2P-Kt: logic programming with objects & functions in Kotlin

Giovanni Ciatto[a], Roberta Calegari[b], Enrico Siboni[c], Enrico Denti[a] and Andrea Omicini[a]

[a]*Dipartimento di Informatica – Scienza e Ingegneria (DISI), Alma Mater Studiorum—Università di Bologna, Italy*

[b]*Alma Mater Research Institute for Human-Centered Artificial Intelligence, Alma Mater Studiorum—Università di Bologna, Italy*

[c]*University of Applied Sciences and Arts of Western Switzerland (HES-SO), Sierre, Switzerland*

Abstract

Mainstream programming languages nowadays tends to be more and more *multi-paradigm* ones, by integrating diverse programming paradigms—e.g., object-oriented programming (OOP) and functional programming (FP). Logic-programming (LP) is a successful paradigm that has contributed to many relevant results in the areas of symbolic AI and multi-agent systems, among the others. Whereas Prolog, the most successful LP language, is typically integrated with mainstream languages via foreign language interfaces, in this paper we propose an alternative approach based on the notion of *domain-specific language* (DSL), which makes LP available to OOP programmers straightforwardly within their OO language of choice. In particular, we present a *Kotlin DSL* for Prolog, showing how the Kotlin multi-paradigm (OOP+FP) language can be enriched with LP in a straightforward and effective way. Since it is based on the interoperable 2P-Kt project, our technique also enables the creation of similar DSL on top of other high-level languages such as Scala or JavaScript—thus paving the way towards a more general adoption of LP in general-purpose programming environments.

Keywords

logic programming, object-oriented programming, multi-paradigm languages, domain-specific languages, Kotlin

1. Introduction

Logic Programming (LP) [1, 2] is a programming paradigm based on formal logic, inspired to the idea of declaratively specifying a program semantics via logic formulæ, so that automatic reasoners can then prove such formulæ by leveraging on different *control* strategies. In the years, LP has contributed to the development of a number of diverse research fields laying under the umbrella of symbolic AI—such as automatic theorem proving, multi-agent systems, model checking, research optimisation, natural language processing, etc.

WOA 2020: Workshop "From Objects to Agents", September 14–16, 2020, Bologna, Italy

giovanni.ciatto@unibo.it (G. Ciatto); roberta.calegari@unibo.it (R. Calegari); enrico.siboni@hevs.ch (E. Siboni); enrico.denti@unibo.it (E. Denti); andrea.omicini@unibo.it (A. Omicini)

https://about.me/gciatto (G. Ciatto); http://robertacalegari.apice.unibo.it (R. Calegari); http://enricodenti.apice.unibo.it (E. Denti); http://andreaomicini.apice.unibo.it (A. Omicini)

0000-0002-1841-8996 (G. Ciatto); 0000-0003-3794-2942 (R. Calegari); 0000-0003-3584-8637 (E. Siboni); 0000-0003-1687-8793 (E. Denti); 0000-0002-6655-3869 (A. Omicini)

Nowadays, LP is one of the major programming paradigms available for software development, along with the imperative, functional, and object-oriented ones. In particular, LP is today one of the best-suited choices for tackling problems involving knowledge representation, logic inference, automated reasoning, search in a discrete space, or meta-programming [3]. Moreover, today LP supports the core of AI components in pervasive and distributed systems, providing intelligence where and when it is needed [4].

This is why the *integration* of LP within the main programming languages is nowadays more interesting than ever. However, to make it actually work, integration should be designed – from a linguistic standpoint – so as to reduce both the development time and the learning curve of developers, as well as the psychological and cultural resistances against the adoption of new paradigms—thus, basically, moving LP up to a manageable level by the OOP developer.

Most mainstream programming languages – such as Java, Kotlin, Scala, Python, JavaScript, C# – have recognised the added value of the integration of diverse programming paradigms under a unique syntax, a coherent API, and a rich standard library. In fact, they all already support both the object-oriented (OOP) and functional (FP) programming paradigms. We believe that the same languages would largely benefit from extending their support to LP languages as well, making them usable in the OOP context in the same way as FP features already are.

Interoperability among Prolog [5] (as the first and most prominent LP language) with other languages from different paradigms is not new: the Prolog community has actually been studying this theme for years [6], as shown by the many forms of integration historically present in most Prolog systems, providing either *(i)* logic-to-OOP interoperability, making object orientation available within Prolog scripts, or *(ii)* OOP-to-logic interoperability, making logic programming exploitable within object-oriented code—or both, as in the case of tuProlog Java Library [7]. It is worth noting here that existing integrations make quite strong assumptions. For instance, item *(i)* assumes the main application is written in Prolog and the interoperation is given via a suitable interface allowing the injection of (small) parts written in other languages, while item *(ii)* implicitly assumes LP and Prolog to be somehow *harmonised* with the language(s) hosting them, at the paradigm, syntactical, and technological levels. Both these assumptions can actually create a barrier against an effective adoption of LP in today applications despite its huge potential.

Therefore, the approach adopted in this work is to devise instead a form of LP-FP-OOP *blended integration*, such that the key features of LP paradigm can be exposed and made available to mainstream language developers in a way that is the most natural in that context. The key requirement, therefore, is the "making LP easy to adopt for OOP programmers", in order to break down the learning curve for non-experts, and pursuit the typical developers' mindset. Going beyond the mere interoperability intended as simple technical habilitation, our approach is meant to embrace the perspective of the OOP developer aiming at exploiting the potential of LP.

To this end, we show how designing Prolog as a *domain-specific language* (DSL) for an OOP language such as Kotlin can make LP close enough to the OOP developers' mindset to overcome most cultural barriers, while at the same mimicking the Prolog syntax and semantics close enough to make its use straightforward for LP developers.

Generally speaking, the integration of Prolog with other paradigms aims at bringing the effectiveness and declarativeness of LP into general-purpose OOP framework. In particular,

OOP designers and programmers can be expected to benefit from LP features for *(i)* data-driven or data-intensive computations, such as in the case of complex-event-processing frameworks; *(ii)* operation research or symbolic AI algorithms, or more generally other algorithms involving a search space to be explored; *(iii)* multi-agent systems, and the many areas in that field leveraging on LP, such as knowledge representation, argumentation, normative systems, etc.

The proposed solution is based on the 2P-Kt technology [8], a re-engineering of the tuProlog project [9, 10] as a Kotlin multi-platform library supporting the JVM, JS, Android, and Native platforms. The case of a Kotlin-based Prolog DSL is presented and discussed.

Accordingly, the paper is organised as follows. Section 2 briefly introduces LP, providing details about tuProlog, 2P-Kt, and Kotlin as well. It also summarises the current state of the integration between LP and mainstream programming languages. Section 3 discusses the rationale, design, and architecture of our Prolog-Kotlin DSL along with some examples. Section 4, briefly illustrates a case study where our DSL is used in combination with functional programming to solve a simple AI task in an elegant way. Finally, in section 5, concludes the paper by providing some insights about the possible future research directions stemming from our work.

2. State of the Art

2.1. Logic programming

Logic programming (LP) is a programming paradigm based on computational logic [11, 12]. In LP languages, a program is typically a set of logical relations, that is, a set of sentences in logical form, expressing *facts* and *rules* about some domain. In Prolog [5] , in particular, both facts and rules are written in the form of *clauses*, i.e. H `:-` B_1, ..., B_n, which are read declaratively as logical implications (H is true if all B_i are true). While both H and B_i are *atomic* formulæ, H is called the *head* of the rule and $(B_1, ..., B_n)$ is called the *body*. Facts are rules without body, and they are written in the simplified form: H.

An atomic formula is an expression in the form $P(t_1, ..., t_m)$ where P is a predicate having m arguments, $m \geq 0$, and $t_1, ..., t_m$ are *terms*. Terms are the most general sort of data structure used in Prolog. A term is either a constant (either a number or an atom), a variable, or a (possibly recursive) function of terms. *Variables* are named placeholders which may occur within clauses to reference (yet) unknown objects. When reading clauses declaratively, variables are assumed to be universally quantified if occurring into a rule head, existentially quantified if in the body.

A logic program is executed by an inference engine that answers a query by trying to prove it is *inferable* from the available facts and rules. In particular, the Prolog interpreter exploits a *deduction* method based on the SLD resolution principle introduced in [2]. The SLD principle, in turn, heavily leverages on the unification algorithm [13] for constructing a most general unifier (MGU) for any two suitable terms. Provided that such a MGU exists, the subsequent application of the resulting substitution to the terms, renders them syntactically equal. These two mechanisms – resolution and unification – constitute the basis of any Prolog solver. In particular, SLDNF is an extension of SLD resolution for dealing with *negation as failure* [14]. In SLDNF, goal clauses can contain negation as failure literals—i.e. the form $not(p)$.

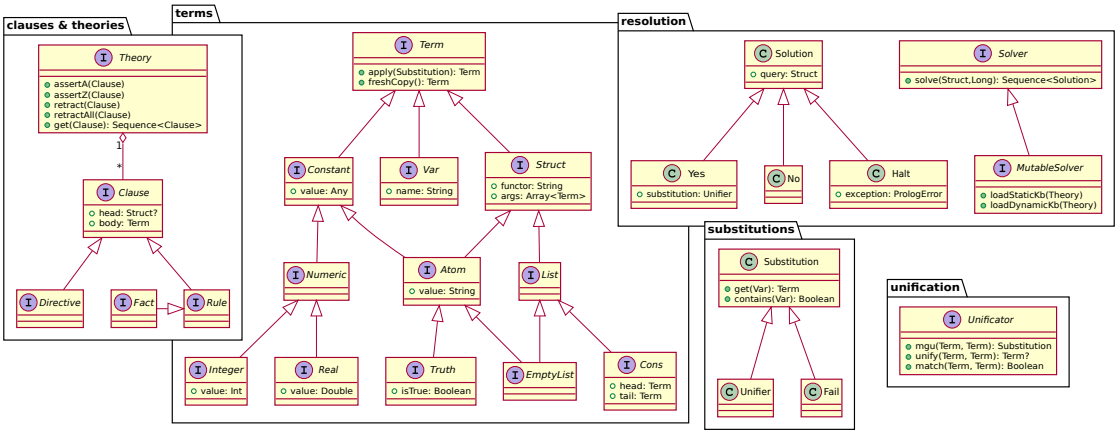


Figure 1: Overview on the public API of 2P-Kt: a type is provided for each relevant concept in LP

2.2. tuProlog and 2P-Kt

tuProlog [10] (2P for short) is a logic programming framework supporting multi-paradigm programming via a clean, seamless, and bidirectional integration between the logic and object-oriented paradigms. 2P-Kt is a Kotlin project for LP, representing a complete rewriting of the tuProlog project, whose ultimate purpose is to provide an open LP ecosystem supporting multiple platforms and programming paradigms.

More importantly, 2P-Kt includes a rich API providing many logic-programming facilities to developers. For instance, it enables the exploitation of clauses and terms to represent symbolic knowledge via FOL in Kotlin. Clauses and terms can be manipulated via logic unification, possibly producing substitutions to be applied to other clauses and terms. Logic theories can be created as indexed and ordered sequences of clauses, enabling the exploitation of classical `assert` or `retract` operations – as well as bare clause retrieval via pattern-matching – in an efficient way. Finally, the full potential of Prolog SLDNF resolution strategy can be exploited, if needed, via lightweight solvers which could be instantiated and queried on the fly, by lazily consuming their solutions.

Accordingly, the 2P-Kt API – sketched in fig. 1 – exposes a Kotlin type for each relevant LP concept: *(i)* logic `Terms` (as well as for each particular sort of term, e.g. `Variables`, or `Structures`, etc.), *(ii)* logic `Substitutions` and `Unification`, *(iii)* Horn `Clauses` (there including `Rules`, `Facts`, and `Directives`), *(iv)* knowledge bases and logic theories (e.g. the `Theory` type), and *(v)* automatic reasoning via Prolog's SLDNF resolution strategy, as provided by the `Solver` interface.

2.3. Prolog integration

In order to integrate Prolog with high-level languages, many Prolog implementations expose a foreign language interface (FLI, table 1). In most cases, the target language is Java, and the FLI is bi-directional—meaning that it supports both "calling Prolog from the target language" and vice versa. In a few particular cases, however, the JavaScript and C# languages are also supported.

Table 1
Prolog implementations and their foreign language interfaces (FLI)

Prolog Implementation	Platform	FLI towards	Nature of FLI	Paradigm of FLI	Source
BProlog [15]	C	Java	JNI	imperative	Official BProlog doc.
Ciao! [16]	C	Java	TCP/IP	imperative	Official Ciao Prolog doc.
ECLiPSe [17]	C	Java	JNI	imperative	Official ECLiPSe doc.
SICStus [18]	C	Java, C#	TCP/IP	object-oriented	Jasper Library
SWI [19]	C	Java	JNI	object-oriented	JPL API
τProlog [20]	JS	JavaScript	Native	object-oriented	Project homepage
tuProlog [9]	JS, JVM Android	Kotlin, Java, JavaScript	Native	object-oriented, functional	Project homepage
XSB [21]	C	Java	TCP/IP	imperative	Interprolog Java Bridge

As we are mostly interested in discussing how and to what extent Prolog exploitation is possible within the target languages, in the reminder of this section we only focus on those FLI allowing to "call Prolog from the target language".

Each FLI mentioned in table 1 is characterised by a number of features that heavily impact the way the users of the target languages may actually exploit Prolog. These are, for instance, the nature of the FLI – which is tightly related to the target platform of the underlying Prolog implementation – and its reference programming paradigm.

By "nature" of the FLI, we mean the technological mechanism exploited by a particular Prolog implementation to interact with the target language. There are some cases – like tuProlog and τProlog – where this aspect is trivial, because the target language is also the implementation language—so Prolog is considered there as just another library for the target language. More commonly, however, Prolog is implemented in C, and the Java Native Interface (JNI) is exploited to make it callable from Java. This is for instance the case of SWI- and ECLiPSe-Prolog. While this solution is very efficient, it hinders the portability of Prolog into particular contexts such as Android, and its exploitation within mobile applications.

Another viable solution is to leverage on the TCP/IP protocol stack. This is for instance the case of SICStus- and Ciao-Prolog. The general idea behind this approach is that the Prolog implementation acts as a remote server offering logic-programming services to the target language via TCP/IP, provided that a client library exists on the Java side making the exploitation of TCP/IP transparent to the users. While this solution is more portable – as virtually any sort of device supports TCP/IP –, it raises efficiency issues because of the overhead due to network/inter-process communications. A more general solution of that sort is instead offered by the LPaaS architecture [22], where the fundamental idea is to have many Prolog engines distributed over the network, accessed as logic-based web services. By the way, the implementation of LPaaS[1] is based on tuProlog.

By "reference programming paradigm" of the FLI we mean the specific programming style proposed by a Prolog implementation to the target language users through its API. Some FLI are conceived to be used in a strictly imperative way: this is e.g. the case of BProlog or Ciao! Prolog, where a Java API is available for writing Prolog queries and issue them towards the underlying Prolog system in an imperative way. Other Prolog implementations, such as SICStus- and

[1] http://lpaas.apice.unibo.it

SWI-Prolog, offer a more object-oriented API which let developer not only represent queries but also terms, and solutions (there including variable bindings) which can be consumed through ordinary OOP mechanisms such as iterators.

In most cases, however, the code to be produced in the target language is far less concise and compact than pure Prolog – unless strings and parsing are extensively adopted –, especially when the size of the Prolog code to be represented grows. So, for instance, the simple Prolog query `?- parent(adam, X)` (aimed at computing who Adam's son is) in Java through SWI-Prolog's JPL interface would be written as:

```
Query query = new Query("parent", new Term[] { new Atom("adam"), new
↪  Variable("X") } );
Map<String,Term> solution = query.oneSolution();
System.out.println("The child of Adam is " + solution.get("X"));
```

While this a totally effective solution on the technical level, we argue that a more convenient integration among LP and the other paradigms is possible. In particular, in this paper we show how 2P-Kt allows for a finer integration at the paradigm level through *domain-specific languages*.

2.4. Kotlin Domain-Specific Languages (DSL)

Domain-specific languages (DSL) are a common way to tackle recurrent problems in a general way via software engineering. When a software artefact (e.g. library, architecture, system) is available to tackle with some sort of problems, designers may provide a DSL for that artefact to ease its usage for practitioners. There, the exploitation of a DSL hides the complexity of the library behind the scenes, while supporting the usage of the artefact for non-expert users too. This is, for instance, the approach of the Gradle build system[2]—which is nowadays one of the most successful build automation systems for the JVM, Android, and C/C++ platforms. It is also the approach followed by the JADE agent programming framework, which is nowadays usable through the Jadescript DSL [23].

Building a DSL usually requires the definition of a concrete syntax, the design and implementation of a compiler or code generator, and the creation of an ecosystem of tools—e.g., syntax highlighters, syntax checkers, debuggers. In particular, compilation or code generation are fundamental as they are what makes a DSL machine-interpretable and -executable.

Some high-level languages, however, such as Kotlin, Groovy, or Scala, enable a different approach. They come with a flexible syntax which *natively* supports the definition of new DSL with no addition to the hosting language required. For instance, Gradle consists of a JVM library of methods for building, testing, and deploying sources codes, plus a Kotlin- or Groovy-based DSL allowing developers to customise their particular workflows.

The advantages of this approach are manifold. First, no compiler or code generator has to be built, as the DSL is already part of the hosting language and it is therefore machine-interpretable and -executable by construction. Second, the DSL automatically inherits all the constructs,

[2] https://gradle.org

API, and libraries of the hosting language—there including conditional or iterative constructs, string manipulation API, etc., which are common and useful for many DSL, regardless of their particular domain. Third, the ecosystem of tools supporting the hosting language – e.g. compilers, debuggers, formatters, code analysers, IDE, etc. – can be reused for the DSL as well, easing its adoption and making it more valuable.

When Kotlin is the hosting language of choice, DSL leverage on a small set of features making the Kotlin syntax very flexible, described below:

operator overloading[3] — allowing ordinary arithmetic, comparison, access, and function invocation operators to change their ordinary meaning on a per-type basis;

block-like lambda expressions[4] — including a number of syntactic sugar options such as *(i)* the possibility to omit formal parameters in case of a single-argument lambda expression, and *(ii)* the possibility to omit the round parentheses in case of a function invocation having a lambda expression as a last argument;

function types/literals with receiver[4] — allowing functions and methods to accept lambda expressions within which the `this` variable references a different object than the outer scope;

extension methods[5] — allowing pre-existing types to be extended with new instance methods whose visibility is scope-sensible.

Of course, Kotlin-based DSL automatically inherit the full gamma of facilities exposed by the Kotlin language and standard library—there including support for imperative, and object-oriented programming, as well as a rich API supporting functional programming through most common high-order operations. Furthermore, if properly engineered, these DSL may be executed on all the platforms supported by Kotlin—which commonly include, at least, the JVM, JavaScript, and Android. This is for instance the case of our DSL proposed in section 3.

While this paper focuses on Kotlin-based DSL, similar results could be achieved in other languages by means of equivalent mechanisms. For instance, in Scala, extension methods have to be emulated via implicit classes, and DSL, in particular, can be built via the "Pimp My Library" pattern [24].

3. A domain-specific language for LP

3.1. Design Rationale

Regardless of the technological choices, the design of our DSL leverages on a small set of principles ($\mathbf{P_i}$) briefly discussed below. In fact, our aim is to ($\mathbf{P_1}$) provide a DSL that is a *strict* extension of its hosting language, meaning that no feature of the latter language is prevented by the usage of our DSL. Dually, we require ($\mathbf{P_2}$) our DSL to be fully interoperable and finely integrated with the hosting language, meaning that all the features of the latter language can be

[3] https://kotlinlang.org/docs/reference/operator-overloading.html
[4] https://kotlinlang.org/docs/reference/lambdas.html
[5] https://kotlinlang.org/docs/reference/extensions.html

exploited from within our DSL. However, we also require ($\mathbf{P_3}$) the DSL to be well encapsulated and clearly identifiable within the hosting language, in order to prevent unintended usage of the DSL itself. Finally, we require ($\mathbf{P_4}$) our DSL to be as close as possible to Prolog, both at the syntactic and semantic level, to ease its exploitation for logic programmers.

In order to accomplish to the aforementioned principles, we choose the Kotlin language as the technological reference for prototyping our proposal. This is because it *(i)* comes with a flexible syntax supporting the definition of DSL, *(ii)* supports a considerable number of platforms (JVM, JavaScript, Android, Native) and therefore enables a wide exploitation of LP – for instance, on smart devices –, *(iii)* includes a number of libraries supporting LP-based applications, such as 2P-Kt.

3.2. The Kotlin DSL for Prolog

Before dwelling into the details of our proposal, we provide an overview of what a Kotlin DSL for Prolog has to offer.

Let us consider the following Prolog theory describing a portion of Abraham's family tree:

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

parent(abraham, isaac).
parent(isaac, jacob).
parent(jacob, joseph).
```

It enables a number of queries, e.g. `ancestor(abraham, X)`, which can be read as "Does there exist some `X` which is a descendant of Abraham?". According to the Prolog semantics, this query may have a number of solutions, enumerating all the possible descendants of Abraham that can be deduced from the above theory—i.e., Isaac, Jacob, and Joseph.

The same result may be attained through the following Kotlin program, which leverages on our DSL for Prolog:

```
prolog {
  staticKb(
    rule {
      "ancestor"("X", "Y") `if` "parent"("X", "Y")
    },
    rule {
      "ancestor"("X", "Y") `if` ("parent"("X", "Z") and
      ↪  "ancestor"("Z", "Y"))
    },
    fact { "parent"("abraham", "isaac") },
    fact { "parent"("isaac", "jacob") },
    fact { "parent"("jacob", "joseph") }
  )
```

```
  for (sol in solve("ancestor"("abraham", "X")))
    if (sol is Solution.Yes)
      println(sol.substitution["X"])
}
```

The program creates a Prolog solver and initialises it with standard built-in predicates. Then it loads a number of facts and rules representing the aforementioned Prolog theory about Abraham's family tree. Finally, it exploits the solver to find all the descendants of Abraham, by issuing the query `ancestor(abraham, X)`.

It is worth noting how the simple code snippet exemplified above is adherent w.r.t. our principles. In fact, the whole DSL is *encapsulated* ($\mathbf{P_3}$) within the

$$\texttt{prolog } \{ \ \langle \textit{DSL block} \rangle \ \}$$

In there, LP facilities can be exploited in combination with the imperative and functional constructs offered by the Kotlin language and its standard-library ($\mathbf{P_1}$ and $\mathbf{P_2}$)—such as the `for-each`-loop used to print solutions in the snippet above. Furthermore, within `prolog` blocks, both theories and queries are expressed via a Kotlin-compliant syntax mimicking Prolog ($\mathbf{P_4}$). The main idea behind such syntax is that each expression in the form

$$\texttt{"functor"}(\langle e_1 \rangle, \ \langle e_2 \rangle, \ \ldots)$$

is interpreted as a compound term (a.k.a. structure) in the form $\texttt{functor}(t_1, \ t_2, \ \ldots)$, provided that each Kotlin expression e_i can be recursively evaluated as the term t_i—e.g. capital strings such as `"X"` are interpreted as variables, whereas non-capital strings such as `"atom"` (as well as Kotlin numbers) are interpreted as Prolog constants. In a similar way, expressions of the form[6]

$$\texttt{rule } \{\texttt{"head"}(\langle e_1 \rangle, \ \ldots, \ \langle e_N \rangle) \ \texttt{`if`} \ (\langle e_{N+1} \rangle \ \texttt{and} \ \ldots \ \texttt{and} \ \langle e_M \rangle)\}$$

are interpreted as Prolog rules of the form

$$\texttt{head}(t_1, \ \ldots, \ t_N) \ \texttt{:-} \ t_{N+1}, \ \ldots, \ t_M$$

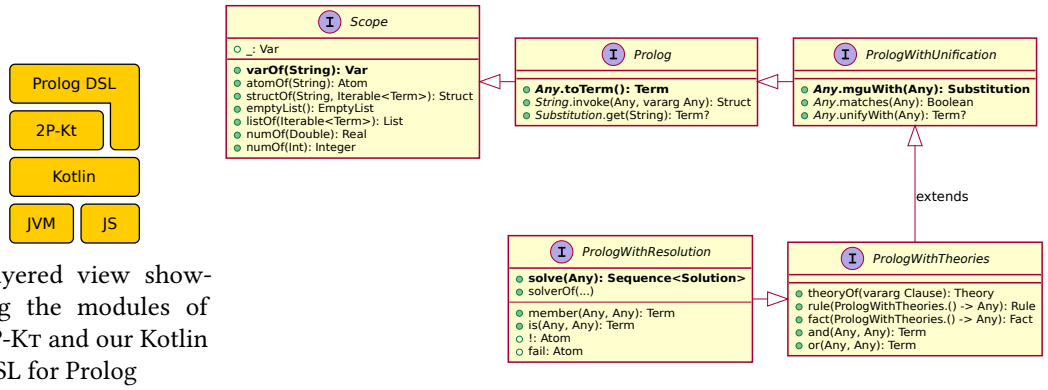
provided that $M > N$ and each Kotlin expression e_i can be recursively evaluated as the term t_i. A similar statement holds for fact expressions of the form `fact { . . . } .`

To support our DSL for Prolog, a number of Kotlin classes and interfaces have been designed on top of the 2P-Kt library, exploiting manifold extensions methods, overloaded operators, etc. to provide the syntax described so far. The details of our solution – there including its architecture, design, and implementation – are discussed in the reminder of this section.

3.3. Architecture, Design, Implementation

The Kotlin DSL for Prolog is essentially a compact means to instantiate and use objects from the 2P-Kt library, through a Prolog-like syntax. More precisely, it consists of a small software layer

[6] backticks make Kotlin parse words as identifiers instead of keywords



(a) Layered view showing the modules of 2P-Kt and our Kotlin DSL for Prolog

(b) Structural view showing the API of our Kotlin DSL for Prolog, leveraging on the 2P-Kt API

Figure 2: Architectural view of our Kotlin DSL for Prolog

built on top of Kotlin and 2P-Kt, as represented by fig. 2a. In particular, the DSL is enabled by the five *factory interfaces* depicted in fig. 2b. We call *factory interface* a type definition whose methods are aimed at *instantiating* objects of related types, as dictated by the Gang of Four' abstract factory pattern [25].

The five factory interfaces are: `Scope`, `Prolog`, `PrologWithUnification`, `PrologWithTheories`, and `PrologWithResolution`. Each factory type extends the previous one with more LP-related functionalities. So, for instance, while instances of `Scope` simply provide the basic bricks to create logic terms, instances of `Prolog` leverage on these bricks to enable the exploitation of the Prolog-like syntax exemplified above. `PrologWithUnification` extends `Prolog` with unification-related facilities, and it is in turn extended by `PrologWithTheories`, which lets developers create both clauses and theories via a Prolog-like syntax. Finally `PrologWithResolution` extends `PrologWithTheories` by adding resolution-related facilities, plus some syntactic shortcuts for writing rules exploiting Prolog standard predicates such as `member/2`, `length/1`, etc.

In the following we provide further details about how each factory contributes to our DSL.

Scope The simplest factory type is `Scope`. As suggested by its name, it is aimed at building terms which must share one or more variables. For this reason, it exposes a number of factory methods – roughly, one for each sub-type of `Term` –, some of which are mentioned in fig. 2b. The main purpose of a scope, however, is to enable the creation of objects reusing the same logic `Variables` more than once. Thus, it includes a method – namely, `varOf(String)` – which always returns the same variable if the same name is provided as input.

For example, to create the term `member(X, [X | T])`, one may write:

```
val m = Scope.empty {
  structOf("member", varOf("X"), consOf(varOf("X"), varOf("T")))
} // m references the term member(X, [X | T])
```

There, the two calls to `varOf("X")` actually return the same object, if occurring within the same scope—whereas they would return different variables if invoked on different scopes. This mechanism is what enables developers to instantiate terms and clauses without having to explicitly refresh variables among different clauses: it is sufficient to create them within different scopes.

Prolog The `Prolog` factory type extends `Scope` by adding the capability of creating terms through a Prolog-like syntax. To do so, it exposes a number of extension methods aimed at automatically converting Kotlin objects into Prolog terms or using Kotlin objects in place of Prolog terms. The most relevant extension methods are mentioned in fig. 2b. These methods are:

- `fun Any.toTerm(): Term`, which is an extension method aimed at making any Kotlin object convertible into a Prolog term, through the syntax `obj.toTerm()`. To convert an object into a term, it leverages on the following type mapping: *(i)* a Kotlin number is either converted into a Prolog real or integer number, depending on whether the input number is floating-point or not, *(ii)* a Kotlin string is either converted into a Prolog variable or atom, depending on whether the input string starts with a capital letter or not, *(iii)* a Kotlin boolean is always converted into a Prolog atom, *(iv)* a Kotlin iterable (be it an array, a list, or any other collection) is always converted into a Prolog list, provided that each item can be recursively converted into a term, *(v)* a Kotlin object remains unaffected if it is already an instance of `Term`, *(vi)* an error is raised if the input object cannot be converted into a `Term`.
- `operator fun String.invoke(vararg Any): Struct`, which is an extension method aimed at overloading the function invocation operator for strings. Its purpose is to enable the construction of compound terms through the syntax `"f"(`$\texttt{arg}_1$`, . . . , `$\texttt{arg}_N$`)`, which mimics Prolog. Its semantics is straightforward: assuming that each $\texttt{arg}_i$ can be converted into a term via the `Any.toTerm()` extension method above, this method creates a N-ary `Structure` whose functor is `"f"` and whose i^{th} is $\texttt{arg}_i$`.toTerm()`. So, for instance, the expression

  ```
  "member"("X", arrayOf(1, true))
  ```

  creates the Prolog term `member(X, [1, true])`.
- `fun Substitution.get(String): Term?`, which is an extension method aimed at overloading the `get` method of the `Substitution` type in such a way that is can also accept a string other than a `Variable`. This enables DSL users to write expressions such as

  ```
  substitution.get("X")
  ```

  instead of having to create a variable explicitly via `varOf("X")`. While we only discuss this method, the main idea here is that every method in the 2P-Kt API accepting some basic Prolog type – such as `Var`, `Atom`, or `Real` – as argument should be similarly overloaded to accept the corresponding Kotlin type as well—e.g. `String` or `Double`. This is what enables a fine-grained integration of our DSL with the Kotlin language and the 2P-Kt library.

It is worth highlighting that every `Prolog` object is also a particular sort of `Scope`. So, converting the same string into a `Variable` twice or more times, within the same `Prolog` object, always yields the exact same variable.

Prolog with unification The `PrologWithUnification` factory type extends `Prolog` by adding the capability of *(i)* computing the most general unifier (MGU) among two terms, *(ii)* checking whether two terms match or not according to logic unification – i.e., checking if a MGU exists unifying the two terms –, and *(iii)* computing the term attained by unifying two terms—assuming an MGU exists for them. To do so, it exposes a number of extension methods aimed at providing unification-related support to Kotlin objects, provided that they can be converted into terms. The most relevant extension methods are mentioned in fig. 2b.

Prolog with theories The `PrologWithTheories` factory type extends `PrologWithUnification` by adding the capability of creating logic clauses (e.g. rules and facts) and theories. To do so, it exposes a number of methods aimed supporting the conversion of Kotlin objects into Prolog clauses – through the syntactic facilities presented so far –, and their combination into theories. The most relevant methods, mentioned in fig. 2b, are the following:

- `fun theoryOf(vararg Clause): Theory`, an ordinary method aimed at creating a logic `Theory` out of a variable amount of clauses.
- `infix fun Any.` `` `if` `` `(Any): Rule`, which is an extension method aimed at creating logic rules via a Prolog-like syntax in the form `head` `` `if` `` `body`. Its semantics is straightforward: assuming that both `head` and `body` can be converted into logic goals via the `Any.toTerm()` extension method above, this method creates a binary `Structure` whose functor is `':-'` and whose arguments are `head.toTerm()` and `body.toTerm()`. Similar methods exists – namely, `and`, `or`, etc. – to create conjunctions or disjunctions of clauses.
- `fun rule(PrologWithTheories.() -> Any): Rule`, an ordinary method aimed at creating a rule in a separate scope, thus avoiding the risk of accidentally referencing the variables created elsewhere. It creates a fresh, empty, and nested instance of `PrologWithTheories` and accepts a function with receiver to be invoked on that nested instance. The function is expected to return a Kotlin object which can be converted into a Prolog rule. Any variable possibly created within the nested scope is guaranteed to be different than any homonymous variable defined elsewhere.
- `fun fact(PrologWithTheories.() -> Any): Fact`, analogous to the previous method, except that it is aimed at creating Prolog facts. So, for instance, one may write

```
val r1 = fact {
    "member"("X", consOf("X", `_`))
}
val r2 = rule {
```

```
        "member"("X", consOf(`_`, "T")) `if` "member"("X", "T")
    }
```

while being sure that the `X` variable used in `r1` is different than the one used in `r2`.

Prolog with resolution The `PrologWithResolution` factory type extends `PrologWithUnification` by adding the capability of performing logic queries and consuming their solutions attained through the standard Prolog semantics. To do so, it exposes a number of methods aimed at supporting *(i)* the instantiation of Prolog `Solvers`, *(ii)* the loading of Prolog theories, either as static or dynamic knowledge bases (KB), and *(iii)* the invocation of Prolog queries on those KB. The most relevant methods, mentioned in fig. 2b, are the following:

- `fun solve(Any, Long): Sequence<Solution>` , which is an ordinary method aimed at executing Prolog queries without requiring a new solver to be explicitly created. It accept a Kotlin object as argument – which must be convertible into a Prolog query –, and an optional timeout limiting the total amount of time the solver may exploit to compute a solution. If the provided arguments are well formed, this method returns a `Sequence` of `Solutions` which *lazily* enumerates all the possible answers to the query provided as input, using Prolog's SLDNF proof procedure.
- `fun staticKb(vararg Clause)` , which is an ordinary method aimed at loading the *static* KB the `solve` method above will operate upon.
- `fun dynamicKb(vararg Clause)` , which is analogous to the previous method, except that it loads the *dynamic* KB the `solve` method above will operate upon.
- `fun member(Any, Any): Struct` which is an ordinary method aimed at easily creating invocations of the `member/2` built-in predicate. While we only discuss this method, the main idea here is that every standard built-in predicate in Prolog has a Kotlin counterpart in `PrologWithResolution` accepting the proper amount or arguments and returning an invocation to that built-in predicate. This is aimed easing the exploitation of the standard Prolog predicates to developers leveraging our DSL. So, for instance, we also provide facility methods for built-in such as `is/2`, `+/2`, `-/2`, `!/0`, `fail/0`, `append/3`, etc.

Instances of `PrologWithResolution` are created and used via the

```
fun <R> prolog(PrologWithResolution.() -> R): R
```

static method, which accepts a lambda expression letting the user exploit the DSL on the fly, and returns the object created by this lambda expression. This is for instance what enables developers to write the code snippet discussed in section 3.2.

4. Case study: N-Queens

We present now a brief example demonstrating how, by integrating multiple programming paradigms, developers may easily produce compact and effective solutions. Suppose one is willing to implement the following Kotlin method

```
fun nQueens(n: Int): Sequence<List<Position>>
```

aimed at computing all the possible solutions to the N-Queens problem. More precisely, the method is expected to enumerate all the possible dispositions of n queens on a $n \times n$ chessboard, such that no queen can be attacked by others. Each solution can be represented by a list of queen positions, in the form $[(1, Y_1), \ldots, (n, Y_n)]$, where each Y_i represent the row occupied by the i^{th} queen—i.e., the one in column i.

Computing all solutions for the N-Queens problem may require a lot of code in most programming paradigms. However, in LP, the solution is straightforward and compact. One may, for instance, leverage the following Prolog code:

```
no_attack((X1, Y1), (X2, Y2)) :-
  X1 =\= X2, % infix operator
  Y1 =\= Y2,
  (Y2 - Y1) =\= (X2 - X1),
  (Y2 - Y1) =\= (X1 - X2). % arithmetic expression

no_attack_all(_, []).
no_attack_all(C , [H | Hs]) :-
  no_attack(C, H),
  no_attack_all(C, Hs).

solution(_, []).
solution(N, [(X, Y) | Cs]) :-
  solution(N, Cs),
  between(1, N, Y), % built-in predicate
  no_attack_all((X, Y), Cs).
```

which can satisfy queries in the form

```
?- solution(N, [(1, Y1), ..., (N, YN)])
```

(provided that some actual N is given) by instantiating each variable `Yi`.

Thanks to our Kotlin DSL for Prolog, one may exploit the elegance of LP in implementing the aforementioned `nQueens` method. For instance, one may implement it as follows:

```
fun nQueens(n: Int) = prolog {
  staticKb(
    rule {
```

```
        "no_attack"(("X1" and "Y1"), ("X2" and "Y2")) `if` (
            ("X1" `=!=` "X2") and // infix operator
            ("Y1" `=!=` "Y2") and
            (("Y2" - "Y1") `=!=` ("X2" - "X1")) and
            (("Y2" - "Y1") `=!=` ("X1" - "X2")) // arithmetic expr
          )
      },
      fact { "no_attack_all"(`_`, emptyList) },
      rule {
        "no_attack_all"("C", consOf("H", "Hs")) `if` (
            "no_attack"("C", "H") and
            "no_attack_all"("C", "Hs")
          )
      },
      fact { "solution"(`_`, emptyList) },
      rule {
        "solution"("N", consOf(("X" and "Y"), "Cs")) `if` (
            "solution"("N", "Cs") and
            between(1, "N", "Y") and // built-in predicate
            "no_attack_all"(("X" and "Y"), "Cs")
          )
      }
    )
    return solve("solution"(n, (1 .. n).map { it and "Y$it" }))
}
```

This implementation produces a *lazy* stream of solutions to the N-Queens problem, given a particular value of n. In doing so, it takes advantages not only of logic- but also of functional-programming paradigm. For instance, on the last line, it exploits the `map` high-order function to build a list in the form $[(1, Y_1), \ldots, (n, Y_n)]$, to be provided as argument of `solution/2`.

A number of minutiæ may be noted as well by comparing the Prolog code with its Kotlin counterpart. For instance, Prolog operators (such as `=\=/2`, `-/2`, etc.) retain their infix notations in the Prolog DSL. This is possible because they are part of the DSL, in the same way as Prolog built-in predicates (such as `between/3`). This implies the Kotlin compiler can prevent standard operators and built-in from being silently mistyped.

5. Conclusions and future works

In this paper we propose a novel way of integrating the logic, object-oriented, functional, and imperative programming paradigms into a single language, namely Kotlin. More precisely, we describe how a domain-specific language (DSL) for Prolog can be built on top of the Kotlin language by exploiting the 2P-Kt library. The proposed solution extends the Kotlin language with LP facilities by only relying on its own mechanisms – therefore no external tool is necessary

apart from Kotlin itself and 2P-Kt –, even if, however, analogous extensions can in principle be constructed for other high-level languages as well—such as Scala.

Our DSL is currently implemented as part of the 2P-Kt project – namely, within the `dsl-*` modules –, and it is hosted on both GitHub [8] and Maven Central. Other than being available to the public for general purpose usage – under the terms of the Apache License 2.0[7] open-source license –, the DSL is currently extensively exploited to implement the unit tests of 2P-Kt itself.

While in this paper we discuss the design rationale and architecture of our DSL by explicitly focusing on Kotlin as our target technology, in the future we plan to generalise our approach, possibly tending to technology independence. Furthermore, we plan to provide other implementations of our DSL, targeting other high-level languages and platforms, in order to make logic programming and its valuable features available in general-purpose programming languages and frameworks.

As concerns possible research directions and applications, in the future, we plan to exploit our DSL in several contexts. For instance, we argue our DSL may ease the usage of the logic tuple spaces offered by the TuSoW technology [26]. Similarly, we believe our DSL may be used as a basic brick in the creation of logic-based technologies for MAS, as the MAS community is eager of general-purpose, logic-based technologies targetting the JVM platform [27]. Along this line, we argue logic-based languages for MAS may benefit from the integration of our DSL – or a similar one – to support the reasoning capabilities of agents. Finally, we plan to exploit our DSL within the scope of XAI [28], to ease the creation of hybrid (i.e., logic + machine-learning) systems, and management on the symbolic side.

Acknowledgments

R. Calegari has been supported by the H2020 ERC Project "CompuLaw" (G.A. 833647). A. Omicini has been partially supported by the H2020 Project "AI4EU" (G.A. 825619).

References

[1] K. R. Apt, The Logic Programming Paradigm and Prolog, Cambridge University Press, 2001, pp. 475–508. doi:10.1017/CBO9780511804175.016.

[2] R. A. Kowalski, Predicate logic as programming language, in: J. L. Rosenfeld (Ed.), Information Processing, Proceedings of the 6th IFIP Congress, North-Holland, 1974, pp. 569–574.

[3] R. Calegari, G. Ciatto, E. Denti, A. Omicini, Logic-based technologies for intelligent systems: State of the art and perspectives, Information 11 (2020) 1–29. doi:10.3390/info11030167, special Issue "10th Anniversary of Information—Emerging Research Challenges".

[4] A. Omicini, R. Calegari, Injecting (micro)intelligence in the IoT: Logic-based approaches for (M)MAS, in: D. Lin, T. Ishida, F. Zambonelli, I. Noda (Eds.), Massively Multi-Agent Systems II, volume 11422 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 21–

[7] https://www.apache.org/licenses/LICENSE-2.0

35. doi:10.1007/978-3-030-20937-7_2, international Workshop, MMAS 2018, Stockholm, Sweden, July 14, 2018, Revised Selected Papers.

[5] A. Colmerauer, P. Roussel, The birth of prolog, in: J. A. N. Lee, J. E. Sammet (Eds.), History of Programming Languages Conference (HOPL-II), ACM, 1993, pp. 37–52. doi:10.1145/154766.155362.

[6] R. Bagnara, M. Carro, Foreign language interfaces for Prolog: A terse survey, ALP Newsletter 15 (2002). URL: https://dtai.cs.kuleuven.be/projects/ALP/newsletter/may02/index.html.

[7] E. Denti, A. Omicini, A. Ricci, Multi-paradigm Java-Prolog integration in tuprolog, Science of Computer Programming 57 (2005) 217–250. doi:10.1016/j.scico.2005.02.001.

[8] 2P-Kt, Github page, https://github.com/tuProlog/2p-kt, Last access: October 22, 2020.

[9] tuProlog, home page, http://tuprolog.unibo.it, Last access: October 22, 2020.

[10] E. Denti, A. Omicini, A. Ricci, tuProlog: A light-weight Prolog for Internet applications and infrastructures, in: I. Ramakrishnan (Ed.), Practical Aspects of Declarative Languages, volume 1990 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2001, pp. 184–198. doi:10.1007/3-540-45241-9_13, 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

[11] J. W. Lloyd (Ed.), Computational logic, Springer, 1990.

[12] A. Nerode, G. Metakides, Principles of Logic and Logic Programming, Elsevier Science Inc., USA, 1996.

[13] A. Martelli, U. Montanari, An efficient unification algorithm, ACM Transactions on Programming Languages and Systems 4 (1982) 258–282. doi:10.1145/357162.357169.

[14] K. L. Clark, Negation as failure, in: H. Gallaire, J. Minker (Eds.), Symposium on Logic and Data Bases, Advances in Data Base Theory, Plemum Press, New York, 1977, pp. 293–322. doi:10.1007/978-1-4684-3384-5_11.

[15] BProlog, home page, http://www.picat-lang.org/bprolog, Last access: October 22, 2020.

[16] Ciao! Prolog, home page, https://ciao-lang.org, Last access: October 22, 2020.

[17] ECLiPSe Prolog, home page, https://eclipseclp.org, Last access: October 22, 2020.

[18] SICStus Prolog, home page, https://sicstus.sics.se, Last access: October 22, 2020.

[19] SWI Prolog, home page, https://www.swi-prolog.org, Last access: October 22, 2020.

[20] τProlog, home page, http://tau-prolog.org, Last access: October 22, 2020.

[21] XSB Prolog, home page, http://xsb.sourceforge.net, Last access: October 22, 2020.

[22] R. Calegari, E. Denti, S. Mariani, A. Omicini, Logic programming as a service, Theory and Practice of Logic Programming 18 (2018) 846–873. doi:10.1017/S1471068418000364, special Issue "Past and Present (and Future) of Parallel and Distributed Computation in (Constraint) Logic Programming".

[23] F. Bergenti, G. Caire, S. Monica, A. Poggi, The first twenty years of agent-based software development with JADE, Autonomous Agents and Multi Agent Systems 34 (2020) 36. doi:10.1007/s10458-020-09460-z.

[24] B. C. Oliveira, A. Moors, M. Odersky, Type classes as objects and implicits, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 341–360. doi:10.1145/1869459.1869489.

[25] E. Gamma, R. Helm, R. E. Johnson, J. Vlissides, Design Patterns: Elements of Reusable

Object-Oriented Software, Addison-Wesley Professional Computing Series, Addison-Wesley, Reading, MA, 1995. URL: https://www.safaribooksonline.com/library/view/design-patterns-elements/0201633612/.

[26] G. Ciatto, L. Rizzato, A. Omicini, S. Mariani, TuSoW: Tuple spaces for edge computing, in: The 28th International Conference on Computer Communications and Networks (ICCCN 2019), IEEE, Valencia, Spain, 2019, pp. 1–6. doi:10.1109/ICCCN.2019.8846916.

[27] R. Calegari, G. Ciatto, V. Mascardi, A. Omicini, Logic-based technologies for multi-agent systems: A systematic literature review, Autonomous Agents and Multi-Agent Systems 35 (2021) 1:1–1:67. doi:10.1007/s10458-020-09478-3, collection "Current Trends in Research on Software Agents and Agent-Based Software Development".

[28] R. Calegari, G. Ciatto, A. Omicini, On the integration of symbolic and sub-symbolic techniques for XAI: A survey, Intelligenza Artificiale 14 (2020) 7–32. doi:10.3233/IA-190036.