2P-Kt: A logic-based ecosystem for symbolic AI

Giovanni Ciatto a,*, Roberta Calegari b, Andrea Omicini a

a Department of Computer Science and Engineering (DISI), Alma Mater Studiorum—Università di Bologna, Italy
b Alma Mater Research Institute for Human-Centered Artificial Intelligence (AlmaAI), Alma Mater Studiorum—Università di Bologna, Italy

Abstract

To date, logic-based technologies are either built on top or as extensions of the Prolog language, mostly working as monolithic solutions tailored upon specific inference procedures, unification mechanisms, or knowledge representation techniques. Instead, to maximise their impact, logic-based technologies should support and enable the general-purpose exploitation of all the manifold contributions from logic programming. Accordingly, we present 2P-Kt, a reboot of the tuProlog project offering a general, extensible, and interoperable ecosystem for logic programming and symbolic AI.

© 2021 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Motivation and significance

Computational logic (CL) is a fundamental research area for artificial intelligence (AI), dealing with formal logic as a means for computing [1]. Its penetration into symbolic AI is nearly pervasive nowadays, and increasingly going deeper within sub-symbolic
AI [2,3]: CL has enabled the development of the former in the past, and it is now pushing the latter towards interpretability and explainability. Be it either exploited to manipulate symbols or to make sub-symbolic solutions human-intelligible, the common expectation is that CL can endow software systems with automated reasoning.

Generally speaking, automated reasoning involves three major aspects: (i) logic, (ii) inference rule, and (iii) resolution strategy. Logics formally define how knowledge is represented and how novel knowledge can be derived from prior one. Each logic comes with several inference rules, dictating how to produce new knowledge under particular circumstances. When coupled with some suitable resolution strategy, inference rules can become deterministic algorithms that computers can execute.

Many logics exist in CL – e.g. propositional, first-order (FOL), temporal, deontic, etc. –, each one targeting a specific domain. For instance, temporal logic enables reasoning about events in time, deontic logic supports reasoning about permissions/prohibitions and their circumstances, while FOL is general-purpose. Furthermore, different inference rules exist for different logics. Some are deductive – drawing conclusions out of premises –, some are inductive – looking for general rules out of several premises–, while others are abductive – speculating on which premises caused some conclusions. Finally, when a resolution strategy exists for some inference rule, it can be translated in some software construct, and used to provide intelligent systems with automated reasoning. Software of that sort is commonly referred to as a part of the logic programming (LP) paradigm.

Despite the wide availability of logics, inference rules, and resolution strategies in the LP literature, only a relatively-small amount of them have been reified into actual logic-based technologies (LBT in short henceforth). The Prolog language [4] represents by far the most successful story here [5]. It consists of a well-defined language [6,7] coming with several implementations [8–14].

While standard implementations of Prolog target FOL via SLDNF inference rule [15,16] and depth-first resolution strategy, most implementors have extended Prolog to support other resolution strategies as well. This is the case of Prolog implementations supporting, e.g., constraint logic programming [17], constraint handling rules [18], tabling resolution [19], etc.

Thanks to the versatility of FOL, it is a common practice in LP to either develop LBT either on top of Prolog or from scratch. Building LBT on top of Prolog is often preferred as they automatically inherit Prolog basic mechanisms, including e.g. the capability to perform (i) data structures representation via logic terms, (ii) knowledge representation via Horn clauses, (iii) logic unification, (iv) efficient in-memory indexing of logic information, (v) reasoning, via a flexible inference rule, and (vi) meta-level programming. This is a smart strategy when LBT must be quickly bootstrapped, yet it may result in poorly-interoperable, Prolog-tailored solutions. Conversely, when Prolog capabilities are poorly-suited for some particular problem, LBT may be designed from scratch. This involves re-designing and re-implementing most LP features ex-novo.

In [20] Sterling states that logic unification is one major contribution of LP to software engineering—thus singling a specific feature out of Prolog for its value and benefits. Similarly, many aspects of LP could be useful in AI by themselves, so each contribution should be conveniently reified into individually-useable software. Accordingly, we aim at creating an open ecosystem for interoperable, general-purpose LP libraries, virtually supporting multiple logics, inference rules, and resolution strategies—and possibly factorising any shared aspect.

The idea of LP as a key technology-enabler of intelligent application was already in place decades ago. The tuProlog project [21] was proposed for this purpose. It consists of a lightweight mal-leable, object-oriented, Java-based implementation of Prolog [22] which can be used as a library for JVM projects. Despite several versions have been proposed – bringing new features, or more platforms support [23] –, and many research projects have been built upon it – such as TuCSoN [24], ReSpecT [25], LPaaS [26], or Tenderfome [27], Arg2P [28], etc. –, it still consists of a monolithic library targeting Prolog alone. However, Prolog is no silver bullet for LBT, and LP should not be reduced to Prolog alone.

Accordingly, here we propose 2P-Kt: a reboot of the tuProlog project providing for a common technological ground for LP. Acknowledging that most mechanisms in LP have the potential to be of general value – not necessarily tailored to any specific logic, inference rule, or resolution strategy –, 2P-Kt consists of a logic-based ecosystem for symbolic AI, designed and implemented by taking openness, modularity, extensibility, and interoperability into account.

The tuProlog project has been completely re-designed and re-written, splitting LP functionalities into minimal, loosely-coupled, Prolog-agnostic, individually-useable, multi-platform modules. The rationale behind this choice is to enable the incremental addition of novel LP functionalities to the 2P-Kt ecosystem – possibly targeting other inference rules and search strategies –, minimising duplication of features and reusing pre-existing ones, while supporting as many programming platforms as possible. On the long run, 2P-Kt aims at becoming a comprehensive technological playground supporting several sorts of logics and inference mechanisms.

Finally, we acknowledge the importance of keeping 2P-Kt widely interoperable at the technological level with as many platforms as possible—to maximise the pool of potential adopters. Following this purpose, 2P-Kt leverages on the Kotlin multi-platform technology: each module currently supports the JVM, JS, and Android platforms—while others are expected to be supported soon.

2. Software description

2P-Kt is deeply rooted in LP, a programming paradigm based on CL [29,30]. In LP, programs are typically theories (a.k.a. knowledge bases, KB), i.e. collections of sentences in logical form, expressing facts and rules about some domain, typically in the form of clauses, i.e.:

\[ \text{Head} \leftarrow \text{Body}_1, \ldots, \text{Body}_n \]

where both Head and Body, are atomic formulae, and the whole sentence is read declaratively as logical implication (right-to-left). If \( n = 0 \), the clause is called a fact, a rule otherwise. An atomic formula is an expression in the form \( P(t_1, \ldots, t_m) \) where \( P \) is a m-ary predicate \((m \geq 0)\), and \( t_j \) are terms. Terms are the most general sort of data structure in LP languages. They can be constant (either numbers or atoms/strings), variables, or recursive elements named structures. Structures are used to represent clauses, lists, sets, or other sorts of expressions.

Logic solvers exploit KB to answer users’ queries via some inference procedure and resolution strategy. For instance, Prolog interpreters exploit a deductive procedure rooted into the SLDNF resolution principle [16,31], and a depth-first strategy. Yet, other options exist like, e.g., abductive [32], inductive [33], probabilistic [34] inference. Each of them represents a particular reification of a logic solver.

A common mechanism in LP is the unification algorithm [35] for constructing a most general unifier (MGU) among any two terms. Provided that a MGU exists, its subsequent application to the terms, makes them syntactically equal. This is a basic brick in
virtually all LP algorithms, regardless of the particular inference rule.

Summarising, LP leverages several mechanisms — terms and clauses representation, knowledge base storage, unification, resolution, etc. —, which constitute the basis of any logic solver. Subsets of these mechanisms may be useful per se. 2P-Kt makes LP mechanisms individually available, while easing the construction of novel mechanisms on top of the existing ones.

2.1. Software architecture

Architecturally, 2P-Kt is a framework supporting LBT development via several loosely-coupled modules. To support reusability, each module factorises related functionalities via compact API of OOP types and methods. As modules are the basic deployable units in 2P-Kt, major LP functionalities are partitioned into modules on a per-usage basis, making them selectively useable as dependencies by other projects. The 2P-Kt ecosystem itself incrementally combines such modules, as depicted in Fig. 1.

To maximise interoperability, 2P-Kr modules are individually available as precompiled libraries both on Maven Central Repository [36] — for JVM-, Android- or Kotlin-based contexts — and on the NPM Registry [37] — for JavaScript-based contexts —, whereas a detailed description of their API is available as a part of 2P-Kr documentation.

If all 2P-Kr modules were merged together, the most relevant aspects of their API could be summarised as in Fig. 2. The diagram shows how relevant LP aspect are reified into types: e.g.

- logic Terms (plus any specific sort of term, e.g. Variables, Structures, etc.),
- logic Substitutions, unification, and MGU (computed by an Unificator),
- Clauses (there including Rules, Facts, and Directives),
- knowledge bases and logic Theory,
- automatic reasoning, via Solvers, and
- logic Solutions—i.e. responses to users’ queries.

Interfaces expose relevant aspects, and keep the system extensible. Developers may for instance define custom implementations for the Unificator and Solver interfaces, to provide novel inference mechanisms involving some variant of unification.

Of course, a detailed diagram would include more features, as 2P-Kr supports: (i) (de)serialization of logic terms and theories into/from standard data-representation formats (e.g. JSON, or YAML), (ii) parsing/formatting terms and theories from/into concrete logic syntaxes such as Prolog’s one, (iii) extension of solvers via libraries of custom LP functionalities, and (iv) exploitation of solvers via command-line (CLI) or graphical (GUI) user interfaces, too.

2.2. Software functionalities

Here we enumerate 2P-Kr functionalities on a per-module basis. Following Gradle convention, we denote modules by :module:Name.

The most fundamental module is :core, which exposes data structures for knowledge representation via terms and clauses, other than methods supporting their manipulation—e.g. construction, unfolding, scoping, formatting, etc. Novel sorts of terms/clauses may be added by developers, by extending any public interface in :core. Furthermore, all types in :core leverage an immutable design, making them well suited for concurrent and multi-threaded scenarios.

Logic terms and clauses are often compared or manipulated via unification. Thus, we encapsulate this mechanism within the :unify module. It provides a general notion of Unificator – i.e. any algorithm aimed at computing MGU out of terms or clauses –, and a default implementation based on [35]. Developers may extend this implementation by configuring when terms should be considered equal. Similarly, they can provide custom Unificator implementations, in case they need a specific unification strategy, or need a different unification algorithm.

Another common need in LP is the in-memory storage of clauses into ordered (e.g. queues) or unordered (e.g. multisets) data structures, and their efficient retrieval via pattern-matching (e.g. unification). The :theory module follows this purpose, by providing notions such as ClauseQueue, ClauseMultiset—both involving an immutable (access-efficient) and mutable (update-efficient) implementation. These types differ from ordinary collections as they support unification-based retrieval and indexing of clauses. Prolog’s notions of theory and static/dynamic KB leverage on these types, exploiting the most adequate implementation in each case.

The practice of LP also involves ancillary operations over terms and clauses, e.g.: (i) formatting — into some costumisable form –, (ii) (de)serialisation — into/from data-representation formats –, and (iii) parsing—out of a particular concrete syntax. While formatting is a :core functionality, attained via TermFormatter, (de)serialisation and parsing require their own modules. Accordingly, module :serialize-core (resp. –theory) supports the (de)serialisation of terms (resp. theories) into JSON or YAML, via human-readable schemas. Thus, it supports distributed applications exchanging logic knowledge over the Internet. Similarly, parsing terms (resp. theories) in Prolog syntax is supported through the :parser-core (resp. –theory) module, leveraging ANTLR technology [38] for language engineering.

Generic API for logic solvers are available too, via the :solve module. Essentially, this module exposes the Solver type, representing any entity capable of performing some sort of logic resolution to provide Solutions to logic queries. However, resolution involves many practical aspects which are orthogonal w.r.t. any particular resolution strategy—e.g. errors management, extensibility via libraries, I/O, etc. Thus, :solve is quite an articulated (yet not directly useable) module.

Developers may build their inference procedure of choice by providing an implementation for the Solver interface, possibly reusing features from :solve in a selective way. Two implementations are currently available as part of 2P-Kr, namely :solve-classic and -streams, both implementing Prolog’s SLDNF resolution strategy. The latter is based on a state-machine-based design [22] and is currently stable, while :solve-streams is an experimental attempt of implementing Prolog via FP, as proposed in [39]. Notably, none of them leverages Warren’s Abstract Machine [40]—the computational model Prolog is commonly built upon.

Generic API for developing Prolog-like predicates in Kotlin are available as well. They exploit FP and OOP to let developers extend solvers with libraries of complex functionalities which are easier to implement in Kotlin than LP. There, data streams are treated as flows of solutions to be consumed by a solver. This makes 2P-Kr well-suited for handling long/infinite streams of data [41].

User experience (UX) is enabled by two more modules, namely :rep1 and :ide, which provide a CLI and GUI, respectively. While they both target JVM-specific UX, an experimental web-based GUI is available at [42], targeting JS-specific UX.

Other modules depicted in Fig. 1 do not need a specific description here: interested readers may read [43] for further details.
3. Illustrative examples

The 2P-Kt GUI (Fig. 3(a)) is a minimal IDE based on JavaFX. It lets users exploit LP interactively, repeatedly editing theories, performing queries, and inspecting the mutable internals of logic solvers. Users can open several files at once, perform queries one-by-one or all-at-once, or inspect the currently loaded libraries, operators, flags, etc. Syntax colouring completes the picture, easing users' writing of logic theories.

The CLI (Fig. 3(b)) lets users use logic solvers via a textual console. It supports both an interactive and non-interactive operation mode. Thus, it can either enter a Read-Eval-Print-Loop accepting logic queries from stdin and progressively prompting solutions to stdout, or simply accept queries and theories as arguments and prompts all possible solutions.

The Playground (Fig. 3(c)) is a proof-of-concept Web application mimicking the IDE. It demonstrates how 2P-Kr can be executed in-browser in a server-less fashion. It only requires Internet connection upon page loading. After that, it does not interact with the server anymore as the 2P-Kt JS scripts provide for a self-sufficient environment. Thus, logic computations need not any sandbox, nor logic solvers need API limitation for security reasons.

Finally, our Kotlin-based DSL for Prolog [43] can be exploited within Kotlin projects (Fig. 3(d)), by using any :dsl-* module as dependencies. It provides logic programmers with a syntactical way to inject LP into Kotlin scripts, making it possible to inherit the many tools available for Kotlin development, e.g. type checking, linting, code completion, debugging, etc.

4. Impact

The 2P-Kt technology may impact on many research areas. As far as LP is concerned, 2P-Kt provides a well-grounded technological basis for implementing/experimenting/extendin the many solutions proposed in the literature—e.g., abductive inference [32], rule induction [33], probabilistic reasoning [34], labelled LP [44].

As shown in [3], the multi-agent systems community has quite an appetite for interoperable and general-purpose LBT. There, 2P-Kt provides a technological substrate supporting agents' reasoning via manifold mechanisms.

2P-Kt is a valuable choice within the field of coordination [45], too: many tuple-based coordination models and technologies leverage LP and LBT [46]. There, 2P-Kt enables the implementation of interoperable Linda tuple spaces—such as in TuSoW [47]—or tuple centres—as we plan to do in TuCSoN [24].

Concerning programming paradigms, while most successful ones are being increasingly blended into modern programming languages, LP remains somewhat isolated [43]. Our Kotlin DSL for LP paves the way towards the integration of LP with other paradigms.

Finally, 2P-Kt has a role to play in the field of XAI [48]. Integrating symbolic and sub-symbolic AI—i.e. using them in synergy, as an ensemble [49]—is a strategic research direction [2], and 2P-Kt offers a sound technological foundation for this purpose [50].
**Future research directions.** 2P-Kr already enables the investigation of relevant research questions involving symbolic manipulation or automated reasoning, thanks to its modularity and interoperability.

Furthermore, 2P-Kr enables exploring how to: (i) integrate different LP aspects, (ii) blend LP with other AI techniques, and (iii) exploit LP to build flexible intelligent systems. Along these lines, our goals involve: (i) creating comprehensive solvers exploiting multiple inference procedures, knowledge-representation means, etc. at once in answering users’ queries, (ii) building hybrid systems where developers can transparently exploit sub-symbolic AI, and (iii) injecting LP into cognitive agents architectures.

As far as goal (i) is concerned, we are designing a unified API for probabilistic, abductive, or concurrent resolution. This would
enable further research towards mixed automated reasoning processes, where multiple inference procedures are dynamically interleaved within resolution.

As far as goal (ii) is concerned, we are designing logic-based API for machine learning and neural networks. Such API allow developers to define, train, assess, and use sub-symbolic predictors via LP. This would enable further research w.r.t. the integration of symbolic and sub-symbolic AI, the automation of machine learning workflows, and the exploitation of induced knowledge in LP.

Finally, about goal (iii), we are integrating multiple logics within BDI architectures. Intelligent agents may then adopt the most adequate reasoning or knowledge-representation means for the situation at hand. Thus, 2P-Kr enables further research towards the exploitation of different logics to support intelligent, context-specific behaviours for software agents, by providing the underlying reasoning facilities.

2P-Kr adoption. While tuProlog has been exploited both in the industry and in the academia [51], 2P-Kr has been used in the academia only.

2P-Kr already works (or is going to work) as the underlying technology of many scientific contributions. Some, such as TuCSoN [24], ReSpecT [25], LPaAs [26], or Tenderfence [27] leveraged on tuProlog for their implementation, and are being migrated on 2P-Kr. Others, such as TuSoW [47], Arg2P [28], or our Kotlin-based DSL [43] already exploit 2P-Kr.

5. Conclusion

This paper introduces 2P-Kr, an open, general, Kotlin Multiplatform ecosystem for LP, supporting manifold mechanisms for automated reasoning, via several loosely-coupled modules. Each module makes some specific LP aspects individually usable. Selectively reusing/extending modules enables bootstrapping novel LBT without re-implementing everything from scratch or producing Prolog-centered monoliths. In particular, 2P-Kr supports mixed inference procedures, involving both symbolic and sub-symbolic techniques.

The 2P-Kr ecosystem is structured by keeping reusability, extensibility and interoperability in mind. Its functionalities include knowledge representation, (de)serialisation, parsing (and formatting facilities), unification, clause in-memory indexing and storage facilities, logic inference via SLDNF, UX, and rich Kotlin API for developers. They all support JVM, JavaScript, and Android platforms.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

R. Calegari has been supported by the H2020 ERC Project ”Compulaw” (G.A. 833647). A. Omicini has been partially supported by the CHIST-ERA IV project ”EXPECTATION” (G.A. CHIST-ERA-19-XAI-005).

The authors would like to thank Enrico Siboni for his contribution in the early phases of 2P-Kr development.

References


