The functional perspective on advanced logic programming

Alexander Vandenbroucke[1]

1 **KU Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium**
`alexander.vandenbroucke@kuleuven.be`

Abstract

The basics of logic programming, as embodied by Prolog, are generally well-known in the programming language community. However, more advanced techniques, such as tabling, answer subsumption and probabilistic logic programming fail to attract the attention of a larger audience. The cause for the community's seemingly limited interest lies with the presentation of these features: the literature frequently focuses on implementations and examples that do little to aid the understanding of non-experts in the field. The key point is that many of these advanced logic programming features can be characterised in more generally known, more accessible terms. In my research I try to reconcile these advanced concepts from logic programming (Tabling, Answer subsumption and probabilistic programming) with concepts from functional programming (effects, monads and applicative functors).

1998 ACM Subject Classification D.1.6 Logic Programming, D.3.1 Formal Definitions and Theory

Keywords and phrases Tabling, Answer Subsumption, Effect Handlers, Functional Programming, Logic Programming, Probabilistic Programming

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.20

1 Introduction

Logic programming is—or has the potential to be—one of the most declarative programming paradigms. In fact, the essentials of logic programming, are generally well-known in the programming language community, and almost every computer scientist has had some exposure to Prolog.

Unfortunately, more advanced features, or more recent advances in logic programming fail to attract the attention of a larger audience beyond the logic programming community. For instance, several Prolog systems, such as XSB [19], Yap [16], B-Prolog [26] and most recently SWI-Prolog [24], support a more advanced form of resolution, SLG-resolution, also called *tabling*. Sadly, this very useful technique is completely unfamiliar to most programming language researchers that are not active in logic programming. This sometimes leads to the technique being reinvented in some very specific setting, for example for parsing.

Similarly, probabilistic logic programming extends regular logical programming to the realm of probabilistic computation, while still retaining the basic logical semantics. For example, the ProbLog system [2] is a simple syntactic extension of Prolog, where Prolog clauses can be annotated with probabilities. Such a system admits declarative specification of many probabilistic problems. ProbLog additionally supports many powerful probabilistic inference modes. However, the larger probabilistic programming community remains ignorant of these features.

The cause for this apparently limited interest from the community lies with the presentation of these features: the literature frequently focuses on implementations and examples that do little to aid the understanding of non-experts in the field. The key point is that many of these advanced logic programming features can be characterised in more generally known, more accessible terms. For example, the behaviour of logic programs is often formalised by fixed points of functions. In particular, Van Emden's concise and elegant fixed point semantics for Prolog, is a prime example of this approach.

The benefits of adopting a more general, abstract presentation are mutual and twofold:

1. By recasting (advanced features of) logic programming in a more general light, a fair comparison with similar functional systems becomes possible.
   For instance, functional logic programming systems claim to be more expressive than their logical counterparts. In a general framework, objective verification of such claims is possible, and moreover cross-pollination can proceed in a natural way.
2. The functional programming community has amassed a wealth of techniques that deal with and use non-standard control-flow. These now become readily available to the logic programmer. Recent examples of this are *delimited control* [3] and *effect handlers* [7, 15].
   Here the benefits are clearly mutual: Delimited control is applied to capture tabling in a functional context. As a side-effect tabling is reduced to its essence, which in turn enables a very compact (logic programming) implementation.

Currently, my research focuses on two main areas: probabilistic (functional & logic) programming languages, and formalising tabling with answer subsumption for logic programs, which is a more advanced version of tabling.

2 Background

2.1 Probabilistic Programming Languages

Probabilities are an indispensable tool for dealing with uncertainty in real-world scenarios. They allow us to quantify missing information and thereby reason with incomplete knowledge. This key insight is the root of many advances in artificial intelligence: from machine learning and data mining, to natural language processing (NLP), information retrieval (IR) and automated reasoning. Traditionally, probabilistic models and their inference routines are tightly coupled in a single implementation, necessitating their re-implementation when the same inference technique is used for a different model. Universal probabilistic programming languages instead provide a generic platform to express probabilistic models and their inference routines. For example, consider a simple ProbLog program that models a fair coin:

```
coin(c).
0.5 :: heads(X) :- coin(X).
```

The result of the query `heads(X)` is a probability distribution which is true (with $X = c$) with probability 0.5.

Obviously, a single lingua franca for probabilistic programming enables much more efficient communication and reuse of algorithms. However, in practice the probabilistic programming landscape is highly fragmented due to the sheer number of incompatible probabilistic programming languages. Often these languages belong to completely different paradigms, from Object Oriented Programming (Microsoft's Infer.NET [11]); Logic Programming (ProbLog [2], PRISM [8]); Functional Programming (Church [5], Anglican [25]); and hybrid systems (Factorie [10], Figaro [13]).

Thus, while probabilistic programming was originally intended to unify AI-discourse on the subject, the lack of provisions for interoperability between the systems has only served to divide it further.

Clearly, what is needed is a single theory or framework that explains the relative capabilities of the different systems. When two systems are equivalent (that is, they possess the same capabilities), it should be possible to translate one system into the other and vice-versa.

Recently there has been much interest, from both functional and logical communities in using *monads* to model the semantics of probabilistic programming languages [4, 14]. Monads are a concept from category-theory, an abstract branch of mathematics. Initially, Moggi [12] proposed them as a way to structure compositional denotational semantics of programs. This compositionality has proven incredibly useful for implementing side-effects in pure functional programming languages such as Haskell [22]. Monads (and other similar category-theoretic structures) may be precisely the tool that is needed to unite the disparate branches of probabilistic programming.

2.2 Tabling with Answer Subsumption

2.2.1 Tabling

Tabling [23, 19] is a well-known and extensively studied extension of standard Prolog. The main benefit of tabling is that it brings the behaviour of many logic programs in line with their standard logical semantics. In more practical terms, it frees the Prolog programmer from worrying about more operational concerns such as clause and goal ordering. Additionally, it can dramatically speed-up the execution of a program, in exchange for higher memory consumption. Tabling has been implemented in various Prolog systems such as XSB [19], Yap [16], B-Prolog [26] and SWI-Prolog [24].

Consider the following program defining a graph containing three nodes arranged in a cycle. The edges are modelled by the `e/2`-predicate, while `p(X,Y)` holds if there is a path between `X` and `Y`.

```
:- table p/2.
e(1,2).
e(2,3).
e(3,1).

p(X,Y) :- p(X,Z),e(Z,Y).
p(X,Y) :- e(X,Y).
```

The `:-table p/2`-directive indicates that tabled resolution should be used when evaluating `p/2`. Under normal Prolog execution, the order of the clauses would cause an infinite loop, while with tabled execution the program produces all possible combinations and then *terminates*. Note that termination cannot be achieved with regular execution, even if we permute the program's clauses and bodies, since the graph contains a cycle. The technique is called tabling, because answers are stored in a data structure, called a *table* while the program is executed. In this fashion the Prolog system can keep track of the answers it has already seen.

2.2.2 Answer Subsumption and Tabling Modes

Some Prolog systems support an extension of tabling that we call *Answer subsumption*, using Swift and Warren's nomenclature [18], often implemented as a set of *tabling modes* [6].

Answer subsumption, specifies how answers should be aggregated in the table. Subsumption refers to the fact that the original answers are replaced by their aggregates, that is, they are subsumed.

Consider the following program where we use answer subsumption to compute the length of the shortest path in a graph.

```
:- table p(index,index,min).

e(1,2).
e(2,3).
e(3,1).

p(X,Y,1) :- e(X,Y).
p(X,Y,D) :- p(X,Z,D1),p(Z,Y,D2), D is D1 + D2.
```

The directive `:-table p(index,index,min)` specifies the tabling mode of each argument of the `p/3` predicate: the first two arguments serve as indexes into the table, while the final argument uses the `min` mode indicating that only the smallest answer must be retained. This means that if the table contains an answer `p(X,Y,D)` for any `X` and `Y` after the program has been executed, then `D` must be the length of the shortest path from `X` to `Y`. Instead of table modes, XSB uses lattice and partial order answer subsumption modes, which allow the user to specify an arbitrary predicate (subject to some mild conditions) to aggregate answers.

Using answer subsumption can yield very compact and efficient programs for optimisation problems, especially those that are instances of Dynamic Programming [6].

Unfortunately, none of the existing implementations that we are aware of are generally sound. Consider the following pure logic program:

```
p(0). p(1).
p(2) :- p(X), X = 1.
p(3) :- p(X), X = 0.
```

The query `?-p(X)` has the finite set of answers `p(0),p(1),p(2),p(3)`, the largest of which is `p(3)`. However, XSB, Yap and B-Prolog all yield different (invalid) solutions when answer subsumption is used to obtain the maximal value. Both XSB and B-Prolog yield `X = 2`, with a maximum aggregation and max table mode respectively. Yap (also with max table mode) yields `X = 0; X = 1; X = 2`, every solution except the right one.

The problem is exacerbated by the fact that none of the systems formally define the semantics of answer subsumption. In a recent ICLP paper [20], we try to resolve this issue by giving a formal semantics for answer subsumption. We then examine under which conditions the systems are sound according to this semantics. Please see Section 3.1.1 for a short overview.

3 Objectives

The research mostly proceeds along two tracks: (1) we investigate the connection between functional programming and tabling with and without answer subsumption; (2) we investigate probabilistic logic programming–as embodied by the ProbLog system–from the functional perspective, in order to develop a general semantics for probabilistic programming languages. The semantics of probabilistic programs directly depends on the least-fixed point semantics mentioned above. Tabling approximates these semantics, and therefore frequently appears as an aspect of these probabilistic programming languages.

3.1 Current Status of the Research and Preliminary Results

3.1.1 Tabling with Sound Answer Subsumption

As mentioned in Section 2, Answer Subsumption, while very useful in practice, lacks a formally defined semantics, which hampers the user's ability to reason about the behaviour of his or her programs. In fact, without a formal semantics, we are reduced to reasoning based on intuitive knowledge of the implementation of a particular system, which is distinctly unportable, and even less satisfying.

In very recent work [20], we have attempted to mitigate this problem by defining what we believe is an appropriate denotational semantics, based on least fixed points of monotone functions on complete lattices. A *complete lattice* is a partially ordered set (poset) $\langle L, \leq_L \rangle$ such that every $X \subseteq L$ has a least upper bound $\bigvee X$, i.e.:

$$\forall z \in L : \bigvee X \leq_L z \iff \forall x \in X : x \leq_L z$$

It is a well known result from lattice theory that least fixed points of monotone functions are guaranteed to exist. Our semantics is based on Van Emden's well known least fixed point semantics, which uses an immediate consequence operator $T_P : \mathcal{P}(H_P) \to \mathcal{P}(H_P)$, where H_P is the *Herbrand base*, the set of all ground atoms of a program P. Then the logical semantics (for definite programs, that is programs not containing negations) is given by its least fixed point, $\mathsf{lfp}(T_P)$.

In our work, we define a similar operator $\widehat{T}_P : \mathcal{P}(H_P) \to \mathcal{P}(H_P)$, that takes answer subsumption into account. We do so by showing that most tabling modes can be modelled by a semi-lattice L, with functions $\eta : H_P \to L$ and $\rho : L \to \mathcal{P}(H_P)$ to convert between ground atoms and L. The semantics of a tabled logic program using answer subsumption is then given by

$$\rho\left(\bigvee_{x \in \mathsf{lfp}(\widehat{T}_P)} \eta(x)\right)$$

That is, we take the least fixed point of $\widehat{T}_P$, then convert this least fixed point to the lattice L where we aggregate it, and finally convert this aggregate to a set of ground atoms. It is important to note that we assume that the program is *stratified*, and $\widehat{T}_P$ is operating on a single stratum. The full details are beyond the scope of this text.

Finally, note that the semantics we have specified differs in an important way from actual subsumption implementations: this semantics only aggregates and subsumes answers *after* the least fixed point has been computed, while implementations generally execute subsumption in lock-step with the derivation of new answers.

In the paper we prove a theorem that specifies when an implementation is sound, i.e. when the difference alluded to above, does not produce different answers. Using the theorem requires that a programmer proves certain properties about their program, which may be difficult for realistically sized programs. Nevertheless, we believe this is an important first step towards formalisation of answer subsumption.

3.1.2 Fixing Non-determinism

In a recent paper [21] we reduce tabling (with and without answer subsumption) to its functional essence. Two key elements remain: recursion and non-determinism. This has the advantage, for instance, that this presentation is not muddled by *answer variance*: Prolog systems must avoid adding an answer if there is already a *variant* of the answer in the table.

Most languages don't have unification (and therefore no notion of variance), thus answer variance is an irrelevant detail that can be ignored.

The contributions of this work are:

- We define a monadic model that captures both non-determinism and recursion. This yields a finite representation of recursive non-deterministic expressions. We use this representation as a light-weight (for the programmer) embedded Domain Specific Language to build non-deterministic expressions in Haskell.
- We give a denotational semantics of the model in terms of the least fixed point of a semantic function $\mathcal{R}[\![\cdot]\!]$. In fact, the semantics closely resembles the T_P operator mentioned previously. The semantics is subsequently implemented as a Haskell function that interprets the model.
- We generalise the denotational semantics to arbitrary complete lattices. We illustrate the added power on a simple graph problem, which could not be solved with the more specific semantics. This new semantics corresponds to tabling with answer subsumption.
- We provide a set of benchmarks to demonstrate the expressivity of our approach and evaluate the performance of our implementation.

3.2 Open Issues and Expected Achievements

3.2.1 Automatic Verification of Sound Answer Subsumption

In the paper mentioned in Section 3.1.1, we define a high-level semantics for answer subsumption based on lattice theory. Then we generalise it to establish a correctness condition indicating when it is safe to use (greedy) answer subsumption implemented by most tabling systems. We show several examples where the existing implementations of answer subsumption fail that condition and derive an erroneous result.

This condition is sufficient, but not necessary: there exist programs that do not satisfy the condition, for which the greedy strategy nevertheless delivers correct results. Since we have not run across any non-contrived examples of such programs, we believe that this apparent lack of necessity is an artefact of our rather coarse semantics, which we intend to refine in the future.

The verification of correctness constitutes a non-trivial effort. Hence, manually proving the correctness condition for realistically sized programs could be unfeasible in practice. Ideally we would have an automated analysis that warns the programmer if it fails to establish the correctness condition.

One promising avenue of research is the fact that the program needs to be stratified, and the correctness condition need only hold for the stratum under consideration.

Currently the stratification is also rather coarse. A more fine-grained stratification should significantly reduce the work involved in proving the correctness condition. For automation purposes, abstract interpretation [1] could be used to statically inspect a program, or if we relax our requirements, a dynamic approach could be taken, that warns the programmer that the obtained answers are unreliable during or after the execution of the program

3.2.2 Algebraic Structures for Probabilistic Programming

Within the functional programming community the use of monads for probabilistic programming is both pragmatic and more theoretical. On the one hand several people, e.g. Scibior et al. [17] have developed efficient monadic interfaces for well-known probabilistic inference algorithms. A functional programmer can then use these interfaces to model a probabilistic problem monadically. On the other, there has been much research towards a measure theoretic formalisation of such monadic probabilistic programs [14].

However, little has been done into other more general algebraic structures related to monads. In particular, so called *applicative functors* or *idioms* [9] appear to model precisely those programs where the structure of the program is static, with respect to probabilistic choices that are made. This is especially relevant for probabilistic logic systems such as ProbLog, where the structure of the clauses is fixed. Moreover, as these structures are more restrictive, they may actually admit faster inference routines. Or conversely, ProbLog's specialised inference may apply to probabilistic programming languages that have applicative structure. There are already some promising early results, for instance, it is cleary that ProbLog programs exhibit applicative structure. However, the implications of these results are not yet fully understood, and are subject of ongoing research.

Acknowledgements

I would like to thank the reviewers for taking the time to provide insightful comments and in-depth remarks.

References

1 Samson Abramsky and Chris Hankin. *Abstract Interpretation of declarative languages*, volume 1, chapter An introduction to abstract interpretation, pages 63–102. 1987.

2 Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, pages 2462–2467, 2007.

3 Benoit Desouter, Marko van Dooren, and Tom Schrijvers. Tabling as a library with delimited control. *TPLP*, 15(4-5):419–433, 2015.

4 Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *ICFP*, pages 2–14. ACM, 2011.

5 Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *UAI*, pages 220–229. AUAI Press, 2008.

6 Hai-Feng Guo and Gopal Gupta. Simplifying dynamic programming via tabling. In *PADL*, volume 3057 of *LNCS*, pages 163–177. Springer, 2004.

7 Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *Haskell*, pages 59–70. ACM, 2013.

8 Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806, pages 585–591. Springer, 2011.

9 Conor McBride and Ross Paterson. Applicative programming with effects. *JFP*, 18(1):1–13, 2008.

10 Andrew McCallum, Karl Schultz, and Sameer Singh. FACTORIE: probabilistic programming via imperatively defined factor graphs. In *NIPS*, pages 1249–1257. Curran Associates, Inc., 2009.

11 http://research.microsoft.com/en-us/um/cambridge/projects/infernet/ Microsoft Research. Infer.NET.

12 Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

13 Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 137, 2009.

14 Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165. ACM, 2002.

15 Amr Hany Saleh. Transforming delimited control: Achieving faster effect handlers. In *ICLP (Technical Communications)*, volume 1433 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

16 Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog system. *TPLP*, 12(1-2):5–34, 2012.

17 Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. Practical probabilistic programming with monads. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, pages 165–176. ACM, 2015.

18 Terrance Swift and David S. Warren. Tabling with answer subsumption: Implementation, applications and performance. In *LAI*, volume 6341 of *LNCS*, pages 300–312. Springer, 2010.

19 Terrance Swift and David S. Warren. XSB: Extending Prolog with tabled logic programming. *TPLP*, 12(1-2):157–187, January 2012.

20 Alexander Vandenbroucke, Maciej Piróg, Benoit Desouter, and Tom Schrijvers. Tabling with sound answer subsumption. *arXiv preprint arXiv:1608.00787*, 2016.

21 Alexander Vandenbroucke, Tom Schrijvers, and Frank Piessens. Fixing non-determinism. In *IFL 2015: Symposium on the implementation and application of functional programming languages Proceedings*, number 27. Association for Computing Machinery, 2016.

22 Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

23 David S. Warren. *Programming in Tabled Prolog*, volume 1. `http://www3.cs.stonybrook.edu/~warren/xsbbook/`, 1999.

24 Jan Wielemaker, S Ss, and I Ii. Swi-prolog 2.7-reference manual. 1996.

25 Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

26 Neng-Fa Zhou. The language features and architecture of B-Prolog. *TPLP*, 12(1-2):189–218, 2012.