LogicKit: Bringing Logic Programming to Swift

Dimitri Racordon
University of Geneva
Centre Universitaire d'Informatique
Switzerland
dimitri.racordon@unige.ch

Didier Buchs
University of Geneva
Centre Universitaire d'Informatique
Switzerland
didier.buchs@unige.ch

ABSTRACT

A new trend in programming languages is to merge multiple paradigms, rather than focusing on one as it was customary in the past. Most modern languages provide native support for imperative and functional programming, object-orientation and even concurrency. The typical mechanism used to blend heterogeneous language concepts is to rely on functions. Functions are a very well understood, usually offer an excellent abstraction over diverging models of computation, but are unsuitable to interface logic programming, unfortunately. Embedding a full logic programming language into a host is also unsatisfactory, as it impedes the ability to make data flow from one paradigm to the other.

As an answer to these issues, we propose LogicKit, a library that aims to bridge the gap between logic programming and other traditional paradigms. LogicKit is a Prolog-inspired language that blends seamlessly into Swift. Predicates are first-class objects in the host language and data can flow in and out of a logic program, without the need for any data serialization/parsing. Our framework is distributed in the form of a Swift library that can be imported in any Swift project effortlessly. We elaborate on our motivation for developing LogicKit and present the library by the means of examples.

CCS CONCEPTS

• **Software and its engineering → Constraint and logic languages**; **Software libraries and repositories**.

KEYWORDS

logic programming, domain specific language, embedded language, multi-paradigm programming, swift, prolog

ACM Reference Format:
Dimitri Racordon and Didier Buchs. 2020. LogicKit: Bringing Logic Programming to Swift. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3397537.3399575

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03…$15.00
https://doi.org/10.1145/3397537.3399575

1 INTRODUCTION

The last decade has seen the birth of numerous new programming languages. Swift was first released to the public in 2014, Rust appeared in 2010 and Kotlin in 2011, just to cite a few. A key feature of these new languages is that they merge multiple programming paradigms, rather than focusing on a single one, as it was once customary. This trend is obviously positive. Traditional "old-style" languages such as Haskell (functional), Java (object-oriented) or Prolog (logic) tend to force developers' to use ill-suited abstractions for problems that do not perfectly fit the offered paradigm. In contrast, in a modern multi-paradigm language, one may leverage object-orientation to classify data and hide implementation details, while at the same time using functional reactive programming [6] to handle graphical interfaces. It appears that this versatility is so seducing that even older languages are now merging new paradigms as well [4, 8].

Although object-orientation, functional programming and to a slightly lesser extent concurrency are usually taken for granted, logic programming is very rarely listed in the features of these new multi-paradigm languages. Logic programming [5] is a paradigm in which a program is written as a description of the problem to solve, rather than as a description of the steps to solve it. The beauty of logic programming is that one no longer has to tell *how* to compute a result, but only to describe its constraints. Hence, this approach is particularly useful to quickly prototype and implement backtracking search algorithms (e.g. a type solver).

Sadly, the lack of support for logic programming in contemporary language leaves this paradigm mostly unknown to mainstream developers. Besides, interoperability between logic programming languages and other languages is also challenging. The traditional approach to interoperability is to rely on a foreign function interface (FFI) as the bridging mechanism. A FFI allows one to seamlessly call a function written in another language, and even potentially evaluated by a different runtime. While this technique may pose some issues related to memory management, solutions to tackle them are well known [2]. Unfortunately, FFIs are usually not suitable to plug a logic programming language into a traditional one, because they do not share the same concept of function. One does not call a function in a logic program, but rather queries the system to check the satisfiability of some formula. Furthermore, the representation of a term in the former has typically no direct equivalent in the latter. In other words, exchanging data between the two languages often requires a sizable adaptation effort.

This short paper introduces LogicKit, a library that aims to bridge the gap between logic programming and the other paradigms offered by the Swift programming language. LogicKit is a Prolog-inspired language that blends seamlessly into Swift. Its main implementation challenge is to offer a seamless experience between the two languages. Therefore, it does not attempt to simply bring a Prolog interpreter into Swift (as JIProlog [1] or Tau Prolog [9], for instance). Instead, data can be freely exchanged and referred to from one language to the other, including predicates. Furthermore, in an effort to maximize its usability, LogicKit is provided as a pure Swift library, so that it can be imported just as any other regular library. It is fully compatible with an out-of-the-box Swift compiler, and does not necessitate the installation of an external tool or library. The remainder of this paper elaborates on our motivations and design decisions. LogicKit's sources are distributed under the MIT License and are available on GitHub (https://github.com/kyouko-taiga/LogicKit).

2 MOTIVATIONS

A logic program is a set of relations that describe facts and rules about the problem to solve, in the form of logical statements. Consider for instance the following Prolog program:

```
1  add(zero, Y, Y).
2  add(succ(X), Y, Z) :- add(X, succ(Y), Z).
```

This program describes the addition on natural numbers, using Peano arithmetic. It states that adding zero to a number `Y` is equal to `Y`, and that if one can conclude that adding the successor of `X` to a number `Y` is equal to `Z`, then one can also conclude that adding `X` to the successor of `Y` is equal to `Z`. The program only describes our knowledge. In logic programming, computation is expressed through queries over this knowledge. For instance, one could ask all pairs of operands whose sum is two with the following query:

```
1  add(X, Y, succ(succ(zero))).
```

A query is a logical statement. In the above, `X` and `Y` are logical variables, for which a Prolog interpreter will search values that satisfy the statement. Since there is not a single answer to this query, one can iterate over each valid answer, and thus retrieve the set of pairs whose sum is two.

The difficulty of marrying this programming style to more traditional paradigms stems from the mismatch between queries and functions. A query is neither a function nor a function call, but rather the description of a goal that an interpreter has to satisfy. Hence, FFIs are de facto disqualified.

The most straightforward way to use a logic program within another environment is to wrap an interpreter. The developer writes a logic program as a separate file, or directly within the host language, typically as a character string, or using more elaborate techniques such as language boxes [3]. This program and the queries thereupon are sent to the interpreter, which sends results back to the host language. Although simple, this approach compels the developer to deal with two different syntaxes, increasing her cognitive overhead. More importantly, it does not provide an obvious way to exchange data from one language to the other. This usually results in a high implementation cost to write user-friendly adapters to generate inputs for the interpreter and parse its outputs, which naturally comes with the usual challenges of data serialization. The following Javascript program illustrates an example, using the Tau Prolog:

```
1  let session = pl.create()
2  session.consult(
3    "add(zero, Y, Y)." +
4    "add(succ(X), Y, Z) :- add(X, succ(Y), Z).")
5  session.query("add(X, Y, succ(succ(zero))).")
6  let subst = null
7  session.answer((s) => { subst = s })
8
9  console.log(subst.links["X"])
10 // Prints "Term { id: "zero", ..."
11 console.log(subst.links["Y"])
12 // Prints "Term { id: "succ", ..."
```

The logic program is expressed in the form of a simple character string, which does not offer any convenient way to manipulate the relations. Tau Prolog returns query results in the form of a substitution table that maps logic variables onto terms, which is provided as an argument to the callback given to `session.answer`. Terms are automatically parsed into Javascript objects, alleviating the burden of extracting results from the interpreter's answer. Nonetheless, the structure and content of these terms depends entirely on the logic program, therefore providing little support to manipulate data defined within the host language[1].

3 LOGICKIT

This section introduces LogicKit through a series of examples. For spatial reasons, we focus only on the core features of the library. A more comprehensive introductory tutorial, as well as the user manual, are available on GitHub.

Like Prolog, LogicKit revolves around a knowledge base (or database) of relations that can be queried to "prove" a given statement. Three constructs are provided:

- *facts*, that denote axioms,
- *rules*, that denote logical deductions, and
- *literals*, that denote atomic values.

A knowledge base is merely a collection of such constructs. Consider for instance the following program:

```
1  enum Bird { case ostrich, pelican, swift }
2
3  let heavier = "heavier"/2
4  let kb: KnowledgeBase = [
5    heavier(Bird.ostrich, Bird.pelican),
6    heavier(Bird.pelican, Bird.swift),
7  ]
```

This program declares a knowledge base made of facts and literals which specifies a weight relation between birds. A predicate `heavier` is declared at line 3, and defined with two facts at lines 5 and 6. We borrow from Prolog's syntax, so that *name*/*n* designates a predicate *name* with an arity of *n*. Both facts refer to literals. Notice that these literals are built from Swift values, and not from a

[1] The framework does provide specific predicates to interact with Javascript though callbacks. While this allows data to flow from one language to the other, we argue that this approach is inconvenient.

textual representation. In fact, a literal can be built from a value of any Swift type for which an equality relation is defined.

The knowledge base can be queried as follows:

```
let answers = kb.ask(
  heavier(Bird.ostrich, Bird.pelican))
print(answers.next() != nil)
// Prints "true"
```

The above query only checks a fact, which is indeed defined in the knowledge base. Note that the `ask` method does not returns a *set* of answers. Although it is not the case in the above example, the reason is that a query may have multiple answers, which are computed lazily. If the answer set is empty, then the query is unsatisfiable.

One can obtain more interesting results by using logic variables and let LogicKit find the proper values to satisfy a query. For instance, the following query asks for the bird that is heavier than a swift:

```
let x: Term = .var("x")
let answers = kb.ask(heavier(x, Bird.swift))
print(answers.next()!)
// Prints "["x": pelican]"
```

We start by creating a logic variable `x`, which is then used as arguments to the predicate `heavier/2` to create a query. The set returned by the method `ask` contains a single answer, given in the form of a dictionary-like data structure mapping the logical variable `x` to its corresponding value.

The true power of logic programming comes from the ability to deduct answers from a given system. In the above example, only one result is returned, although one could easily deduce that if an ostrich is heavier than a pelican, then it is also heavier than a swift. Fortunately, we can enrich the knowledge base to allow LogicKit to make such deductions. Consider for the following program:

```
let heavier = "heavier"/2
let faster  = "faster"/2
let a: Term = .var("a")
let b: Term = .var("b")
let c: Term = .var("c")
let kb: KnowledgeBase = [
  heavier(Bird.ostrich, Bird.pelican),
  heavier(Bird.pelican, Bird.swift),
  heavier(a, c)
    |- heavier(a, b) && heavier(b, c),
  faster(a, b) |- heavier(b, a),
]

let x: Term = .var("x")
let answers = kb.ask(faster(x, Bird.ostrich))
```

Two rules are added, respectively defining the transitivity of the `heavier/2` predicate, as well as a `faster/2`, based on the assumption that the flying speed of a bird is inversely proportional to its weight[2]. Thus, running the query `faster(x, Bird.ostrich)` will yield two results, indicating that both the pelican and the swift are faster than the ostrich.

[2]We kindly ask bird enthusiasts to forgive such a dubious assumption.

Recall that all constructions are first-class Swift objects, which can be created and manipulated as any other value. Hence, one may very easily create a knowledge base from some data available to the host language, for instance obtained from a distant server:

```
let movie = "movie"/3
server.fetch("/movies/") { data in
  let k = KnowledgeBase(
    knowledge: data.map { m in
      movie(m.name, m.duration, m.rating)
    })
}
```

The above example retrieves a collection of movie data from a fictive API and builds a knowledge base containing information extracted from the fictive API's response.

As knowledge bases are also first-class Swift objects, one can easily insert and remove relations from them, effectively reproducing the behavior of Prolog's support for metaprogramming. We argue that this approach is even cleaner, as it is easier to debug. Since the modification of the knowledge base is expressed within the host language, one can leverage the latter's debugging tools to observe all mutations.

Another way to create interactions between the logic program and the host language is by leveraging literal values. Internally, LogicKit represents predicates as terms. The name of the predicate corresponds to the name of a term, while arguments are encoded as subterms. This representation lets LogicKit's resolution engine use unification [7] to determine the value of each logic variable. However, representing all data structures and operations thereupon as predicates can be cumbersome, in particular if such data structures are to be later manipulated by the host language in a more traditional way. LogicKit provides a built-in predicate `satisfies/1` to address this issue. This predicate accepts a regular Swift function representing the predicate's semantics. Unlike other predicates, `satisfies/1` cannot be used to infer the value of a logic variable, but it can be used to check if it satisfies a predicate written in pure Swift. Recall that literals can be built from any value whose type defines an equality relation. Hence, a common use-case is to use `satisfies/1` to check properties that are not defined as a logical relation on a literal. Consider for instance the following example:

```
protocol AST {}
struct Id: AST, Hashable {
  let name: String
}

let isAST = "isAST"/1
let a: Term = .var("a")
let kb: KnowledgeBase = [
  isAST(a) |- satisfies { t in
    t.unwrap() is AST
  }
]

let answers = kb.ask(isAST(Id(name: "foo")))
```

In the above program, the predicate `isAST/1` holds if and only if its argument's type conforms to the protocol `AST`.

4 CONCLUSION

We have presented LogicKit, a Swift library that aims to bridge the gap between logic programming and other traditional paradigms. Rather than embedding an interpreter for a logic programming language into Swift, LogicKit provides the tools to write a logic program that blends seamlessly with its host language. Consequently, the host environment does not have to rely on data serialization to interact with an interpreter. Instead, data can flow effortlessly in and out of a logic program. We have introduced LogicKit and its features by the means of simple examples. We purposely used a syntax reminiscent to Prolog to demonstrate that our library allows to write clean logic programs. Nonetheless, all constructions are first-class Swift object, which can be manipulated as any other value by the host language.

Perspective for future work include a better support for static type checking. LogicKit has been developed with Prolog in mind, which is itself a dynamic language. However, Swift offers a powerful type system that could be leveraged to write safer programs and remove the need for most of the runtime type checks that the library currently performs. In particular, typing predicate arguments and logic variables could provide an excellent tool to detect typos and other careless mistakes. Other improvements relate to the library's expressiveness. LogicKit provides far less so-called built-in predicates than a full-fledged logic programming language. As discussed in Section 3, some might not be necessary or even desired. Nonetheless, the addition of some support to control backtracking (e.g. Prolog's cut) could simplify the expression of problems more easily expressed with negation.

LogicKit is currently used by Bachelor students, in the context of a course on language semantics. Its sources are distributed under the MIT License, and are available on GitHub (https://github.com/kyouko-taiga/LogicKit).

REFERENCES

[1] Ugo Chirico. 2015 (accessed December 1, 2019). *JIProlog*. JIProlog. http://www.jiprolog.com

[2] Marcus Crestani. 2010. *Foreign-Function Interfaces for Garbage-Collected Programming Languages.* Technical Report. Eberhard-Karls-Universität.

[3] Lukas Diekmann and Laurence Tratt. 2019. Default disambiguation for online parsers. In *International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira (Eds.). ACM, New York, 88–99. https://doi.org/10.1145/3357766.3359530

[4] Jaakko Järvi and John Freeman. 2008. Lambda functions for C++0x. In *Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, Roger L. Wainwright and Hisham Haddad (Eds.). ACM, New York, NY, USA, 178–183. https://doi.org/10.1145/1363686.1363735

[5] John W. Lloyd. 1987. *Foundations of Logic Programming, 2nd Edition.* Springer, Berlin, Germany. https://doi.org/10.1007/978-3-642-83189-8

[6] Fatih Nayebi. 2017. *Swift Functional Programming* (2nd ed.). Packt Publishing, Birmingham, UK. https://books.google.ch/books?id=70EwDwAAQBAJ

[7] Jörg H. Siekmann. 1989. Unification Theory. *Journal of Symolic Computation* 7, 3/4 (1989), 207–274. https://doi.org/10.1016/S0747-7171(89)80012-4

[8] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. 2014. *Java 8 in Action: Lambdas, Streams, and Functional-style Programming* (1st ed.). Manning Publications Co., Greenwich, CT, USA.

[9] José Antonio Riaza Valverde. 2019 (accessed December 1, 2019). *Tau Prolog*. JIProlog. http://tau-prolog.org