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

Giovanni Ciatto* Roberta Calegari° Enrico Siboni† Enrico Denti‡ Andrea Omicini⋆

*‡⋆Dipartimento di Informatica – Scienza e Ingegneria (DISI)
°Alma Mater Research Institute for Human-Centered Artificial Intelligence
Alma Mater Studiorum—Università di Bologna, Italy
{giovanni.ciatto, roberta.calegari, enrico.denti, andrea.omicini}@unibo.it

†University of Applied Sciences and Arts of Western Switzerland (HES-SO)
enrico.siboni@hevs.ch

21st Workshop "From Objects to Agents" (WOA)
Sept. 16, 2020, Bologna, Italy

Next in Line. . .

Context

AI side

- AI is shining, brighter than ever
  - mostly thanks to the advances in ML and sub-symbolic AI

⇒ symbolic AI is gaining momentum because of XAI

  ! hybrid solution mixing logic & data-driven AI are flourishing [3]

MAS side

The MAS community is eager for logic-based technologies [2]

- to support agents' knowledge representation, reasoning, or execution
- or to prove MAS properties

! despite few mature tech exist, and even fewer are actively maintained

Motivation

The problem with logic-based technologies

There is technological barrier slowing

- the adoption of logic programming (LP) as paradigm
- the exploitation of logic-based technologies

while programming *in the large*

- $\overbrace{\text{mainstream programming languages}}^{\textit{e.g. Scala, Kotlin, Python, C\#}}$ are blending several paradigms

  e.g. imperative, object-oriented (OOP), and functional programming (FP)

  - except LP!

- $\underbrace{\text{mainstream platforms}}_{\textit{e.g. JVM, .NET, JS, Python}}$ are poorly interoperable with logic-based tech.

Motivating example – SWI-Prolog's FLI for Java

- Prolog [4] implementors rely on Foreign Language Interfaces (FLI) [1]
  - (mostly targetting Java, or C)
- For instance, SWI-Prolog comes with a FLI for Java[1]:

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

→ No paradigm harmonization between Prolog and $\underbrace{\text{the hosting language}}_{\textit{i.e. Java}}$

[1] https://jpl7.org

Contribution of the paper

- Show that OOP, FP, and LP can be blended into a single language
- Propose a DSL blending Kotlin (OOP + FP) and Prolog (LP)
- Pave the way to the creation of similar DSL in mainstream languages

Next in Line...

Focus on. . .

Whet your appetite

Our Kotlin DSL for Prolog vs. actual Prolog

```
prolog {
  staticKb(
    rule {
      "ancestor"("X", "Y") `if` "parent"("X", "Y")     // ancestor(X, Y) :- parent(X, Y).
    },                                                  //
    rule {                                              //
      "ancestor"("X", "Y") `if` (                       // ancestor(X, Y) :-
        "parent"("X", "Z") and "ancestor"("Z", "Y")     //     parent(X, Z), ancestor(Z, Y).
      )                                                 //
    },                                                  //
    fact { "parent"("abraham", "isaac") },              // parent(abraham, isaac).
    fact { "parent"("isaac", "jacob") },                // parent(isaac, jacob).
    fact { "parent"("jacob", "joseph") }                // parent(jacob, joseph).
  )                                                     //
                                                        //
  for (sol in solve("ancestor"("abraham", "D")))        // ?- ancestor(abraham, D),
    if (sol is Solution.Yes)                            //        write(D), nl.
      println(sol.substitution["D"])
}
```

! try it here: https://github.com/tuProlog/prolog-dsl-example

Focus on. . .

Design Principles

P_1 – The DSL strictly extends the hosting language

→ no feature of the hosting language is forbidden within the DSL

P_2 – The DSL is interoperable with hosting language

→ all features of the hosting language are allowed within the DSL

→ LP is *harmonised* with the hosting language paradigm(s)

P_3 – The DSL is well encapsulated within the hosting language

→ i.e. only usable within well-identifiable sections

P_4 – The DSL is as close as possible to Prolog

→ both syntactically & semantically

Focus on. . .

Functioning in the Kotlin case I

The DSL is only enabled within `prolog {` $\langle$*DSL block*$\rangle$ `}` expressions

Expressions of the form: `"functor"(`$\langle e_1 \rangle$`,` $\langle e_2 \rangle$`, ...)`

are interpreted as terms: `functor`$(t_1, t_2, \ldots)$

provided that $\forall i : \langle e_i \rangle$ can be converted into t_i

Expressions of the form:

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

are interpreted as rules: `head`$(t_1, \ldots, t_N)$ `:-` $t_{N+1}, \ldots, t_M$

provided that $\forall i : \langle e_i \rangle$ can be converted into t_i

- similar syntax for facts

Functioning in the Kotlin case II

Within `prolog { ... }` blocks

- `staticKb(`*Clause*$_1$`, ` *Clause*$_2$`, ...)` sets up the local static KB
- `dynamicKb(`*Clause*$_1$`, ` *Clause*$_2$`, ...)` sets up the local dynamic KB
- `solve(`*Query*`, ` *Timeout*`)` returns a *lazy stream* of solutions
- `assert(`*Clause*`)` appends a new clause to the local dynamic KB
- ...

Kotlin to Prolog conversions

Kotlin	Prolog
lowercase string	atom
uppercase string	variable
int, long, short, byte	integer
double, float	real
boolean	atom
list, array, iterable	list

Next in Line...

Recipe for a Prolog-like DSL

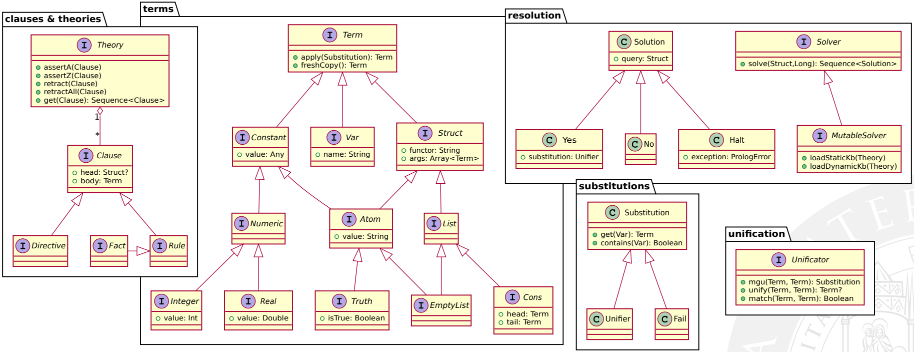
1. A language with a flexible API
   e.g. Kotlin, Scala, Python, Groovy, etc.

2. Full fledged API for Prolog, supporting that language
   e.g. 2P-Kt ! see next slide

3. Exploit flexibility to hide the exploitation of the API

2P-Kt – Overview

Comprehensive, modular, re-usable API covering most aspects of LP:



- runs on several platforms (e.g., JVM, NodeJS, Browsers, Android)
- more info here: `https://github.com/tuProlog/2p-kt`

Kotlin mechanisms for DSL I

1. Operator overloading

```
interface Term {
  operator fun plus(other: Term): Struct =
    Struct.of("+", this, other)
}

// now one can write:
val term3: Term = term1 + term2
```

Kotlin mechanisms for DSL II

2. Block-like lambda expressions

```
solutions.filter({ it -> it is Solution.Yes })
         .map({ it -> it.substitution["X"] })
         .joinToString(" ", { it.toString() })

 // ↑ can be rewritten as ↓

 solutions.filter { it is Solution.Yes }
          .map { it.substitution["X"] }
          .joinToString(" ") { it.toString() }
```

Kotlin mechanisms for DSL III

3. Extension methods

```
fun Any.toTerm(): Term = // converts Kotlin objects into terms

operator fun String.invoke(vararg args: Term): Struct =
    Struct.of(this, args.map { it.toTerm() })

// now one can write
"member"("X", arrayOf(1 .. 3)) // member(X, [1, 2, 3])
```

Kotlin mechanisms for DSL IV

4. Function types/lambdas with receivers

```
class PrologScope {
  fun Any.toTerm(): Term = // ...
  operator fun String.invoke(vararg args: Term): Struct = // ...

  operator fun Any.plus(other: Any): Struct =
    this.toTerm() + other.toTerm()
}

// type with receiver  ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
fun <R> prolog(action: PrologScope.() -> R): R =
    PrologScope().action()

//      ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓  lambda with receiver
prolog { "f"("1") + "f"("2") } // in braces this is PrologScope
```

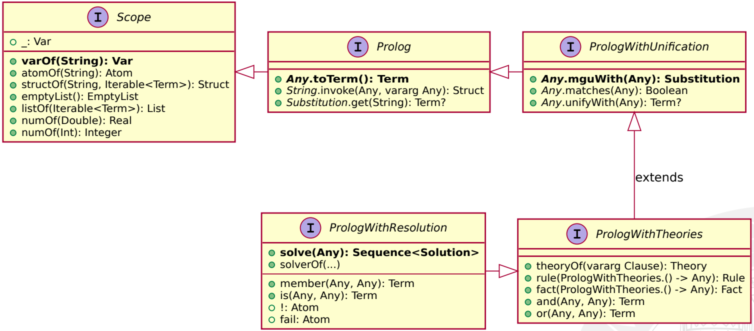
Kotlin mechanisms for DSL V

This is a Kotlin-specific discussion!

Other languages may support DSL through different mechanisms

e.g. implicits in Scala

DSL design on top of 2P-Kt



- Onion design with incremental features
- Built on top of 2P-Kt and Kotlin

Next in Line...

Conclusions & future works

Summing up, in this study we...

- argued LP should be integrated in modern languages/paradigms
- designed an in-language, DSL-based solution
- prototyped an actual Kotlin-based DSL for Prolog

In the future, we will try to...

- design Prolog-like DSL for other languages (e.g. Scala)
- design an agent-oriented (possibly BDI?) DSL
- extend 2P-Kt to support other sorts of inference mechanisms
- design a logic-based API for sub-symbolic AI

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

Giovanni Ciatto[*] Roberta Calegari[°] Enrico Siboni[†]
Enrico Denti[‡] Andrea Omicini[⋆]

[*‡⋆]Dipartimento di Informatica – Scienza e Ingegneria (DISI)
[°]Alma Mater Research Institute for Human-Centered Artificial Intelligence
ALMA MATER STUDIORUM—Università di Bologna, Italy
{giovanni.ciatto, roberta.calegari, enrico.denti, andrea.omicini}@unibo.it

[†]University of Applied Sciences and Arts of Western Switzerland (HES-SO)
enrico.siboni@hevs.ch

21st Workshop "From Objects to Agents" (WOA)
Sept. 16, 2020, Bologna, Italy

References I

[1] Roberto Bagnara and Manuel Carro.
Foreign language interfaces for Prolog: A terse survey.
ALP Newsletter, 15(2), May 2002.

[2] Roberta Calegari, Giovanni Ciatto, Viviana Mascardi, and Andrea Omicini.
Logic-based technologies for multi-agent systems: A systematic literature review.
Autonomous Agents and Multi-Agent Systems, In press.
Special Issue “Current Trends in Research on Software Agents and Agent-Based Software Development”.

[3] Roberta Calegari, Giovanni Ciatto, and Andrea Omicini.
On the integration of symbolic and sub-symbolic techniques for XAI: A survey.
Intelligenza Artificiale, In press.

[4] Alain Colmerauer and Philippe Roussel.
The birth of prolog.
In John A. N. Lee and Jean E. Sammet, editors, *History of Programming Languages Conference (HOPL-II)*, pages 37–52. ACM, April 1993.