Polylogic

Mikkel Kragh Mathiesen

University of Copenhagen, Copenhagen, Denmark
mkm@di.ku.dk

Abstract

Negation in logic programming is often awkward, requiring workarounds like stratified negation or negation by failure. We propose *polylogic* as an alternative that supports general negation while respecting solution multiplicities. The theory has an algebraic semantics in terms of *De Morgan spaces*, a new lattice-like structure over a module. Despite this significant increase in power, polylogic can be realised in any Prolog system that supports constraint handling rules. In particular, we present a SWI-Prolog library which corresponds to the standard model of a De Morgan space.

Introduction The common treatment of negation in logic programming, exemplified by ISO Prolog, is *negation by failure*: $\neg P$ succeeds exactly once if a proof search for P finds no results, otherwise $\neg P$ fails. Even when succeeding no logical variables are instantiated and the answer contains no information about *how* $\neg P$ is true. This kind of negation is rather ill-behaved. It satisfies few of the equations that we normally expect from negation and is considered *impure* by virtue of being able to, for example, check whether a logical variable is instantiated.

Other solutions like *stratified negation*[1] provide a more coherent treatment, but places strict requirements on its use and is typically found in very specialised logic languages like Datalog.

We propose *polylogic* as a general approach to negation that properly tracks the evidence involved in deducing negative conclusions. Additionally, polylogic provides a framework for dealing efficiently with multiplicities, negated or otherwise.

Polylogic Consider a standard Prolog-like logic with 0 (false), $\top$ (true), $+$ (disjunction) and $\wedge$ (conjunction). We use 0 and $+$ rather than $\bot$ and $\vee$ since disjunction and conjunction in Prolog behave more like a semiring than a lattice. Polylogic extends this logic with new constructs $\bot$, $\vee$, $\neg$ and $-$. The idea is that 0 and $+$ are not actually 'false' and 'disjunction', but 'no evidence' and 'addition of evidence' respectively. Truth is negated using $\neg$ while evidence is negated using $-$. Proper falsity and disjunction are now provided by $\bot$ and $\vee$.

Altogether polylogic has primitives 0, $\top$ and $\bot$; and operations $+$, $-$, $\wedge$, $\vee$ and $\neg$. A statement represents a truth value and some amount of evidence for that conclusion. A truth value consists of both examples and counter-examples to an assertion.

- $+$ combines evidence by taking both sides into account.
- $-$ turns evidence *for* into evidence *against* and vice versa.
- $\wedge$ combines truth, such that two examples yield an example and any other combination yields a counter-example.
- $\vee$ combines truth, such that two counter-examples yield a counter-example and any other combination yields an example.
- $\neg$ negates truth by turning examples into counter-examples and vice versa.

In particular:

- 0 is no evidence at all.
- $\top + \bot$ is both evidence for an example and a counter-example.
- $-\top$ is evidence against an example.
- $\top - \bot$ is evidence for an example and evidence against a counter-example.

Any natural number n can be interpreted as a proposition by taking the sum of n copies of $\top$. Negative integers are interpreted similarly, but using $-$ to negate the result. With this we can write $nP = n \wedge P$ for any integer n and proposition P to denote the n-fold sum of copies of P.

Examples As an example of why this is useful take the well-known conundrum in classical logic regarding the statement "all birds can fly". A penguin enters and a contradiction ensues. In polylogic it is safe to admit the fact "all birds can fly" as long as one separately asserts the fact "but penguins do not fly". Specifically we take the phrasing "but penguins do not fly" as evidence *for* penguins *not flying* and evidence *against* penguins *flying*. This last piece of evidence cancels out with the evidence from the general statement, leaving us only with evidence for penguins not flying.

Another example demonstrates how multiplicities are handled. Suppose we make a query $\mathbf{path}(a, b) \wedge \mathbf{path}(b, c)$ on a database containing $\mathbf{path}(a, b) + \neg\mathbf{path}(a, b) + 3\mathbf{path}(b, c)$. The answer is then:

$$\mathbf{path}(a, b) \wedge \mathbf{path}(b, c) = (\top + \bot) \wedge 3\top = 3(\top \wedge \top) + 3(\bot \wedge \top) = 3\bot + 3\bot = 6\bot$$

In this way we correctly count all the ways of deriving a contradiction.

De Morgan spaces We now introduce the algebraic structure into which polylogic is interpreted. Fix a ring R. A *De Morgan space*[1] comprises

- An R-module M
- A commutative monoid object $C = (M, \wedge, \top)$
- A commutative monoid object $D = (M, \vee, \bot)$.
- An isomorphism of monoids $\neg : C \cong D$.

If an element $x : M$ behaves classically in the sense that $x \wedge x = x \vee x$ we say that x is *consistent*. In this case we define $x^2 = x \wedge x = x \vee x$.

A De Morgan space must satisfy the following equations whenever both sides are defined.

- $\neg\neg x = x$
- $x^2 \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
- $x^2 \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$

Models For any distributive lattice L let $\mathrm{FM}(|L|)$ be the free module generated by elements of L. Extend the lattice operations linearly and the result is a De Morgan space. In particular, the initial lattice $\{\bot, \top\}$ generates what we shall call the 'standard model' of a De Morgan space. All elements in this model have the form $\alpha \cdot \bot + \beta \cdot \top$ where $\alpha, \beta : K$. A consistent element has either $\alpha = 0$ or $\beta = 0$. Intuitively, these are the elements that are not both true and false at the same time.

[1]A more apt name for this structure might be *linear lattice*, but that is unfortunately already used for an unrelated concept.

Not all models arise in this manner. Even the behaviour of the generators $\bot$ and $\top$ is to some degree up for discussion. By the axioms we have $\top \wedge \top = \top$, $\top \wedge \bot = \bot$ and $\bot \wedge \top = \bot$, but the value of $\bot \wedge \bot$ can be freely chosen. In the standard model we have $\bot \wedge \bot = \bot$. Other interesting choices include $\bot \wedge \bot = -\top$, where $\wedge$ behaves like multiplication of complex numbers, and $\bot \wedge \bot = 0$, where conjunction 'bottoms out'.

Implementation Polylogic can be implemented in Prolog with *constraint handling rules*[4]. The basic idea is to declare a constraint `scale`/2. An active constraint of the form `scale`(m, n) corresponds to the proposition $m\top + n\bot$. For instance, the query discussed earlier with the result $6\bot$ would be reported as `scale`$(0, 6)$. Note how using constraints to count multiplicities is not only more expressive, but also more efficient; no need to generate 'true' n times when `scale`$(n, 0)$ is available.

Our implementation realises the standard De Morgan algebra, so the simplification for conjunctions looks as follows:

```
scale(M1, N1), scale(M2, N2) <=> scale(M1 * M2, M1 * N2 + M2 * N1 + N1 * N2).
```

The implementation of $\neg$ is slightly complicated, but essentially $\neg P$ runs the goal P and then exchanges the two components of `scale`.

The primary export of the polylogic library is $\neg$. Other operations can be derived from it, such as disjunction $P \vee Q$ which is simply defined as $\neg(\neg P \wedge \neg Q)$.

Polylogic interacts well with the rest of Prolog. Code using only the pure fragment of Prolog works seamlessly, but will of course not produce negated results. A library like CLP($\mathbb{Z}$) has a rich enough interface to allow writing a thin wrapper that generates both examples and counter-examples. Aggregation predicates like `findall` will disregard constraints, so the polylogic library defines `poly_findall` which collects multiplicities and negations properly.

Related work Polylogic is closely related to the notion of *polyset* [3]. In fact, a predicate of one argument that uses polylogic constructs can be thought of a polyset. The algebraic semantics and the implementation techniques also generalise smoothly to polysets.

There exist a plethora of generalisations of lattices. They generally seek to maintain idempotency, whereas polylogic eschews idempotency in favour of linearity. Probably the most similar structure is a *bilattice* [2], operating with both truth and knowledge similar to our truth and evidence. There is no concept of negative knowledge, though, and negation therefore works differently.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1995.

[2] Melvin Fitting. Bilattices and the semantics of logic programming. *J. Log. Program.*, 11(2):91–116, June 1991.

[3] Fritz Henglein and Mikkel Kragh Mathiesen. Module theory and query processing (extended abstract). In *Proc. Mathematically Structured Functional Programming (MSFP)*, MSFP '20, 2020.

[4] Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. 05 2004.