order of the nodes of the search-tree, hence the order of execution of goals and subgoals. If the search-tree has infinite branches, there is no way to visit beyond the first one, which will be explored indefinitely. This explains why the execution does not terminate when the traversal visits an infinite branch. It also explains why not all solutions may be computed if there is an infinite branch with some success branches afterwards.

Consider the following database and the goal `p(U, V).`

```
p(X, Y) :- q(X), r(X, Y). | q(a) :- true. | r(b, b1) :- true.
p(X, Y) :- s(X).          | q(b) :- true. | r(c, c1) :- true.
s(d) :- true.             | q(c) :- true. |
```

Figure 4.3 shows the corresponding Prolog search-tree with the chosen predication underlined. Fresh renaming is denoted by new variables and local substitutions are represented before the node, beside the incoming arc.

The current substitution computed at the second success leaf is {`U/c`, `V/c1`, `X/c`, `Y/c1`}. The corresponding answer substitution is {`U/c`, `V/c1`}. The tree walk produces the following answer substitutions, in this order: {`U/b`, `V/b1`}, {`U/c`, `V/c1`}, {`U/d`}.

Search-tree visit and construction algorithm. The following algorithm synthetises the general resolution algorithm and the concept of search-tree, describing the construction and visit of a search-tree simultaneously. The algorithm represents the execution of a goal with a given database [6].

It is defined in two parts: the "down walk" similar to the general resolution algorithm and the "backtracking" which corresponds to the choice of a not yet visited computation path. So, instead of simply "stopping", the algorithm will continue towards a backtracking step.

Let P be the current database.

1. Start from the root as *current node*, labelled by the initial goal G, which is a sequence of predications, as *current goal*, and by the empty substitution as *local substitution*,

2. If the goal G of the current node is `true` then **backtrack** (*success*), otherwise

3. Let A be the first predication in G,

4. If A is `true`, delete it, and proceed to step (2.) with the new current goal being the tail of the sequence G (if the tail is empty then replace it by `true`), otherwise

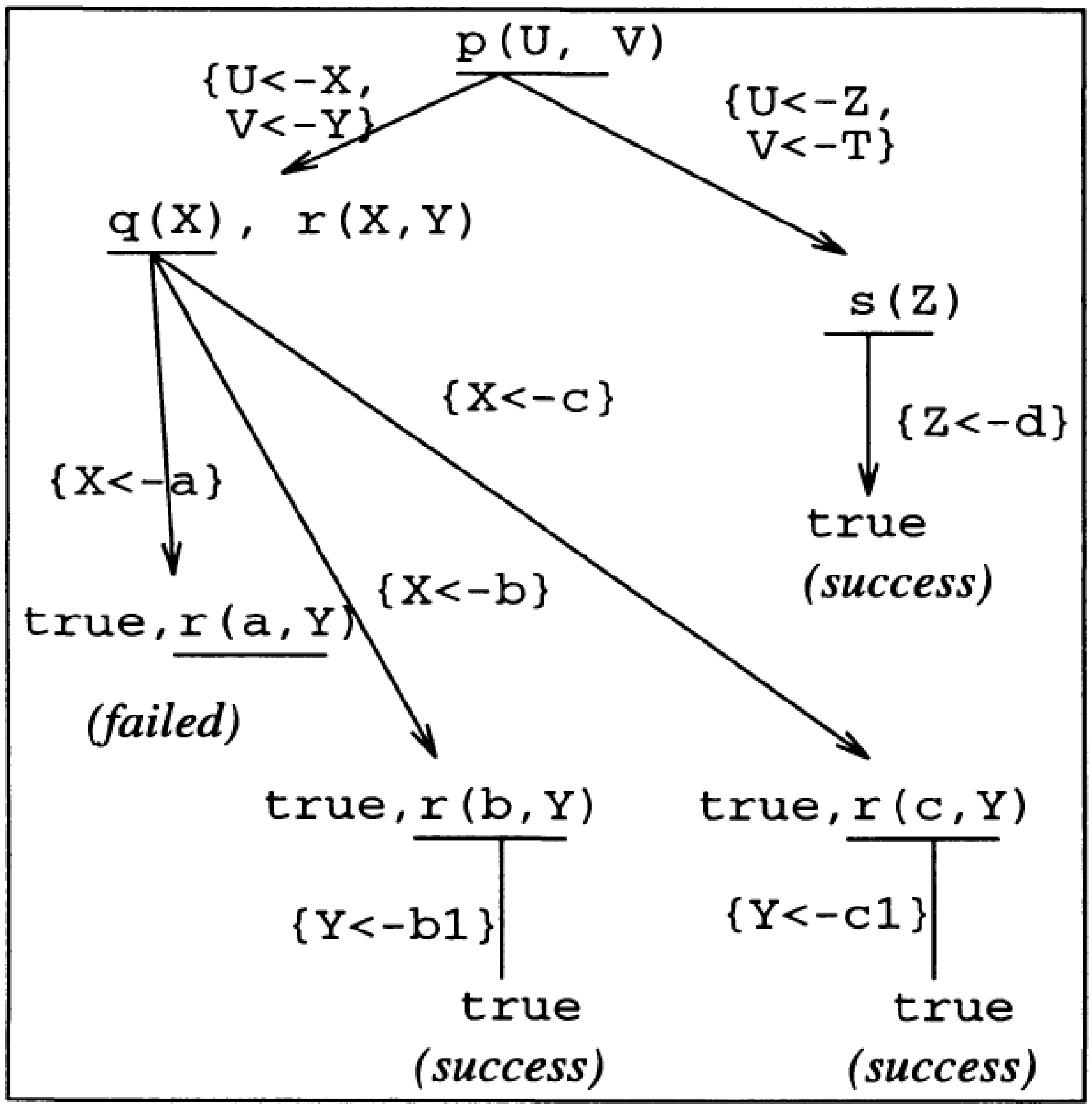

Figure 4.3: A Prolog search-tree example. Image taken from *Prolog: The Standard - Reference Manual* [6]

5. If no renamed clause in P has a head which unifies with A then **backtrack** (*failure*), otherwise

6. Add to the current node as many children as there are freshly renamed clauses H `:-` B in P whose head is unifiable with A with the same order as the clauses in P.

   The child nodes are labelled with a local substitution σ, which is the *MGU* of A and H (H `:-` B being the corresponding freshly renamed clause), and the current goal G' which is the instance by σ of G in which A has been previously replaced by B and which has been flattened.

7. The current node becomes the first child and proceed to step (2).

The new current substitution is obtained by composing all the local substitutions along the path from the root up to the current node (inclusive).

If a node has more than one child, it is *non-deterministic*. Such a node for which A is *re-executable* is called a *choice point*. If a node has only one child after its first visit it is a *deterministic node*. A node is said to be *completely visited* after all the branches issuing from it have been completely developed.

This algorithm describes how to walk down until success or failure is reached. The continuation (**backtrack**), which consists of visiting again a node which has not yet been completely visited, is called *backtracking*.

After constructing a branch terminated by a success or failed leaf-node, the possible nodes which may be visited are nodes with still non-visited children "on the right" of that branch (considering a left-to-right order of the children, this is illustrated in Figure 4.4). These nodes may be reached by seeking the first ancestor node with a not yet visited child.

The new current node is the first child not yet visited and the execution continues at step (2). If there are no more non-visited children, the execution of the initial goal is achieved.

The execution model for Prolog Standard. Up to now it has been assumed that the procedures were user-defined procedures. The model extends straightfor-

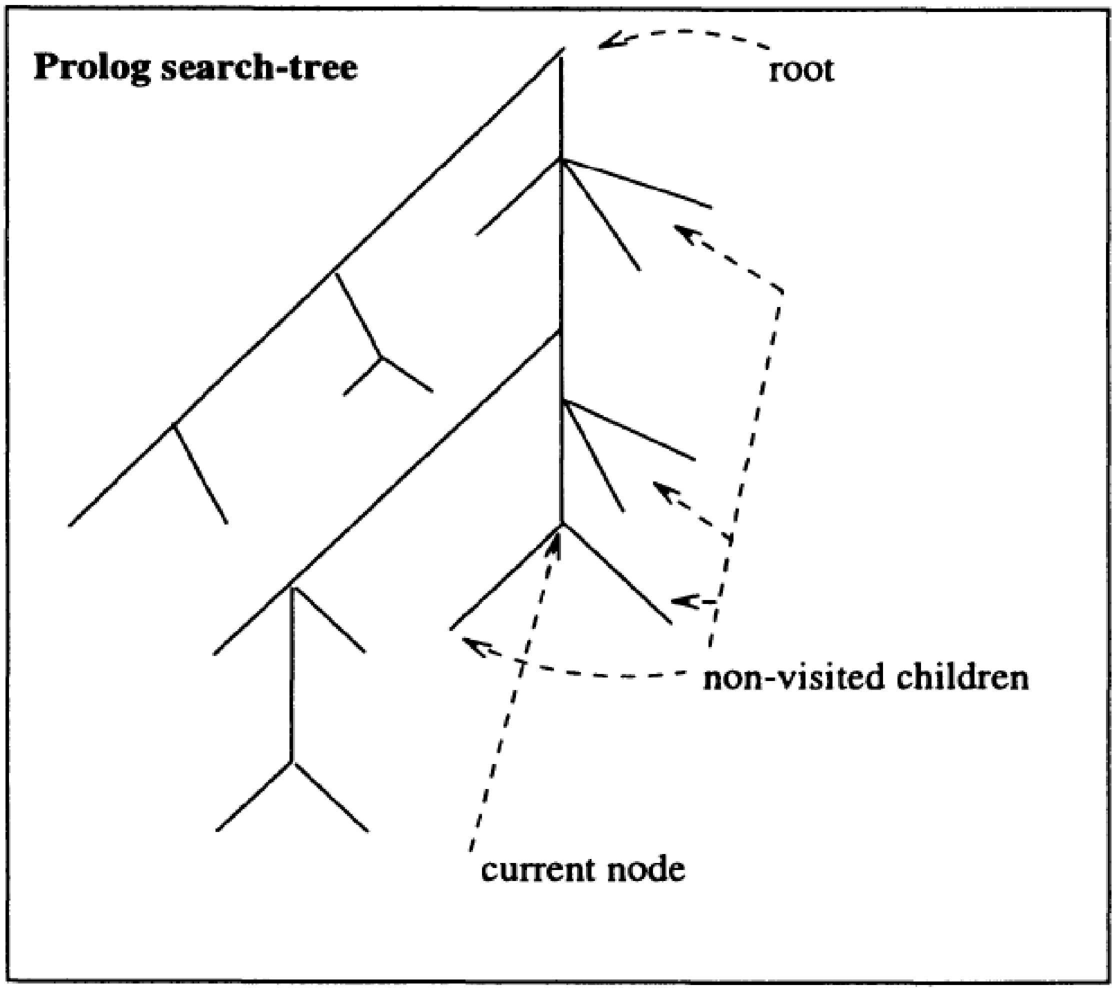

Figure 4.4: A Prolog search-tree and its non-visited children. Image taken from *Prolog: The Standard - Reference Manual* [6]

wardly to built-in predicates: the search-tree is visited and constructed according to the Prolog search-tree visit and construction algorithm as long as no built-in predicate is chosen.

This algorithm is thus adapted to describe the execution of a goal in the context of a given environment (clause database and Prolog flags) [6].

Steps (5) and (6) of the search-tree visit and construction algorithm are modified as follows to take into account built-in predicates and their side-effects.

Three cases have to be distinguished (notice that A is different from `true`, since this case is already considered in steps (2) and (4)):

- If A *corresponds to a user-defined procedure which exists in the database*, then the execution continues as indicated in steps (5) and (6) of the Prolog search-tree visit and construction algorithm.
- If A *does not correspond to any existing procedure*, then the action depends on the value of the Prolog flag `unknown`; if its value is:
  - `error`: an error is generated whose effect corresponds to the execution, at the same node, of the built-in predicate `throw(existence_error(procedure,` PI`))`, where $PI \in \mathcal{L}(\langle Compound \rangle)$ is the predicate indicator of the chosen predication A; the effect of this predicate is defined in the description of the standard built-in predicate `throw/1` [6].
  - `warning`: an implementation dependent warning is generated, and the current goal fails (*failure*).
  - `fail`: the current goal fails (*failure*).

  In both last cases the execution continues doing *backtracking.*
- If A *corresponds to a built-in predicate defined in the Standard Prolog or a custom system procedure (in an extension of Standard Prolog)*, then it is executed and it can:
  - *succeed* with no other alternative, with local substitution σ. Thus a new unique child is added whose labels are the local substitution and

the instance by σ of the tail of the current goal. If the tail of the current goal is empty, the goal label is just `true`. The execution continues at step (2) with the new child as the current node.

- *succeed* with more than one alternative, with possibly different local substitutions. Thus several children are created according to the order specified in the description of the built-in predicate. Each child is labelled with the corresponding local substitution as specified in the description of the built-in predicate and a goal which is the instance by the local substitution of the tail of the current goal. If the tail of the current goal is empty, the goal label is just `true`. The execution continues at step (2) with the first child as the current node.
- *fail*, thus the execution continues as indicated in backtracking.
- generate an *error*. The execution is interrupted and an error is generated whose effect corresponds to the execution at the same node of built-in predicate `throw/1` whose argument is `error(error_term, impl_def)` where `error_term` in described with the error cases in the built-in predicate description and `impl_def` is an implementation defined term. If several errors are generated by a built-in predicate, the error that is reported is implementation dependent.

The side-effect of cut. Here we give some details of the built-in predicate "cut" (`'!'/0`) aimed at performing some control by pruning the search-tree.

"Cut" always succeeds, but it has a drastic side-effect on the search-tree: it deletes some search-tree branches in order to force a predication to execute quickly without constructing and visiting all sub-search-trees.

For example, if the first clause of the clause database previously defined, is replaced by `p(X, Y) :- q(X), !, r(X, Y).`, Figure 4.5 shows that the search-tree corresponding to the goal `p(U, V)`, depicted in the Figure 4.3, now has only one failed branch.

The effect of the "cut" is thus to erase all the hanging nodes between the current node and issued from the parent node of the goal which was containing it.

Doing so, all the nodes which have been made deterministic will be skipped when backtracking takes place.

The side-effect of throw-catch interaction. Finally we give some details of built-in predicates `throw/1` and `catch/3`, aimed respectively at throwing and managing exceptions and whose behaviour changes the search-tree.

Throw. The execution of `throw(`E`)` with $E \in \mathcal{L}(\langle Term \rangle) \setminus \mathcal{L}(\langle Variable \rangle)$ seeks in the search-tree for the closest ancestor node whose chosen predication has the form `catch(Goal, Catcher, RecoverGoal)`, which is still executing its `Goal` argument (and not `RecoverGoal`) and such that a freshly renamed copy E' of E unifies with `Catcher` by substitution σ.

- If there is no such ancestor, then a `system_error` is raised and the behaviour is implementation dependent.
- If there is such ancestor node whose goal label is `catch(Goal, Catcher, RecoverGoal` then *(i)* all the nodes between the current node and the ancestor one, are made deterministic (none of these nodes can thus be selected by *backtracking*) and *(ii)* a second child is added to the ancestor node whose labels are the substitution σ and the goal `call(RecoverGoal)`σ, and *backtracks*. The execution will thus continue, with the new child as current node, at step (2.) of algorithm for serach-tree visit and construction (Section 4.5.1).

Catch. Captures errors generated through explicit `throw/1` usage, during the execution of a given goal – first `catch/3` argument – or implicitly when an error is raised by the processor—having the same `throw/1` effect. Then executes a recover goal (third `catch/3` argument, after instantiation). The meaning of such primitive, must be considered with the built-in predicate `throw/1`.

Solving the goal `catch(`G`, `C`, `R`)`, where $G \in \mathcal{L}(\langle Term \rangle) \setminus \mathcal{L}(\langle Variable \rangle)$ and $\{C, R\} \subset \mathcal{L}(\langle Term \rangle)$, the execution of `call(`$G$`)` is triggered. If an error occurs during its execution, which is "caught" by C, the resulting instance of R is executed (see Throw definition).

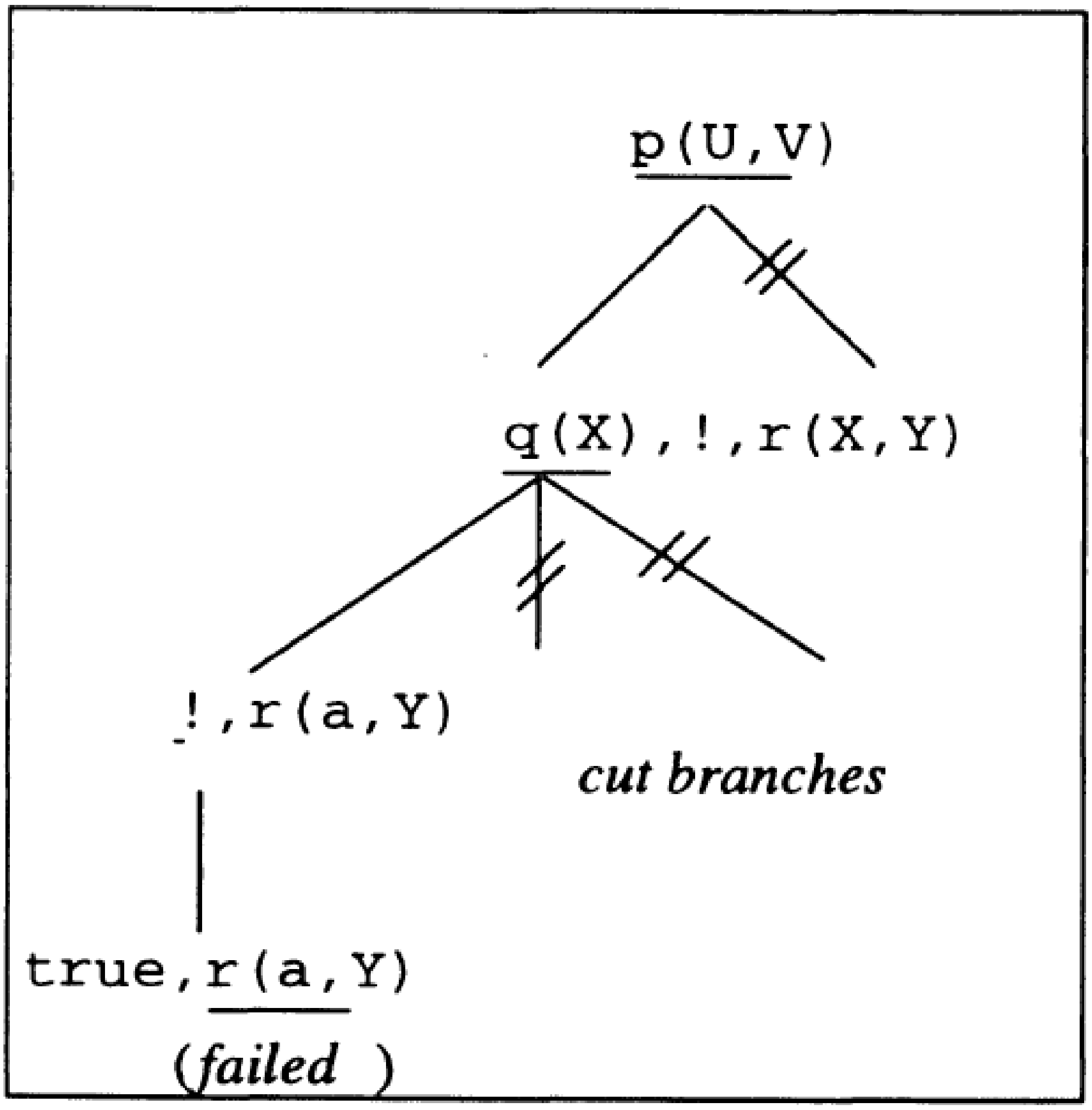

Figure 4.5: A search-tree example showing the effect of Cut. Image taken from *Prolog: The Standard - Reference Manual* [6]

More precisely, if the current goal is `catch(`G`, `C`, `R`)`, it creates a new child whose labels are the empty substitution and the goal `call(`G`)`. The execution will continue at step (2.) of algorithm for search-tree visit and construction (Section 4.5.1).

4.5.2 Main "solve" types

Solution

The `Solution` type captures the concept of outcome of a Prolog computation. It always has a reference to the solved goal and to the substitution that was applied to solve it. It has three sub-types:

- `Yes`, meaning that the Prolog goal was executed successfully;
- `No`, representing the Prolog failure response;
- `Halt`, reporting that the computation stopped abruptly due to an unhandled error.

Solve requests and responses

In order to be able to support asynchronous interactions with a Prolog goal solver, we separated the concepts of `SolveRequest` and `SolveResponse` and gave them a first-class representation. The design was inspired by *stateless web interactions*, asynchronous by construction, and by *REST* principles.

`SolveRequest` represents a request to solve a Prolog goal, carrying with it all the information necessary for its solution, such as:

- the goal `Indicator`,
- the goal argument list,
- the `ExecutionContext` (described below) into which the goal should be solved.

`SolveResponse` represents a response to a previously submitted solve request, carrying with it the mere `Solution` along with the possible side-effects caused by the goal execution. These side-effects will be computed starting from the `SolveRequest` represented state.

Those request and response concepts are designed to be dealt with, only if the programmer is developing a Prolog library, that is an extension of Prolog language. Programmatic usage of the solver should not expose these internal details, somehow low-level.

Primitive

`Primitive` is a function that given a `SolveRequest`, with a particular `Indicator`, produces a *lazy sequence* of `SolveResponse`s. A primitive is so called, because it is not implemented in Prolog language, but in the native language in which the Prolog solver is written. It is very useful to write some Prolog predicates implementations in a procedural language. Every programming paradigm is really good at doing something particular and with primitives we give the chance, to library developers, to use the best-fit paradigm in writing functionalities.

Function

`Function` concept is somehow similar to the primitive one but this time we have `ComputeRequest` and `ComputeResponse`. However, these have the same conceptual meaning of solve requests and responses. A `ComputeResponse` instead of carrying a solution, carries the function computed `Term`. A `Function` maps a "compute request" into a *single* "compute response".

The function concept will be employed to represent Prolog Standard arithmetic. In fact some terms represent arithmetic expressions when they are in the following positions:

- the right-hand argument of the built-in predicate `is/2`,
- both arguments of the *arithmetic comparison* built-in predicates `'=:='/2`, `'=\='/2`, `'<'/2`, `'>'/2`, `'=<'/2`, `'>='/2`.

In that case they will be formed as a term, *using arithmetic functors and numbers only*, otherwise the evaluation of the expression raises an exception [6].

Library

A `Library` is a container for:

- `Operators`, because each library could declare theirs,
- Prolog theories, in the form of `ClauseDatabase`, defining custom library predicates,
- `Primitives`, defining custom library predicates implemented in an imperative way,
- `Functions`, defining custom functions to be used in evaluating Prolog expressions.

During resolution process more than one library could be loaded at the same time. This poses the problem of name clashes between predicate indicators.

We provide also a `LibraryAliased` type, that is a `Library` with a characteristic alias attached. This enables the solver to search predicates with their plain name and also with their "fully-qualified" name (i.e. alias plus the predicate indicator name).

ExecutionContext

The `ExecutionContext` type captures the homonym concept. It contains important context information that determines the solver behaviour, such as:

- Loaded libraries, that will be used during resolution process along with user defined theory.
- Enabled flags, that will slightly modify the resolution process behaviour and set some solver parameters.

- Static Knowledge-Base, a `ClauseDatabase` containing user's not-modifiable clauses.
- Dynamic Knowledge-Base, a `ClauseDatabase` containing user's modifiable clauses; the modifications can be done using the appropriate built-in predicates.
- Substitution, that will contain the current substitution in the specific context.

The `ExecutionContext` is a block of information that is replicated for each node of the search-tree, labelling it.

Solver

The `Solver` type represents the "Prolog engine" that will navigate the search-tree finding solutions. Given an initial `ExecutionContext`, possibly empty, it will lazily solve a given goal.

The laziness in the resolution process is necessary because the Prolog execution model could potentially produce infinite computation paths. So when a solution is found, before proceeding in the search-tree exploration, the solution has to be reported.

We also provide the possibility for the `Solver` user, to specify a *maximum execution duration*. This will help bounding the amount of time needed to search a solution, in those cases where if no solution is found rapidly, it will not matter anymore.

Exceptions and PrologError

Prolog language has exceptions. Our design commits at implementing them as exceptions in the implementation language, with a one to one mapping. The `TuPrologRuntimeException` is the basic type for exceptions thrown during resolution process. It will always contain an `ExecutionContext` that helps to understand because and where, during the resolution, the exception happened.

Some notable exception sub-types could be:

- `HaltException`: when it is thrown, the resolution process should halt immediately.
- `TimeOutException`: it is thrown when the execution max duration has been exceeded and so the execution should terminate.
- `PrologError`: this is an exception thrown at Prolog language level. The Prolog ISO standard in fact describes its own exceptions and the way they should be manged. All Prolog language level exception should be sub-types of `PrologError`.

  The Prolog ISO standard also states that when a Prolog exception cannot be handled properly, a `SystemError` (sub-type of `PrologError`) should be raised. If even this exception could not be managed we halt the resolution process by means of an `HaltException`.

A Prolog library programmer can throw an exception either by using the native language keyword `throw` along with the exception instance or launching the resolution of a `throw/1` predicate.

4.6 Solve-classic module

The solve-classic module contains the same inferential core of *tuProlog* but in a slightly reviewed design. The solver provided by this module could be seen, to all effects, as an enhanced *tuProlog* [1].

4.6.1 Inferential core design

The inferential core of classic *2p-kt* solver is designed as a *Finite State Machine* (FSM), whose basic architecture is depicted in Figure 4.6. The machine is composed by *(i)* an initial state, assumed as the solving procedure starts, *(ii)* seven main states representing the activities performed during the resolution process, and *(iii)* four final states which identify the different ways a demonstration may be ended.

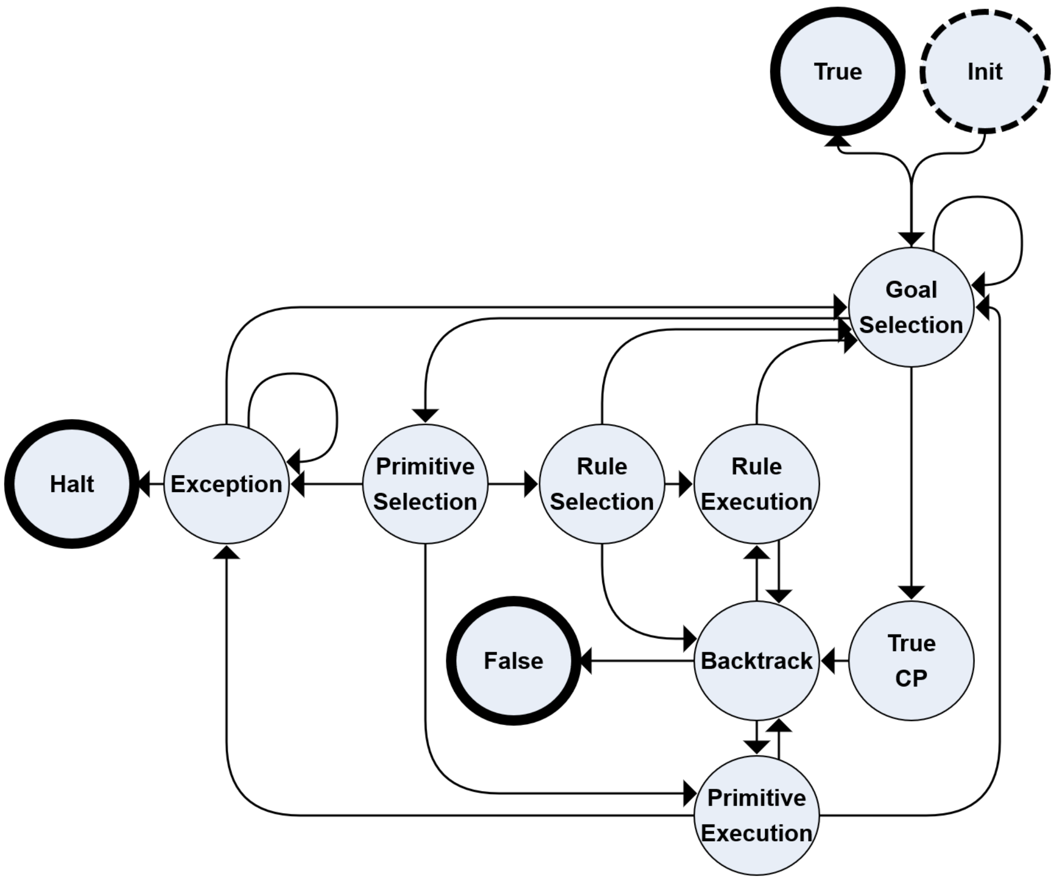

Figure 4.6: The solve-classic solver core *Finite State Machine*.

One of the differences from *tuProlog* design is the increased number of states. We tried to confine better responsibilities of each state, for example separating selection of primitives from their execution. On the other hand we remained adherent to the original design, leaving the management of some Prolog primitives to the inferential core itself, like `','/2`, `'!'/0`, `fail/0`, `throw/1` and `catch/3`. This complicates to some extent the inferential core design.

Furthermore, is worth pointing out that the following described FSM design, is made with in mind the idea that the state machine *is one during the whole resolution process.*

Init state. The *Init* state is the main entry point of the FSM when starting a demonstration process and, as its name declares, initialises the core by extracting the subgoals to evaluate from the query, and by setting up an object which represents the *execution context* for the current subgoal. That subgoal is chosen by the next state, to which the *Init* state immediately passes control.

Goal Selection state. The *Goal Selection* state checks if there is a next goal to be executed from the subgoal list.

- If such a goal *does not exist* because the resolvent is empty, the current context is checked:
  - if it is the root context, and there are *open alternatives* to be explored, then the demonstration has ended and the machine ends up in *TRUE_CP* state, where a further solution to the initial query can be asked;
  - if it is the root context, and there are *no open alternatives*, then the demonstration has ended and the machine stops into *TRUE* state;
  - else, the machine shifts to *Goal Selection* state again, adjusting current context to hold parent next goals, and the resolution process continues.
- If the resolvent is not empty and there is a next goal, the machine shifts to *Primitive Selection* state.

Primitive Selection state. The *Primitive Selection* state deals with evaluating current subgoal which a previous state has selected from the resolvent.

- If the indicator of such subgoal is bound to a primitive predicate, its execution is triggered, a new goal scope is opened (thus a choice point is saved), and the machine gets shifted to the *Primitive Execution* or the *Exception* state, depending respectively on the success or error of primitive triggering.
- If the indicator of the subgoal does not represent a primitive, its evaluation can only be performed by browsing the logic theory contained in the engine

and selecting a compatible clause. These actions are performed in the *Rule Selection* state.

Primitive Execution state. The *Primitive Execution* state deals with the solutions produced by primitive triggering. When the solution is:

- *Yes*, the control passes to *Goal Selection* state, with a context modified to point to the next goal in list.
- *No*, the machine shifts into *Backtrack* state.
- *Halt*, the machine shifts into *Exception* state.

Exception state. The *Exception* state would be assumed by the machine whenever an error occurs during the execution of a Prolog subgoal. The effect of a run-time error should be the same as the invocation of a `throw/1` primitive.

- If the thrown exception is a *prolog error*, the *Exception* state job should be to carry out the procedural side effect of `throw/1`: causing the normal flow of control to be transferred back to an existing call of `catch/3` which matches the thrown error. Keeping the Prolog term representing the error, the *Exception* state would need to trigger a backwards visit of resolution tree composed of *execution context* objects, in order to find the appropriate `catch/3` subgoal, which needs to feature a second argument unifiable with the error to be caught. The search is made recursively:
  - if current context is the root one, no parent goal can handle the error, then the machine stops the computation in the *HALT* state.
  - if current goal is not `catch/3`, then the machine shifts again to *Exception* state, switching current context with its parent context.
  - if the second argument is *not unifiable* with the thrown *prolog error*, then the machine behaves as described in previous point.

  - if the second argument is *unifiable*, the unifier is composed with local substitution producing σ; then σ is applied to `catch/3` third argument *RecoverGoal*; *RecoverGoal*σ is used as source for new goals to be submitted for resolution and σ becomes the new local substitution; the machine hence shifts to *Goal Selection* state with the newly computed context, to continue its execution.

  During the search, each traversed *execution context* would be pruned, so as not to be executed or selected by backtracking mechanism anymore.

- If the thrown error is *not a prolog error*, then the machine stops the computation in the *HALT* state.

Rule Selection state. When entering the *Rule Selection* state, given the current goal G:

- If G is `true`, then the machine shifts to *Goal Selection* state adjusting context to point to the next goal in list.
- If G is `fail`, then the machine shifts to *Backtrack* state.
- If G is `'!'/0` (Cut), then the machine goes into *Goal Selection* state, updating current context to point to the next goal and deletes (cuts) all eligible choice points.
- Otherwise a set of rules compatible with current subgoal G is gained from the logic theory stored in the engine.
  - if that set is *empty*, then a backtrack is triggered by setting the machine's next state to *Backtrack* state;
  - otherwise, having found at least a compatible rule with the current subgoal, a new goal scope is opened (thus a new choice point is saved) and the machine gets shifted to *Rule Execution* state.

Rule Execution state. The *Rule Execution* state deals with evaluating a previously selected rule H `:-` B. Here the unifier computation, between current goal and the currently selected rule head (H), is performed:

- If the the two terms are *unifiable*, with σ, then the rule body (B) evaluation is prepared: a new execution context is created with $B\sigma$ as subgoal source and σ as local substitution, then the machine shifts to *Goal Selection* state.
- If the two terms are *not unifiable*, then the machine shifts to *Backtrack* state.

Backtrack state. Once entered the *Backtrack* state, a check on the set of opened choice point is performed:

- If that set is empty, a transition to the *FALSE* state is performed, to immediately make the demonstration fail.
- If the set is not empty, the next alternative is fetched from last added choice point; the new current context is updated to consider the fetched alternative as "no more selectable" and:
  - if the last added choice point was a *rule choice point*, then the machine shifts to *Rule Execution* state;
  - else the last choice point added was a *primitive choice point*, hence the machine goes into *Primitive Execution* state.

End states. Among the four mentioned end states (*TRUE_CP*, *TRUE*, *FALSE*, *HALT*), the *TRUE_CP* one is not properly an end state. While the others can't produce a subsequent state, because signaling a type of computation end, the *TRUE_CP* one can transition to *Backtrack* state, although it also has the function to signal a successful computation end.

4.6.2 Main "solve-classic" types

The actual design of the finite state machine is realised using the *State pattern* [26]. This design pattern is typically used to allow an object to alter its behaviour when its internal state changes, by modelling states as objects which encapsulate the different behaviours. In our case, the solver plays the role of the object whose state changes during the course of a computation, and whose state is therefore modelled as a set of objects with their own behaviour.

State. The class hierarchy representing the core states is based in the interface `State`, providing the `next(): State` method which contains the state's behaviour in the concrete sub-classes and which returns the state to move to. From `State` all other classes representing the core's states are derived: `StateInit`, `StateRuleSelection`, `StateBacktrack` and so on, with a clean name pattern, matching state names in the FSM. `StateEnd` will extend `State` adding properties to retrieve the computed `Solution`. `StateException` will have an additional property to contain the exception that caused the state machine to go into that state.

ChoicePointContext. `ChoicePointContext` is a type needed by states to keep track of if and which choices has been chosen during resolution process and hence are useful during *backtracking*. There are two kind of `ChoicePointContext`s: the primitive ones keep track of which responses of primitives have been already considered; the rule ones keep track of which rules have been already chosen for execution. Every choice point context can be asked if there are open alternatives, i.e. some not already chosen computational paths.

4.7 Solve-streams module

The solve-streams module brings with it a very different solver type from that present in solve-classic. This solver has been designed from scratch, with Prolog search-tree and functional programming in mind.

4.7.1 Rationale

Backtracking step removal. When navigating the Prolog search-tree, as already said, we encounter choice points. The point is: *what if, instead of making a choice at some point, we always make all possible choices and "forget" about that choice point?* Instead of needing to remember it, and to backtrack till this point to make another choice and execute, we could prepare a sequence of all execution contexts that should be carried out from this point, and merely use them when the execution of current "branch" has been completed.

This basic idea removes the need of an *explicit backtracking* mechanism, because when at "choice point time" we prepare all the execution contexts, they don't have any side effect of any branch execution inside them, because none has been already executed. So the *implicit backtracking* mechanism resides in selecting a previously prepared context, with no side effect inside it.

The tricky part is that of implementing primitives that alter flow control like `'!'/0` (Cut) and `throw/1` (Throw). These primitives should somehow affect the inferential core in a way that it can determine which already prepared context should be discarded, hence not executed.

The solution we have found, can be a "side effect" field in the `SolveResponse` type that can be read by the "upper scope executor" and, if applicable, it can discard its prepared contexts and continue.

Seeking minimality. Trying to be as minimal as possible, we thought at which mechanisms were really needed at the inferential core. We firstly ended up with this list:

- A way to determine if a computation can end with success.
- A mechanism to unify the current goal with database rule heads and execute their bodies.

These mechanisms describe the simplest inferential core, that makes a rigid pattern matching of a rule head with a goal and makes its body the new current goal, until the current goal matches the success condition expressed at first point, or otherwise fails.

To widen the capabilities of such inferential core, enabling it to the execute the whole Prolog language, we just added to the list:

- A mechanism to find and execute primitives.

So primitives revealed themselves as the docking point for every other Prolog feature.

4.7.2 Inferential core design

The inferential core of streams *2p-kt* solver is designed as a FSM, whose basic architecture is depicted in Figure 4.7. The machine this time is composed by *(i)* an initial state, assumed as the solving procedure starts, *(ii)* two main states representing the activities performed during the resolution process, and *(iii)* three final states which identify the different ways a demonstration may be ended.

Starting from the idea explained before, every state computes all its possible subsequent states at the same time.

Is worth pointing out that the following described FSM desing, is thought with in mind that *during the resolution process the state machine can start another instance of itself*, nesting the execution.

Init state. The *Init* state is the main entry point of the FSM when starting a demonstration process. It initialises the state machine preparing it for potential "side effects" execution. Then, it takes the current goal G and:

- If the *success check strategy* says that G is solved, shifts to *TRUE* state.
- If G is *not a well formed* goal, shifts to *FALSE* state.
- If G is *a well formed* goal, prepares the goal for execution (i.e. if it is a variable, then it wraps the variable in `call/1`), sets the current goal to the prepared goal, and finally goes into *Goal Evaluation* state.

The *Init* state always goes into a single subsequent state.

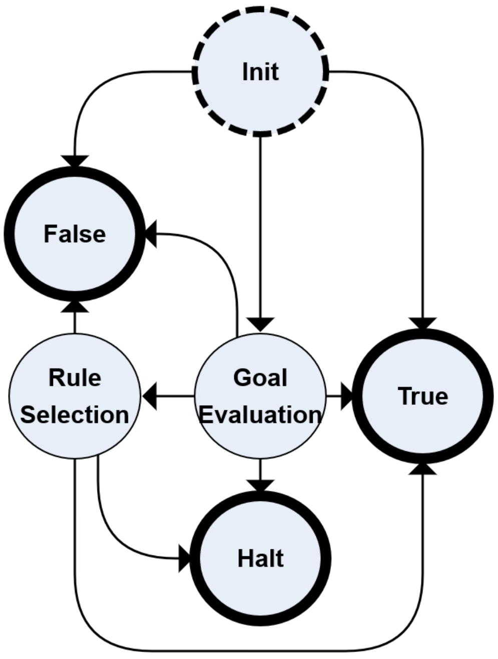

Figure 4.7: The solve-streams solver core *Finite State Machine*.

Goal Evaluation state. The *Goal Evaluation* state deals with evaluating the current goal as a primitive, if applicable.

- If the indicator of current goal is present in loaded libraries primitives, then the primitive execution is triggered:
  - if the primitive succeeds executing, it shifts into a state depending on the `Solutions` type; that is, if the solution is *Yes*, *No*, *Halt* respectively goes into *TRUE*, *FALSE* or *HALT* states; we call that mechanism *solution forwarding*; the number of next states depends on the number of solutions returned by primitive execution;
  - if the primitive throws a *prolog error*, a brand new resolution process is started with the goal `throw(`*prolog error*`)`, leaving the duty of implementing "throw" behaviour to an external `throw/1` primitive; for responses coming from this sub-execution we use *solution forwarding* again.
  - if the primitive throws an `HaltException`, shifts to a single *HALT* state.
  - if any of the forwarded solutions coming from the two initial points, are `Halt` solutions, then all remaining *prepared contexts* are discarded.
- If the indicator of current goal doesn't match any loaded primitive, then it shifts to the single *Rule Selection* state.

Rule Selection state. When entering the *Rule Selection* state, current goal is used to load all matching rules from the engine `ClauseDatabase`.

- If no matching rule is found, it ends up in a single *FALSE* state.
- If matching rules are found, they are prepared for execution, then ordered through a given *clause choice strategy.*

  For each rule H `:-` B, let σ be the unifier computed between current goal and H; $B\sigma$ is the sub-goal to be executed, thus an *execution context is prepared* for it.

Then, for each *prepared execution context*:

1. a sub-state-machine is initialized with it and its execution is triggered, making it start from *Init* state;
2. when the sub-state-machine execution ends up in a final state responding to our commissioned sub-goal;
   (a) the "side effect" field is checked to know whether other *prepared context* should be discarded or not; if they should be discarded, then this "discard information" is saved for further usage;
   (b) the sub-solution is *forwarded*, if it isn't a *No* solution or it is a *No* solution but hasn't alternatives;
   (c) if the sub-solution is an `Halt` solution, then the other *prepared contexts* are discarded; the same happens if the "discard information" is present.

The number of *Rule Selection* subsequent end states depends on the number of succeeded rule bodies execution, plus a failed one, if present.

End states. All mentioned end states (*TRUE*, *FALSE*, *HALT*) are properly end states, meaning that no further computation is performed starting from them. After an end state is reached, that branch of the Prolog search-tree is considered to be completely explored and subsequent *prepared contexts* should be executed, further exploring the search-tree.

Or-parallelism. The proposed design paves the way for optimisations exploiting some kind of computation parallelism.

In particular the *or-parallelism* originates from parallelising the execution of rules in *Rule Selection* state. In fact, if more than one rule head matches the current goal, parallel computations could be carried out for each rule body, making some sort of speculative execution (speculative because, in case a `'!'/0` is executed, all cut subsequent computations results have to be dropped).

4.7.3 Default primitives

The “solve-streams” inferential core is very simple and has no embedded primitive in it. To reach “solve-classic” one capabilities and adhere to Prolog Standard execution model, all basic primitives should be provided as part of the solver. Note that Prolog Standard execution model takes these primitives for granted.

Conjunction. This primitive execution is triggered when solving a goal whose principal functor is `','/2`.

Its behaviour can be described as follows:

1. Given the current conjunction goal, all “in-conjunction” sub goals are extracted and ordered with a given *predication choice strategy*; an empty substitution σ is prepared to be composed with sub goals produced substitutions, in order to “accumulate knowledge” over previously executed sub goals;
2. Let G be the first sub goal in conjunction;
3. $G\sigma$ is prepared for execution producing G';
4. A new solver instance is created to solve G', resulting in its solutions;
5. For each G' solution S (notice that at every loop iteration, previous side effects should be removed):
    (a) If S is *Yes* and we have other “in-conjunction” goals to be solved and no `throw/1` has been executed successfully, then this procedure is executed again from (3.) with G being the next sub goal and σ being current σ enriched with S new substitutions.
    (b) Otherwise if S is not *No* or it is *No* and there are no other open alternatives, then the S solution is added to those given in output.

At any point of execution if a Cut or Throw is executed, all previously opened alternatives are cut; this in particular means that in point (5.) all G' solutions S after a cut execution should not be considered, breaking the “for each” loop.

Cut. This primitive execution is triggered when solving a goal whose principal functor is `'!'/0`. In our design it has the only responsibility of responding *True* solution specifying as "side effect", the Cut one.

Call. This primitive execution is triggered when solving a goal whose principal functor is `call/1`. It is needed at this level, near the inferential core, because it has particular behaviour with `'!'/0`, and `catch/3` primitive is defined on top of it.

The effect of a Cut occurring inside a `call/1` goal is limited to this goal, i.e. it should have no effect outside of it. `call/1` is said to be *opaque* (or *not transparent*) to Cut.

Its behaviour can be described as follows:

1. If the `call/1` argument G is a `Var`, then the *prolog error* `instantiation_error` is thrown.

2. Otherwise, if G is *not well-formed* goal, then the *prolog error* `type_error(callable, G)` is thrown, where `G` is the `call/1` argument.

3. Otherwise, a new solver instance is created to solve G, after being prepared for execution, and its solutions are *forwarded* as responses of the `call/1` request; all changes made by `'!'/0` during this resolution are confined inside the `call/1` "scope".

Throw. This primitive execution is triggered when solving a goal whose principal functor is `throw/1`.

The `throw/1` primitive works as a Cut between the triggering point in the search-tree till the matching `catch/3`. However this type of "cut" can't be confined by `call/1` primitive as the normal Cut.

Its behaviour can be described as follows:

1. If the `throw/1` argument A is a `Var`, then the *prolog error* `instantiation_error` is thrown.

2. Otherwise the stack of solving goals, till this point, is checked for the presence of an ancestor `catch/3` goal, whose second argument unifes with A:
   (a) if there is no ancestor catch or the unifier can't be computed, because the two terms does not unify, then:
      i. if the thrown error is a `system_error`, then the resolution process is halted;
      ii. otherwise a `system_error` is thrown.
   (b) if the two terms unify, by unifier σ, then σ is composed with local context substitution; the primitive succeeds with *Yes* solution and the solution "side effect" is set to the Throw one.

Catch. This primitive execution is triggered when solving a goal whose principal functor is `catch/3`. This primitive is complementary to the `throw/1` one.

Its behaviour can be described as follows:

1. The first argument G of `catch/3` is extracted;
2. G is wrapped into a `call/1` primitive call, becoming `call(`G`)`;
3. A new solver instance is created to solve `call(`G`)`;
4. For each solution S coming from that execution:
   (a) if this `catch/3` goal was not selected by a `throw/1` primitive, all responses are *forwarded* as they are; note that `catch/3` without any `throw/1` primitive executed, behaves just like `call/1`;
   (b) otherwise if this `catch/3` goal was selected by a `throw/1`, then the last `catch/3` argument is taken and S substitution applied to it;
   (c) the "recover goal" R, thus obtained, is wrapped into a `call/1` primitive, resulting in `call(`R`)`;
   (d) current `catch/3` is made no more selectable from other `throw/1` calls;

(e) then a new solver instance is created to solve `call(`R`)` and all solutions coming from this execution are *forwarded* as solutions of the principal `catch/3` goal.

4.7.4 Main "solve-streams" types

The actual design of the described FSM is realised also this time exploiting the *State pattern* [26].

State: The `State` type defines the base interface for every state. It provides a `behave(): Sequence<State>` that returns the lazily initialized sequence of subsequent states.

IntermediateState: The `IntermediateState` type defines a non-final state, thus carrying a `SolveRequest` used during resolution. *Init*, *Goal Evaluation* and *Rule Selection* states are all `IntermediateState`s.

FinalState: The `FinalState` type defines a final state, hence carrying a `SolveResponse` to some request. *TRUE*, *FALSE* and *HALT* states belong to this category.

SideEffectManager: It is a type capturing the concept of "side effect" that has to be reported to external executors. It will encapsulate the logic for side effects detection and execution.

SolverStrategies: It is a type capturing all the strategies used during resolution process. It is an application of the well-known *Strategy pattern*. In particular we have three strategies, for the solver:

- *success check strategy*, that determines when a goal should be considered solved; the default is an equality check of current goal with `true` atom;
- *clause choice strategy*, that determines in which order clauses are selected when loaded from the `ClauseDatabase`; the default is the Prolog standard clause choice, i.e. the first database matched clause is taken first;

- *predication choice strategy*, that determines in which order the predications in a conjunction are selected for execution; the default is the Prolog standard predication choice, i.e. the left-most predication is solved first.

A different `SolverStrategies` could be passed to the solver to change its default inner workings, without modifying its source code.

4.8 Dsl-core module

The dsl-core module contains the modeling of a *Domain Specific Language* (DSL) for *2p-kt core module* types manipulation.

This DSL is designed to simplify as much as possible the creation of *core* classes mimicking the *Prolog syntax*. It will provide a *prolog* "scope" where a pseudo-*Prolog syntax* can be used to create objects. This scope can be thought as a `Scope` as defined in core module, hence inheriting its concept of variables scoping, with enhanced capabilities.

Some extra-`Scope` needed features are:

- A function to convert any type to the correct Prolog type. It will be used inside any other functionalities to ensure that every type gets converted to the proper `Term`.

- An override of host language default operators to make them behave as creators of `Struct`s. For example, in DSL `5 + 1` should be converted to `'+'(5, 1) Struct`.

- An enhancement of host language `String` type, to enable programmers to use it as a Prolog functor. For example in DSL `"my_functor"("my_arg1", "my_arg2")` should be legal, and should be evaluated to the corresponding `Struct` without apices.

4.8.1 Term creation modalities

With this module we add further modalities to create objects, so it's worth to classify all of them, from scratch, in a clear way. During this classification we will show that higher level construction modalities will be less verbose and much more clear; our running example will be to programmatically create the term `f(A, B) :- g(A), g(B)`.

Core module provides two basic levels of `Term` objects creation:

(i) `Type.of(argument: CorrectType, ...): Type`.

Every core module type has its own **of** *static factory method* attached, which can create objects passing correctly typed arguments. Implementations constructors won't be accessible by design, so this is the only basic way to create an object.

Our example object should have been created by the following method calls: `Rule.of(Struct.of("f", Var.of("A"), Var.of("B")), Struct.of("g", Var.of("A")), Struct.of("g", Var.of("B")))`.

But since every call of `Var` constructor returns a different instance, this won't produce what we expected. In fact the first two `A` and `B` variables, will be different instances from the second ones. This is why in core module we have the `Scope` type and hence a second level of construction methods.

(ii) `Scope.typeOf(argument: CorrectType, ...): Type`.

Inside a `Scope` we gain access to new methods to create objects easier. Pragmatically these methods:

- handle correclty variables;
- forward the creation request to the previous level *(i)* constructors (obviously not for variables whose instances are retained and reused as needed);
- remove the burden to programmers to always type "Type dot of".

Our example object is now correctly created by the following method calls: `ruleOf(structOf("f", varOf("A"), varOf("B")), structOf("g", varOf( "A")), structOf("g", varOf("B")))`.

Now we come to the dsl-core module, which enhances the `Term` creation experience, providing two other levels:

(iii) `Prolog.typeOf(argument: AnyType): Type`.

Inside a *prolog scope* we gain access to the extra-`Scope` features described before and, in addition to all `Scope` methods, each of them is overloaded with a version that accepts arguments not necessarily of correct type. These overloaded methods convert their parameters to correct type, and then forward the call to previous level *(ii)* constructors.

Our example object can now be correctly created by the following method calls: `ruleOf(structOf("f", "A", "B"), structOf("g", "A"), structOf( "g", "B"))` or better `ruleOf("f"("A", "B"), "g"("A"), "g"("B"))`.

We also want to point out that the creation of variables here, uses the current *prolog scope*; this means that if in two different parts of the same *prolog scope* we wrote the above two rule code, the `A`s and `B`s will always be the same, even if in different clauses. This could be a problem if a user would like to write different clauses with different variable scopes, in the same *prolog scope*. In fact the fourth creation level solves this "problem".

(iv) `Prolog.type { DSL written object, then casted to Type }: Type`.

In a *prolog scope* we also have access to this other type of methods. These methods will accept in input a function whose receiver should be a *prolog scope*, then they internally create an and-hoc empty scope and call the given function with it. This is how the variable scope isolation can be achieved.

Our example object can now be correctly created by the following method call: `rule { "f"("A", "B") ‘if‘ "g"("A") and "g"("B") }`.

If this code is duplicated and executed inside the same *prolog scope*, variables of first rule will be different from variables of the second one.

4.9 Dsl-unify module

The dsl-unify module extends dsl-core one functionalities to enable programmers to easily unify terms inside a *prolog scope*.

The *prolog scope* provided by this module features overloaded `Unificator` methods (`mguWith`, `matches` and `unifyWith`) accepting non `Term`s as parameters. These parameters are then converted to `Term`s and forwarded to the common `Unificator` methods.

4.10 Dsl-theory module

The dsl-theory module extends dsl-unify one functionalities to enable programmers to easily write Prolog programs (theories) inside a *prolog scope*.

The *prolog scope* provided by this module adds two ways to create a `ClauseDatabase`:

- a `theoryOf` method that somehow adheres to the *(ii)* type of constructors, hence calling internally the *(i)* i.e. `ClauseDatabase.of(...)`.
- a `theory` method accepting multiple scoped functions producing clauses, which is similar to *(iv)* type of constructors.

The *(iii)* type constructor is not provided. This is because implementing it will break the semantics of the constructor of databases. All variables in the `ClauseDatabase` would be shared, and this is not a good design choice. It could be implemented in a way that doesn't break the semantics, but it would require double computational time, and because we offer a better alternative, it's not worth it.

Chapter 5

2p-kt Implementation

In this chapter we present how the described design has been implemented in *Kotlin* language. Each section of this chapter dives into implementation details of each project module.

5.1 Core module

All core types described in core module design (Section 4.2) are implemented as interfaces. This permits multiple inheritance, enabling us to exactly implement fig. 4.2 hierarchy.

Every interface has its own actual implementation. Implementations are immutable and are internal to the module, i.e. they cannot be instantiated by module users.

Every interface (except for `Term`) has its own companion object containing some *static factory methods*, named `of`, that handle the actual creation of instances. The hierarchy of interfaces is crafted to manage the correct creation of instances even if a super-interface is used to create a sub-type. For instance, if we call the `Struc.of("a")` method, it returns an `Atom` implementation instance and not a `Struct` one. This smart management in instance creation makes possible to check for instance type, not only by means of `isXXX` getters, but even with language type checking "`is`" operator.

All interfaces also define proper methods to access their immutable internals.

Term. The `Term` type defines some notable methods:

- `structurallyEquals(other: Term): Boolean`, that is a method to enable structural comparison of Terms. This is very useful when implementing `ClauseDatabase` clause retrieval, because there structural checks are made.
- `apply(substitution: Substitution): Term`, that is a method to apply some variable bindings to a Term. In `Term` interface there's a default implementation of this method covering all cases, hence no sub-type has to implement or override it.
- `freshCopy(): Term`, that is a method implementing *variable renaming*. This method is very useful during resolution process when clauses are loaded from the database, and their variables should be renamed before unifying the head.

Var. The `Var` type implementation returns a different instance every time the factory method `of` is invoked. It internally holds a counter of created instances, and the provided variable name is concatenated with the variable instance number, composing the variable identifier. This way there cannot be two `equals` variable instances coming from two different calls of the factory method.

`Term` methods are implemented this way:

- `structurallyEquals` returns `true` if the other `Term` is also a `Var`.
- `freshCopy` returns a newly created variable with the old variable name.

Struct. `Struct` implementation is the base class, either directly or indirectly, for most of the other classes. It provides common basic behaviour implementing:

- `structurallyEquals`, that returns `true` if functor and arity are the same and all arguments are recursively `structurallyEquals`.

- `freshCopy`, that returns a newly created `Struct` implementation with same functor and recursively `freshCopy` applied arguments.

Clause. `Clause` in our implementation is the class having the responsibility to "prepare for execution" its instances (Section 4.4).

The actual behaviour is obtained through an implementation of `TermVisitor`. Given a collection holding the notable functors defined for clause bodies (`','/2`, `';'/2` and `'->'/2`), it transforms all visited variables V inside it, occurring in the position of a predication, into the required structure `call(`V`)`.

Scope. The `Scope` implementation holds a map between variable names and variable instances. It provides bridge methods for all main types factory methods, without doing more that that, except for the `varOf(name: String)` method. In fact before using the `Var.of()` factory, it checks if the provided variable name is present inside current scope instance; if it is present, returns the stored instance, otherwise it creates a new `Var` instance and stores it for further retrieval.

Substitution. The `Substitution` implementation is technically a sub-type of `Map<Var, Term>`. Hence it naturally holds "mappings" and inherits all `Map` type methods and characteristics. Its methods are implemented actually delegating a `Map` instance, retrieved upon construction.

To comply with the standard the construction provided `Map` is checked:

- all variable chains are shortened by means of a `trimVariableChains` method; circular mappings are managed and become identity mappings, then
- all identity mappings (i.e. a mapping from a `Var` instance to the same instance) are removed.

`Substitution` has two sub-types: `Unifier` and `Fail`. The former is a type representing a substitution as described in Prolog Standard, the latter is a *singleton instance* used when there isn't a unifier. This approach is meant to reduce the usage of `null` and exception throwing.

`Substitution.plus(other: Substitution): Substitution` method implements the *composition* of substitutions. Unlike in the standard the composition of substitutions can lead to a `Fail` in case there are contradictions emerging from the bindings. The composition also leads to a `Fail` when one of the two substitutions is failed.

`Substitution` provides the "`of`" *static factory method*, as the other core classes. The overloads accepting possible conflicting substitutions, thus operating a composition, also check for contradictions.

5.2 Unify module

The unify module implements `Equation` and `Unificator` abstractions.

`Equation` type companion object contains some factory methods "`allOf`" that reify the *abstract unification algorithm* "splitting" transformation rule. Thus, given two `Struct`s, these methods return a sequence of equations corresponding to the equation of arguments at same indexes of the structured term. We decided to implement this behaviour inside `Equation` because at construction time it is easier to recognise the correct type of equation (`Identity`, `Assignment` or `Contradiction`).

`Unificator` type is represented by an interface, partially implemented by `AbstractUnificationStrategy`. The latter class is a base class implementing a simple and basic unification algorithm [27] through `mgu(term1: Term, term2: Term, occursCheck: Boolean): Substitution` method. `AbstractUnificationStrategy` does not give a body to method `checkTermsEquality`, which is a *template method* that decides when two `Term`s should be considered equal, hence dictating when the unification should succeed or not.

The concept of "unifying `Terms`" is simply modifiable implementing a different version of `checkTermsEquality`. In fact we provide two basic unification strategies: `strict` that implements the Prolog Standard unification, and a `naive` variant where numbers for example are made unifiable by value and not by type (i.e. float `1.0` unifies with integer `1`, unlike in Prolog Standard).

The `mguWith`, `matches` and `unifyWith` methods are provided in *infix* variant to be easily used as operators between `Terms`.

5.3 Theory module

Theory module implements the `ClauseDatabase` abstraction. As a real database, it has methods to read, write and remove `Clauses`:

- `assertA`, adds the given clause before the othres
- `plus`, aliases the `assertA` method
- `assertZ`, adds the given clause after others
- `contains`, checks for presence of matching clause (using `structurallyEquals`)
- `get`, retrieves all matching clauses (found using `structurallyEquals`)
- `retract`, deletes one matching clause (using `structurallyEquals`) and returns a `RetractResult`
- `retractAll`, deletes all matching clauses (using `structurallyEquals`) and returns a `RetractResult`

The `RetractResult` type holds the information of a "retract" operation (removal operation). It can be a `Success`, and hence contains the new database and the removed `Clauses`, or it can be a `Failure`, containing the same database with no modifications.

The `ClauseDatabase` is frequently consulted during the resolution process. The need for good read/write performances led us to the implementation of an efficient retrieval algorithm.

5.3.1 Rete algorithm implementation

The Rete algorithm [28] solves the problem of matching patterns against a multitude of objects.

The idea behind the algorithm is to create a "classification tree". In this tree we have two type of nodes:

- *non-leaf* nodes: they are in charge of classifying the received request of object insertion (or pattern retrieval) and forward it to the correct child node; they contain pointers to other nodes;
- *leaf* nodes: they contain the objects of interest.

The classification takes place over a set of *predefined properties* in common between objects and patterns. Every non-leaf node classifies objects and patterns using one property assigned to it. The order in which non-leaf nodes are connected to each other, thus which properties are evaluated before the others, is predefined.

Our implementation has general classes, reusable for possibly other Rete trees implementations:

- `ReteNode`, an interface defining common operations of insertion, retrieaval and deletion.
- `AbstractReteNode`, an abstract class with common behaviour for every Rete node; the implementation relies on `HashMap`s to index child nodes.
- `AbstractIntermediateNode`, class that represent a generic *non-leaf* node.
- `AbstractLeafNode`, class representing a generic *leaf* node.

Because we had to index `Clauses` (by their head), our selected classification properties were in order: *functor, arity, arguments.* Note that if a node is not already present it is created in place, otherwise the old instance is always reused. Intermediate node types are:

- `RootNode`, in charge of classifying clauses by their head functor.
  - If the clause is a `Rule` it forwards the request to the proper `FunctorNode`.
  - If the clause is a `Directive` it forwards the request to the leaf node `DirectiveNode` (because no distinction can be made for directives over the `null` head).

- `FunctorNode`, classifies rules by their head functor arity; it forwards the request to the proper `ArityNode`.
- `ArityNode`, classifies rules by their head functor arguments.
  - If the head functor arity is not zero, it forwards the request to `ArgNode` with zero index.
  - Otherwise it forwards the request to `NoArgsNode`.
- `NoArgsNode`, immediately forwards the request to the pointed `RuleNode`.
- `ArgNode`, has a double behaviour:
  - If the head functor has other arguments after the currently indexed one, forwards the request to another `ArgNode` with current index incremented by one.
  - If the current node is indexing the last head functor argument, forwards the request to the pointed `RuleNode`.

Leaf node types `RuleNode` and `DirectiveNode` are merely containers for respectively `Rules` and `Directives`.

5.4 Solve module

All types described in solve module design (Section 4.5.2) are implemented here.

Solution. The solution type is implemented in a `Solution` *sealed class* with properties `query`, `substitution` and a computed property `solvedQuery` obtained as the application of substitution to the query.

Its three sub-classes are `Yes`, `No` and `Halt` *data classes*; the last two override `substitution` property to be always `Substitution.Fail`. `Halt` requires, as additional parameter, the `TuPrologRuntimeException` that halted the solver computation.

Solve request and responses. Our implementation provides a *sealed class* `Solve` that has two implementations: `Request` and `Response`.

`Response` is a simple *data class* carrying the mandatory property `Solution` and other optional properties that should be instantiated whenever a modification is made to `libraries`, `flags`, `staticKB`, `dynamicKB` or to `sideEffectManager`.

`Request` is also a *data class* and carries with it all information explained in design chapter plus a `requestIssuingInstant` and an `executionMaxDuration` to be able to interrupt no more meaningful computations. The `Request` implementation also provides a group of methods to create a `Response` from it, lightening the programmer from the burden of setting to which request is the response responding to. These methods, whose signature include all the optional `Response` parameters, are:

- `replySuccess` which takes a substitution as first parameter (needed to construct the `Solution.Yes`);
- `replyFail` which takes no other parameter than the optional `Response` ones, becasue to create a `Solution.No` no other information, than the query already present in `Request`, is needed;
- `replyException` which takes a `TuPrologRuntimeException` as first parameter (needed to construct the `Solution.Halt`);
- `replyWith` which takes as first parameter a `Boolean` signaling if the response should be positive or negative; this creates a solution with *empty unifier* in positive case;
- `replyWith` which takes as first parameter the `Solution` to be set in `Response`.

While implementing `Request` class, instead of using the core module `Indicator` class, we created an enhanced version of it, with in mind the support for *vararg* predicates, and we called it `Signature`. Hence a `Signature` is an enhanced `Indicator`.

Other `Request` implementation detail is that it features a type variable, allowing solver implementations to have their own context type in requests without needing to explicitly cast it. The `Request` class signature is in fact: `Request<out E: Executioncontext>`.

Primitive. A `Primitive` in our implementation is simply a `typealias` for a function that takes in input a request and produces the sequence of lazily computed responses: `(Solve.Request<ExecutionContext>) -> Sequence<Solve.Response>`. Along with this type-alias we provide `PrimitiveWrapper` abstract class. It contains useful utilities to manage primitives, making easier their implementation and usage; in particular:

- `signature` property returns the supported signature to which the current primitive should be applied;
- `wrappedImplementation` abstract property, contains the primitive function implementation itself; since its used in other properties this is an application of *Template Method* pattern.
- `descriptionPair` property, associates the correct signature to the primitive implementation.

PrologFunction. The function feature implementation is very similar to that of primitives and request/response; this time we have a base `Compute` *sealed class* with two sub-classes `Request` and `Response`.

The `Response` *data class* this time has only one field of type `Term`, representing the computation result.

The `Request` *data class* has same design of the `Solve` twin. This time we have only one utility method to create a `Response` starting from a `Request`: `replyWith(term: Term)`.

A `PrologFunction` is a `typealias` for `(Compute.Request<ExecutionContext>) -> Compute.Response`.

Even this time we provide a `FunctionWrapper` class to make easier the usage and implementation of these functions. The class reuses part of the `PrimitiveWrapper` design.

Along with functions we provide two evaluators:

- `ExpressionEvaluator`, which is a generic term evaluator, implemented as a `TermVisitor` returning a `Term`, thus extending `TermVisitor<Term>`; its behaviour can be described as follows:

  1. Always call, on current term, the `staticCheck` method whose duty is to check if the evaluation can happen;
  2. When visiting a `Struct` S
     (a) if the signature does not match any loaded function signature, then S is returned as is;
     (b) otherwise, let F be the selected function;
     (c) all arguments of S are visited with current evaluator instance;
     (d) a `dynamicCheck` is applied to each result, checking if the computation can continue;
     (e) a `Compute.Request` is created with S signature and all the already evaluated S arguments as request arguments;
     (f) then, F is called with the newly created request.

  This implementation `staticCheck` and `dynamicCheck` methods do nothing; they are an application of *Template Method* design pattern.

- `ArithmeticEvaluator`, which realises a Prolog Standard arithmetic evaluator; it implements the *template methods*:

  – `staticCheck`, which checks the term argument and:
    * if it is a `Var`, throws an `InstantiationError`;
    * if it is an `Atom`, throws a `TypeError`;
    * if it is a `Struct` but not one among the allowed functions in expressions, throws a `TypeError`.

  - `dynamicCheck`, which checks the given term and:
    * if it is not a `Numeric`, throws a `TypeError`;
    * if it is not an `Integer` argument, evaluated inside a bitwise operator, throws a `TypeError`

In order to consider `'/'/2` as an arithmetic operator ad not as an `Indicator`, the `visit(term: Indicator): Term` has been implemented as well, making it call `super.visitStruct(term)`, hence triggering `term` arithmetic evaluation.

Library. The library concept is captured into the interface `Library`, then we have the `LibraryAliased` one which represents a library with some attached alias. Their behaviour is definied primarily at interface level, in fact `LibraryImpl` and `LibraryAliasedImpl` are simple classes adding only `equals`, `hashCode` and `toString` methods. Since libraries are thought to be used together, we also provide the interface `LibraryGroup` which defines some operations over a group of libraries, such as adding or updating them.

`Libraries` is the implementation of a `LibraryGroup` of `LibraryAliased` libraries; it implements all the methods to retrieve, add and update libraries and their contents.

ExecutionContext. The `ExecutionContext` interface implements the designed concept, adding only a property `prologStackTrace` which is very useful when an error should be reported to the user.

Solver. The `Solver` interface also implements the designed concept. It features a `solve(goal: Struct, maxDuration: TimeDuration = TimeDuration.MAX_VALUE): Sequence<Solution>` that lets the user specify the goal to be solved, the optional maximum duration of computation and returns a lazy sequence of `Solution` instances.

TimeRepresentation. To make the implementation independent from any time representation, we created two type aliases `TimeInstant` and `TimeDuration`. Their actual implementation is a `Long` specifying: in the former case the milliseconds from first January of 1970 as usual in programming languages, in the latter case the duration in milliseconds.

If, for some reason, the time representation should be changed, then this would be the main point where the change should happen.

Execptions and PrologError. All the designed exceptions are implemented as classes extending the *Kotlin* `RuntimeException`.

Moreover `TuPrologRuntimeException` adds, to runtime exceptions:

- a `prologStackTrace` property, to easily access the homonym `ExecutionContext` property;
- an `updateContext` method, to easily copy an exception returning a new instance with provided context.

The `PrologError` abstract class extends `TuPrologRuntimeException`. It is the base class for all Prolog language errors. Every `PrologError` has a `type` descriptive structure and an `extraData` term that can contain arbitrary implementation dependant data. `PrologError` companion object has a *static factory method* `of` which, given a `type` structure, is in charge of creating the correct specific instance of the error, as happens in core module classes hierarchy. If no corresponding error is recognised, an anonymous `PrologError` instance is created.

Some of the commonly used error representations have been implemented: `InstantiationError`, `TypeError`, `EvaluationError` and `SystemError`.

Common built-ins. In this solver independent module, we also provide some default built-ins which are not "solver internals" dependent (i.e. these predicates whose work does not interfere with the execution flow).

For example we have:

- All arithmetic primitives: `'=:='/2`, `'=\='/2`, `'<'/2`, `'>'/2`, `'=<'/2`, `'>='/2` and `is/2`;
- All arithmetic functions, not listed for brevity.
- All those built-ins whose implementation could be achieved by writing some simple Prolog rules like, for example:
  - `';'/2` implementation, achieved with:

    ```
    (A ; _) :- A.
    (_ ; B) :- B.
    ```

  - `'->'/2` implementation, achieved with:

    ```
    (Cond -> Then) :- call(Cond), !, Then.
    ```

5.5 Solve-classic module

The actual solve-classic implementation follows its design very closely (Section 4.6.2).

State. At the hierarchy top we have the `State` interface, extended by an `AbstractState` class. This class is the base class for all other concrete classes. The designed addition of `solution` and `exception` properties for final states, is implemented respectively in interfaces `EndState` and `ExceptionalState`, both derived from `State`.

Note that *TRUE*, *TRUE_CP* and *FALSE* are all represented with the same `StateEnd` *data class*. When `StateEnd.next(): State` is called, then the currently held context is checked by means of `hasOpenAlternatives` property:

- if it has open alternatives, it makes the state shift to `StateBacktracking`, implementing *TRUE_CP* designed behaviour;

- if it hasn't open alternatives, `next(): State` throws a *Kotlin* `NoSuchElementException`, signaling a wrong usage, hence implementing the *TRUE*, *FALSE* final states behaviour.

HALT state is implemented in `StateHalt` *data class.*

Cursor. To help managing solver inner workings, solve-classic implementation defines a new type of collection, called `Cursor`.

A `Cursor` is like an `Iterator`, but instead of interacting with it requiring the "next element" and if it "has a next element", the `Cursor` type can be queried to know if it points to a `current` element or if it `isOver` (i.e. no current element pointed), and that pointer can be moved forward by calling `next` method which returns a new instance of `Cursor` pointing to the next element.

Some basic implementations for `Cursor` interface are present and used internally, as needed, by *static factory methods*:

- `EmptyCursor`, which is a cursor pointing to nothing and with no `next` capability (i.e. exception is thrown if `next` is called);
- `NonLastCursor`, which is a cursor that has a next element to be pointed to;
- `MapperCursor`, which is a cursor that lazily maps `current` object with provided mapper function, and as `next` cursor returns another `MapperCursor` with same mapper function;
- `ConjunctionCursor`, which is a cursor to compose `Cursor`s in a way that the second `Cursor` is exploited only when the first one `isOver`.

ExecutionContext. The solve-classic `ExecutionContext` is implemented in an `ExecutionContextImpl` *data class.* Over the interface properties it features the ones below:

- `query`, which will contain the user provided initial query;
- `goals`, which will contain the `Cursor` on top of goals to be executed;

- `rules`, which will contain the `Cursor` over current rules open alternatives;
- `primitives`, which will contain the `Cursor` over current primitives execution open alternatives;
- `startTime` and `maxDuration`, which are properties to implement solver time management;
- `choicePoints`, which will contain the head of a linked list of choice points;
- `parent`, which will contain current context's parent;
- `depth`, which will contain how much deep in the search tree we are with current context (i.e. how many rules or primitives executions have been triggered);
- `step`, which will contain how many computational steps (measured in "state shifts") have been carried out till current execution context.

`ExecutionContextImpl` also has the following computed properties:

- `isRoot`, which returns whether `depth == 0`;
- `hasOpenAlternatives`, which forwards the call to current context choice point;
- `isActiavtionRecord`, which tests whether current context doesn't have a parent or has a parent with `depth` equals current `depth` minus one;
- `pathToRoot`, which computes the sequence of contexts following the `parent` property till the resolution root;
- `prologStackTrace`, which returns the Prolog stack trace from current context to root context.

ChoicePointContext. A `ChoicePointContext` is a *sealed class* containing:

- the `alternatives: Cursor` to be explored;
- the `executionContext` in which the choice point was encountered;
- the `parent: ChoicePointContext` of current one;
- the `depth` at which the `ChoicePointContext` was encountered;
- a property `pathToRoot` that contains a lazily initialised sequence of all choice points opened from this to the resolution tree root;
- a property `hasOpenAlternatives` to query if any `alternatives: Cursor` present from current choice point till the root, has other alternatives to be explored.

It has two *data class* implementations: `Rules` and `Primitives`. They are meant to contain respectively multiple rules choice points and multiple primitive responses choice points.

Two extension methods are provided to create modified instances of these immutable classes:

- `appendRules`, which creates and returns a new `ChoicePointContext` with provided rules alternatives `Cursor` and with `parent` set to the receiver object, hence leghthening the choice points list; `depth` field is also increased by one;
- `appendPrimitives`, which does the same thing but with provided primitives alternatives.

ClassicSolver. The `ClassicSolver` class is the `Solver` implementation provided by solve-classic module.

`ClassicSolver` is a *data class* with four constructor parameters:

- `libraries`, to hold loaded libraries at start;
- `flags`, to hold enabled flags at start;

- `staticKB`, to hold all static predicates loaded upon construction;
- `dynamicKB`, the same as previous point, but for changeable *Knowledge Base*.

The `solve` method is implemented as follows:

- All the computation happens inside a `SequenceScope`, which is a *Kotlin* functionality to lazily yield all the elements of a sequence.
- The method starts initialising an `ExecutionContextImpl` with provided solver constructor parameters.
- A variable `state` is initialised with a `StateInit` instance holding the newly constructed context.
- Then the method execution enters an infinite loop in which:
  1. the method `next(): State` is called on `state` variable and its result is assigned to `state` itself;
  2. if `state` is an `EndState` instance, then the contained `solution` is yielded to the `SequenceScope`; note that until another element is requested to the sequence the computation is suspended after the `yield` call (this is how lazyness is achieved);
  3. if `state` has no other alternatives, `break` the infinite loop.

DefaultBuiltins. The solve-classic solver implementation comes with some specific `DefaultBuiltins` that serve to its correct inner workings.

In addition to `CommonBuiltins` it adds:

- some static rules:
  - `(A, B) :- A, B` to help the unpacking of conjunctions in multiple goals;
  - `call(X) :- X` to unpack every `call/1` primitive usage;

  - `catch(G, _, _) :- G` to manage the `catch/3` situation with no errors. In fact in solve-classic solver implementation the catch management is made for the "non error" part thanks to a rule, while the "error" part is managed internally by the inferential core itself as described in its design.

- the built-in primitive `Throw` that, in this implementation, only needs to return a `Solution.Halt` with the correct `PrologError` instance and the inferential core will do the rest.

5.6 Solve-streams module

The actual implementation, even this time, follows quiet closely the design (Section 4.7). We will focus on notable implementation details.

5.6.1 FSM execution and states

One thing to keep in mind as you move from solve-classic to solve-streams, is that:

- The solve-classic implementation is one big state machine from start to end, with a big context holding all information needed during resolution process. The "communication" between states can happen easily creating a copy of the context, with a modified field, that will be read by subsequent states.

- The solve-streams implementation, instead, is a small state machine that in two of its states can create sub-instances of itself (to solve sub-goals). The "communication" between states becomes much more difficult this way, because information passing should happen through `Solve.Request` and `Solve.Response` fields.

State. The `State` interface defines, besides the already designed method `behave(): Sequence<State>`, these elements:

- a `solve: Solve` field, that will contain either the `Request`, which the state is computing for, or the `Response` which the state is holding to respond to a request;
- a `hasBehaved: Boolean` property, which will contain whether the current state has behaved or not.

Starting from `State` the hierarachy of state classes counts:

- The interface `IntermediateState` extending `State` and overriding `solve` field type to be a `Solve.Request<ExecutionContext>`, because intermediate states only can have requests to carry out.
- The interface `FinalState` extending `State` and overriding `solve` field to be of type `Solve.Response`, because final states only can contain a response to some request.
- The abstract class `AbstractState`, which is a `State` with an execution strategy (to be used internally, executing the behaviour) and whose `hasBehaved` property is `false`.
- The interface `TimedState`, which is a `State` to which can be asked the current time.
- The class `AlreadyExecutedState`, which extends `State` and it's a wrapper class for some other `State` to signal that it has behaved; its property `hasBehaved` always returns `true`.
- The abstract class `AbstractTimedState`, which extends `AbstractState` and implements the timed behaviour that every non-final state should have. It provides a *template method* `behaveTimed(): Sequence<State>` which is called only if the execution max duration has not been exceeded. In case it has been exceeded the state shifts immediately into `StateEnd.Halt`, returning a sequence containing only that as next state.

- Then we have the main states implementations `StateInit`, `StateGoalEvaluation` and `StateRuleSelection`, which follow the design description, all of them extending `AbstractTimedState`.

- At last we have `StateEnd` *sealed class* extending `AbstractState` and `FinalState`, and their three *data class* implementations: `StateEnd.True`, `StateEnd.False` and `StateEnd.Halt`. Their behave method returns an `emptySequence`, because end states have no subsequent state.

  We also provide some extension methods with `ItermediateState` as receiver, along with `StateEnd`, to enhance the state behaviour writing, in particular end state shifting. Following the `replyXXX` methods style in `Solve.Request`, here we have:

  - `stateEndTrue`, to create a successful end state, accepting as first argument a substitution;
  - `stateEndFalse`, to create a failed end state;
  - `stateEndHalt`, to create an halt end state, accepting as first argument the exception thrown;
  - `stateEnd`, to dynamically create the correct end state depending on first `Solution` parameter;
  - `stateEnd`, to forward the provided `Solve.Response` with correct end state.

StateMachineExecutor. Because of the particular nature of our state machine, whose behaviour produces all next possible states, we implemented its execution algorithm into a dedicated `StateMachineExecutor` class.

Roughly speaking the execution of such a state machine is a continuous "flat mapping" of state behaviours, until the last state has behaved. It is like trying to flatten the search-tree exploration into a single sequence of visited states. The resulting sequence will be to all effects "the history" of that tree navigation from top to bottom, left to right.

The internal execution algorithm is recursive and given a state S, can be described as follows:

1. If S has already behaved, return a sequence containing only S.
2. Otherwise, start building a lazy sequence that initially contains only S, then:
    (a) retrieve all possible next states of S;
    (b) for each S' of them, execute this algorithm from (1.) and add the result, of its explosion, to the lazy sequence being built.

The public `execute(state: State): Sequence<State>` always drops first state returned from the internal algorithm, because it is the input state, hence not interesting. It also unwraps states wrapped in `AlreadyExecutedState`.

Moreover we have an internal method called `executeWrapping(state: State): Sequence<State>` which calls `execute` and then wraps every returned state into `AlreadyExecutedState`, needed inside `StateRuleSelection`. This method is needed since in `StateRuleSelection`, with this architecture, at some point we want to know the sub-goal `Solve.Responses`, to forward them as responses of current `Solve.Request`. At first sight, we could merely use `execute` method onto instantiated `StateInit` with the sub-request, but this would lead to double execution of that `StateInit`; the first time inside the `StateRuleSelection` and the second time from the outer `execute`, which has brought the execution till the `StateRuleSelection` itself. This is why inside any state, if a *sub-execution* is needed, the `executeWrapping` should be used. In fact wrapping states into `AlreadyExecutedState` is an implementation trick to make the outer `execute` return them as they are. Every "part of state sequence" wrapped in `AlreadyExecutedState` is put in the output sequence, untouched.

5.6.2 Solver and support classes

SideEffectManagerImpl. The `SideEffectManagerImpl` class is very important for solve-streams implementation. It is the communication vehicle between states for "notifying" that a side effect on execution flow should happen. It contains the following methods:

- Cut related:
  - `cut`, to apply the Cut side effect;
  - `stateInitInitialize`, to initialise every `StateInit` to be ready for side effects;
  - `enterRuleSubScope`, to prepare a new `SideEffectManager` while entering a rule sub-scope;
  - `extendParentScopeWith`, to be used when exiting a rule sub-scope;
  - `resetCutWorkChanges`, to reset Cut side effects when exiting a `call/1` primitive.
- Throw/Catch related:
  - `throwCut`, to apply the Throw side effect;
  - `isSelectedThrowCatch`, to know if the current `catch/3` was selected by a `throw/1`;
  - `retrieveAncestorCatch`, to find among parents an ancestor matching `catch/3`;
  - `ensureNoMoreSelectableCatch`, to ensure that a `catch/3` which has been selected, won't be selected again;
  - `shouldExecuteThrowCut`, to know if a Throw has been executed.
- related to Cut and Throw/Catch:
  - `creatingNewRequest`, to reorganise internal data structures during new sub-request creation;
  - `shouldCutExecuteInRuleSelection`, to query if the Cut side effect should be applied; it should be used in `StateRuleSelection` only, because makes some specific checks.

ExecutionContextImpl. The solve-streams `ExecutionContext` is implemented in an `ExecutionContextImpl` *data class*. Beyond the interface properties, it features:

- a field `solverStrategies` to hold the provided strategies;
- a field `sideEffectManager` to hold the current insatnce.

Solver. The `Solver` implementation provided by solve-streams is `StreamsSolver`. It is a *data class* holding start parameters like initial `libraries`, `flags`, `staticKB` and `dynamicKB`.

The **public** `solve` method implementation does the following steps:

1. Creates a `Solve.Request` with the goal `Struct` and an `ExecutionContextImpl` with provided solver parameters and the specified `maxDuration`.
2. Forwards this request to the internal implementation of `solve`.
3. Returns all provided `Solve.Responses`, mapping them to the respective `Solutions`.

The **internal** `solve` method implementation:

1. Creates a new `StateInit` instance with provided `Solve.Request`.
2. Executes that state machine through `StateMachineExecutor.execute()`.
3. Filters the resulting state sequence by `FinalState` whose response query is equals to the request one, then finally maps it to its held `Solve.Response`.

SolverUtils. The `SolverUtils` *Kotlin* file contains several useful methods which simplify the writing of common behaviours, recurring during solver implementation. We have:

- `Term.isWellFormed(): Boolean`, which tells if receiver `Term` is well-formed;

- `Term.prepareForExecutionAsGoal(): Struct`, which prepares for execution the receiver `Term`, using under the hood the `Clause.prepareForExecution()` method;

- `Sequence.orderWithStrategy(...)`, which lazily orders the receiver sequence with provided strategy argument;

- `moreThanOne(s: Sequence<*>): Boolean`, which lazily checks if the provided sequence has more than one element or not;

- `Solve.Request.newSolveRequest(...): Solve.Request`, which creates a copy of receiver request, modifying fields accordingly to all provided parameters; for example it adds provided substitution to the already present ones, it adjusts the execution max duration and current time and sets the new request goal in returned request;

- `Solve.Request.replyWith(r: Solve.Response): Solve.Response`, which is useful to forward a response to the upper `Solve.Request`.

5.6.3 DefaultBuiltins

As already explained in Section 4.7.3, we had to package the solver with a plethora of basic primitives: Conjunction, Cut, Call, Throw, Catch and Not.

They are all *Kotlin* `object`s extending `PrimitiveWrapper` to ease their writing, and their behaviour follows closely what has been described in design section.

Not. The `Not` primitive has been added to solve-streams solver default ones, for efficiency reasons. Instead of leveraging on the `CommonRule` couple (`not(X) :- call(X), !, fail.` and `not(_).`) that would require to instantiate several state machines, we provided a simple and straightforward primitive that inverts the result of the single argument goal execution.

5.7 Dsl-core module

The implementation of *2p-kt Domain Specific Language* (DSL) is provided in this module, following the explained design (Section 4.8).

The `Prolog` interface, which extends `Scope`, is the reification of that design, with all required methods (including the default implementations); among the others in fact, we have:

- `Any.toTerm(): Term`, which is the universal converter of any type to `Term` representation; it raises an `IllegalArgumentException` if no conversion is possible;

- `String.invoke(term: Any, vararg terms: Any): Struct`, which enables the programmers to use `String`s as if they were structure functors;

- all language arithmetic operators overridden in a way such that the operator becomes the functor for the two (converted) terms: `Any.operator(term: Any): Struct`;

- a lot of `infix` methods to make easier to read a DSL written program; all of them have an `Any` receiver argument and an `Any` argument and return a `Struct` whose functor is dependent on the method name; for example `and` creates *conjunction*, `or` creates *disjunction*, `‘=‘` creates a term `’=’(term1, term2)`, and so on;

- some `Scope` methods overloaded, to make them accept `Any` arguments and hence widen their applicability;

- `Any.‘if‘(other: Any): Rule`, which is an infix method to gracefully write Prolog rules;

- two infix methods `Var.to(term: Any): Substitution` and `String.to( term: Any): Substitution`, which make more compact the `Substitution` creation;

- a companion object method `empty(): Prolog` to create a clean *prolog scope*.

Finally a module global function `<R> prolog(function: Prolog.() -> R): R` is provided, implementing the *scoping* of this DSL. In fact all described methods work without problems if inside such a scope, because defined into `function` receiver type.

5.8 Dsl-unify module

This dsl-core extension DSL, enhanced with unification constructs (Section 4.9), is implemented as the interface `PrologWithUnification`, which extends `Prolog` and `Unification`.

Inside it default methods, overloading `Unificator` ones, are provided widening the parameters typing to `Any`; thus we have:

- the three normal functions, with `Any` type parameters:
  - `mgu(term1: Any, term2: Any): Substitution`
  - `matches(term1: Any, term2: Any): Boolean`
  - `unify(term1: Any, term2: Any): Term?`
- the three `infix` versions, with `Any` type parameters:
  - `Any.mguWith(term: Any): Substitution`
  - `Any.matches(term: Any): Boolean`
  - `Any.unifyWith(term: Any): Term?`

`PrologWithUnification` companion object provides, besides the `empty()` factory method, an `of(unificator: Unificator): PrologWithUnification` one to specify a different unification strategy, if the default one is not desired.

Finally a module global function, specific for this DSL extension, is provided to be used when unification features are needed inside a *prolog scope*: `<R> prolog( function: PrologWithUnification.() -> R): R`.

5.9 Dsl-theory module

This is the dsl-unify extension DSL, adding theory creation capabilities (Section 4.10), is implemented through the interface `PrologWithTheories`, which extends `PrologWithUnification`.

Inside it we have the two designed methods to create Prolog theories:

- `theoryOf(vararg clauses: Clause): ClauseDatabase`, which implements the level *(ii)* constructor;
- `theory(vararg clauseFunctions: Prolog.() -> Any): ClauseDatabase`, which implements the level *(iv)* constructor.

Finally the module global function `<R> prolog(function: PrologWithTheories.() -> R): R`, is provided to enable programmers to use the enhanced theories creation *prolog scope*, when needed.

Chapter 6

2p-kt Validation

After the previous chapters, where we modelled the system to finally implement it, we now proceed with the description of its validation. More precisely, in section 6.1 we discuss the test design adopted in our testing suite then we briefly present an overview of the implementation (section 6.2). Then, in section 6.3 we discuss the requirement compliance of *2p-kt*.

6.1 Test design

Each project module has its own testing module. Testing modules contain *automated tests* to verify module classes compliance with the expected behaviour and intercept possible future regressions when code will be modified.

In our testing code-base we implemented three of many types of *functional testing*:

Unit tests: These are tests whose main purpose is to exercise single functionalities of one class (i.e. single methods or even single lines of code). They requires detailed knowledge of the internal program design and code, hence are written by programmers while implementing the under test class.

Integration tests: These are tests whose main purpose is to verify that two or more parts together, work as expected.

System tests: These are tests that exercise the whole system behaviour, as if it was used by the final user. It is a *black-box* type of testing, where the result is checked for correctness.

Furthermore, our design opted to not abuse of *inheritance* mechanism to achieve the application of **DRY** (*Don't Repeat Yourself*) principle. Even if it is very cheap to implement, it is very hard to maintain. For example, inheriting a test class may inherit even useless behaviour, or behaviour that for a specific class instance should not exist, and a test may be overridden to do nothing. The solution to this problems could be to create a test class hierarchy that in the worst case mirrors the tested hierarchy, thus duplicating the programmer work. Hence not very convenient for a newborn project.

We decided to opt for *delegation*. Test classes delegate some of their repeated behaviours to an external `object` whose purpose is exactly that of providing a functionality-testing service. Thus every test class can implement its specific tests without the problems described above, selecting every time which test helper method to use from the "service" `object`.

The pattern we implemented in our tests is the following, where "type" word should be replaced by the actual type under test:

1. We have a test class named `TypeTest`, which is a *unit test* for `Type`. This class is in the same package of the tested class, but in the mirrored testing module.

2. Then, we have a "service" object named `TypeUtils`, which contains useful methods to help testing `Type` and possible derived or closely related types. If test data is shared among different test classes, `TypeUtils` contains it, permitting its reuse. This object is in a `testutils` sub-package of tested class package.

This way there's no need for a hierarchy of test classes, and tests are more clear, because all the under test behaviour is specified in one class, and not divided into different classes.

6.2 Test implementation overview

In this section we provide an overview of tests implementation for each tested module. Some modules not completely tested are not listed in sections below. These ones test implementation, are object of future work on this project.

6.2.1 Core module tests

In core module testing, we have:

- a *unit test* for each type;
- a *unit test* for each interface `companion object` (which usually implements factory functionality); factories are tested also to provide correct sub-classes instances when needed;
- an *integration test* for notable `Atom` instances like `true`, `fail`, `[]` and `{}`;
- a testing utility class `AssertionUtils`, with some useful methods to help testing all classes, like `onCorrespondingItems` which executes the given function on corresponding items of two given `Iterables`;
- a testing utility class `TermTypeAssertionUtils`, with methods to test `Term` hierarchy classes runtime type and `isXXX` getters.

In this module testing every single field, property, method and constructor is tested in all possible cases. In it we have the finest testing implemented throughout the project. Tests total count amounts to **489**.

6.2.2 Unify module tests

In unify module testing, we have:

- `EquationTest` *unit test*, which deeply tests the `Equation` type;
- `EquationUtils`, which contains correct and not correct samples of equations, shollow equations or deep equations, and utility assertion methods

like `assertAllIdentities`, `assertAnyContradiction`, and so on to apply **DRY** principle in `EquationTest`;

- `AbstractUnificationStrategyTest` *unit test*, which tests the unification algorithm over equation samples provided by `EquationUtils` plus other ad-hoc thought examples to stress particular situations;
- `UnificatorTest` *unit test*, which tests the different types of default unification algorithms implemented;
- `UnificatorUtils`, which contains utility data useful for testing `AbstractUnificationStrategy` and `Unificator`, and utility methods like `assertMguCorrect`, `assertMatchesCorrect`, and so on.

Tests total count amounts to **49**.

6.2.3 Theory module tests

In theory module testing, we have testing classes for the *Rete algorithm* implementation and for the `ClauseDatabase` implementation.

- All Rete node types are tested in multiple situations.
- All `ClauseDatabase` creation modalities and instance methods are tested through `ClauseDatabaseTest` and `ClauseDatabaseImplTest`.

Tests total count amounts to **157**.

6.2.4 Solve module tests

In solve module testing, we have a *unit test* for each class. The only exceptions are some simple common primitives and some function implementations which miss a corresponding *unit test*, and will be covered by future works tests.

Solve-test module. Solve module and its dependent sub-modules testing are assisted by a `solve-test` module, crafted to test "solve" capabilities. The `solve-test` module depends on `solve`, from which inherits main types, but even from `dsl-theory` to help the writing of test data.

In `solve-test` we have these classes:

- `DummyInstances`, containing fake objects to be used in parameter filling;
- `TestUtils`, which contains:
  - some commonly useful methods in testing solver hosting modules like, `assertSolutionEquals`
  - a DSL-theory raw extension for test writing, enhancing the programming experience:
    - `Struct.yes(vararg withSubstitution: Substitution): Solution.Yes`, which creates a `Solution.Yes` starting from the receiver structure and the provided substitutions;
    - `Struct.no(): Solution.No`, which creates a `Solution.No` starting from the receiver structure;
    - `Struct.halt(exception: TuPrologRuntimeException): Solution.Halt`, which does the same thing with the provided exception;
    - `Struct.hasSolutions(vararg Struct.() -> Solution)`, which creates a `Pair` from the provided goal structure to a `List` of its `Solutions`;
    - `Solution.changeQueryTo(query: Struct): Solution`, which copies the receiver `Solution` into another instance with only the `query` field changed;
    - `Iterable<Solution>.changeQueriesTo(query: Struct): Iterable<Solution>`, which does the same thing of previous method but over an `Iterable` of solutions.
- `TestingClauseDatabases`, which contains all custom created databases to test various solver functionalities;

- `PrologStandarExampleDatabases`, which contains all Prolog Standard example databases of implemented primitives, to ensure the standard behaviour is respected;
- `SolverFactory`, which is an interface defining properties and a method to create a custom solver instance to be tested;
- `SolverTestPrototype`, that is a base class to which all solver implementations will delegate *system testing*. Here we implement all system testing methods, that each solver should comply with, calling them in their specific tests.

Tests total count amounts to **211**.

6.2.5 Solve-streams module tests

In solve-streams module testing, we have:

- *unit test* classes related to the *Finite State Machine*, which test each state possible outcomes;
- *integration test* class `StateIntegrationTesting`, which tests the whole state machne behaviour;
- *unit test* classes related to each primitive implementation;
- *integration test* classes like `CutAndConjunctionIntegrationTest`, which tests the integration of few primitives;
- *system test* class `StreamsSolverSystemTesting`, which implements `SolverFactory` delegating to `SolverTestPrototype` all its testing methods. In this class to test the solver behaviour as a whole, `DefaultBuiltins` are loaded before testing starts.

Tests total count amounts to **150**.

6.2.6 Solve-classic module tests

At time of writing, the only implemented test in solve-classic is the most important test `ClassicSolverSystemTesting`, which tests the overall good functioning.

6.3 Requirements compliance

Architectural requirements. *2p-kt* adheres to all architectural requirements which required *(i)* **multi-platform support** and extendability towards other platforms – achieved with the adoption of *Kotlin* multi-platform as development technology – and *(ii)* a **strong modularity**—achieved implementing the proposed design, in which modules have single responsibilities.

Functional requirements. *2p-kt* partially adheres to **Prolog Standard support** requirement. More precisely, this thesis project successfully *supports the implementation* of all standard built-ins, but at time of writing *only part* of Prolog ISO standard built-ins are actually implemented. Hence, the execution of some ISO Prolog predicates is not currently possible. The full Prolog ISO standard support is object of future works (Section 7.2) on this project, due to the amount of work required.

However, our implementation adheres to all other functional requirements, including:

- **Immutability by default**: All project classes are immutable, by design.
- **Detailed Term hierarchy**: We designed a very detailed Term hierarchy following the Prolog Standard reference and further adding some other types (Figure 4.2).
- **Term instances should be of most specific class**: We designed the core classes to be created through *static factory methods* internally dispatching the correct instance creation.

- **Solver computation max duration**: We endowed the common Solver interface of a method accepting a max duration parameter, whose meaning must be honored by solver implementations.

Non-functional requirements. *2p-kt* adheres to requirement **maximise programmatic usability of concepts** providing some *DSL* modules, addressing the requirement concern.

Regarding the **project code testing** requirement it can be considered satisfied, because indirectly all the public API of the project has been tested. As we described in section 6.1 and section 6.2, a specific unit testing has been carried out for `core`, `unify`, `theory`, `solve`, `solve-streams` modules. A system test has been provided for `solve-classic` module and then we have indirectly tested `dsl-core`, `dsl-unify` and `dsl-theory` modules extensively using them in system tests.

Chapter 7

Conclusions

In this last chapter, we firstly summarise the work that has been done with this master thesis in section 7.1. Finally, we discuss some possible future works that could be carried out to enhance the project implementation in section 7.2.

7.1 Summary

In this thesis we have successfully designed and implemented *2p-kt* which is a lightweight, modular, multi-platform framework for the Prolog language, overcoming all identified *tuProlog* issues (section 3.1), hence successfully rebooting the project. Indeed, it features nine modules (fig. 4.1) which encapsulate a specific feature, possibly enabling a selective usage of each one.

We endowed *2p-kt* of two interchangeable resolution engines, based on the same common API. The former features the classical *tuProlog* resolution (section 4.6). The latter features a newly conceived Prolog resolution implementation (section 4.7), paving the way for resolution parallelisation and Prolog "choice strategies" modification.

In literature or in commerce, there's no logic framework working on multiple platforms in a lightweight fashion, hence without an external library (section 2.2). The overall *2p-kt* can be currently compiled down and used as a library in JVM and JavaScript platforms, but every other platform targeted by *Kotlin multi-platform* is virtually supported, hence including Native ones. Thus, another contribution of this thesis, is that of providing a Prolog framework with native multi-platform support like no other does.

However, currently, *2p-kt* has some limitations. Due to lack of time, the Prolog ISO standard built-in predicates implementation has not yet been completed and will be object of future works on this project. Nonetheless, the way for their implementation is pave.

Ultimately, we are overall satisfied with the work that has been carried out with this thesis.

7.2 Future Works

In this section we list some of possible future works.

Complete Prolog Standard ISO support. The *2p-kt* implementation currently complies with a subset of Prolog ISO standard. Missing standard predicates should be implemented in future, to reach full ISO compliance.

Prolog resolution parallelism. It is known in literature how the Prolog resolution process parallelisation is a long-standing problem, in particular because of "problems" with backtracking mechanism. Because in our design we enforced immutability, in future it could be evaluated how this choice impacts the construction of a parallel resolution engine. We think that `solve-streams` Solver could be a good starting code-base to experiment this kind of evolution.

New resolution modules. Our architecture allows new Solver modules implementation, benefiting from the basic components already provided. It could be interesting to explore the implementation of alternative modules not (strictly) based on a depth-first search. For instance, tabled resolution, breadth-first search approaches or constraint programmming approaches.

Interactive Interpreter. As for other Prolog systems, an interactive interpreter could be a good way to experiment little pieces of Prolog code. Since *2p-kt* is a modular project, a module for such a feature could be a good addition.

Tail recursion optimisation. Current *2p-kt* solvers implementations, do not have any tail recursion optimisation. Hence for some Prolog programs, the run-time memory allocation could be sub-optimal.

Parser module. A `parser` module should be added to *2p-kt* to complete its functionalities. Such a module should depend on `theory` and implement the Prolog text parsing functionality, giving in output the parsed theory object.

Test coverage. *2p-kt* unit tests do not cover the full project implementation, in particular there are no unit tests for `solve-classic` and DSL modules (`dsl-core`, `dsl-unify`, `dsl-theory`). We do not aim at 100% test coverage, since it is a myth, but it would be better to cover at least all main public classes implementation.

Signature vararg support. Another minor interesting feature, not already implemented but "prepared" for that, it's the support for *vararg Prolog predicates*. Even though this is not a Prolog ISO standard feature, it would be stimulating to try implementing such a feature.

Bibliography

[1] G. Piancastelli, A. Benini, A. Omicini, and A. Ricci, "The architecture and design of a malleable object-oriented prolog engine," in *Proceedings of the 2008 ACM symposium on Applied computing*, pp. 191–197, ACM, 2008.

[2] C.-P. Wirth, J. H. Siekmann, C. Benzmüller, and S. Autexier, "Jacques herbrand: Life, logic, and automated deduction.," 2009.

[3] J. Ferreirós, "The road to modern logic—an interpretation," *Bulletin of Symbolic Logic*, vol. 7, no. 4, pp. 441–484, 2001.

[4] J. A. Robinson *et al.*, "A machine-oriented logic based on the resolution principle," *Journal of the ACM*, vol. 12, no. 1, pp. 23–41, 1965.

[5] R. Kowalski, "Predicate logic as programming language," in *IFIP congress*, vol. 74, pp. 569–544, 1974.

[6] P. Deransart, L. Cervoni, and A. Ed-Dbali, *Prolog: The Standard: Reference Manual.* Berlin, Heidelberg: Springer-Verlag, 1996.

[7] K. R. Apt, "The logic programming paradigm and prolog," *arXiv preprint cs/0107013*, 2001.

[8] A. Maffi, "Blockchain and beyond: Proactive logic smart contracts," Master's thesis, Alma Mater Studiorum - Università di Bologna - Campus di Cesena, 2018.

[9] J. Wielemaker and V. S. Costa, "On the portability of prolog applications," in *International Symposium on Practical Aspects of Declarative Languages*, p. 69, Springer, 2011.

[10] N.-F. Zhou, "B-prolog user's manual," *Version 7.8. Dostupné z: www. probp. com/download/manual. pdf*, 1994.

[11] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla, "An overview of ciao and its design philosophy," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 219–252, 2012.

[12] M. Wallace and A. Veron, "Two problems-two solutions: one system-eclipse," in *IEE Colloquium on advanced software technologies for scheduling*, pp. 3–1, IET, 1993.

[13] D. Diaz and P. Codognet, "The gnu prolog system and its implementation," in *SAC (2)*, pp. 728–732, 2000.

[14] X. T. GmbH, "Jekejeke runtime reference." `http://www.jekejeke.ch/idatab/rsclet/prod/en/docs/05_run/10_docu/02_reference/package.pdf`, October 2019.

[15] B. D. Steel, "Win-prolog 7.0 - technical reference." `http://www.lpa.co.uk/ftp/7000/win_ref.pdf`, 2019. LPA association url: `http://www.lpa.co.uk/win.htm`.

[16] M. Carlsson and P. Mildner, "Sicstus prolog—the first 25 years," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 35–66, 2012.

[17] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "Swi-prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[18] K. Sagonas, T. Swift, and D. S. Warren, "Xsb as an efficient deductive database engine," in *ACM SIGMOD Record*, vol. 23, pp. 442–453, ACM, 1994.

[19] V. Santos Costa, L. Damas, and R. Rocha, "The yap prolog system," *arXiv preprint arXiv:1102.3896*, 2011.

[20] JetBrains s.r.o., "Kotlin programming language." `https://kotlinlang.org/`, October 2019.

[21] TIOBE Software BV, "Tiobe index." `https://www.tiobe.com/tiobe-index/`, October 2019.

[22] B. Venners and B. Eckel, "The trouble with checked exceptions: Versioning with checked exceptions." `https://www.artima.com/intv/handcuffs2.html`, August 2003.

[23] B. Venners and B. Eckel, "The trouble with checked exceptions: The scalability of checked exceptions." `https://www.artima.com/intv/handcuffs3.html`, August 2003.

[24] E. Denti, A. Omicini, and A. Ricci, "tuprolog: A light-weight prolog for internet applications and infrastructures," in *International Symposium on Practical Aspects of Declarative Languages*, pp. 184–198, Springer, 2001.

[25] J. Bloch, *Effective java.* Addison-Wesley Professional, 2017. Item 17.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Microarchitectures for reusable object-oriented design," *Reading: Addison-Wesley*, 1994.

[27] A. Martelli and U. Montanari, "An efficient unification algorithm," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 2, pp. 258–282, 1982.

[28] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," in *Readings in Artificial Intelligence and Databases*, pp. 547–559, Elsevier, 1989.