*TPLP* **22** (6): 776–858, 2022. 

doi:10.1017/S1471068422000102 First published online 17 May 2022

# *Fifty Years of Prolog and Beyond**

PHILIPP KÖRNER and MICHAEL LEUSCHEL
*Institut für Informatik, Universität Düsseldorf, Universitätsstr. 1, D-40225 Düsseldorf, Germany*
(*e-mails:* p.koerner@uni-duesseldorf.de, leuschel@uni-duesseldorf.de)

JOÃO BARBOSA and VÍTOR SANTOS COSTA
*Department of Computer Science, Faculty of Science of the University of Porto, Porto, Portugal*
(*e-mails:* joao.barbosa@fc.up.pt, vscosta@fc.up.pt)

VERÓNICA DAHL
*Computing Sciences Department, Simon Fraser University, Burnaby, Canada*
(*e-mail:* veronica_dahl@sfu.ca)

MANUEL V. HERMENEGILDO and JOSE F. MORALES
*IMDEA Software Institute and Universidad Politécnica de Madrid (UPM), Madrid, Spain*
(*e-mails:* manuel.hermenegildo@imdea.org, josef.morales@imdea.org)

JAN WIELEMAKER
*Centrum voor Wiskunder en Informatica (CWI), Amsterdam, Netherlands*
(*e-mail:* J.Wielemaker@cwi.nl)

DANIEL DIAZ
*Centre de Recherche en Informatique, University Paris-1, Paris, France*
(*e-mail:* daniel.diaz@univ-paris1.fr)

SALVADOR ABREU
*NOVA-LINCS, University of Évora, Evora, Portugal*
(*e-mail:* spa@uevora.pt)

GIOVANNI CIATTO
*Department of Computer Science and Engineering, Alma Mater Studiorum—Univerità di Bologna, Bologna, Italy*
(*e-mail:* giovanni.ciatto@unibo.it)

*submitted 30 June 2020; revised 27 February 2022; accepted 6 March 2022*

## Abstract

Both logic programming in general and Prolog in particular have a long and fascinating history, intermingled with that of many disciplines they inherited from or catalyzed. A large body of research has been gathered over the last 50 years, supported by many Prolog implementations. Many implementations are still actively developed, while new ones keep appearing. Often, the

*Verónica Dahl is thankful for the support provided for this work by NSERC's Discovery grant 31611024. José Francisco Morales and Manuel Hermenegildo are partially funded by MINECO MICINN PID2019-108528RB-C21 *ProCode* project, by the Madrid P2018/TCS-4339 *BLOQUES-CM* program, and by the Tezos foundation. Salvador Abreu acknowledges support from FCT through strategic project UIDB/04516/2020 (NOVA LINCS). The authors would like to thank Egon Börger, Ugo Chirico, Bart Demoen, Maarten van Emden, John Patrick Gallagher, Sebastian Krings, John Lloyd, Michael Maher, Fernando Pereira, Luís Moniz Pereira, Zoltan Somogyi, German Vidal, David H.D. Warren, David S. Warren, and Isabel Wingen for their input and fruitful discussions. They also thank Juan Emilio Miralles

features added by different systems were motivated by the interdisciplinary needs of programmers and implementors, yielding systems that, while sharing the "classic" core language, in particular, the main aspects of the ISO-Prolog standard, also depart from each other in other aspects. This obviously poses challenges for code portability. The field has also inspired many related, but quite different languages that have created their own communities. This article aims at integrating and applying the main lessons learned in the process of evolution of Prolog. It is structured into three major parts. First, we overview the evolution of Prolog systems and the community approximately up to the ISO standard, considering both the main historic developments and the motivations behind several Prolog implementations, as well as other logic programming languages influenced by Prolog. Then, we discuss the Prolog implementations that are most active after the appearance of the standard: their visions, goals, commonalities, and incompatibilities. Finally, we perform a SWOT analysis in order to better identify the potential of Prolog and propose future directions along with which Prolog might continue to add useful features, interfaces, libraries, and tools, while at the same time improving compatibility between implementations.

*KEYWORDS*: Prolog, logic programming systems, portability, rationale, evolution, vision

## 1 Introduction

Logic programming languages in general, and Prolog in particular, have a long and fascinating history, having catapulted computing sciences from its old number-crunching, algorithm-focused and mostly imperative paradigm into the new, unique paradigm of inferential engines. Rather than measuring performance through the number of calculations per second, we can now do so through inferences per second – a qualitative leap, with import well beyond the natural language processing uses for which Prolog had been first conceived.

Logic programming's truly novel characteristics distinguish it not only from traditional imperative programming but also from functional programming, some of whose aims and techniques it shares. The year TPLP celebrates its 20-year anniversary also marks the milestone of 50 years of evolution since the first steps toward Prolog, the first version of which was completed in 1972. Logic programming and Prolog have progressed over the years deeply intermingled with the evolution of the different areas they both resulted from, as well as those that they enabled.

The Prolog language in particular has attracted sustained academic and practical interest since its origins, yielding a large body of research. The language has been supported by numerous Prolog implementations, many of which are still in active development, while new ones keep appearing all the time. The large number of features added by different systems during this evolution were often motivated by the diverging needs of respective implementors. As a result, while sharing a core language including the main

for proofreading. The authors are particularly grateful for the effort made by the group of people (more than 120 individuals) that drafted the ISO-Prolog Standard and piloted its later evolution, and specially to Jonathan Hodgson, Roger Scowen, Tony Dodd, Pierre Deransart, Ali Ed-Dbali, Paulo Moura, and Ulrich Neumerkel, the current editor. The authors would also like to thank the anonymous reviewers for their thoughtful criticism and suggestions, which gradually helped improve this paper. Apologies to the many that are not mentioned. The authors also endorse Paul McJones' efforts to maintain a historical archive on Prolog, at http://www.softwarepreservation.org/projects/prolog/, and thank all contributors.

aspects of the ISO-Prolog standard, most Prolog systems also depart from each other in significant ways. This fertile evolution has also spawned many other new languages and paradigms that have created their own communities.

This article is structured in three major parts. In the first part, in Section 2, we outline the evolution of Prolog systems and the community approximately up to the development of the ISO standard in 1995, focusing on historic developments and scientific milestones. This provides a condensed description of the history of Prolog including the steps that got us to the first standard, along with the main motivations behind each step.

In the second part, in Section 3, we discuss the need for the ISO standard and analyze, with this standard in mind, how the Prolog implementations and community have evolved since. The section aims at documenting the vision, research, and development focus for each implementation. Since most systems have incorporated significant functionality beyond the Prolog ISO-standard, we survey these nonstandard features, with a special emphasis on portability. That section gathers very diffused information in one place; tables and small paragraphs allow for convenient comparison of implementations. We also consider logic programming languages that have considerably departed from Prolog, but were obviously strongly influenced by it.

In the third and last part, Section 4, we gaze into the crystal ball and answer a few questions such as: How might Prolog and its community evolve in the future? Can we better unify the new aspects that are offered by different implementations? How should efforts for increased portability be organized? Does it make sense to aim for a unified language? And, what tools could be provided to ease development? Furthermore, we propose a plan for future steps that need to be taken to evolve Prolog as a language. The plan is founded on needs expressed by the community in a consultation and on a comparison with successful evolution of other languages. Our goal is to achieve further standardization and easier portability of code among implementations. We argue for why this is important and discuss why previous similar attempts have failed.

## 2 Part I: The early steps of Prolog

This first part of the paper provides two major contributions. First, it provides a general definition of what "Prolog" is and what a current Prolog implementation generally looks like. This allows us to focus in the paper on this class of systems, although we also mention briefly other related ones. The second contribution of this section is a description of the evolution of Prolog systems approximately up to and including the appearance and gradual adoption the ISO standard.

Given the high variability of features and technologies which characterized prestandard Prolog systems, some early systems may not perfectly fit a modern definition of Prolog. Nevertheless, we choose to include these also in our historical discussion – and consider them in any case Prolog systems – because of their importance in shaping Prolog as we know it today. More concretely, we consider Prolog systems essentially all those discussed in this paper, except of course the "systems influenced by Prolog" of Section 3.5.2.

Note that it is beyond the scope of our article to reconstruct all the initial steps and theoretical developments that led to Prolog. Fortunately, good recollections can be found for example by Kowalski* (1988, 2013), Cohen* (1988), Van Roy (1994), Colmerauer*

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

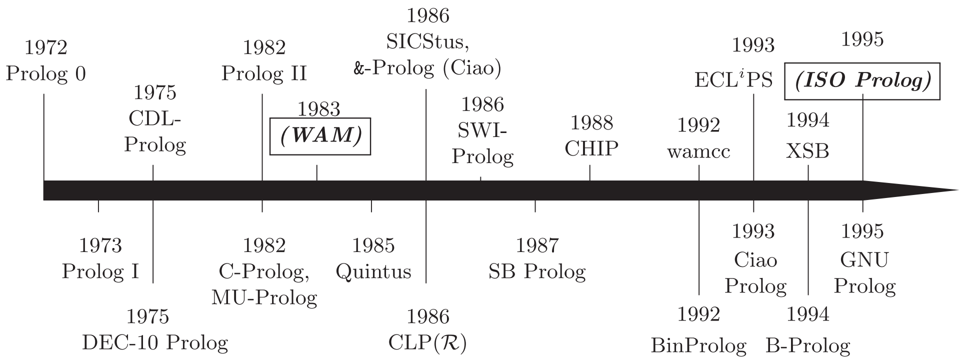


Fig. 1. Approximate timeline of some early Prolog systems (up to the ISO Standard).

(1996) and van Emden* (2006).[1] Here, we discuss those first steps that are useful for understanding the origin and evolution of the Prolog systems that have survived to the present day.

We also note that this paper is not aimed at providing a completely exhaustive list of Prolog systems: the list is very large and constantly changing, and many other implementations, such as Waterloo Prolog (Roberts 1977), UNSW Prolog (Sammut and Sammut 1983), or the recently discontinued Jekejeke Prolog, to name just a few, have helped to spread Prolog throughout the world, but we simply cannot cover them all. Instead, we have tried to concentrate on implementations that constitute a milestone in the evolution of the language or offer some specially interesting characteristics. We redirect interested readers to the historical sources archive maintained by McJones (2021) to learn about many of the earlier systems.

Figure 1 provides a timeline overview of some of the most impactful of the early Prolog systems treated in this section, that is, approximately up to the development of the ISO standard.

Throughout the paper we attempt to assign meaningful dates to the different Prolog systems covered. This is not always straightforward, and the dates should not be given too much significance. The strategy that we have followed is as follows: we first looked for some authoritative source explicitly stating the date the Prolog system was developed or made publicly available. We consider as "authoritative source" any paper from any logic programming-related conference or journal, as well as the Prolog system's home page, or any technical report or manual of the system. In absence of these sources, we looked for resources on the web mentioning the Prolog system along with a date, and we selected the earliest date among all resources.

### *2.1 Defining Prolog*

Prolog is arguably the most prominent language under the logic programming (LP) umbrella. However, as we elaborate in the remainder of this paper, the evolution of Prolog

[1]The asterisk by the name indicates people who are recognized for their impactful research as *Founders of Logic Programming* by the Association of Logic Programming (ALP): http://www.cs.nmsu.edu/ALP/the-association-for-logic-programming/alp-awards/.

did not follow a linear path. Many contributions have been presented in the history of LP as implementations, extensions, variants, or subsets of Prolog. Interestingly, while in some other programming paradigms the custom is to create new language names when making modifications or extensions to a given language, the Prolog tradition has been instead to keep the name Prolog across this long history of very substantial evolution.

In the following, we attempt to draw a line between what can be considered a Prolog implementation and what not. We do so by defining Prolog from several perspectives. We first provide a conceptual and minimalist definition of the essential features of Prolog (in a post-ISO-standard world). We then overview a number of important (yet non-essential) features that any full-fledged implementation of Prolog should include. Finally, we present a technical test users may perform to verify whether a technology can be considered as Prolog or not.

The objective of our definition is in any case inclusive, in the sense that we aim at encompassing all systems that preserve the essence that is generally recognized as Prolog, while allowing the many extensions that have taken place and hopefully those that may be adopted in the future.

*Conceptual Definition* Any Prolog implementation must *at least* support:

1. Horn clauses with variables in the terms and arbitrarily nested function symbols as the basic knowledge representation means for both programs (a.k.a. knowledge bases) and queries;
2. the ability to manipulate predicates and clauses as terms, so that meta-predicates can be written as ordinary predicates;
3. SLD-resolution (Kowalski 1974) based on Robinson's* principle (1965) and Kowalski's procedural semantics (Kowalski 1974) as the basic execution mechanism;
4. unification of arbitrary terms which may contain logic variables at any position, both during SLD-resolution steps and as an explicit mechanism (e.g., via the built-in `=/2`);
5. the automatic depth-first exploration of the proof tree for each logic query.

Notably, item 1 aims at excluding strict subsets of Prolog which do not support function symbols or require knowledge bases to be ground. Item 2 rules out custom rule engines for Horn clauses which do not support meta-programming, while requiring Prolog implementations to support meta-predicates. In other words, real Prolog systems must at least support an efficient mechanism such as `call/1`, enabling programmers to write predicates accepting terms as arguments, to be interpreted as goals. ISO-compliant implementations, for instance, employ meta-predicates to support negation, disjunction, implication, and other aspects which are not naturally supported by Horn clauses. Item 4 requires implementations to expose the unification mechanism to the users, and it cuts off subsets of Prolog employing weaker forms of pattern matching (e.g., where variables can only appear once and only at the top-level). Items 3 and 5 constrain Prolog solvers to a backward (goal-oriented) resolution strategy where a proof tree is explored via some traversal strategy. ISO-compliant implementations support a sequential, depth-first, deterministic exploration of the proof tree, via backtracking. This is commonly achieved by selecting clauses in a top-down and subgoals in a left-to-right fashion. Other implementations may support further strategies: for instance, tabled Prologs can deviate from pure depth-first traversal for tabled predicates. Other Prologs may implement alternative search strategies in addition to depth-first, possibly for certain predicates. The important issue here is to have at least a (default) mode in which the system is a true programming language, predictable in terms of cost in steps and memory.

*Common Relevant Features* Any Prolog implementation may also support:

6. some control mechanism aimed at letting programmers manage the aforementioned exploration;
7. negation as failure (Clark 1978), and other logic aspects such as disjunction or implication;
8. the possibility to alter the execution context during resolution, via ad hoc primitives;
9. an efficient way of indexing clauses in the knowledge base, for both the read-only and read-write use cases;
10. the possibility to express definite clause grammars (DCG) and parse strings using them;
11. constraint logic programming (Jaffar and Lassez 1987) via ad hoc predicates or specialized rules (Frühwirth 2009);
12. the possibility to define custom infix, prefix, or postfix operators, with arbitrary priority and associativity.

There, item 6 dictates that users should be provided with some mechanism to control the proof tree exploration. ISO-compliant implementations provide the cut for this purpose, while other Prologs may expose further mechanisms. For instance, in tabled Prologs users must explicitly specify which rules are subject to tabling, and in this way they retain some degree of control about the proof tree exploration. Similarly, delay declarations like `when/2` allow one to influence the selection rule employed for SLD-resolution. Item 7 provides a practical way to realize negation on top of Horn clauses and SLD, which Keith Clark* gave a nonprocedural semantics to – namely, completion semantics – showing that negation as failure is theoretically sound (Clark 1978). Furthermore, negation as well as other logic operators contribute to the perception of Prolog as a *practical* programming language. Item 8 requires implementations to support, via *side effects*, the dynamic modification of fundamental aspects that affect the resolution process, possibly as the resolution is going on. These aspects may include the knowledge base (a.k.a. the *dynamic* clause database), the flags, or the pool of currently open files, and their modification should be exposed to the user via ad hoc meta-predicates. For instance, ISO-compliant implementations rely upon built-in predicates like `assert/1`, `retract/1`, `set_prolog_flag/1`, etc. to serve this purpose. In particular, to make both the access and modification of clauses efficient, item 9 plays a very important role: the satisfaction of this optional requirement is what distinguishes toy implementations from full-fledged Prolog systems. Finally, while not strictly essential, items 10 and 11 are two very successful features many modern Prolog system support. Item 12 is a nice-to-have feature which allows a more natural notation when extending Prolog systems with custom functionalities, without requiring a bare new language to be designed from scratch.

In particular, probabilistic extensions of Prolog such as ProbLog (de Raedt *et al*. 2007), and `cplint` (Riguzzi 2007) benefit from custom operator definitions.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

Of course, many other features may enrich (or be lacking from) a Prolog implementation. Consider for instance, full ISO library support, presence/lack of a module system, and so on. While these are technical aspects that greatly affect the efficiency, effectiveness, and usability of Prolog implementations, we do not consider them as fundamental.

*Technical Test* As a rule of thumb, one can check if a logic solver can be considered as a Prolog system or not via the following test. The test requires that the well-known `append/3` predicate can be written exactly as follows:

```
append([], X, X).
append([H | X], Y, [H | Z]) :- append(X, Y, Z).
```

and can be queried in variety of ways, e.g., to append two lists: `append([1, 2], [c, d], R)`, deconstruct a list as in: `append(A, B, [1, 2])`, or with any arbitrary instantiation of the arguments, such as `append([X | T], [c], [Z, Z, Z])`.

Note that the above test excludes some logic programming languages, such as Datalog (Maier *et al.* 2018), as it does not support functors (just constants); traditional ASP (Answer Set Programming) (Brewka *et al.* 2011), as it does not cater for fully recursive first-order terms with no bound; or Mercury (Somogyi *et al.* 1996), as it is based on pattern-matching and not on unification and only caters for linear terms. CORAL (Ramakrishnan *et al.* 1994), on the other hand, is an interesting edge case: while its default proof tree exploration strategy does not meet our definition, it can be instantiated to behave like Prolog. Thus, we would consider the entire system not to be a Prolog (but it would qualify as an extension). Other systems we choose not to consider as Prolog systems (but rather as systems derived from Prolog) are Gödel (Hill and Lloyd 1994), Curry (Hanus *et al.* 1995), and Picat (Zhou *et al.* 2015). Nevertheless, we do discuss these systems in some detail in Section 3.5, where we discuss Prolog derivatives. However, as mentioned before, we consider all other systems that are discussed in the paper to be Prologs.

### *2.2 Ancestors of Prolog*

Prolog descends from three main branches of research: AI programming, automatic theorem proving, and language processing.

The field of AI was born around 1956 and quickly gave rise to the functional programming language LISP (McCarthy 1962). A host of other AI languages followed, sometimes grouped under the denomination of *Very High Level Languages*. These languages had features such as symbolic processing and abstraction that set them apart from more mundane languages.

Automatic theorem proving made a big step forward in a seminal paper by Alan Robinson introducing the resolution inference rule (Robinson 1965). Resolution extends modus ponens and modus tollens and includes unification. Resolution can be used to obtain a semi-decision procedure for predicate logic and is at the heart of most inference procedures in logic programming.

In the wake of these advances, an early visionary in the development of the logic programming field was Cordell Green, who already in the late 60s envisioned how to extend resolution to automatically construct problem solutions, and implemented this vision in particular for automatically answering questions based on first-order logic, illustrating it as well for plan formation, program synthesis, and program simulation, thus presaging the possibility of moving symbolic programming beyond functions and into logic (Green 1969a;b). This represented perhaps the first zenith of logic in AI (Kowalski 1988). Also notable is Ted Elcock, whose 1967 Aberdeen System, Absys, developed with Michael Foster, while not having directly influenced the development of Prolog, was a declarative programming language that anticipated some of Prolog's features such as invertability, negation as failure, aggregation operators, and the central role of backtracking (Elcock 1990).

Meanwhile, Alain Colmerauer was seeking to automate human–machine conversation, which led him to develop Q-systems (Colmerauer 1970a; Colmerauer and Roussel 1996; Colmerauer 1970b), a tree rewriting system that for many years served for English-to-French translation of Canadian meteorological reports. His aim of modifying Q-systems so that a complete question-answering system (rather than just the analyzer part of it) could be written in logic inspired him, among others, to create Prolog.

Floyd's work on nondeterministic algorithms (Floyd 1967) (cf. the survey by Cohen 1979) was another important influence, as was Kowalski and Kuehner's SL resolution (1971). SL resolution is a refinement of resolution which is still both sound and refutation complete for the full clausal form of first order logic, and underlies the procedural interpretation of Horn clauses (Kowalski 1974).

A further simplification for the case of Horn clauses – SLD resolution (Kowalski 1974) – resulted from Kowalski's efforts to reconcile the declarative nature of logic based representations of knowledge with PLANNER's procedural approach (Hewitt 1969). The semantics of Horn clauses was explored by Kowalski and van Emden (1976).

### *2.3 The birth of Prolog*

Colmerauer's aim of creating a human–machine communication system in logic had led him to further research French language analysis with Pasero (1973), and to numerous experiments with Philippe Roussel and Jean Trudel on automated theorem proving methods. Having learned about SL resolution, he invited Kowalski to visit Marseille in the summer of 1971. The visit led to Roussel's use of SL resolution in his thesis on formal equality in automated theorem proving (1972). In addition to its attractions as a theorem prover, SL resolution had the additional attraction that its stack-type operating mode was similar to the management of procedure calls in a standard programming language, making it particularly well suited for implementation by backtracking à la Floyd, which Colmerauer adopted for efficiency, so as to avoid having to copy and save the resolvents. Yet, for language processing, Q-systems still seemed indispensable. During Kowalski's 1971 visit to Marseille, Kowalski and Colmerauer discovered that a certain way of representing formal grammars in clausal logic enables certain general-purpose proof procedures for first-order logic to behave as special-purpose parsing methods: SL-resolution as top-down parsing, hyper-resolution as bottom-up parsing, similar to Q-systems.

Then, Colmerauer defined a way to encode grammar rules in clauses, known today as the difference-list technique, and introduced extra parameters into the nonterminals to propagate and compute information, through which the analyzer could extract, as in Q-systems, a formula representing the information contained in a sentence.

Colmerauer and Kowalski's collaboration led in 1972 to a discovery analogous, for programs, to that made previously for grammars: that a certain style for representing programs in the clausal form of logic enables SL resolution to behave as a computational procedure for executing computer programs.

For this to happen, though, a simplification for efficiency of Kowalski's SL-resolution was implemented at the cost of incompleteness: linear resolution was constrained to unify only between the head literals of ordered clauses with ordered literals.[2] This made Colmerauer's aim of creating a human-machine communication system possible. The result was not only the first Natural Language (NL) application of what we now know as Prolog, but most importantly, the basis of Prolog itself: a linear resolution system restricted to Horn clauses that could answer questions (i.e., solve problems) nondeterministically in the problem domain described by the clauses input (Colmerauer *et al.* 1973).

However, the Marseille group was unaware of Horn clauses at the time. But Kowalski recognized that Marseille's principal "heresy" (in Colmerauer' words), a strategy of linear demonstration with unifications only at the heads of clauses, was justified for Horn clauses. Kowalski also clarified further simplifications that so far were only implicit: the elimination of "ancestor resolution" (which only works with non-Horn clauses) and the elimination of the "factoring" rule. Together with Maarten van Emden, he went on to define the modern semantics of Horn-clause programming (van Emden and Kowalski 1976).

### *2.4 The early Prolog systems*

Prolog implementations evolved in interaction with ad hoc, initially meta-programmed extensions of the language itself, created for the often interdisciplinary needs of applications. In time, these extensions became, or evolved into, standard features of the language. In this section, we chronicle such early developments.

#### *2.4.1 Prolog 0, Prolog I (1972–1973)*

*Basic Features:* As reported by Cohen (1988) and later by Colmerauer and Roussel (1996), the first system ("Prolog 0") was written in Algol-W by Roussel in 1972. Practical experience with this system lead to a much more refined second implementation ("Prolog I") at the end of 1973 by Battani, Meloni, and Bazzoli, in Fortran. This system already had the same operational semantics and most of the built-ins that later became part of the ISO standard, such as the search space pruning operator (the "cut"), relevant for Prolog to become a practical AI language. Efficiency was greatly improved by adopting the structure-sharing technique by Boyer and Moore (1972) to represent the clauses generated during a deduction.

*Higher-order logic extensions:* Basic facilities for meta-programming higher-order logic extensions were present in Prolog systems from the very beginning, and many later

[2] "head literals" are not to be confused with what we now call the "head" of a clause: at the time, conditions and conclusions were distinguished by a "+" or "-" sign preceding the literal in question, not by which side of Prolog's "if"" symbol they were in- there was no such symbol in their clauses.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

systems include extended higher-order capabilities beyond the basic `call/1` predicate – for example, λProlog (Nadathur and Miller 1988), BinProlog (Tarau 1992; 2012), Hyprolog (Christiansen and Dahl 2005). Some of the most influential early extensions are discussed into the following subparagraphs.

Constraints: Interestingly, Prolog 0 already included the `dif/2` ($\neq$) predicate, as a result of Roussel's thesis (1972). The predicate sets up a constraint that succeeds if both of its arguments are different terms, but *delays the execution* of the goal if they are not sufficiently instantiated.

Coroutining: Although `dif/2` was neither retained in Prolog I nor became part of the ISO standard, it meant a first step towards the extension of unification to handle constraints: while it introduced the negation of unification, it also allowed an early form of coroutining. Building on this work, Verónica Dahl* introduced a `delay` meta-predicate serving to dynamically reorder the execution of a query's elements by delaying a predicate's execution until statically defined conditions on it become true, and used it to extend Prolog with full coroutining – that is, the ability to execute either a list of goals or a first-order logic formula representing a goal, by proving them in efficient rather than sequential order. With Roland Sambuc, she developed the first Prolog automatic configuration system, which exploited coroutining, for the SOLAR 16 series of computers (Dahl and Sambuc 1976).

Safe Negation as Failure: Dahl also used `delay/2` to make negation-as-failure (NaF) (the efficient but generally unsafe built-in predicate of Prolog I which consists of assuming `not(p)` if every proof of `p` fails) safe, simply by delaying the execution of a negated goal until all its variables have been grounded. This approach to safe negation and to coroutining made its way into many NL consultable systems, the best known being perhaps Chat 80 (Warren and Pereira 1982), and more importantly, into later Prologs, as we discuss later.

Deductive Databases: Dahl then ushered in the deductive database field by developing the first relational database system written in Prolog (Dahl 1977; 1982). Other higher-order extensions to Prolog included in this system or in the one by Dahl and Sambuc (1976), such as `list/3` (now called `setof/3`), have become standard in Prolog.

*Metamorphosis Grammars:* Metamorphosis Grammars (MGs) (Colmerauer 1975) were Colmerauer's language processing formalism for Prolog. They constituted at the time a linguist's dream, since they elegantly circumvented the single-head restriction of Prolog's Horn clauses, thus achieving the expressive power of transformational grammars in linguistics, which, as type-0 formal grammars, allow more than one symbol in their left-hand side. This allowed for fairly direct, while also executable, renditions of the linguistic constraints then in vogue: a single rule could capture a complete parsing state through unification with its left-hand (multi-head) side, in order to enforce linguistic constraints through specifying, in its right-hand side, how to re-write it.

The first applications of MGs were compilation (Colmerauer 1975); French consultation of automatic configuration systems (Dahl and Sambuc 1976), where a full first-order logic interlingua was evaluated through coroutining; and Spanish consultation of database systems (Dahl 1977; 1979), where a set-oriented, three valued logic interlingua (Colmerauer 1979; Dahl 1979) was evaluated, allowing among other things for the detection of failed presuppositions (Dahl 1977; 1982). Coroutining was used in the system by Dahl and Sambuc (1976) not only for feasibility and efficiency, as earlier described, but also to permit different paraphrases of a same NL request to be reordered into a single, optimal execution sequence.

A simplification of MGs, Definite Clause Grammars (DCGs), was then developed by Fernando Pereira* and David H.D. Warren*,[3] in which rules must be single-headed like in Prolog, while syntactic movement is achieved through threading syntactic gap arguments explicitly. DCGs were popularized in 1980 (Pereira and Warren 1980) and became a standard feature of Prolog. It is worth highlighting that the "DCG" name does not refer to the fact that they can translate to definite clauses (since all four subsets of MGs can, just as a side effect of being included in MGs), but to their restriction to single heads, which makes them similar in shape to definite clauses.

More specialized Prolog-based grammars started to emerge. Their uses to accommodate linguistic theories, in particular Chomskyan, were studied as early as 1984 (Dahl 1986), leading to the new research area of "logic grammars" (Abramson and Dahl 1989).

*Further Theoretical Underpinnings:* In 1978, Keith Clark published a paper that showed NaF to be correct with respect to the logic program's completion (Clark 1978). Simultaneously, Ray Reiter* provided a logical formalization of the related "Closed World Assumption" (Reiter 1978), which underlies NaF's sanctioning as false of anything that cannot be proved to be true: since in a closed world, every statement that is true is also known to be true, it is safe – in such worlds – to assume that what is not known to be true is false. This then led to substantial research on nonmonotonic reasoning in logic programming, and to inspiring foundational work on deductive databases by Reiter himself, as well as Herve Gallaire*, Jack Minker*, and Jean-Marie Nicolas (1984). The work of Cohen (1979) on nondeterminism in programming languages was also influential in these early stages.

### *2.4.2 CDL-Prolog (1975)*

As discussed by Peter Szeredi* (2004), a group at NIM IGÜSZI in Hungary was trying to port the Marseille system to the locally available machine in 1975. At the same time, Szeredi, who was part of another group at NIM IGÜSZI, completed his first (unnamed) Prolog implementation using the Compiler Definition Language (CDL) developed by Cornelis Koster, one of the authors of the Algol 68 report, marking the beginning of a series of substantial contributions to Prolog.

### *2.4.3 DEC-10 Prolog (1975)*

In 1974, David H.D. Warren visited Marseille and developed a plan generation system in Prolog, called Warplan (Warren 1974). He then took Prolog with him as a big deck

[3] Not to be confused with David S. Warren, see later.

of punched cards and installed it on a DEC-10 in Edinburgh, where he enhanced it with an alternative "front-end" (or "superviser") written in Prolog, to better tailor it to the Edinburgh computing environment and the wider character set available (the Marseille group had been restricted by a primitive, upper-case-only, teletype connection to a mainframe in Grenoble). He distributed this version to many groups around the world.

He then set out to address what he perceived as a limitation of the implementations of Prolog up to that point in time: they were comparatively slower and more memory hungry than other high-level AI-languages of the time and, in particular, than LISP. In what would eventually become his PhD thesis work (Warren 1977), David H.D. Warren pieced away on one side the elements of Prolog that could be implemented in the same way as the most efficient symbolic programming languages (activation records, argument passing, stack- and heap-based memory management, etc.), and applied to them well-established compilation techniques. Then, for those elements of Prolog that were more novel, such as unification and backtracking, he developed or applied specific compilation and run-time techniques such as optimization of unification by clause head precompilation, fast recovery of space on backtracking, trailing, or structure sharing-based term representation (Boyer and Moore 1972) (the latter already present in Marseille Prolog). He used as target again the DEC-10 with the TOPS-10 operating system, and exploited architectural features of the DEC-10 such as arbitrary-depth indirect memory access, particularly suited for the structure-sharing technique. The product of this effort was the first compiler from Prolog to machine code. This resulted in a large leap in performance for Prolog, both in terms of speed and memory efficiency, rivaling that of Lisp systems. This was documented in Warren *et al*. (1977), in what was to be a landmark publication on Prolog in a mainstream Computer Science venue. A version of this compiler dated 1975 is part of the archive maintained by McJones (2021).

Fernando Pereira and Luís Moniz Pereira*, both at LNEC in Lisbon, also made major contributions to the development of the complete DEC-10 Prolog system, which also included now "classic" built-ins such as `setof/3` and `bagof/3`. A significant element in DEC-10 Prolog's popularity was the availability of an example-rich user guide (Warren 1975; Pereira *et al*. 1978). All these features, coupled with the improved syntax and performance, and the fact that the DEC-10 (and later DEC-20) were the machines of choice at the top AI departments worldwide, made DEC-10 Prolog available to (and used by) all these departments, and in general by the AI research community. This led to DEC-10 Prolog becoming very popular and it spread widely from about 1976 onward. By 1980, the system also featured a garbage collector and last-call optimization (Warren 1980). Also in 1980, David H.D. Warren and Fernando Pereira adapted it to TENEX/TOPS-20, which had then become the operating system(s) and machines most widely used for AI research.

The contributions made by the authors of DEC-10 Prolog were fundamental for the coming of age of Prolog: they proved that Prolog could not only be elegant and powerful but it could also come with the usability, speed, and efficiency of a conventional programming language. As a result, DEC-10 Prolog had a large influence on most Prologs after it, and its syntax, now known also as the "Edinburgh syntax," and many of its features constitute a fundamental component of the current Prolog ISO standard.

However, for all its merits, the one drawback of DEC-10 Prolog was that it was deeply tied to its computer architecture and thereby inherently not portable to new machines, in particular to the then-emerging 32-bit computer architectures with virtual memory. This prompted the development of other, more portable Prologs, described below, and eventually the Warren Abstract Machine (see Section 2.5).

#### *2.4.4 Unix Prolog (1979)*

As discussed by Mellish (1979), there were a number of Prolog interpreters at the time that used the DEC-10 syntax but were internally quite different.

The objective of these other systems was to develop a portable, yet still reasonably-performing Prolog system, written in a mainstream source language, and that could be compiled on more mainstream, 32-bit machines (including later Unix systems such as, for example, the DEC VAX family, which became ubiquitous).

The first system to achieve portability was Unix Prolog by Mellish (1979), written for PDP-11 computers running Unix, which was also ported to the RT-11 operating system. Unlike Marseille Prolog and DEC-10 Prolog, it used structure-copying rather than structure-sharing. It led to Clocksin and Mellish writing an influential textbook (1981) which describes a standard "core" Prolog, compatible with both DEC-10 Prolog and Unix Prolog.

#### *2.4.5 LPA Prolog (1980)*

Logic Programming Associates (LPA) was founded in 1980 out of the group of Kowalski at the Department of Computing and Control at Imperial College London, including, among others, Clive Spenser, Keith Clark, and Frank McCabe (LPA Ltd 2021). LPA distributed micro-PROLOG which ran on popular 8-bit home computers of the time such as the Sinclair Spectrum and the Apple II and evolved to be one of the first Prolog implementations for MS-DOS. LPA Prolog evolved to support the Edinburgh syntax around 1991 and is still delivered today as a compiler and development system for the Microsoft Windows platform.

#### *2.4.6 MU-Prolog (1982)*

In 1982, another implementation named MU-Prolog (Naish 1982; 1986) was developed by Lee Naish at Melbourne University. It was initially a simple interpreter to understand the workings of Prolog, as the author could not find a Prolog system for his hardware.

The system offered efficient coroutining facilities and a delay mechanism similar to those discussed in Section 2.4.1 to automatically delay calls to negation and if-then-else constructs, as well as meta-logical (e.g., `functor/3`) and arithmetic predicates. Its rendition of the `delay` predicate, here called `wait`, allows for declarations to be provided manually but also generated automatically.

MU-Prolog was one of the first shipping database connections, module systems, and dynamic loading of shared C libraries, as well as sound negation (through `delay/2`, as in Section 2.4.1) and a logically pure `findall/3` predicate, a consequence of its variable binding-controlled delayed goal execution. MU-Prolog was later succeeded by

NU-Prolog (Thom and Zobel 1987), bringing MU-Prolog's features to the Warren Abstract Machine (see Section 2.5).

### *2.4.7 C-Prolog (1982)*

As a first foray into getting Edinburgh Prolog on 32-bit address machines, Luís Damas created an Edinburgh-syntax Prolog interpreter for an ICL mainframe with Edinburgh-specific time sharing system (EMAS) and systems programming language (IMP). This interpreter used the structure sharing approach by Boyer and Moore (1972) and copied as far as possible the built-in predicates of DEC-10 Prolog. When Fernando Pereira got access to a 32-bit DEC VAX 11/750 at EdCAAD in Edinburgh in 1981, he rewrote EMAS Prolog in C for BSD 4.1 Unix. This required many adaptations from the untyped IMP into the typed C, and he also made it even closer to DEC-10 Prolog in syntax and built-in predicates. The whole project became known as C-Prolog later on (Pereira 1983). The archive maintained by McJones (2021) contains a readme file from 1982.

Although implemented as an interpreter, C-Prolog was reasonably efficient, portable and overall a very usable system. Thus it quickly became influential among the Edinburgh implementations, helping to establish "Edinburgh Prolog" as the standard. It contributed greatly to creating a wider Prolog community and remained extensively used for many years.

## *2.5 From Prolog compilation to the WAM*

Following David H.D. Warren's first Prolog compiler, described in Section 2.4.3, there were a number of other compiled systems up until 1983, including Prolog-X (Bowen *et al.* 1983) and later NIP, the "New Implementation of Prolog" (for details, cf. the survey by Van Roy 1994).

In 1983, funded by DEC in SRI, who wanted to have the Prolog performance of the DEC-10/20 implementation ported to the VAX line, David H.D. Warren devised an *abstract machine*, i.e., a memory architecture and an instruction set that greatly clarified the process of implementing a high-performance Prolog system (Warren 1983). This machine became widely known as the Warren Abstract Machine, the WAM. The proposal was basically a reformulation of the ideas of the DEC-10 compiler, which translated Prolog source to a set of abstract operations which were then expanded to machine code (Warren 1977), but expressed in a more accessible way. In particular, it was described in legible pseudo-code, as opposed to DEC-10 machine code. Warren made some changes with respect to the DEC-10 system, such as passing parameters through registers instead of the stack. Also, instead of the structure sharing approach used in the DEC-10 work, the WAM used the structure copying approach by Bruynooghe* (1976). The WAM also included the idea of compiling to intermediate code (bytecode), as introduced by the programming language Pascal and its p-code (Nori *et al.* 1974), which made compiled code very compact and portable, an approach that is still advantageous today with respect to native code in some contexts. The first software implementation of the WAM was for the Motorola 68000 implemented for Quintus by David H.D. Warren, which he also adapted to the VAX line. Evan Tick, later designed a pipelined microprocessor organization for Prolog machines based on the WAM (Tick 1984).

Copies of the SRI technical report describing the WAM were passed around widely among those who had an interest in Prolog implementations, and the WAM became the standard blueprint for Prolog compilers and continues to be today. The WAM was made even more widely accessible and easier to understand with the publication of Aït-Kaci's *Tutorial Reconstruction* (1991), building on earlier tutorials by Hermenegildo (1989) and Nasr.

Much work was done after that on further optimization techniques for WAM-based Prologs, achieving very high levels of sequential performance. This very interesting topic is outside the scope of this paper, but is covered in detail in the excellent survey by Van Roy (1994), and much of this work is by Van Roy himself. Further work, beyond the survey, includes, for example, dynamic compilation (da Silva and Santos Costa 2007), instruction merging (Nässén *et al*. 2001) (pioneered by Quintus), advanced indexing (Santos Costa *et al*. 2007; Vaz *et al*. 2009), optimized compilation (Van Roy and Despain 1992; Morales *et al*. 2004; Carro *et al*. 2006), optimized tagging (Morales *et al*. 2008), etc. Also, the compilation of Prolog programs to WAM code was proven mathematically correct by Börger and Rosenzweig (1995), and the proof was machine verified by Schellhorn and Ahrendt (1998), Schellhorn (1999).

### *2.6 The FGCS initiative*

In 1982 Japan's Ministry of International Trade and Industry (MITI) started the Fifth Generation Computer Systems (FGCS) initiative in order to boost Japan's computer industry. The technical objective was to build large parallel computers and apply them in artificial intelligence tasks, with logic programming as the basis, and in particular Prolog. The research was conducted across Japanese computer industries and at a specific research center, ICOT. Among the first results were hardware sequential Prolog machines called PSI (for Personal Sequential Inference), similar to those developed for Lisp at the time by companies such as Lambda Machines, Thinking Machines, Xerox, and Borroughs. A series of parallel machines were also developed in the project.

However, at the point of combining parallelism and logic programming, a language shift occurred. During a visit to ICOT, Ehud Shapiro developed what he defined as a *subset* of concurrent Prolog (Shapiro 1983; 1987). This referred to the fact that, in order to reduce the implementation complexity stemming from the interactions between concurrency and Prolog's backtracking, the latter was left out in this initial design. As in other concurrent logic programming languages at the time, such as Parlog (Clark and Gregory 1986), *committed choice* was supported instead, where only the first clause whose *guard* succeeds is executed. This guard part consists of a marked set of literals (normally built-ins) at the beginning of the clause. This inspired the Guarded Horn Clauses (GHC) language of Ueda (1985), as the Kernel Language 1 (KL1) (Ueda and Chikayama 1990), which became the core language of the FGCS project.

While the "kernel" denomination of KL1 indicated a desire to eventually recover the declarative search capabilities of Prolog, the basic characteristics of KL1 remained throughout the FGCS project. With the departure from Prolog, an essential part of the language's elegance and functionality was lost, and this arguably detracted from the potential impact of the FGCS.

It can be argued that the fifth generation project was successful in a number of ways. From the technical point of view, in addition to the programming language work, it produced many results in parallel machines, scheduling, parallel databases, parallel automated theorem proving, or parallel reasoning systems. Perhaps most importantly, it accelerated much work elsewhere. This included a significant line of research into concurrent (constraint) logic languages (see Section 3.5.1). But, more relevant herein, all the work on parallel implementation of Prolog, which in the end was done at other centers throughout the world rather than in Japan (we return to this briefly in Section 2.7).

Beyond the technical part, the FGCS project developed very valuable expertise in computer architecture, parallelism, languages, software, etc. and nurtured a whole generation of researchers in areas that were hitherto not so well covered in Japan. Furthermore, the FGCS project spurred a number of similar initiatives around the world that led to important legislative changes and funding schemes that last until today. For example laws were developed that allowed companies to collaborate on "pre-competitive" research. This gave rise to the Microelectronics Computer and Technology Corporation (MCC) in the US and to the European Computer Research Center (ERC) in Europe, where hardware Prolog machines were also developed, and, most importantly, the EU ESPRIT program that has continued to the present day in the form of the current framework programs. An account of the outcomes of the FGCS project was presented by Shapiro *et al.* (1993).

### *2.7 Parallelism*

In parallel to the FGCS project, logic programming and Prolog were soon recognized widely as providing good opportunities for parallel execution, largely because of their clean semantics and potentially flexible control. This spurred a fertile specialized research and development topic, and several parallel implementations of Prolog or derivatives thereof were developed, targeting both shared-memory multiprocessors and distributed systems. As mentioned before, many concurrent Prolog derivatives were also developed. Going over this very large and fruitful field of research is beyond the scope of this paper; good accounts may be found in the articles by Gupta *et al.* (2001), de Kergommeaux and Codognet (1994) and Kacsuk and Wise (1992). There is also a survey on this topic in this same special issue of the TPLP journal (Dovier *et al.* 2022). However, it is worth mentioning that two of the current Prolog systems, SICStus and Ciao, have their origins in this body of work on parallelism.

*Or-Parallelism: SICStus, Aurora, MUSE (1985)* Around 1985, the Swedish Institute of Computer Science (SICS) was founded and Mats Carlsson joined SICS to develop a Prolog engine that would be a platform for research in or-parallelization of Prolog, that is, the parallel exploration of alternative paths in the execution. This work was performed in the context of the informal "Gigalips" project, involving David H.D. Warren at SRI and researchers from Manchester and Argonne National Laboratory, as well as and-parallel efforts (described below). This resulted in quite mature or-parallel Prologs, such as Aurora (Lusk *et al.* 1990) and MUSE (Ali and Karlsson 1990). The objective of these Prologs was to achieve effective speedups through or-parallel execution transparently

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

for the programmer and supporting full Prolog. This led to SICS distributing SICStus Prolog, which quickly became popular in the academic environment.

*And-Parallelism: RAP-WAM and &-Prolog (1986), a.k.a. Ciao Prolog* Since 1983, the University of Texas at Austin conducted research on and-parallelization of Prolog, that is, executing in parallel steps within an execution path, complementary to or-parallelism. The appearance of the WAM led to &-Prolog's abstract machine, the RAP-WAM (Hermenegildo 1986), which extended the WAM with parallel instructions, lightweight workers, multiple stack sets, task stealing, etc. Richard Warren, Kalyan Muthukumar, and Roger Nasr joined the project, which continued now also at MCC (also funded by DEC). The RAP-WAM was recoded using early versions of SICStus, also becoming part of the "Gigalips" effort. &-Prolog extended Prolog with constructs for parallelism and concurrency, and incorporated a parallelizing compiler (Muthukumar and Hermenegildo 1990; Muthukumar *et al*. 1999) which performed global analysis using the *ProLog Abstract Interpreter*, PLAI (Warren *et al*. 1988; Muthukumar and Hermenegildo 1989), based on abstract interpretation (Cousot and Cousot 1977). This allowed the exploitation of parallelism transparently to the user, while supporting full Prolog, and, on shared-memory multiprocessors, was the first proposed WAM extension to achieve effective parallel speedups (Bueno *et al*. 1999). This infrastructure was later extended to support constraint logic programs (García de la Banda *et al*. 1996; 2000). &-Prolog evolved into Ciao Prolog (cf. Section 2.10.2).

## 2.8 Constraints

As discussed by Colmerauer (1984), in 1982, a new version of Prolog, Prolog II (Colmerauer 1982a; van Emden and Lloyd 1984), was developed in Marseille by Alain Colmerauer, Henri Kanoui, and Michel van Caneghem, for which they shared in 1982 the award *Pomme d'Or du Logiciel Français*. This release brought two major contributions to the future paradigm of Constraint Logic Programming (CLP) (Jaffar and Lassez 1987; Jaffar and Maher 1994; Marriott and Stuckey 1998): moving from unification to equations and inequations over rational trees, and innovative extensions to constraint solving and its semantic underpinnings driving into richer domains.

### 2.8.1 The CLP scheme and its early instantiations

CLP was presented by Jaffar and Lassez (1987) in their landmark paper as a language framework, parameterized by the *constraint domain*. The fundamental insight behind the CLP scheme is that new classes of languages can be defined by replacing the unification procedure in the resolution steps by a more general process for solving *constraints* over specific *domains*. Jaffar and Lassez proved that, provided certain conditions are met by the constraint domain, the fundamental results regarding correctness and (refutation) completeness of resolution are preserved. Traditional LP languages and Prolog are particular cases of the scheme in which the constraints are equalities over the domain of Herbrand terms and can be represented as CLP($\mathcal{H}$). The CLP framework was first instantiated as the CLP($\mathcal{R}$) system (Jaffar *et al*. 1992), which implemented linear equations and inequations over real numbers, using incremental versions of Gaussian elimination and the Simplex algorithm. CLP($\mathcal{R}$) was widely distributed, becoming a popular system. In the meantime, the research group at ECRC (the European Computer Research Centre)[4] developed CHIP (Dincbas *et al.* 1988) (for *Constraint Handling in Prolog*) over the late 1980s, which interfaced Prolog to domain-specific solvers stemming from operations research and successfully introduced constraints over finite domains, CLP($\mathcal{FD}$). CHIP also introduced the concept of *global constraints* (Beldiceanu and Contejean 1994), which is arguably a defining feature of CLP and Constraint Programming. Other instances of the CLP scheme supported constraints over *intervals*, as implemented by BNR-Prolog (Older and Benhamou 1993), and constraints over booleans, which are usually implemented as a specialization of finite domains and are useful to express *disjunctive constraints*, whereby a set of constraints may be placed which encode multiple alternatives, without resorting to Prolog-level backtracking.

### *2.8.2 Later Marseille Prologs*

*Prolog III (1990)* Colmerauer (1990) focused on improving some limitations of Prolog II. It now included the operations of addition, multiplication, and subtraction as well as the relations $\leq, <, \geq$, and $>$. It also improved on the manipulation of trees, together with a specific treatment of lists, a complete treatment of two-valued Boolean algebras, and the general processing of the relation $\neq$. By doing so, the concept of unification was replaced by the concept of constraint solving in a chosen mathematical structure. By mathematical structure, we mean here a domain equipped with operations and relations, the operations being not necessarily defined everywhere.

*Prolog IV (1996)* Colmerauer (1996) generalized to discrete and continuous domains the technique of constraint solving by enclosure methods. The solving of an elementary constraint, often qualified local, consists in narrowing the domain ranges of its variables, which generally are intervals. In a system where numerous constraints interact, interval narrowing and propagation is performed iteratively, until a fixed point is reached. It also moved closer to the ISO standard syntax.

### *2.8.3 Opening the box*

While the early instantiations on the CLP scheme, such as CLP($\mathcal{R}$), the CLP scheme predecessor Prolog II, BNR Prolog, Prolog III and IV, etc. were all specialized systems, new technology incorporated into Prolog engines for supporting extensions to unification, such as meta-structures (Neumerkel 1990) and attributed variables (Holzbaur 1992), enabled a *library-based approach* to supporting embedded constraint satisfaction in standard Prolog systems. This approach was first materialized in Holzbaur's libraries for supporting CLP over reals, as in CLP($\mathcal{R}$), as well as the rationals, CLP($\mathcal{Q}$) (Holzbaur 1995). On the CLP($\mathcal{FD}$) side, work progressed to replace the segregated "black box" architecture of CHIP by a transparent one (Hentenryck *et al.* 1994), in which the underpinnings of the constraint solver are described in user-accessible form (*indexicals*): such is the proposal discussed and implemented by Diaz and Codognet (1993), Carlson *et al.* (1994), and

[4] The European counterpart of MCC, see Section 2.7.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

Codognet and Diaz (1996). Having elementary constraints to compile to is an approach which has largely been adopted by the attributed variable-based Prolog implementations of CLP($\mathcal{FD}$), present in most Prolog systems. SICStus and GNU Prolog incorporate high-performance native implementations, which nevertheless follow this conceptual scheme.

*Constraint Handling Rules (1991, cf. Frühwirth 2009, p. xxi)* On the trail of providing finer-grained control over the implementation of constraints, Frühwirth (1992; 2009) introduced *Constraint Handling Rules* (CHR), in which syntactically enhanced Prolog clauses are used to describe and implement the progress of the constraint satisfaction process. CHR is both a theoretical formalism related to first-order and linear logic, and a rule-based constraint programming language that can either stand alone or blend with the syntax of a host language. When the host language is Prolog, CHR extends it with rule-based concurrency and constraint solving capabilities. Its multi-headed rules allow expressing complex interactions succinctly, through rule applications that transform components of a shared data structure: the "constraint store". A solid body of theoretical results guarantee best known time and space complexity, show that confluence of rule application and operational equivalence of programs are decidable for terminating CHR programs, and show that a terminating and confluent CHR program can be run in parallel without any modification and without harming correctness. Applications are multiple, since CHR, rather than constituting a single constraint solver for a specific domain, allows programmers to develop constraint solvers in any given domain.

It should be noted that CLP has spurred the emergence of a very active research field and community, focusing on Constraints, with or without the Logic Programming part.

## *2.9 Tabling*

Tabling is a technique first developed for natural language processing, where it was called Earley parsing (Kay 1967; Earley 1970). It consists of storing in a table (a.k.a. chart in the context of parsing) partial successful analyses that might come in handy for future reuse

Its adaptation into a logic programming proof procedure, under the name of Earley deduction, dates from an unpublished note from 1975 by David H.D. Warren, as documented by Pereira and Shieber (1987). An interpretation method based on tabling was later developed by Tamaki and Sato (1986), modeled as a refinement of SLD-resolution.

David S. Warren[5] and his students adopted this technique with the motivation of changing Prolog's semantics from the completion semantics to the minimal model semantics.

Indeed, the completion semantics cannot faithfully capture important concepts such as the transitive closure of a graph or relation. The minimal model semantics is able to capture such concepts. Moreover, tabled execution terminates for corresponding programs such as for the transitive closure of a cyclic graph. This makes Prolog more declarative.

Tabling consists of maintaining a table of goals that are called during execution, along with their answers, and then using the answers directly when the same goal is

[5]Not to be confused with David H.D. Warren.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

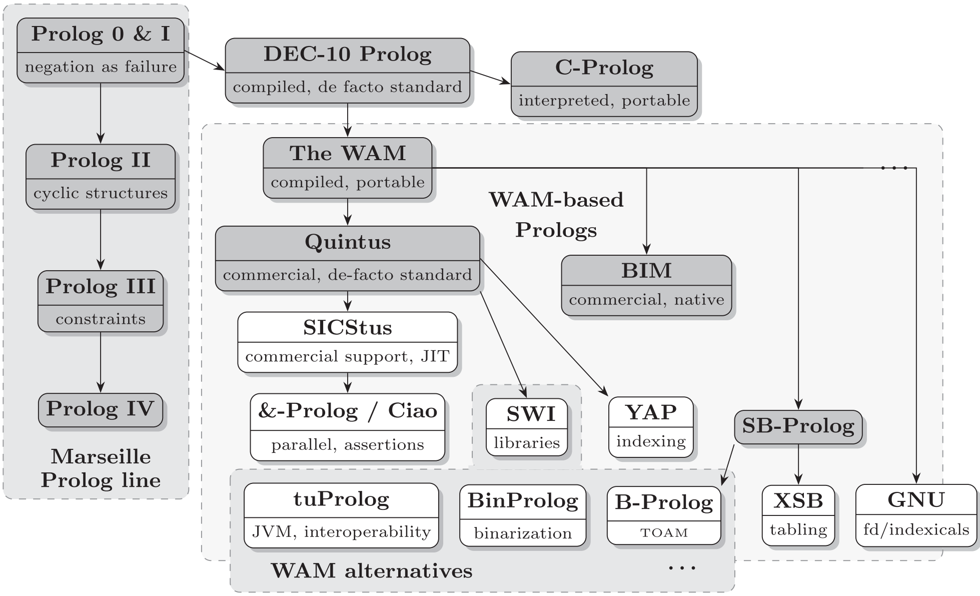


Fig. 2. Prolog Heritage. Systems with a dark gray background are not supported any more. Arrows denote influences and inspiration of systems. The bottom section of each block includes just some highlight(s); see the text for more details. Quick legend: JIT = "Just in Time [Compiler]", JVM = "Java Virtual Machine", TOAM = "Tree-Oriented Abstract Machine".

subsequently called. Tabling gives a guarantee of total correctness for any (pure) Prolog program without function symbols, which was one of the goals of that work.

*XSB Prolog (1994)* The concept of tabled Prolog was introduced in XSB Prolog (Sagonas *et al.* 1994). This resulted in a complete implementation (Rao *et al.* 1997) of the *well-founded semantics* (Van Gelder *et al.* 1991), a three-valued semantics that represents values for true, false and unknown.

### *2.10 Prolog implementations after the WAM*

As mentioned before, the WAM became the standard for Prolog compilers and continues to be today. In this section, we review how the main Prolog systems developed more or less until the appearance of the ISO standard. An overview of the most influential Prolog systems and their influence is given in Figure 2.

#### *2.10.1 Early proprietary Prologs*

The WAM aroused much interest and many Prolog implementations started out as an exercise to properly understand it while others were aimed directly at commercialization. Three of the early commercial Prolog systems were Quintus Prolog, BIM-Prolog, and VM/Prolog by IBM.

*Quintus Prolog (1984)* Shortly after the WAM was proposed, *Quintus* Computer Systems was founded by David H.D. Warren, William Kornfeld, Lawrence Byrd, Fernando Pereira

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

and Cuthbert Hurd, with the goal of selling a high-performance Prolog system for the emerging 32-bit processors. One of the earliest documents available about Quintus is a specifications note (Warren *et al.* 1984). Quintus used the DEC-10 Prolog syntax and built-ins and was based on the WAM. Currently, Quintus is distributed by SICS (2021). Quintus quickly became the *de facto* standard at the time, influencing most Prolog systems that were created afterwards. For many years, it offered the highest-performance implementation and was the standard in terms of syntax, built-ins, libraries, and language extensions. Its success inspired many more Prolog systems to emerge, including the ones we discuss below.

*BIM-Prolog (1984)* In 1984, BIM (a Belgian software house) in cooperation with the Katholieke Universiteit Leuven, and under the guidance of Maurice Bruynooghe, started a project aiming at implementing a commercial Prolog system: BIM-Prolog. A collection of documents is still available on the internet (Bruynooghe 2021) and notable contributions were made among others by Bart Demoen, Gerda Janssens, André Mariën, Alain Callebaut, and Raf Venken. BIM Prolog was funded by the Belgian Ministry of Science Policy and was based on the WAM. One of the earliest resources available is an internal report (Janssens 1984). BIM-Prolog developed into a system with the first WAM-based compiler to native code (as opposed to, e.g., threaded code by Quintus), with interfaces to several database systems (Ingres, Unify, etc.), a graphical debugger in the style of dbx-tool, a bidirectional interface to C, decompilation even of static code, and multiargument indexing of clauses – which overcame the common practice of indexing Prolog clauses via their head's first argument alone. Its first release was on SUN machines, and later it was ported to Intel processors. BIM was involved in the later ISO standardization effort for Prolog. BIM went out of business in 1996.

*IBM Prolog (1985)* Several Prolog systems that ran on specific IBM hardware remained unnamed and were referred to as IBM Prolog. Here, we focus on Prolog systems distributed by IBM. In 1985, IBM announced a tool named VM Programming in Logic or VM/Prolog (Symonds 1986), which was its Prolog implementation for the 370, focusing on AI research and development. Its development started in 1983 by Marc Gillet at IBM Paris according to Van Roy (1994). In 1990, a 16-bit Prolog system for OS/2 was announced, including a database and dialog manager. It was able to call programs written in other IBM languages, such as macro assembler, C/2 and REXX scripts. While its syntax was based on its predecessor, it also provided support for the Edinburgh syntax considering the ongoing ISO standard development. It was maintained until 1992, at which time it was succeeded to by the 32-bit implementation IBM SAA AD/Cycle Prolog/2 (Benichou *et al.* 1992). IBM withdrew from the market in 1994.

*SICStus Prolog (1986)* A preliminary specification of SICStus existed in 1986 (Carlsson 1986), drawing inspiration from DEC-10 Prolog as well as from Quintus. As already mentioned, SICStus was at first an open-source project aimed at supporting or-parallelism research, and became the basis of much other research, turning into an invaluable tool for other research groups as well as for commercial applications. In addition to the open-source nature, powerful reasons for this popularity were the compatibility with the DEC-10 and Quintus Prolog de-facto standards, very good performance, and compact generated code. Execution profiling and native code compilation were also added later.

At the end of the 80s, the Swedish Funding Agency and several companies funded the industrialization of SICStus, which eventually became a commercial product. In 1998, SICS acquired Quintus Prolog and a number of its features made their way into newer SICStus Prolog versions. SICStus is ISO-conforming and provides support for web-based applications. It also supports several constraint domains, including a powerful finite domain solver. Notably, SICStus is still alive and well as a commercial product, and its codebase is still actively maintained.

#### *2.10.2 Open-source and research-driven Prolog systems based on the WAM*

Further, generally open-source Prologs were developed featuring extensions and alternatives arising from the needs of specific application areas or from experimentation with issues such as control, efficiency, portability, global analysis and verification, and, more recently, interoperability and multparadigm support and interaction. This section examines some of these.

*YAP Prolog (1985)* As further discussed by Costa *et al*. (2012), the YAP Prolog project started in 1985. In contrast to other systems discussed here, early versions of it were cast as a proprietary system which was later released as open source software. Luís Damas, the main developer, wrote a Prolog compiler and parser in C (still used today). Since the emulator was originally written in m68k assembly, the result was a system that was and felt fast. As the 68k faded away, Damas developed a macro language that could be translated into VAX-11, MIPS, Sparc, and HP-RISC. Unfortunately, porting the emulator to the x86 was impossible, so a new one was designed in C, making it also easier for some excellent students to contribute. Rocha implemented the first parallel tabling engine (Rocha *et al*. 2005) and Lopes the first Extended Andorra Model emulator (Lopes *et al*. 2012). This work was well received by the community, but proved difficult to use in scaling up real applications. The problem seemed to be that many YAP applications used Prolog as a declarative database manager. In order to support them, the team developed JITI (Costa *et al*. 2007), a just-in-time multiargument indexer that uses *any* instantiated arguments to choose matching clauses, hoping to avoid shallow backtracking through thousands or millions of facts. JITI's trade-off is extra space – the mega clause idea reduces overhead by compacting clauses of the same type into an array (Costa 2007), and the exo-emulation saves space by having a single "smarter" instruction to represent a column of a table (Costa and Vaz 2013).

*Ciao Prolog (1993) a.k.a. &-Prolog (1986)* As mentioned before, &-Prolog started in 1986, based initially on early versions of SICStus. The early 90s brought much evolution, leading to its re-branding as Ciao Prolog (Hermenegildo and CLIP Group 1993). One of the main new aims was to point out future directions for Prolog, and to show how features that previously required a departure from Prolog (such as those in, e.g., Mercury, Gödel, or AKL, and from other paradigms), could be brought to Prolog without losing Prolog's essence. A new module system and code transformation facilities were added that allowed defining many language extensions (such as constraints, higher-order, objects,

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

functional notation, other search and computation rules, etc.) as libraries in a *modular* way (Hermenegildo *et al.* 1994; Hermenegildo *et al.* 1996; Cabeza and Hermenegildo 2000), and also facilitated global analysis. Also, the progressively richer information inferred by the PLAI analyzers was applied to enhancing program development, leading to the Ciao assertion language and pre-processor, CiaoPP (Hermenegildo *et al.* 1999; Puebla *et al.* 2000a; Hermenegildo *et al.* 2005), which allowed *optionally* specifying and checking many properties such as types, modes, determinacy, nonfailure, or cost, as well as auto-documentation. A native, optimizing compiler was also developed, and the abstract machine was rewritten in a restricted dialect of Prolog, ImProlog (Morales *et al.* 2005; 2016).

*SB-Prolog (1987)* SB-Prolog was a Prolog system that, according to the CMU Artificial Intelligence Repository (1995) became available in 1987, and had been started as an exercise to understand the WAM. It was made freely available in the hope that its source code would be of interest to other Prolog researchers for understanding, use, and extension. Indeed, it became the foundation of two other Prolog systems, XSB (cf. Section 2.9) and B-Prolog (cf. Section 2.11). The goal of XSB Prolog (Sagonas *et al.* 1994) at its release in 1993 was to allow new application areas of Prolog. As an example, a recent survey of its applications to NLP is given by Christiansen and Dahl (2018).

*Andorra I (1991)* Sometimes also referred as *Andorra Prolog*, Andorra I is a Prolog system developed by Costa *et al.* (1991). This system exploited both (deterministic) AND-parallelism and OR-parallelism, while also providing a form of implicit coroutining, and ran on the shared-memory multiprocessors of the time, the Sequent Symmetry. OR-parallelism was supported by using binding arrays to access common variables and the Aurora scheduler (Lusk *et al.* 1990). The implementation of AND-parallelism, that (dynamically) identified which goals in a clause are determinate and can be evaluated independently in parallel, came to be known as the *Andorra Principle* and is akin to the concept of *sidetracking* (Pereira and Porto 1979), itself a form of coroutining. Adherence to Prolog operational semantics meant that subgoal order sometimes needs to remain fixed and, also, that a cut may impact parallel execution. Implementing an efficient Prolog system which could exploit both forms of parallelism led to difficulties, for which solutions would follow in the guise of different computational models, namely the Extended Andorra Model (Warren 1990) and the Andorra Kernel Language (AKL) (Janson and Haridi 1991).

*GNU Prolog (1999) a.k.a. Calypso (1996), and wamcc (1992)* As stated on the GNU Prolog home page (GNU Prolog 2021), the development of GNU Prolog started in January 1996 under the name Calypso. A few years later, in 1999, the first official release of GNU Prolog saw the light (Diaz *et al.* 2012).

GNU Prolog is derived from wamcc (Codognet and Diaz 1995), a system developed in 1992–1993 as a foundational framework for experiments on extensions to Prolog, such as intelligent backtracking techniques, coroutining, concurrency, and constraints. The `wamcc` Prolog system was designed to be easily maintainable, lightweight, portable, and freely available, while still reasonably fast. Its approach consisted in using the WAM as an intermediate representation in a multipass compilation process, producing C code which was

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

subsequently compiled by GCC, to yield native code which was then linked to produce an executable. `wamcc` was used as the basis for the development of CLP($\mathcal{FD}$) (Codognet and Diaz 1996), which introduced transparent user-defined propagators for finite domain ($\mathcal{FD}$) constraint solving (CLP). In a later stage, when the ISO standard was being introduced in 1995, the CLP($\mathcal{FD}$) system was redesigned to become more standards-compliant and to increase its compile-time performance to compete with that of interpreted Prolog systems. As C was used as the intermediate language, compiled Prolog programs would map to considerably larger C programs which were very slow to compile using GCC, with little benefit as the code had very explicit and low-level control flow (e.g., labels and gotos.) This situation led to the replacement of C by a much simpler specialized miniassembly language (Diaz and Codognet 2000) as the intermediate language for compiling Prolog. After it was approved by the Free Software Foundation, this system became GNU Prolog.

*ECL$^i$PS$^e$ (1993)* The earliest resource about the *ECL$^i$PS$^e$* logic programming system (Schimpf and Shen 2012) is the technical paper by Wallace and Veron (1993). Originally, it was an integration of ECRC's SEPIA, an extensible Prolog system, Mega-Log, an integration of Prolog and a database, and (parts of the) CHIP systems. It was then further developed into a Constraint Logic Programming system with a focus on hybrid problem solving and solver integration. It is data driven, allowing for array syntax and structures with field names. The system also contains a logical iteration construct that eliminates the need for most of the basic recursion patterns. Since September 2006, ECL$^i$PS$^e$ is an open-source project owned by Cisco.

### 2.11 Alternatives to the WAM

While most Prolog systems are based on the WAM, some alternatives were explored and are also used to date. In the following, we briefly describe some impactful implementations.

*SWI-Prolog (1986)* As stated on the project home page (SWI-Prolog 2021), SWI-Prolog (Wielemaker *et al.* 2008) started in 1986 as a recreational project, though the main reason for its development was the lack of recursive calls between Prolog and C in Quintus. Hence, it soon gained a significant user community (which may be the largest user base for a Prolog system today), partially because it spread over the academic ecosystem as it could be used by universities for teaching.

At its very core, SWI-Prolog is based on an extended version of the ZIP virtual machine by Bowen *et al.* (1983), that is, a minimal virtual machine for Prolog implementing a simple language consisting of only seven instructions. SWI-Prolog-specific extensions aim at improving performance in several ways: ad hoc instructions are introduced to support unification, predicate invocation, some frequently used built-in predicates, arithmetic, control flow, and negation-as-failure. Prolog can easily be compiled into this language, and the abstract machine code is easily decompiled back into Prolog. This feature is often exploited to interleave compiled and interpreted code execution – which may be needed, for example, in debug mode.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

In the past, SWI-Prolog has incorporated technologies first implemented in other systems, for example, delimited continuations following the design by Schrijvers *et al.* (2013), BinProlog-like continuation passing style as designed by Tarau (1992), or tabling based on XSB.

*LIFE (1991)* LIFE (Logic, Inheritance, Functions, and Equations) was an experimental language developed by Hassan Aït-Kaci and his group, first at MCC and later at the DEC Paris Research Lab (Aït-Kaci and Podelski 1991; Aït-Kaci 1993). It extended Prolog with type inheritance and functions. Functions were implemented using *residuation*, where they are delayed until their arguments are sufficiently instantiated. Also, extensions to the WAM were developed for implementing unification of feature terms in order-sorted theories.

*BinProlog (1992)* Paul Tarau started work on BinProlog in 1991, researching alternatives to the then relatively mature WAM. The first trace of BinProlog in the literature is the paper from Tarau (1992). In particular, Tarau was interested in a simpler WAM and in exploring what could be removed without too harsh performance losses. He was also specializing the WAM for the efficient execution of binary programs by compiling a program's clauses into binary clauses, passing continuations explicitly as extra arguments. This approach has advantages for implementing concurrency and distributed execution and related mechanisms such as engines. However, it also has a cost, since it conflicts with tail- and last-call optimization. It is thus a veritable WAM alternative trading efficiency for flexibility.

BinProlog supports multithreading and networking. With human language processing needs in mind, hypothetical reasoning was built-in as well. This took the form of intuitionistic and linear (affine) implication plus a novel form of implication (timeless assumptions, designed and first meta-programmed by Verónica Dahl for dealing with backward anaphora) (Dahl and Tarau 1998), later incorporated as well into Constraint Handling Rule Grammars (CHRG) and into Hyprolog (which will be discussed in Section 3.5.1).

*B-Prolog (1994)* The first version of *B-Prolog* (Zhou 2012) was released in 1994, as the reader may verify by scrolling the version history publicly available on the B Prolog Updates Note (2021). It uses a modified version of the WAM named TOAM (Tree-Oriented Abstract Machine). The TOAM differs from the WAM in three key aspects: first, arguments are passed through the stack rather than registers. Second, it uses a single stack frame for a predicate call rather than two (Zhou 1996). Lastly, the first-argument indexing of the WAM is improved by using matching trees inspired by Forgy's Rete algorithm (Forgy 1989).

*tuProlog (2001) and* 2P-Kt *(2021)* Denti *et al.* (2001) proposed tuProlog, another successful attempt at supporting Prolog without compiling to the WAM. It consists of a lightweight Prolog implementation targeting the Java Virtual Machine (JVM). In particular, tuProlog relies on an original state-machine-based implementation of the SLDNF resolution mechanism for Horn Clauses, aimed at interpreting Prolog programs on the fly (Piancastelli *et al.* 2008), without requiring any compilation step.

More recently, the whole project has been re-written by Ciatto *et al.* (2021a) as a Kotlin multiplatform project (codename 2P-Kt) targeting both the JVM and JS platforms. The Prolog state machine has been slightly extended as well to support the lazy enumeration of data streams via backtracking, as discussed by Ciatto *et al.* (2021b).

### *2.12 Early steps in building the community*

After the first successes in Marseille, the availability of Prolog systems such as DEC-10 Prolog and, especially, the explosion of widely available Prolog systems that followed the appearance of the WAM, an international community grew around Prolog and LP.

The First International Logic Programming Conference (ICLP) was held in Marseille, in September 1982 and the second in Uppsala, Sweden in July 1984. There have also been editions of the Symposium on Logic Programming and the North American Conference of Logic Programming.

In 1984, the first issue of the Journal of Logic Programming (JLP) with Alan Robinson as founding Editor-in-Chief was published, marking the solidification of the field of logic programming within computer science.

The Association for Logic Programming (ALP) was founded at the 3rd ICLP conference in 1986 and JLP became its official journal. The ALP is the main coordination body within the community, sponsoring conferences and workshops related to logic programming, granting prizes and honors, providing support for attendance at meetings by participants in financial need, etc. The ALP was followed by other country-specific associations.

In 2001, the first issue of Theory and Practice of Logic Programming (TPLP) was published. TPLP is a successor to JLP established in response to considerable price increases by Elsevier and to provide the logic programming community with a more open-access journal (Apt 2001), and took over from JLP as the official journal of the ALP. In 2010 ICLP started publishing its proceedings directly as special issues of TPLP, pioneering (together with VLDB and SIGGRAPH) a tendency that is now followed by top conferences in many areas (Hermenegildo 2012).

## 3 Part II: The current state of Prolog

Currently, there are many Prolog implementations – the `comp.lang.Prolog` FAQ (2021) lists 35 different free systems at the time of writing (last modified May 2021). While many of those are not maintained anymore or have even become unavailable, many others are actively maintained and extended regularly. Thanks to the ISO-standardized core of Prolog, significant core functionality is shared among the different Prolog systems. However, implementations also diverge from the standard in some details. For the nonstandardized interfaces and the additional libraries, differences become more marked, which leads to incompatibilities between different Prologs. In addition, most systems have incorporated functionality that goes well beyond the Prolog ISO-standard.

In the following, the background, benefits, and shortcomings of the ISO standard, as well as contributions based on it, are discussed (Section 3.1). Section 3.2 then discusses the more active current Prolog implementations and what renders them unique, documenting their visions for Prolog and their main development or research focus. Section 3.3

analyzes which nonstandard features are available throughout the many current Prolog systems, and what the state of these features is with a special emphasis on the differences. Section 3.4 gives preliminary conclusions on the current state of Prolog features, whether they are important for portability, and if these differences can be easily reconciled. Finally, Section 3.5 takes a look over the horizon to discuss which nonstandard features have inspired other successful languages, as well as other interesting concepts that may be or become relevant for the Prolog community.

### 3.1 The ISO standard and portability of Prolog code

As discussed in Section 2, the success of the WAM gave rise to many Prolog systems in the 80s and early 90s. Yet, at this point, the Prolog language was evolving without central stewardship. While originally the two traditional camps in Marseille and Edinburgh steered their developments at their respective location, many Prolog systems around the world aimed for extensions and new uses of the language. However, the Edinburgh/DEC-10 syntax and functionality became progressively accepted as the de-facto standard, helped by the wide dissemination of systems such as C-Prolog and later the very influential Quintus system. Many popular systems, such as SICStus, YAP, Ciao, SWI, and XSB tried to maintain compatibility with this standard.

*The Core ISO Standard:* Work on a Prolog standard started in 1984 and was organized formally in 1987 (Neumerkel 2013). Its major milestone was the ISO Prolog standard in 1995 (ISO/IEC 13211-1 1995) that we will refer to as Part 1 or *core* standard (Deransart *et al*. 1996). It solidified the Edinburgh/Quintus de-facto standard, and greatly helped establish a common *kernel*. Furthermore, it greatly increased the confidence of users in the portability of code between Prolog systems (especially important among industrial users), and the hope of having libraries that would be able to build on top of rich functionality and be shared as well. This was achieved to some extent, and indeed many libraries (for example, the excellent code contributed to the community by Richard O'Keefe Johnson and Rae 1983; Ireson-Paine 2010) are present today in almost identical form in many systems.

However, adoption of the ISO standard was not a painless process. It was a compromise among many parties that did not describe any particular single system and, thus, compliance did force the different Prolog systems to make changes that, even if often minor, were not always backwards compatible. This made many Prolog vendors face the difficult decision of choosing between fully adopting the standard and breaking existing user code, or allowing slight deviations that made user code remain compatible. Therefore, it took understandably some time for the standard to be adopted. At some point, the community even became concerned that the ISO standard might not be taken seriously, and some voices asked whether it *should not* be taken seriously. In a post in the ALP newsletter, Bagnara (1999) pointed out that many implementations differed from the standard in at least some way, and that those differences were poorly documented. Other shortcomings were pointed out by several Prolog system main contributors, such as, for example, by Carlsson and Mildner (2012), Diaz *et al*. (2012), and Wielemaker and Costa (2011). However, both aspects, adoption of the core standard and precise documentation of where each system departs from the standard, have improved progressively. In practice, most

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

systems tend to follow the kernel part of the standard, and many systems continue to evolve even today to complete aspects in which they are not compliant.

In addition to the battery of tests provided by Deransart *et al.* (1996), a useful tool to analyze compliance with the core standard was developed in by Szabó and Szeredi (2006), where a test suite of about 1000 tests was developed and run on several Prolog systems, offering detailed test results. This suite is widely used for detecting areas of noncompliance in Prolog systems and was useful to improve compliance. It has also been useful as a means for detecting areas in which the ISO standard can be improved, based on failing tests, ambiguities, typos, and inconsistencies that have been found in the standard document. These tests and others were coded as assertions and unit tests in the Ciao assertion language by Mera *et al.* (2009); encoding and testing compliance (both statically and dynamically) was one of the design objectives of the Ciao assertion language. Ulrich Neumerkel has greatly contributed by testing and analyzing a comprehensive set of features and programs over many systems (Neumerkel 1992; 1994; Hoarau and Mesnard 1998; Mesnard *et al.* 2002; Triska *et al.* 2009). His attention to detail has been essential in bringing ISO standard compatibility to many implementations.[6]

The fact that some aspects of the core standard still require additional work has been pointed out also by several authors. For example, Moura (2005), in another issue of the ALP newsletter, points out that revisions of the standard are necessary to close gaps in the built-in predicates (as well as other issues like exceptions, scopes of declared operators, or meta-predicates). He demands a strong standard and also requests mature libraries. In later reflections during a special session at ICLP 2008, Moura (2009b) reported on his interesting experiences when implementing Logtalk, which, as mentioned before, is portable between many Prolog compilers. Over the course of a decade, Moura was able to locate hundreds of bugs and incompatibilities between all the targeted Prologs. However, Moura also asserts that Prolog developers[7] have generally addressed these issues, and there has been considerable improvement in portability. Another very valuable result of these efforts by Moura is a very comprehensive set of compatibility tests and libraries covering the target Prolog systems.

*Beyond the Core Standard: The Module System* Despite leaving room for improvement, the adoption of the core standard can be considered a reasonable success. However, an important remaining shortcoming is that it does not address many features that modern Prolog systems offer, such as modules. The Prolog module system was addressed by the ISO standardization group in the second part of the standard (ISO/IEC 13211-2 2000) which appeared five years later. However, while the core standard was wise enough to reflect the de-facto standards of the time, this second part proposed a module system that was a radical departure from any module system used by any Prolog at the time. Instead, by the time this part came out, the Prolog developer community had already settled for the de-facto module standard, which was the Quintus module system. Thus, this second part of the standard was largely ignored. Fortunately, the Quintus-like

[6] On this topic, Neumerkel maintains the site `https://www.complang.tuwien.ac.at/ulrich/iso-prolog/`.

[7] We will use the term Prolog *developers* to refer to developers *of* Prolog. Developers *in* Prolog will be refered to as Prolog *programmers*.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

module system is still widely supported currently. Some systems include extensions to this de-facto standard that while sometimes incompatible, generally preserve backward compatibility. This has allowed the development of some libraries that rely only on the Prolog ISO core, which are present in almost identical form in many systems. Only a few systems have module systems with radically differing semantics or no module system at all.

*Beyond the Core Standard: Libraries and Extensions* Other aspects that are outside the core standard are libraries and many extensions such as coroutining, tabling, parallel execution, exceptions, and constraints.

While, as mentioned before, many libraries are present in most systems, unfortunately the adoption of the de-facto module standard did not result in the establishment of a full set of common libraries across systems. Often, nonstandard Prolog features are required by more involved libraries, and in other cases, similar yet independent developments were not reconciled.

Schrijvers and Demoen (2008) published an article aptly named *Uniting the Prolog Community* discussing issues related to portability and incompatibility, specially in the structure and content of libraries. This prompted two Prolog implementations, SWI-Prolog and YAP, to work together more closely on the issue. A basic compatibility was established, that allowed writing Prolog programs in a portable manner by abstraction, emulation, and a small amount of conditional, dialect-specific code (Wielemaker and Costa 2011). The overall approach works fairly well and, as demonstrated in two case studies (reported in the same paper), large libraries can be ported between both Prologs with manageable effort.

Some time later, in 2009, the Prolog Commons Working Group (2021) was established, with the objective of developing a common, public domain set of libraries for systems that support the Prolog language. The group met a number of times in Leuven (Swift 2009), with attendance from developers of most major Prolog systems, and some useful progress was made during this period. Leading Prolog system developers worked toward a set of common libraries, a common mechanism for conditional code compilation, and closer documentation syntax. During early discussions other interesting topics were raised, such as the necessity of a portable bidirectional foreign language interface for C. The work of the group also resulted in 17 libraries and 8 more in development.

While the standard initiatives like the Prolog Commons Working Group have been taking major steps forward in the coordination of the Prolog developer community, there has been unfortunately less activity lately on standardization. Clearly, a higher involvement of the Prolog developer community in the evolution of the standard and/or alternative standardization efforts such as the Prolog Commons seems to be a necessity.

### *3.2 Rationales and unique features of Prolog implementations*

Naturally, there are different interests between the Prolog systems that are maintained for commercial usage and those maintained for research: while the former generally seek maturity and stability, the latter generally concentrate on advancing the capabilities and uses of the language. This divergence of interests raises the question of what features *do* work similarly between Prologs, what *can* be adapted, and what can be considered as

Table 1. *Unique features and Foci of Prolog systems*

| System | Uniqueness |
|---|---|
| B-Prolog | action rules, efficient CLP supporting many data structures |
| Ciao | multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (types, modes), performance/ scalabilty, language interfaces, parallelism |
| ECL$^i$PS$^e$ | focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog |
| GNU Prolog | extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs |
| JIProlog | semantic intelligence/NLP applications |
| Scryer | new Prolog in development, aims at full ISO conformance |
| SICStus | commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation |
| SWI-Prolog | general-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL$^i$PS$^e$ and XSB |
| tuProlog | bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library |
| XSB | commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming) |
| YAP | focus on scalability, advanced indexing, language integrations (Python, R), integration of databases |

a "de-facto" standard that is valid today. Along this line, we argue that two interesting questions deserving attention are the following:

- can maintainers of Prolog systems agree on additional features and common interfaces?
- can the Prolog Commons endeavor or similar efforts be continued?

Accordingly, this section takes another look at currently active Prolog implementations. In contrast to Section 2, it ignores their initial motivation and historic development, concentrating on their current development foci and unique features. A brief summary is presented in Table 1 and expanded in the following.

*B-Prolog* (Zhou 2012; BProlog Homepage 2021) is a high-performance implementation of ISO-Prolog. It extends the language with several interesting concepts, such as action rules (Zhou 2006), which allow delayable subgoals to be activated later. B-Prolog also offers an efficient system for constraint logic programming that supports many data structures.

*Ciao Prolog* (Hermenegildo *et al.* 2012; Ciao Prolog Homepage 2021) is a general-purpose, open source, high-performance Prolog system which supports the ISO-Prolog and other de-facto standards, while at the same time including many extensions. A characteristic feature is its set of program development tools, which include static and dynamic verification of program assertions (see Section 3.3.3), testing, auto-documentation, source debuggers, execution visualization, partial evaluation, or automatic parallelization.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

Another characteristic feature is extensibility, which has allowed the development of many LP and multiparadigm language extensions which can be turned on and off at will for each program module, while maintaining full Prolog compatibility. Other important foci are robustness, scalability, performance, and efficiency, with an incremental, highly optimizing compiler, that produces fast and small executables. Ciao also has numerous libraries and interfaces to many programming languages and data formats.

*ECL$^i$PS$^e$* (Wallace and Schimpf 1999; Apt and Wallace 2007; ECLiPSe Prolog Homepage 2021) is a system that aims for backward compatibility with ISO Prolog (and, to some extent, compatibility with the dialects of Quintus, SICStus and SWI-Prolog), but also tries to evolve the language. Its research focus is constraint logic programming. The system integrates the popular MiniZinc constraint modeling language (Nethercote *et al.* 2007), by means of a library that allows users to run MiniZinc models, as well as third-party solvers. While ECL$^i$PS$^e$ is open-source software, commercial support is available.

*GNU Prolog* (Diaz *et al.* 2012; GNU Prolog 2021) is an open-source Prolog compiler and extensible constraint solver over finite domains ($\mathcal{FD}$). It compiles to native executable code in several architectures, by means of an intermediate platform-independent language which reduces compilation time. GNU Prolog strives to be ISO-compliant and compiled programs are lightweight and efficient, as they do not require a run-time interpreter. The system's design eases the development of experimental extensions and attains good performance, despite being built on a simple WAM architecture with few optimizations and a straightforward compiler.

*JIProlog* (Chirico 2021) was the first Prolog interpreter for a Java platform and, some years later, it was the first Prolog interpreter for a mobile platform with its implementation for J2ME. Its strengths include bidirectional Prolog-Java interoperability (meaning that Java programs can call Prolog and vice-versa), the possibility to let Prolog programs interoperate with JDBC-compliant data base management systems, and the possibility to run the Prolog interpreter on Android. Along the years, JIProlog has been exploited in the construction of expert systems, as well as semantic web or data mining applications.

*Scryer Prolog* (2021) is a quite recent Prolog implementation effort whose WAM-based abstract machine is written in the Rust language. Scryer is open source and aims for full ISO compliance. Since Scryer Prolog is a new, from scratch implementation, it has the opportunity to select different trade-offs and implementation choices. Scryer is still heavily in development at the time of writing, and thus its features are in relative flux. We have thus not included it in the feature overview table (Table 2). Nevertheless, it does already have at least preliminary support for features such as modules, tabling, constraint domains, indexing, attributed variables, and coroutines. The reader is directed to the evolving Scryer Prolog documentation to follow-up on this system.

*SICStus Prolog* (Carlsson 1986; Carlsson and Mildner 2012; SICStus Prolog Homepage 2021) is now a commercial Prolog system. It adheres to the ISO standard and has a strong focus on performance and stability. An additional trait of the system is its sophisticated constraint system, with advanced libraries and many essentials for constraint solvers, such as coroutines, attributed variables, and unbounded integers. The `block` coroutining declaration is particularly efficient. It also incorporates many of the characteristics, features, and library modules of Quintus Prolog. Since release 4.3, SICStus also contains a JIT (just-in-time) compiler to native code, but currently has no multithreading or tabling support. SICStus is used in many commercial applications – cf. Carlsson and Mildner (2012, Section 6).

*SWI-Prolog* (Wielemaker *et al*. 2012; SWI Prolog Homepage 2021) is a general-purpose Prolog system, intended for real-world applications. For this, it has to be able to interface with other (sub-)systems. Thus, the development focus lies on multithreaded programming, implementations of communication network protocols such as HTTP, and on libraries that can read and write commonly used data formats such as RDF, HTML, XML, JSON, and YAML. Notably, for the last two formats, specific data structures need to be supported, which has motivated the divergence from the ISO standard in favor of real strings, dictionaries, distinguishing the atom `'[]'` from the empty list `[]`, and non-normal floating-point numbers (Inf, NaN). Strings, extended floating-point numbers, and support for rational numbers have been synchronized with ECL$^i$PS$^e$. Its top priorities are robustness, scalability and compatibility with both older versions of SWI-Prolog and the ISO standard, as well as with YAP, ECL$^i$PS$^e$ (data types), and XSB (tabling).

*tuProlog* (Denti *et al*. 2001; tuProlog Home 2021) is a relatively recent, research-oriented system which is the technological basis of several impactful works at the edge of the multiagent systems and logic programming areas, such as TuCSoN (Omicini and Zambonelli 1998), ReSpecT (Omicini and Denti 2001), and LPaaS (Calegari *et al*. 2018). The main purpose of tuProlog is to make Prolog and LP ubiquitous (Denti *et al*. 2013). It provides basic mechanisms such as knowledge representation, unification, clause storage, and SLDNF resolution *as a library* via multiplatform interoperability (e.g., to JVM, .NET, Android, and iOS) and multi-paradigm integration. Further, tuProlog also supports the direct manipulation of objects from within Prolog. Recent research efforts are focused on widening the pool of *(i)* logics and inferential procedures such as argumentation support (Pisano *et al*. 2020) and probabilistic LP, *(ii)* platforms it currently runs upon, for example, JavaScript (tuProlog Home 2021), and *(iii)* programming paradigms and languages it is interoperable with, cf. Ciatto *et al*. (2020).

*$\tau$Prolog* (2021), also referred to as Tau Prolog, is another noteworthy implementation of Prolog focusing on easing the use of logic programming in the Web. In particular, it provides a JavaScript-native library that facilitates the use of Prolog in web applications, both from the browser- and the server side. In other words, $\tau$Prolog pursues a similar intent with respect to tuProlog and JIProlog: bringing Prolog interpreters to high-level platforms and languages, except it focuses on another platform, JavaScript. Accordingly, $\tau$Prolog makes it very easy to run a Prolog interpreter in a web page, even without a server behind the scenes.

*XSB* (Sagonas *et al*. 1993; 1994; Warren 1998; XSB Prolog Homepage 2021) is a research-oriented system, but its development is also influenced by continued use in commercial applications. Its most distinctive research contribution is tabled resolution (Swift and

Warren 2012) which has, since, been adopted in other systems. In the XSB manual, the developers explicitly refrain from calling it a Prolog system, as it extends the latter with concepts such as SLG-resolution and HiLog programming. We return to tabling below in Section 3.3.3.

*YAP* (Costa *et al*. 2012) is a general-purpose Prolog system focused on scalability, mostly based on advanced clause indexing, and on integration with other languages, specifically Python and R. There is a strong interest in trying to make Prolog as declarative as possible, looking at ways of specifying control (such as types), and in program scalability by considering modules. There is also a long term goal of integrating databases into Prolog by having a driver that allows YAP to use a database as a Prolog predicate. In the future, YAP's strength lies in the ability to write and maintain large applications. Its team has worked on three key points toward this goal: interfacing with other languages (Angelopoulos *et al*. 2013), interfaces to enable collaboration between Prolog dialects, and tools for declaring and inferring program properties, such as types (Costa 1999).

### *3.3 Overview of features*

In this section, we survey the availability of features that are often appreciated by Prolog programmers. The goal is to find out whether there are commonalities and even a "de facto standard" with respect to features among most Prolog systems. An overview of which features are available in what Prolog system is given in Table 2. We give our conclusions regarding portability in Section 3.4. Due to the sheer number of existing Prolog systems, we consider only the more actively developed and mainstream ones. In the following, we will briefly discuss each surveyed feature, classifying them in four different groups: core features that usually cannot be reasonably emulated on top of simpler features; extensions to the language semantics and the execution model; libraries written on top of the core and (optionally, one or more) extensions; tools and facilities to debug, test, document, and perform static analysis.

*A Note on Portability:* The above classification sheds some light onto the challenges of attaining portability of sophisticated Prolog code. Compatibility at the core features (1) is relatively easy, and this enables the sharing of a substantial number of libraries (2). Extensions (3) represent a more complicated evolving landscape, where some of them require deep changes in the system architecture. It stands to reason that the existence of multiple Prolog implementations (or alternative *cores*) might be a *necessary* good step, that should be regarded as a healthy sign rather than an inconvenience. Nevertheless, this requires a periodic revisit, dropping what did not work and promoting cross-fertilization of ideas. On the other hand, tools (4), despite being more complex, have also the advantage of being more flexible: sometimes they can run on one system while still being usable with others (e.g., IDEs, documentation, analysis, or refactoring tools).

#### *3.3.1 Core features*

*Module System* As mentioned before, while most Prolog systems support structuring the code into modules, virtually no implementation adheres to the modules part of the ISO

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

Table 2. *Feature Overview of Several Maintained Prolog(-like) Systems. Constraint Logic Programming (CLP) abbreviations: $\mathcal{FD}$ = finite-domain, $\mathcal{Q}$ = rational numbers, $\mathcal{R}$ = real numbers, $\mathcal{B}$ = boolean variables. Indexing Strategy abbreviations: FA = first argument, N-FA = non-first argument, MA = multiple-argument, JIT = just-in-time, all = all aforementioned strategies*

| System | Open source | Modules | Non-standard data types | Foreign language interfaces |
|---|---|---|---|---|
| B-Prolog | ✗ | ✗ | arrays, sets, hashtables | C, Java |
| Ciao | ✓ | ✓ | ✗ | C, Java, Python, JavaScript |
| ECLiPSe | ✓ | ✓ | arrays, strings | C, Java, Python, PHP |
| GNU Prolog | ✓ | ✗ | arrays | C, Java, PHP |
| JIProlog | ✓ | ✓ | ✗ | Java |
| SICStus | ✗ | ✓ | ✗ | C, Java, .NET, Tcl/Tk |
| SWI | ✓ | ✓ | dicts, strings | C, C++, Java |
| $\tau$Prolog | ✓ | ✓ | ✗ | JavaScript |
| tuProlog | ✓ | ✗ | arrays | Java, .NET, Android, iOS |
| XSB | ✓ | ✓ | ✗ | C, Java, PERL |
| YAP | ✓ | ✓ | ✗ | C, Python, R |

| System | CLP | CHR | Tabling | Parallelism | Indexing | Type / Mode |
|---|---|---|---|---|---|---|
| B-Prolog | $\mathcal{FD}$, $\mathcal{B}$, *Set* | ✓ | ✓ | ✗ | N-FA | ✗ |
| Ciao | $\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$ | ✓ | ✓ | ✓ | FA, MA | ✓ |
| ECLiPSe | $\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, *Set* | ✓ | ✗ | ✓ | most suitable | ✗ |
| GNU Prolog | $\mathcal{FD}$, $\mathcal{B}$ | ✗ | ✗ | ✗ | FA | ✗ |
| JIProlog | ✗ | ✗ | ✗ | ✗ | undocumented | ✗ |
| SICStus | $\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$ | ✓ | ✗ | ✗ | FA | ✗ |
| SWI | $\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$ | ✓ | ✓ | ✓ | MA, deep, JIT | ✗ |
| $\tau$Prolog | ✗ | ✗ | ✗ | ✗ | undocumented | ✗ |
| tuProlog | ✗ | ✗ | ✗ | ✓ | FA | ✗ |
| XSB | $\mathcal{R}$ | ✓ | ✓ | ✓ | all, trie | ✗ |
| YAP | $\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$ | ✓ | ✓ | ✗ | FA, MA, JIT | ✗ |

| System | Coroutines | Testing | Debugger | Global Variables | Mutable Terms |
|---|---|---|---|---|---|
| B-Prolog | ✓ | ✗ | trace | ✓ | ✗ |
| Ciao | ✓ | ✓ | trace / source | ✓ | ✓ |
| ECLiPSe | ✓ | ✓ | trace | ✓ | ✗ |
| GNU Prolog | ✗ | ✗ | trace | ✓ | ✓ |
| JIProlog | ✗ | ✗ | trace | ✗ | ✗ |
| SICStus | ✓ | ✓ | trace / source | ✗ | ✓ |
| SWI | ✓ | ✓ | trace / graphical | ✓ | ✓ |
| $\tau$Prolog | ✗ | ✗ | ✗ | ✗ | ✗ |
| tuProlog | ✗ | ✗ | spy | ✗ | ✗ |
| XSB | ✓ | ✗ | trace | ✗ | ✗ |
| YAP | ✗ | ✗ | trace | ✓ | ✗ |

standard. Instead, most systems have decided to support as *de-facto* module standard the Quintus/SICStus module system. However, further convenience predicates concerning modules are provided by some implementations only and often have subtle differences in their semantics.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

Interesting cases include GNU Prolog which initially chose not to implement a module system at all, although a similar functionality was later brought in by means of contexts and dynamic native unit code loading (Abreu and Nogueira 2005); Logtalk which demonstrates that code reuse and isolation can be implemented on top of ISO Prolog using source-to-source transformation (Moura 2003); Ciao which designed a strict module system that, while being basically compatible with the *de-facto* standard used by other Prolog systems, is amenable to precise static analysis, supports term hiding, and facilitates programming in the large (Cabeza and Hermenegildo 2000; Stulova *et al*. 2018); and XSB, which offers an *atom-based* module system (Sagonas *et al*. 1994). The latter two systems allow controlling the visibility of terms in addition to that of predicates.

*Built-in Data Types* The ISO Prolog standard requires support for atoms, integers, floating-point numbers, and compound terms with only little specification on the representation limits, usually available as Prolog flags.

In practice, most limits evolve with the hardware (word length, floating-point units, available memory, and raw performance), open software libraries (e.g., multiple precision arithmetic), and system maturity. Since the standard does not specify minimum requirements for limits, special care must be taken in the following cases:

- *Integers* may differ between Prolog systems. E.g., a given system may not support arbitrary precision arithmetic. Furthermore, the minimum and maximum values representable in standard precision may be smaller than implied by word length (due to tagging).
- *Maximum arity* of compound terms may be limited (`max_arity` Prolog flag). Despite the arity of user terms usually falling within the limits, this is an issue with automatic program manipulation (e.g., analyzers) or libraries representing arrays as terms.
- *Atoms* have many implementation-defined aspects, such as their maximum length, number of character codes (such as ASCII, 8-bit or Unicode), text encoding (UTF-8 or other), whether the code-point 0 can be represented, etc.
- *Strings* are traditionally represented as list of codes or characters (depending on the value of `double_quotes` flag), or as dedicated data types. Although this can be regarderd as very minor issue, combining different pieces of code expecting different encodings is painful and error prone.
- *Garbage collection* of atoms may not be available. This may lead to nonportability due to resource exhaustion in programs that create an arbitrary number of atoms.
- *Floating point numbers* are not specified to be represented in a specific way. The IEEE double standard is most prevalent across all Prolog systems. However, support for constants such as `NaN`, `-0.0`, `Inf` as well as rounding behavior may differ. ECL$^i$PS$^e$, for example, does interval arithmetic on the bounds.

For convenience, systems may offer other data types by means of (nonstandard) extension of the built-in data types, for example, *rational numbers* (useful for the implementation of CLP($\mathcal{Q}$)), key-value dictionaries, and compact representations of strings. There is no consensus on those extensions or portable implementation mechanisms, thus more work is needed in this area.

*Foreign (Host) Language Interface* Like any programming language, Prolog is more suited for some problems than for others. With a foreign language interface, it becomes easier to embed it into a software system, where it can be used to solve part of a problem or access legacy software and libraries written in another language. Since this is also a nonstandard feature, the interfaces themselves, as well as the targeted languages, differ quite a lot among different systems. An important case is interfacing Prolog with the *host* implementation language of the system (e.g., C, Java, JavaScript). The main issues revolve around the external representation for Prolog terms (usually in C or Java) and whether nondeterminism is visible to the host language or should be handled at the Prolog level. The latter aspect is usually resolved by hiding backtracking from the foreign language program, except in where there is a natural counterpart: such is the case when the language has consensual built-in support for features like iterators. A more detailed survey describing and comparing different features is given by Bagnara and Carro (2002).

### *3.3.2 Libraries*

*Constraint Satisfaction* As mentioned in Section 2.8, advances such as finite domain implementation based on indexicals and, specially, progress in the underlying technology in Prolog engines for supporting extensions to unification, such as meta-structures (Neumerkel 1990) and attributed variables (Holzbaur 1992), enabled the *library-based approach* to supporting embedded constraint satisfaction that is now present in most Prolog systems. Since many constraint domains have been implemented as libraries, such as $\mathcal{R}$ and $\mathcal{Q}$ (linear equations and inequations over real or rational numbers), $\mathcal{FD}$ (finite domains), $\mathcal{B}$ (booleans), etc.

Systems vary in how the constraint satisfaction process gets implemented. In SICStus and ECL$^i$PS$^e$, the constraint library is partly implemented in C, in the case of GNU Prolog in a dedicated DSL designed to specify propagators. Several other systems, such as Ciao, SWI-Prolog, XSB, and YAP, use Prolog implementations built on top of attributed variables (as mentioned above), such as those of Holzbaur (1992) or Triska (2012), or local ones. While the C-based implementations provide a performance edge, the Prolog implementations are small, portable, and may use unbounded integer arithmetic when provided by the host system.

CHR (Frühwirth 2009), described also in Section 2.8.3, is available in several Prolog systems as a library which, rather than working on a single, specific domain, enables the writing of rule-based constraint solvers in arbitrary domains. CHR provides a higher-level way of specifying propagation and simplification rules for a constraint solver, although possibly at some performance cost.

*Data Structures* Different Prolog systems also ship varying numbers of included libraries such as code for AVL trees, ordered sets, etc. Because they are usually written purely in standard Prolog, those implementations can usually be dropped in place without larger modifications.

### *3.3.3 Extensions*

*Tabling* As discussed in Section 2.9, tabling can be used to improve the efficiency of Prolog programs by reusing results of predicate calls that have already been made, at the cost of additional memory. It improves the termination properties of Prolog programs by delaying *self-recursive* calls. Tabling was first implemented in XSB and currently a good number of other Prolog implementations support it (e.g., B-Prolog, Ciao, SWI, YAP). XSB and recent SWI-Prolog versions improve support for negation using stratified negation and well-founded semantics. Both systems also provide *incremental* tabling which automatically updates tables that depend on the dynamic database, when the latter is modified. Some systems (YAP, SWI-Prolog) support *shared* tabling which allows a thread to reuse answers that are already computed by another thread. Ciao supports a related concept of concurrent facts for communication between threads (Carro and Hermenegildo 1999), combines tabling and constraints (Arias and Carro 2016), and supports negation based on stable model semantics (Arias *et al*. 2018).

*Parallelism* Today, new hardware generations do not generally bring large improvements in sequential performance, but they do often bring increases in the number of CPU cores. However, ISO-Prolog only specifies semantics for single-threaded code and does not specify built-ins for parallelism. As already discussed in Section 2.7, several parallel implementations of Prolog or derivatives thereof have been developed, targeting both shared-memory multiprocessors and distributed systems. Support for or-parallelism is not ubiquitous nowadays, although systems like SICStus were designed to support it and this feature can possibly be easily recovered. Ciao still has some native support for and-parallelism and concurrency, and its preprocessor CiaoPP still includes auto-parallelization. A different, more coarse-grained form of parallelism is multithreading (this is what the parallelism column in Table 2 gathers).

*Indexing* Indexing strategies of Prolog facts and rules are vital for Prolog development as they immediately influence how Prolog predicates are written with performance in mind. The availability of different indexing strategies is an important issue that affects portability of Prolog programs: if a performance-critical predicate cannot be indexed efficiently, run-time performance may be significantly affected. There are several strategies for indexing:

- First-argument (FA) indexing is the most common strategy where the first argument is used as index. It distinguishes atomic values and the principal functor of compound terms.
- Nonfirst argument indexing is a variation of first-argument indexing that uses the same or similar techniques as FA on one or more alternative arguments. E.g., if a predicate call uses variables for the first argument, the system may choose to use the second argument as the index instead.
- Multiargument (MA) indexing creates a combined index over multiple instantiated arguments if there is not a sufficiently selective single argument index.
- Deep indexing is used when multiple clauses use the same principal functor for some argument. It recursively uses the same or similar indexing techniques on the arguments of the compound terms.
- Trie indexing uses a prefix tree to find applicable clauses.

In addition to the above indexing techniques, one can also distinguish systems that use directives to specify the desired indexes and systems that build the necessary indexes

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

just-in-time based on actual calls. One should note that the first form of indexing (FA) is the only one which may be effectively relied upon when designing portable programs, for it is close to universal adoption.

*Type and Mode Annotations* As Prolog is a dynamic language, it can be hard to maintain larger code bases without well-defined (and checkable) interfaces. Several approaches have been proposed to achieve or enforce sound typing in Prolog programs (Mycroft and O'Keefe 1984; Dietrich and Hagl 1988; Gallagher and Henriksen 2004; Schrijvers *et al.* 2008). While these approaches are closer to strong typing, they have not caught on with mainstream Prolog.

Many Prolog systems offer support, for example, for mode annotations, yet the directives usually have no effects except their usage for documentation. The few Prolog or Prolog-like systems that really address these issues and incorporate a type and mode system include Mercury (Somogyi *et al.* 1996) and Ciao (Hermenegildo *et al.* 1999). The former is rooted in the Prolog tradition, but departs from it in several significant ways (see Sections 2.1 and 3.5). The latter aims to bridge the static and dynamic language approaches, while preserving full compatibility with traditional nonannotated Prolog. A fundamental component is its assertion language (Puebla *et al.* 2000b) that is processed by CiaoPP (cf. Section 2.10.2). CiaoPP then is capable of finding nontrivial bugs statically or dynamically and can statically verify that the program complies with the specifications, even interactively (Sanchez-Ordaz *et al.* 2021). The Ciao model can be considered an antecedent of the now-popular *gradual-* and *hybrid-typing* approaches (Flanagan 2006; Siek and Taha 2006; Rastogi *et al.* 2015) in other programming languages.

*Coroutining* As discussed in Section 2.4.1, coroutining was first introduced into Prolog in order to influence its default left-to-right selection rule. From a logical perspective, logic programs are independent of the selection rule. However, from a practical, programming language perspective, procedural factors might influence efficiency, termination, and even faithfulness to the intended results. Consider for instance a program that tries to calculate the price of an as yet unknown object X (this could result from, say, some natural language interface's ordering of different paraphrases). Coroutining can ensure that the program behaves as intended by reordering these goals so that objects are instantiated before attempting to calculate their prices.

Coroutining is an important feature of modern Prolog systems, allowing programmers to write truly reversible predicates and improve their performance. It represents a step forward towards embodying the equation of "Algorithm = Logic + Control" by Kowalski (1979). Early mechanisms for coroutining focused on variations of the `delay` primitive by Dahl and Sambuc (1976), which dynamically reorders the execution of predicates according to statically, user-defined conditions on them. The `freeze` variation was present in Prolog II (Colmerauer 1982b; Boizumault 1986), which delays the execution of its second argument (understood to be a goal) until its first argument, a single variable occuring in the second argument, is bound. A more flexible variation is the `wait` primitive present in MU-Prolog (Naish 1985), which was subsequently generalized to `when`. This concern evolved into the `block` declaration found in modern Prologs, which supports the annotation of an entire predicate (rather than of each individual call), resembling a mode declaration. This approach thus leads to more readable programs and more efficient

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

code. It can be argued that the coupling of goal evaluation to the binding of variables ultimately led to the development of Constraint Logic Programming (CLP) (3.3.2).

### *3.3.4 Tools*

*Unit Testing Structures* One of the most important tools in software development is a proper testing facility. Some Prolog systems ship a framework for unit testing, and while the basic functionality is shared, usually they do not adhere to the same interface. SWI-Prolog ships a library named `plunit`, while SICStus uses a modified version of it. Both versions are entirely written in Prolog, yet they rely on system-specific code to function properly. Ciao relies on the `test` assertions of its assertion language, which also include test case generation. ECL$^i$PS$^e$ offers a library named `test_unit` with several test primitives that assert whether calls should succeed, raise errors, etc. Other systems seem to rely on the fact that a small ad hoc testing facility is rather easy to implement.

*Debugging* A good debugger is vital to understand and fix undesired behaviors of programs. Prolog control flow is different from that of most other programming languages because it has backtracking. To address this, most Prolog systems provide some form of tracing debugger based on the 4-port debugger introduced for DEC-10 Prolog by Byrd (1980), which allows for the tracing of individual goals at their call, exit, redo, and failure ports (states). Most systems allow setting of spy points (similar to breakpoints) and some provide a very Prolog-specific and powerful debugging tool: the *retry* command which allows one to "travel back in time" to the entry point of a call that, for some reason, misbehaved. The latter feature assumes Prolog programs without side effects. A few systems, such as SWI, SICStus, and Ciao, additionally offer source-level debugging that allows following the steps of execution directly on the program text, thus providing a more conventional view.

## *3.4 Takeaways*

When considering Table 2, one can see that, despite undeniable differences among Prolog systems, many Prolog systems offer similar features.

### *Available and Mostly Compatible Features*

Mostly compatible *module* systems have been widely adopted, even if they virtually all diverge from the ISO document. The existence of a de-facto module standard makes it possible to write production-quality libraries that are portable across most Prolog systems.

Facilities for *multithreaded programming* are also common: A number of systems offer predicates based on the corresponding technical recommendation document (Moura 2007), sometimes with some differences in semantics or syntax.

Most systems also offer libraries for *constraint programming*, though they differ in performance and expressiveness. Yet, no standard interface or even a proposal for one exists.

Almost all Prolog systems embrace dynamic typing, and *type and mode* annotations are used for documentation or optimization, yet are not enforced or verified at all. Ciao,

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

with its combination of the dynamic and static approaches, is the significant exception here.

While support for *tabling* is present in various systems, the features and interfaces can differ. Programs that rely on it (beyond simple answer memoization that can be implemented fairly simply using the dynamic database), specially if they use special features, can be hard to port. Thus, progress needs to be made toward better portability in this area.

*Discrepancies*

One gets a mixed result when considering support for other features: some systems, such as ECL$^i$PS$^e$ offer extensive library support of *data structures*, whereas others remain rather basic, without a large standard library.

*Coroutining* is not available on all systems, and the various primitives (`when`, `block`, `freeze`, `dif`) sometimes have some variations among those systems that do support it. A similar situation occurs for *global variables* and *mutable terms*. Similarly, *testing frameworks* are missing in several systems, but usually can be provided in form of a portable library.

Almost all Prolog systems support at least one *foreign language interface*, in order to leverage existing libraries and to widen the domains where logic programming can be applied. Yet, there are different strategies on how the interfaces interact with Prolog, and, thus, the interfaces often differ between implementations.

An issue that can also hinder portability is the large discrepancy in *indexing strategies*. Solutions so far are of a very technical nature, rather than aimed toward a common interface, so work is needed if this issue is to be resolved.

*Conclusions*

Overall, most Prolog systems are not *too* different in what they offer. Many differences could be bridged by agreeing on certain interfaces, or, for example, sharing library predicates for testing, or data structures. Differences in constraint solving capabilities are harder to reconcile, as some solvers are of commercial nature. However, CHR's embodiment of constraints is fairly ubiquitous and permits the implementation of constraint solvers in arbitrary domains of interest. Missing technical features, such as tabling or indexing strategies, may hinder portability or performance, but the relevance of this issue can greatly depend on the application. It is also possible to integrate tabling with constraints à la CHR (Schrijvers and Warren 2004). As differences with ISO-Prolog usually are very small now, the Prolog implementations' cores are very similar today.

### 3.5 Influence on other languages

The concepts and ideas that have been explored during the long history of evolution of Prolog systems have influenced and given rise to other languages and systems, both within the LP paradigm and out to other programming paradigms. In the following (Section 3.5.1), we describe languages within the LP paradigm that are heavily inspired by or emerged from Prolog. These systems generally fail our definition of Prolog in Section 2.1.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

However, sometimes they bear witness to useful features or extensions that have not (yet) made their way into Prolog itself. Still others, such as, for example, `s(ASP)`/`s(CASP)` or Co-Inductive LP, are really extensions of Prolog whose support in Prolog systems could be generalized in the future, as has happened already with constraints or tabling. They could thus also have been listed in Section 3.3.3. Regarding the impact that Prolog and Prolog systems have had *beyond* LP, it is outside the scope of this paper to do a full analysis of this very interesting topic, but we briefly review in any case in Section 3.5.2 a few examples of other such influences outside the LP paradigm.

### *3.5.1 Influences on other languages in the LP paradigm*

*Datalog* is a subset of Prolog, which does not allow compound terms as arguments of predicates. This topic has been worked on since the late 70s, although the term was coined later by David Maier and David S. Warren in the 80s. Datalog can be viewed as an extension of relational databases, allowing recursion within predicates. Datalog plays an important role in the research field of deductive databases (Ramakrishnan and Ullman 1993). Datalog has found new applications in many areas (Huang *et al*. 2011), such as information extraction (Shen *et al*. 2007), program analysis and synthesis (Whaley *et al*. 2005; Alpuente *et al*. 2010; Jordan *et al*. 2016; Madsen and Lhoták 2020), security (Bonatti 2010), graph processing (Seo *et al*. 2015), reasoning over ontologies (Baumeister and Seipel 2010), or natural language processing (Dahl *et al*. 1995).

*λProlog* was developed in the late 80s, by Miller and Nadathur (1988). It was defined as a language for programming in higher-order logic based on an intuitionistic fragment of Church's theory of types. It spawned several modern refinements and implementations (λProlog Home Page 2021). Higher-order extensions have also made their way into less specialized Prologs, as we discuss in Section 2.10.2. Miller and Nadathur (2012) discuss uses of higher-order logic in logic programming to provide declarative specifications for a range of applications. λProlog applications are still surfacing, with the main focus being on meta-programming (Nadathur 2001), program analysis (Wang and Nadathur 2016), and theorem proving (Miller 2021).

*Committed-choice Languages* As mentioned in Section 2.6, the implementation complexity of combining Prolog's backtracking with concurrency and/or parallelism led to the development of logic languages supporting "committed choice" where only the first clause whose guard succeeds is executed, instead of the multiple execution paths supported by Prolog. This includes GHC (Guarded Horn Clauses) (Ueda 1985), KL1 (Ueda and Chikayama 1990), Parlog (Clark and Gregory 1986), and Concurrent Prolog (Shapiro 1983; 1987). Erlang (Armstrong 2007) (see later) also has its origins in this line of work. This line of work also yielded concurrent constraint languages, such as `cc(fd)` (Hentenryck *et al*. 1994), and distributed constraint languages, such as AKL (Janson and Haridi 1991) and Oz/Mozart (Roy *et al*. 2020) (see later). The concepts brought about by the committed-choice languages, such as guards and data-flow synchronization based on one-way unification and constraint entailment (Maher 1987), have in turn made their way back into Prolog systems as part of extensions for concurrency and distributed

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

execution (e.g., in &-Prolog/Ciao Cabeza and Hermenegildo 1996; Hermenegildo *et al.* 1996 or ACE Gupta and Pontelli 1999).

*Turbo-Prolog* (Hankley 1987) can be considered a precursor of other strongly typed logic programming languages. It was released as a full development environment in 1986 for PC/MS-DOS. It was strongly typed, had support for object-oriented programming, and it compiled directly to machine code. At that time, this pragmatic approach provided a safe and efficient language, but lacked important dynamic features of Prolog required in many applications. It has been continued as PCD Prolog and Visual Prolog, focusing on supporting good integration with Microsoft Windows APIs.

*Gödel* (Hill and Lloyd 1994) is a logic programming language that first appeared around 1992. It is strongly typed, with a type system based on many-sorted logic, allowing for parametric polymorphism. It implemented a sound negation, delaying negated calls until they were ground. Gödel also supports meta-programming, using ground representation for meta-programs, which has the advantage of having a declarative semantics. This enabled the development of a self-applicable partial evaluator called SAGE by Gurr (1994). The Gödel system was built on top of SICStus Prolog, employing a different syntax style. The development of the language came to a halt in the 1990s.

*Curry* (Hanus *et al.* 1995) was developed in 1995. It is a functional logic programming language that is mostly based on Haskell, but includes some features from logic programming languages such as nondeterminism and constraint solving. Curry is based on the technique of *narrowing*, which is also the basis of other functional logic programming work (Antoy and Hanus 2010). The language has recently been used for typesafe SQL queries (Hanus and Krone 2017), for research and for teaching both the logic and the functional paradigm (Hanus 1997).

*Oz* Another multiparadigm language is Oz (Henz *et al.* 1993), incorporating concurrent logic programming, functional programming, constraints, and objects. The design of the language started in 1991 and its logic programming aspects were greatly influenced by AKL, the Andorra Kernel Language (cf. Section 2.10.2). A recent synopsis of Oz's history is available in the article by Roy *et al.* (2020). The current incarnation of Oz is available as an open source implementation called Mozart.

*Mercury* (Somogyi *et al.* 1996) was created in 1995 as a functional logic programming language, with characteristics from both Prolog and Haskell. The main reasons for its development were threefold. First, idiomatic Prolog code at the time had short predicate and variable names, and also lacked type annotations and comments. This often rendered it hard for the reader to infer the meaning, types, and modes of a program. Second, multimode predicates and those without static-type information could not be compiled to the most efficient WAM code. Third, a lot of LP research was concerned with logic programs without any impure operations, and thus were not applicable to general Prolog programs (e.g., executing `read(X), call(X)`). Thus, Mercury features a strong, static, polymorphic type system, and a strong mode and determinism system. It has a separate compilation step, which allows for a larger variety of errors to be detected before actually running a program, and for generation of faster code. By removing nonlogical features, such as `assert/1` and `retract/1`, a pure language was obtained, I/O could be implemented declaratively, and existing research could be implemented unaltered. Some interesting research has been done in the areas of program analysis and optimization (Becket and Somogyi 2008), as well as parallelism (Bone *et al.* 2012).

*Assumption Grammars and Assumptive Logic Programming* (Dahl *et al.* 1997; Dahl and Tarau 1998; 2004) extend Prolog with hypothetical reasoning, needed in particular for natural language processing applications (Dahl *et al.* 1997). They include specialized linear logic implications, called assumptions (Dahl and Tarau 1998; 2004), that range over the computation's continuation, can be backtracked upon, and can either be consumed exactly once (linear), at most once (affine linear), any number of times (intuitionistic), or independently of when they have been made: before or after consumption (timeless). The latter is a novel form of implication designed and first meta-programmed by Dahl for dealing with backward anaphora. Their uses for abduction were also researched by Dahl and Tarau (2004).

*Answer Set Programming* (ASP) is arguably one of the largest successes of logic programming. It is a logic programming paradigm that focuses on solving (hard) search problems, by reducing them to computing stable models. Note that ASP is *not* a Turing-complete programming language, but rather a language to represent aforementioned problems. It is based on the stable models semantics and uses answer set solvers to provide truth assignments as models for programs. The usual approach for ASP is to ground all clauses so that propositional logic techniques like SAT-solving can be applied to find stable models for the program. Unlike Prolog's query evaluation, ASP's computational process always terminates. The denomination "answer set" was first coined by Lifschitz (1999). Its early exponents were Niemelä (1999) and Marek and Truszczynski (1999); more recent developments include Smodels (Syrjänen and Niemelä 2001), DLV (Leone *et al.* 2006), the Potassco toolset (Gebser *et al.* 2014; 2008), and WASP Alviano *et al.* (2013). The interested reader may rely on the survey by Brewka *et al.* (2011) and on the special issue of the AI Magazine dedicated to ASP (Brewka *et al.* 2016).

*s(ASP) and s(CASP)* `s(ASP)` (Marple *et al.* 2017) is a *goal-directed, top-down* execution model which computes stable models of normal logic programs with arbitrary terms, supporting the use of lists and complex data structures, and, in general, programs which may not have a finite grounding. It supports both deduction and abduction. `s(ASP)` uses a non-Herbrand universe, coinduction, constructive negation, and a number of other novel techniques. Unlike in ASP languages, variables are (as in (C)LP) kept during execution and in the answer sets. `s(CASP)` (Arias *et al.* 2018) is the extension of `s(ASP)` to constraint domains, while also including additional optimizations in the implementation. `s(ASP)` and `s(CASP)` can be seen as Prolog extended with negation as failure where this negation follows the stable model semantics. If negation is not used, the behavior is as in Prolog. Both `s(ASP)` and `s(CASP)` have been implemented in Prolog, originally in Ciao and also ported to SWI. For some applications, this approach leads to improved performance and expressivity compared to existing ASP systems. Another advantage is that it naturally provides explanations. At the same time, for some other classical

ASP applications, the grounding to propositional logic remains currently the technology of choice. Goal-directed evaluation of ASP was also addressed in the work of Bonatti *et al.* (2001, 2008). In comparison, the `s(ASP)` and `s(CASP)` work handles unrestricted predicates and queries, with constraints. Other early work on query-driven computation of stable models includes that of Chen and Warren (1996) and the XNMR system within XSB.

*Constraint Handling Rule Grammars* (CHRG) by Christiansen (2002), extend Prolog with sophisticated language processing capabilities because, just like Prolog does, they allow writing grammar rules that become executable. They differ from the previous grammatical default of Prolog (*DCGs*) in that they work bottom up, are robust (i.e., in case of errors the recognized phrases so far are returned, rather than silently failing by default), can inherently treat ambiguity without backtracking, and, just as Hyprolog (see below), can produce and consume arbitrary hypotheses. This makes it straightforward to deal with abduction, which is useful for diagnostics, integrity constraints, operators à la Assumption Grammars. They can also incorporate other constraint solvers. Applications go beyond traditional NLP, including e.g., biological sequence analysis (Bavarian and Dahl 2006).

*Hyprolog* (Christiansen and Dahl 2005) is an extension to Prolog and Constraint Handling Rules (CHR) which includes all types of hypothetical reasoning in Assumption Grammars, enhances it with integrity constraints, and offers abduction as well. It compiles into Prolog and CHR through an implementation by Henning Christiansen available for SICStus, Prolog III and IV, and for SWI-Prolog. It can access all additional built-in predicates and constraint solvers that may be available through CHR, whose syntax can be used to implement integrity constraints associated to assumptions or abducibles. Due to the compiled approach, which employs also the underlying optimizing compilers for Prolog and CHR, the Hyprolog system is among the fastest implementations of abduction.

*Co-Inductive logic programming* (Simon *et al*. 2006) was proposed in order to allow logic programming to work with infinite terms and infinite proofs based on greatest fixed-point semantics. The *co-logic programming* paradigm by Gupta *et al*. (2007) is presented as an extension of traditional logic programs with both inductive and co-inductive predicates. This can be used, for example, for model checking, verification, and nonmonotonic reasoning (Gupta *et al*. 2011). These concepts were implemented based on modifying YAP's Prolog engine and are also related to `s(ASP)/s(CASP)`.

*Probabilistic Logic Programming* (PLP) is a research field that investigates the combination of LP with the probability theory. A comprehensive overview on this topic is provided by Riguzzi (2018). In PLP, theories are logic programs with LPAD, that is, logic programs with annotated disjunctions (Vennekens *et al*. 2004), hence they may contain facts or rules enriched with probabilities. These may, in turn, be queried by the users to investigate not only which statements are true or not but also under which probability. To support this behavior, probabilistic solvers employ ad hoc resolution strategies

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

explicitly taking probabilities into account. This makes them ideal to deal with uncertainty and the complex phenomena of the physical world.

From a theoretical perspective, the distribution semantics by Sato (1995) is one of the most prominent approaches for the combination of logic programming and probability theory. Sato, in particular, was among the first authors exploiting Prolog for PLP, by building on the ideas of Poole (1993). The very first programming language laying under the PLP umbrella was PRISM, by Sato and Kameya (1997), which supported not only probabilistic inference but learning as well. Since then, many PLP solutions have been developed supporting this semantics, such as ProbLog by de Raedt *et al.* (2007) and `cplint` by Riguzzi (2007). These were often implemented on top of existing Prolog implementations. For instance, ProbLog consists of a Python package using YAP behind the scenes, while `cplint` is based on SWI-Prolog. They reached a considerable level of maturity and efficiency by exploiting binary decision diagrams (Akers 1978) or variants of them to speed up probabilistic inference.

*Logtalk* (Moura 2011) can be considered to be an object-oriented logic programming language as well as an extension to Prolog. Its development started in 1998 with the goal of supporting programming in the large. Because it is object-oriented, Logtalk supports classes, prototypes, parametric objects, as well as definite clause grammars, term-expansion mechanisms, and conditional compilation. It addresses several issues of Prolog that have not been met with a standardized solution, including portability of libraries and tools (e.g., for unit testing, documentation, package management and linting), by compiling the code to a very wide range of Prolog systems.

*Picat* (Zhou *et al.* 2015) is a logic-based multiparadigm language. Its development started in 2012, stemming from B-Prolog (Zhou 2012) and having its first alpha release in 2013 (Zhou and Fruhman 2021). It aims to combine the efficiency of imperative languages with the power of declarative languages. It is dynamically typed and uses rules in which predicates, functions, and actors are defined with pattern-matching. It also incorporates some features of imperative languages, such as arrays, assignments, and loops. Its main focus of research is constraint solving (Zhou 2021).

*Womb Grammars* (WG) (Dahl and Miralles 2012), endow Prolog + CHRG with constraint solving capabilities for grammar induction, within a novel paradigm: they automatically map a language's known grammar (the source) into the grammar of a different (typically under-resourced) language. This is useful for increasing the survival chances for endangered languages, with obvious positive socioeconomic potential impact (over 7000 languages are spoken in the world, of which, according to Ethnologue (2021), 2895 are endangered). They do so by feeding the source grammar a set of correct and representative input phrases of the target language plus its lexicon and using the detected "errors" to modify the source grammar until it accepts the entire corpus. WG have been successfully used for generating the grammars of noun phrase subsets of the African language Yorùbá (Adebara and Dahl 2016) (for which a grammar that validated the system's findings does exist) and the Mexican indigenous language Ch'ol (Dahl *et al.* to appear) (for which no grammar had been yet described).

### 3.5.2 Some influences on languages and systems beyond LP

*Theorem proving* Some aspects of the WAM, such as the compilation of clause heads, were adopted by different theorem provers, such as the Boyer-Moore theorem prover (Kaufmann and Boyer 1995), as a result of the prover team and Prolog teams working together at MCC in the mid to late 80s. Another example of influence of these Prolog systems is the use of Prolog technology in theorem provers, for instance, by Stickel (1984) or provers directly implemented in Prolog (Manthey and Bry 1988; Stickel 1992; Beckert and Posegga 1995).

*Java* The design of the WAM and various other aspects of Prolog implementation influenced the design of the Java abstract machine, since some of the designers of this machine had formerly worked at Quintus and were Prolog implementation experts. For instance, the semantics of type checking for Java's class files is provided as a Prolog script by Lindholm *et al*. (2021).

*Erlang* The quite successful programming language *Erlang* (Armstrong 2007) has its roots in Prolog and the concurrent constraint languages that derived from Prolog and was developed with the goal of improving the development of telephony applications. The first version of the Erlang interpreter was written in Prolog, which is the reason for syntactic similarities. Erlang is still used nowadays by many companies, including Cisco, Ericsson, IBM, and WhatsApp (Paxton 2021).

*Language Embeddings* Some languages outside LP nowadays include a Prolog or logic programming library or mode. Classical examples are the different embeddings of Prolog in Scheme, such as Schelog (Schelog Homepage 2018) and Racket's RackLog sublanguage (RackLog Homepage 2021), which are generally based on the work of Felleisen (1985) and Haynes (1986) (the former also done in part at MCC) and Carlsson (1984). These can provide useful Prolog-like functionality, although the performance is generally not comparable with, for example, native WAM-based systems.

Further influence outside the logic programming paradigm is apparent in languages with inferred types and polymorphic type systems, which sometimes include a rule system to specify and constrain the types. For example, the *concepts* of C++ 2020 (ISO/IEC 14882 2020) provide predicates that form rules to statically determine which of a set of implementations of a polymorphic function ought to be used, according to context.

## 4 Part III: The future of Prolog

While Section 3 establishes that some incompatibilities between Prolog systems are not too difficult to overcome, this section explores a different perspective: what are the perceived issues and potential future directions for Prolog to grow. In order to provide insights on the future of Prolog, we conducted a SWOT analysis. Its results can be found in Table 3. In the following, we discuss strengths (Section 4.1) and opportunities (Section 4.2), followed by weaknesses of the language (Section 4.3) currently, and external factors that may be threats to the adoption of Prolog, its future development, or to the compatibility of Prolog systems (Section 4.4). In Section 4.5, we aim at providing

Table 3. *SWOT analysis*

**Strengths** (Section 4.1)

- clean, simple language syntax and semantics
- immutable persistent data structures, with "declarative" pointers (logic variables)
- arbitrary precision arithmetic
- safety (garbage collection, no NullPointer exceptions, ...)
- tail-recursion and last-call optimization
- efficient inference, pattern matching, and unification, DCGs
- meta-programming, programs as data
- constraint solving (Section 3.3.2), independence of the selection rule (coroutines (Section 3.3.3))
- indexing (Section 3.3.3), efficient tabling (Section 3.3.3)
- fast development, REPL (Read, Execute, Print, Loop), debugging (Section 3.3.4)
- commercial (Section 2.10.1) and open-source systems
- active developer community with constant new implementations, features, etc.
- sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- successful applications
  - program analysis,
  - domain-specific languages
  - heterogeneous data integration
  - natural language processing
  - efficient inference (expert systems, theorem provers), symbolic AI
- many books, courses and learning materials

**Weaknesses** (Section 4.3)

- syntactically different from "traditional" programming languages, not a mainstream language
- learning curve, beginners can easily write programs that loop or consume a huge amount of resources
- static typing (see, however, Section 3.3.3)
- data hiding (see, however, Section 3.3.1)
- object orientation (see, however, Section 4.5.4)
- limited portability (see Section 4.5.1)
- packages: availability and management
- IDEs and development tools: limited capabilities in some areas (e.g., refactoring; Section 4.5.2)
- UI development (usually conducted in a foreign language via FLI (Section 3.3.1))
- limited support for embedded or app development

**Opportunities** (Section 4.2)

- new application areas, addressing societal challenges 4.2:
  - neuro-Symbolic AI
  - explainable AI, verifiable AI
  - Big Data
- new features and developments
  - probabilistic reasoning (3.5.1)
  - embedding ASP (3.5.1) and SAT or SMT solving
  - parallelism (2.7, 3.3.3) (resurrecting 80s, 90s research)
  - full-fledged JIT compiler

**Threats** (Section 4.4)

- comparatively small user base
- fragmented community with limited interactions (e.g., on StackOverflow, reddit), see 4.4.1
- active developer community with constant new implementations, features, etc.
- further fragmentation of Prolog implementations, see 4.4.1
- new programming languages
- post-desktop world of JavaScript web-applications
- the perception that it is an "old" language
- wrong image due to "shallow" teaching of the language

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

a foundation for community discussion and stimulus toward future development of the language. To this end, we make proposals and raise questions on which features could be useful future extensions for Prolog. Finally, in Section 4.6, we summarize and briefly discuss some possible next steps.

## *4.1 SWOT: Strengths of Prolog*

### *Ease of Expression*

Prolog is a language with a small core and a minimal, yet extremely flexible syntax. Even though some features can only be understood procedurally, such as the cut, the semantics remains very simple. Combined with automatic memory management and the absence of pointers or uninitialized terms makes Prolog a particularly safe language. Flexible dynamic typing completes the picture by placing Prolog among the most high-level programming languages available to date – a feature that makes it very close to how humans think and reason, and therefore ideal for Artificial Intelligence (AI).

The inspection and manipulation of programs at run-time also leads to faster programmer feedback and enables powerful debugging, in particular when coupled with Prolog's interactive toplevel (a.k.a. REPL or Read-Eval-Print-Loop, although in the case of Prolog the print part is richer because of multiple solutions).

Declarativity is yet another important feature of Prolog: most programs just state *what* the computer should do and let the Prolog engine figure out the *how*. For pure logic programs, the resolution strategy may also be altered, and possibly optimized, without requiring the source code of those programs to be changed. Program analysis and transformation tools, partial evaluators, and automatic parallelizers can be used for Prolog programs and can be particularly effective for purely declarative parts. For partial evaluation, however, the techniques have still not yet been integrated into Prolog compilers (the discussion by Leuschel and Bruynooghe 2002, Section 7 is mostly still valid today), with the exception of Ciao Prolog's preprocessor (Hermenegildo *et al.* 2005) and abstract machine generator (Morales *et al.* 2005; 2016). However, they have found their way into just-in-time compilers for other languages (Bolz *et al.* 2011).

Prolog's data structure creation, access, matching, and manipulation are performed via the powerful and efficiently implemented unification operation. The logical terms used by Prolog as data structures are managed dynamically and efficiently. Logical variables within logical terms can encode "holes," which can then be passed around arbitrarily and filled at other places by a running program. Furthermore, logical variables can be bound (*aliased*) together, closing or extending pointer chains. This gives rise to many interesting programming idioms such as difference lists and difference structures, and in general to all kinds of pointer-style programming, where logical variables serve as "declarative pointers" (since they can be bound only once). This view of logical variables as declarative pointers and related issues have been discussed by Hermenegildo (2000). Furthermore, Prolog's automatic memory management ensures the absence of nuisances such as NullPointer exceptions or invalid pointer manipulations. Many Prolog systems also come with arbitrary precision integers, which are used transparently without requiring user guidance.

Prolog compilers and the many program analysis and transformation tools mentioned above are almost always written in Prolog itself, which is an excellent language for

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

writing program processors. Interestingly, since the semantics of programming languages can be easily encoded as (Constraint) Horn Clauses (CHCs), Prolog tools can often be applied directly to the analysis of other languages (Méndez-Lojo *et al.* 2007). The survey by De Angelis *et al.* (2021) in this same special issue of the TPLP journal provides a comprehensive overview of work using analysis and transformation of (constrained) Horn clauses and techniques stemming from logic programming for program verification, including those techniques most related to Prolog.

*Efficiency*

In addition, Prolog is a surprisingly efficient language. Beginners will often write very inefficient Prolog programs. Yet, carefully written Prolog programs can build on many of the features provided by modern implementations, such as last call optimization (which generalizes tail recursion optimization), efficient indexing and matching, and fine-tuned memory management with efficient backtracking. For applications which are well-suited to Prolog, such as program analysis (Méndez-Lojo *et al.* 2007), program verification (Leuschel 2008), or theorem proving (see Section 3.5.2), this can lead to programs which are both more flexible and better-performing than counterparts written in traditional languages (Leuschel 2020).

*Successful Applications*

Thanks to its simple foundation, the language makes it straightforward to read Prolog programs as data objects and it is almost trivial to implement meta-interpreters, as well as custom or domain-specific languages (such as Erlang Armstrong 2007, which was initially implemented in Prolog). Therefore, it can be (and has been) used as a means to represent knowledge bases or bootstrap declarative languages in knowledge-intensive environments. Prolog has also been used for several successful formal methods tool developments, such as, for example, Leuschel and Butler (2008), Lopez-Garcia *et al.* (2018). Moreover, Prolog supports the implementation of novel sorts of expert systems or logic solvers, relying for instance on probabilistic, abductive, or inductive inference, which can be simply realized as meta-interpreters. This is another reason why Prolog is well suited for symbolic AI applications. It can also be used to integrate and reason over heterogeneous data (Wielemaker *et al.* 2008).

Prolog has also been successfully used for parsing, both for computer languages and for natural languages (see also Section 2.4.1). A relatively recent success story is IBM's Watson system (Lally *et al.* 2012) which used Prolog for natural language processing, adapting techniques developed in logic grammars over the years for solving difficult computational linguistics problems, such as coordination (McCord 2006; Dahl and McCord 1983). Regarding parsing algorithms, Prolog's renditions of tabling admit especially succinct while elegant and efficient formulations, as demonstrated for the CYK algorithm on p. 37 of Frühwirth's book on CHR (Frühwirth 2009). Indeed, Prolog lends itself particularly well for grammar development and grammatical theory testing, which has potential both for compilers and for spoken and other human languages. The simplest grammatical Prolog version, DCGs, extends context-free grammars with symbol arguments, while the first grammatical version, MGs, extends type-0 grammars with symbol arguments.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

Variants that are adequate to different linguistic theories can, and have been, developed (Dahl 1992; Dahl *et al.* 1993; Dahl 1990; 1986). Most crucially, semantics can be straightforwardly accommodated and examined through symbol arguments, which allows for the increasingly important features of transparency and explainability to materialize naturally. Coupled with Prolog's meta-programming abilities, grammar transformation schemes can help automate the generation of linguistic resources that most languages in the world typically lack, as shown by Dahl and Miralles (2012). Finally, tabling can be used for efficient parsing (Simpkins and Hancox 1990) of a wide range of grammars, even context-sensitive ones.

Many more examples of practical applications can be found in the literature, in particular in the conference series PADL (Practical Applications of Declarative Languages, running since 1999) and INAP (International Conference on Applications of Declarative Programming and Knowledge Management, since 1998). ICLP, the premier conference in logic programming, regularly includes papers and sessions on applications and some editions have a special applications track.

*Active Community*

As we have seen throughout this paper, there are many implementations of the Prolog language, many of them quite mature and still being actively developed, with new features and libraries added continuously, while new implementations keep appearing, with new aims or targeting different niches.

There are many books and tutorials on the Prolog language. Good examples are texts by Clocksin and Mellish (1981), Sterling and Shapiro (1994), O'Keefe (1990), Clocksin (1997), Blackburn *et al.* (2006), Bratko (2012), or Triska (2021). There is also significant teaching material publicly available in the form of slides, exercises, examples, contest problems, etc., as well as plenty of topical discussions in on-line fora. There are also some interactive learning environments and playgrounds, for example, GUPU (Neumerkel and Kral 2002), or those of SWI, Ciao, or LogTalk, although this is certainly an area that would be well worth improving.

A Prolog programming contest (which has now expanded to include other LP and CLP dialects) is held every year in the context of ICLP.

Even with Prolog being somewhat outside the mainstream of programming languages, it is taught at many universities for a simple reason: it introduces new concepts and features that are quite different from those of object-oriented as well as functional programming languages. Accordingly, it provides computer scientists with not only a simple yet powerful tool to understand and write elegant algorithms but also a new way of thinking about programming. Getting to know the ropes of Prolog expands one's horizons and allows programmers to significantly improve their way of solving problems. One could easily argue that a computer scientist is not really complete without being familiar with First-Order Logic, Resolution, Logic Programming, and Prolog.

### *4.2 SWOT: Opportunities*

There are several opportunities to considerably improve the performance of Prolog by resurrecting earlier research on parallelism (Sections 2.7, 3.3.3). A JIT compiler can also be beneficial and is provided, for example, by SICStus Prolog. There are certainly opportunities for combining just-in time compilation with specialization to achieve even better performance.

It is very natural to integrate into Prolog features like probabilistic reasoning (Section 3.5.1), ASP (Section 3.5.1), or other logic-based technologies like SAT solving and SMT solving. Making these features routinely available would make Prolog more appealing for a wider class of applications.

### *Artificial Intelligence*

*Symbolic AI:* Prolog is undoubtedly among the most impactful ideas in *symbolic* AI. However, *subsymbolic* or *data-driven* AI is nowadays attracting most of the attention and resources, mostly because of the recent progress that has been achieved in machine and deep learning.

Despite these advances, state-of-the-art data-driven AI techniques are far from perfect. A common problem that shows up in critical fields such as Healthcare (Panch *et al.* 2019), Finance (Johnson *et al.* 2019), or Law (Tolan *et al.* 2019) is that learning-based solutions tend to acquire the inherent *biases* of the contexts they are trained into. This often results in decision support systems exposing sexist, racist, or discriminatory behaviors, thus unwittingly permeating the digital infrastructure of our societies (Noble 2018).

Similarly, subsymbolic techniques have been criticized for their inherent *opacity* (Guidotti *et al.* 2019). In fact, while most techniques in this field (neural networks, support vector machines, etc.) are very good at learning from data, they easily fall short when it comes to *explicitly* representing what they have learned. For this reason, such techniques are often described as black boxes in the literature (Lipton 2018).

*Explainable AI:* While all such issues are being tackled by the eXplainable AI community (XAI) (Gunning 2016) by using a plethora of subsymbolic tricks (Guidotti *et al.* 2019), an increasing number of works recognize the potential impact of symbolic AI models and technologies in facing these issues, such as the works by Calegari *et al.* (2018, 2020) or Cyras *et al.* (2021). It seems clear that *symbolic* inferential capabilities will be crucial for transitioning into the sustainable and humanity-serving AI that is urgently needed (Calegari *et al.* 2020).

Accordingly, we highlight two possible research directions where Prolog and LP may contribute further to the current AI picture. One direction concerns the exploitation of LP either *(i)* for making machine and deep learning techniques more *interpretable* or *(ii)* for constraining their behavior, reducing biases. There, Prolog and LP may be exploited as a *lingua franca* for symbolic rules extracted from subsymbolic predictors (Calegari *et al.* 2019; Ciatto *et al.* 2020), or as a means to impose constraints on what a subsymbolic predictor may (or may not) learn (Serafini *et al.* 2017). The other direction concerns the exploitation of subsymbolic AI as a means to speed up or improve some basic mechanism of LP and Prolog. For example, in the field of *inductive* logic programming (Muggleton and de Raedt 1994) neural networks have been exploited to make the inductive capabilities of induction algorithms more efficient or effective (d'Avila Garcez and Zaverucha 1999; Basilio *et al.* 2001). For instance, in the work of França *et al.* (2014), the induction task is translated into an attribute-value learning task by representing subsets of relations as numerical features, and CILP++ neuro-symbolic system is exploited to make the process faster.

Along this path, more general approaches attempt to unify, integrate, or combine the symbolic and subsymbolic branches of AI, for the sake of advancing the state of the art. This is the case for instance of the neuro-symbolic initiatives (de Raedt *et al.* 2020; Lamb *et al.* 2020; Pisano *et al.* 2020; Tarau 2021) where LP and neural networks are combined or integrated in several ways following the purpose of engineering more generally intelligent systems capable of coupling the inferential capabilities of LP with the flexible pattern-matching capabilities of neural networks.

*Inductive Logic Programming:* Inductive Logic Programming (ILP), first coined by Muggleton and de Raedt (1994) is a subfield of machine learning that studies how to learn computer programs from data, where both the programs and the data are logic programs. Prolog in particular is typically used for representing background knowledge, examples, and induced theories. This uniformity of representation gives ILP the advantage, compared to other machine learning techniques, that it is easy to include additional information in the learning problem, thus enhancing comprehensibility and intelligibility. Muggleton first implemented ILP in the PROGOL system.

ILP has shown promise in addressing common limitations of the state-of-the-art in machine learning, such as poor generalization, a lack of interpretability, and a need for large amounts of training data. The first two affect quality and usability of results, and the latter affects accessibility and sustainability: the requirements of storing and processing exponentially growing amounts of data already make its processing prohibitive. In this context, ILP shows especial promise as a tool enabling a shift from using hand-crafted background knowledge to *learning* background knowledge, including learning recursive programs that generalize from few examples. These issues, as well as future promising directions, have been recently surveyed by Cropper *et al.* (2020).

Further developing the ideas of ILP to encompass the automated learning of *probabilistic* logic programs is key to Statistical Relational AI (StaRAI), a successful hybrid field of AI (de Raedt *et al.* 2016), for which regular workshops have been held since 2010. Tools based on this paradigm aim to handle complex and large-scale problems involving elaborate relational structures and uncertainty.

### *Bridges to Established Research Areas*

Novel opportunities may then arise by bridging Prolog with other well-established research areas. This is, for instance, what happened with the Multi-Agent System community, where Prolog and LP have been extensively exploited in the last decades as the technological or conceptual basis for tens of agent-oriented technologies (Calegari *et al.* 2021). Similarly, the Prolog language has been proposed within the scope of Distributed Ledger Technologies (a.k.a. Blockchains), by Ciatto *et al.* (2018), or as a means to provide more declarative sorts of *smart contracts*, as suggested by Ciatto *et al.* (2019) and Ciatto *et al.* (2020). There, LP pursues the goal of making smart contracts declarative, hence easing their adoption, and increasing their expressiveness. More generally, beginning with the Prolog implementation of the British Nationality Act, Prolog has been used extensively for applications to computational law and is proabably still the dominant approach. For example, the logic programming language in Oracle Policy Automation[8] was originally implemented in Prolog.

In data science, several data sources need to be cleaned and combined before applying statistical analysis and machine learning. This preprocessing step is often the most labor intensive phase of a data science project. Prolog, particularly when extended with tabling support, is a suitable tool for data preprocessing. It can transparently access data from different sources without needing to repeatedly import all the data, for example, from a relational database management systems. Subsequently, a view on the data can be established from small composable predicates that can be debugged separately and interactively. These ideas have been explored in SWISH datalab (Bogaard *et al*. 2016) which provides a web front end for cooperative development of both Prolog data preprocessing steps and subsequent statistical analysis using R, and used in applications in Biology using large data (Angelopoulos and Wielemaker 2019). Other Prologs, such as Yap, include support for dealing with large data sets (Costa 2007; Costa and Vaz 2013).

Summarizing, it might prove very useful for the community to anticipate what features may attract programmers for ends such as improving AI, the Internet, programming languages, or knowledge intensive systems in general.

### *4.3 SWOT: Weaknesses*

Prolog is syntactically quite different from traditional mainstream programming languages. One could argue that this is a necessary side effect of many of its strengths, and that it is possibly a reason why Prolog has a place in many computer science curricula. However, it also means that Prolog can appear strange or unfamiliar to many beginners or even seasoned programmers. Mastering Prolog also implies a considerable learning curve and it certainly takes a while for a beginner to become truly productive in what is often a radically new language and paradigm.

While Prolog's dynamic typing can be an advantage for rapid prototyping and metaprogramming, it is also often considered a weakness. Optional static typing (see Section 3.3.3) would definitely help in catching many bugs early on.

Data hiding is also more difficult in Prolog. In the de-facto standard module system used by most Prologs one can decide which predicates are exported, but not which data types (i.e., functors and constants) are exported. One can typically not prevent another module from manipulating internal data structures (see, however, Section 3.3.1). This issue goes hand-in-hand with the limited support for object orientation (see, however, Section 4.5.4).

Another issue is the limited support for user interface (UI) development within Prolog. There was more attention to this issue in the past (e.g., BIM-Prolog had the Carmen library for this purpose), and there were interesting approaches in which declarativeness and/or constraints were exploited for this purpose, but nowadays UIs are usually developed in a foreign language via FLI (Section 3.3.1). Similarly, there is limited support for developing mobile applications or embedded software with Prolog, even if a number of

[8] https://documentation.custhelp.com/euf/assets/devdocs/cloud19b/PolicyAutomation/en/Default.htm#Guides/Welcome/Welcome.htm.

Prologs can run on small devices and Prolog program optimization techniques have been shown to be up to the task (Carro *et al.* 2006).

Portability of Prolog code can also be nontrivial, despite the ISO standard (see Section 4.5.1). Also, the Prolog community does not have a standard package manager, making it more difficult to distribute libraries or modules. (On the upside, Prolog also does not have all the version management hell and security issues prevalent in other languages.)

Finally, the support for Prolog in some integrated development environments can be less mature than for mainstream languages like Java (see Section 4.5.2). In particular, the available support for refactoring is often quite limited.

## *4.4 SWOT: Threats*

Some threats to Prolog come from competing programming languages. Indeed, Prolog may be perceived as an "old" language, and new programming languages such as previously Java, or now Rust or Go, may be more appealing to new generations of programmers. Also, for some application domains, like web-based applications, which have obviously become increasingly important in recent years, other languages like JavaScript are much more popular and easier to deploy than Prolog, although this is being addressed by current Prolog implementations.

Another threat to the perception of Prolog comes from the fact that, when teaching it, if the presentation does not go deep enough to convey the real power and elegance of the language and the programming paradigm, it may instead leave the incorrect impression of being a shallow or just academic tool.

On the other hand, as mentioned before, Prolog systems are still very actively developed, with new features and new implementations appearing continuously. While this is obviously positive, some threats we see for Prolog stem precisely from a resulting divergence of implementations (which tends to fragment the community into several, changing user camps) and a lack of strong stewardship (which, in turn, may fuel further divergence of systems).

We discuss the aforementioned issues in more detail in the following two sections.

### *4.4.1 Fragmentation of the community*

A large threat to the future of Prolog as an accessible programming language is further fragmentation of the community. This manifests itself in two dimensions:

*User Bases* Today, the Prolog language lacks a strong, united presence on the Web, especially when compared to other languages, such as the Python Homepage (2021), Rust Homepage (2021), R Homepage (2021), Clojure Homepage (2021) or Julia Homepage (2021). Hence, there is limited exchange between users of Prolog, in particular of different Prolog systems. Only a few shared platforms exist, such as the venerable comp.lang.prolog newsgroup, the Prolog subreddit, or StackOverflow. They have in common that they are not very active and are mostly used for announcements or beginner questions. There are some exceptions, such as the SWI-Prolog discourse forum, hosting active discussions on the implementation, but, again, it is a barrier since it is essentially local to this system. The lack of an overarching presence, thus, lowers Prolog's visibility, hinders development

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

of shared libraries, and might be an entry barrier for new Prolog users. It may even deter programmers from adapting LP technology due to outdated or wrong information on Wikipedia pages on Prolog or Logic Programming in general, or due to missing FAQs, API documentation, and tutorials.

The current fragmentation of the Prolog community may be seen as a threat to the language standardization effort and, thus, the advancement of Prolog. Compilation of standardization documents takes a very long time and is currently driven by a few volunteers. While only experts can evaluate the impact of changes in the standard on existing Prolog systems, the community may assist such efforts by pinpointing differences between systems, prioritizing features that should be considered next, or by providing test cases. However, the community is hard to reach in a unified way and such contributions are often limited to motivated individuals and remain scarce.

*Implementors* While implementors of Prolog systems meet at the CICLOPS workshops and other conferences, they do not have a good shared infrastructure to revise syntax, discuss libraries, tools and technical questions, or to offer existing code or tests. Without such workflows and regular discussion of future directions, Prolog systems may further diverge in features, libraries, and possibly even from the ISO standard if no concerted effort is made to find common best solutions. New developers need a forum to ask questions and benefit from lessons learned. Otherwise, implementation work may be duplicated, no common Prolog tooling will be developed, etc.

### *4.4.2 Lack of strong stewardship*

The largest single threat we see for the future compatibility of Prolog systems and attractiveness of the language in the long term is the lack of a strong stewardship. All implementors we were able to reach for comments agree with the need for compatibility, are willing to discuss issues and work on their systems to address them. However, the most lacking resource is time, as compatibility work diverts efforts from research and development.

A dedicated entity, which acts as a steward, calls meetings on a regular basis to ensure progress, and oversees open issues is missing and necessary, but establishing it is not without challenges. Sufficient financial and/or institutional backing to motivate implementors and to fund at least one steward position would be an asset. Additional human resource positions may be needed to address specific aspects, such as interoperability, source code compatibility, and website maintenance.

Two major collective efforts, the ISO process and the Prolog Commons group, have already been established that can be regarded as such entities and persist with varying success:

*The ISO Process* The ISO working group has the strong mission to provide a robust and concise standard that makes Prolog attractive for the industry, giving it a competitive advantage. The core standard was a huge leap in the right direction, providing a strong basis for compatibility. However, further progress has been rather slow, which may be due to the nature of ISO as well as the voluntary nature of the work of the participants and their aspirations for stability and high-quality work. Unfortunately, only a few of the

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

actual system implementors are currently active in the ISO standardization efforts. Also, the process is complex and may be too slow despite steady progress, since even more and more features and libraries are developed independently.

*Prolog Commons* The Prolog Commons project started as a series of informal implementor meetings to improve the portability of Prolog code, in a more agile and interactive setting than the ISO process. This impetus pushed many participating systems in converging directions, producing changes in their documentation systems, improved compatibility/emulation layers, and plans for further integrations. The reason that the project has not fully materialized into a common code base is, again, the lack of stewardship, meeting schedules, and deadlines, combined with the available time of the implementors.

### *4.5 Improving Prolog*

In preparation of this article, we reached out to researchers from different application areas of Prolog in order to gauge the most pressing topics, as well as to Prolog implementors in order to assess the potential for convergence of Prolog systems. Based on the results of this survey and on the SWOT analysis in Sections 4.1–4.4, we now discuss the issues and areas of improvements which we feel are most pressing.

#### *4.5.1 Portability of existing features*

The differences between the various Prolog implementations, either in the set of features provided (cf. Table 2) or in the way the features work, lead to a *code portability problem* between Prolog implementations. Circumventing this problem rather than solving it results in fragmenting the community into smaller, noncompatible subgroups. This makes it hard for users to find support and stay interested.

Strong and universally accepted standards for available features may raise interest from programmers and industry. Yet, if some combination of them are missing in many Prolog systems, it will negatively impact the perception of Prolog. One of the nonstandard offers of Prolog is constraint logic programming. Yet, choosing a Prolog system requires certain trade-offs concerning the available features. Many of these features concern performance, such as tabling, efficient coroutining via block annotations, and multiargument indexing. Some programmers may not want to give up what they are used to from other programming languages, like multithreaded, concurrent and distributed programming, standard libraries for formatting and pretty printing, efficient hash map data structures, and universally available data structures such as AVL trees or sets. Finally, some features are needed to embed Prolog as a component in a larger software system, for example, nonblocking IO, interfaces to other programming languages or (de-)serialization support, such as fastwrite/fastread, XML, JSON, YAML.

#### *4.5.2 Improved development tools*

An important area of future improvement is in the development tools available for the language. The dynamic nature and complex control of Prolog raises new challenges in the implementation of some of these features, but its clear logical foundations provide

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

an advantage for others. We see a potential and a need for the following improvements to the Prolog tooling ecosystem:

- Increased capabilities in debuggers (e.g., constraint debugging or graphical interaction), performance profiling tools, and testing frameworks. The combination of static and dynamic verification, debugging, and testing of Ciao is relevant here (Hermenegildo *et al.* 2005; Sanchez-Ordaz *et al.* 2021).
- Better experience using interactive shells: Prolog could also profit from being integrated within notebooks such as Jupyter, as has been done in SWISH (Wielemaker *et al.* 2019).
- More regular, user-friendly, and standardized ways of interfacing with other languages. Features that should be accounted for include nondeterminism, convenient and safe data types, and memory management coexistence.
- More capable IDEs, in particular with good refactoring tools, which help the programmer to safely reorder or change the arguments of predicates, rename, or move predicates to other modules, etc., in the line of SICStus' SPIDER (Carlsson and Mildner 2012). Prolog itself may be used as the GUI programming language in developing the IDE.
- Linters that enforce community-sourced coding guidelines based on, for example, the proposals by Covington *et al.* (2012) and Nogatz *et al.* (2019).
- Improved code location facilities, such as Ciao's *Semantic Search* (Garcia-Contreras *et al.* 2016).
- Other static program analysis tools, enabling for instance the inspection of dependencies among modules, and the existence of call hierarchies among predicates. Also, dynamic process inspection tools aimed at visualizing call stacks, choice points queues, proof trees, etc. (see also Section 4.5.3, State Inspection).

#### *4.5.3 Application- and domain-specific needs*

In this section, we consider a few domains that we think may influence Prolog in the future and try to anticipate future developments that are needed to satisfy their needs.

*Parsing* The parsing domain has implications both for computing sciences, in that it can be applied to compilers or other program transformations, and for sciences and the arts, in that many kinds of human languages (e.g., written, spoken, or those of molecular biology or music) can be computationally processed for various ends through parsing.

As far as pure parsing is concerned, one can of course easily write top-down recursive descent parsers in Prolog using DCGs. However, encoding deterministic parsing with lookahead requires the careful use of the cut, which is tedious and error-prone. Ideally, the cuts could be automatically generated by standard compiler construction algorithms (Nullable, First, Follow) and a Prolog-specific parser generator. Bottom-up parsers are also easy to write using, for instance, CHR. Another way to improve Prolog's parsing capabilities is through memoing, which avoids the infinite loops that left-recursive grammars are prone to, thus needing to resort less to the cut, and avoids recomputations of unfinished constituents in the case of alternative analyses where one analysis is subsumed by another. This is done by storing them in a table so they need not be recomputed upon backtrack (e.g., in "Marie Curie discovered Polonium in 1898," the partial analyses of the

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

verb phrase as just a verb or as a verb plus an object are stored in a table for reuse rather than disappearing upon failure and backtrack). Memoing can be easily implemented in Prolog, for example, through assumptions, as discussed by Christiansen and Dahl (2018), or through CHR, as discussed by Frühwirth (2009), or using tabling as in XSB Prolog.

*NLP and Neural Networks* Neural-network approaches to NLP use word embedding strategies to generalize from known facts to plausible ones (e.g., BERT Devlin *et al.* 2019, GPT-3 Floridi and Chiriatti 2020). They typically train only on form, in that they retrieve those responses that are statistically pertinent, with no regard to meaning. Their results are unstable, since they rely on ever larger and changing internet-mined data. While this approach has achieved noteworthy performance milestones in machine translation, sentence completion, and other standard benchmarking tasks, it offers a priori no way to learn meaning (Bender and Koller 2020) and relies on undocumented, un-retraceable, or otherwise partial (and therefore unaccountable) data, which tends to perpetuate harm without recourse (Birhane 2021). It is also resource-intensive, both computationally and energetically, and prone to spectacular failure (Marcus and Davis 2020). It seems that overcoming such drawbacks will need inferential programming capabilities, which integrations with deductive reasoning might help achieve. We anticipate that efforts in that direction, which are already happening (Sun *et al.* 2020), will be increasingly needed.

Another NLP area which we anticipate will require much attention and that Prolog can be ideal for is that of under-resourced human languages. Very few of the over 7000 languages in existence have at their disposal the computational tools that are needed for their adequate processing. Since texts on the Web are also overwhelmingly in mainstream languages, and the machine learning approaches that are in vogue typically rely on mining massive volumes of text, when these do not exist (or as soon as the existing ones become more protected) we need more logical approaches, such as grammatical inference by grammar transformation.

*Problem Solving, Solvers* Constraint programming blends nicely within Prolog (see Section 3.3.2), in the form of CLP($\mathcal{FD}$), CLP($\mathcal{B}$), CLP($\mathcal{R}$), or CLP($\mathcal{Q}$), and also in the CHR form, which lets users define constraint solvers for their own domain of interest. However, the binding to SAT, SMT, or ASP solvers is often still quite awkward. In particular, ASP with its Prolog syntax could be made available as a seamless extension to Prolog. Similarly, SAT and SMT solving could also be linked in a seamless way to Prolog facts or clauses. In an ideal world, one could even link various solvers via shared variables and coroutines.

*Visualization and GUIs* Visualization is nowadays most often performed via the foreign language interface or by exporting data to external tools (e.g., dot text files for GraphViz). While BIM-Prolog (cf. Section 2.10.1) had a declarative graphical toolkit, unfortunately Prolog systems have moved away from this approach. Other communities, however, are discovering the advantages of a "declarative approach" to visualization and user-interface design (e.g., React Gackenheimer and Paul 2015 in the JavaScript world). Maybe it is time again to implement visualizations or user interfaces within Prolog itself.

*State Inspection* Prolog 0 already included primitives to programmatically inspect the computation state. These can be useful for example in debugging or in parsing, in order to implement context-sensitive grammars. Current Prologs allow different degrees of state inspection, including for example the classic facilities that enable meta-programming. Such primitives could acquire new relevance under the increasing needs for further transparency and inspectability in AI applications.

#### *4.5.4 Prolog aspects that need joint, public, and earnest discussion*

There are a number of issues that would need early discussion in the process of giving a new impetus to Prolog standardization. While we raise questions here, we cannot speak on behalf of the entire community. Thus, we think that a visible (in the sense of commonly-known) platform for public discussion between implementors and users is required. The Prolog language should eventually evolve on results of discussions and needs of (potential) programmers. For some issues we discuss below, several solutions have been offered by the research community. However, the Prolog community has to discuss what features shall be adapted as standard. Further, one might also consider whether purely declarative implementations are preferable or even feasible, since some features, such as assert and retract, do not even have a declarative, logical semantics. The following points are some examples:

*Types, Modes, and Other Properties* Often, during development, a Prolog program may terminate without finding a solution, get stuck in an infinite loop, backtrack unexpectedly, etc. Often, this sort of situation is due to type errors in the code. Should a type (and mode, etc.) system be part of an improved Prolog, allowing for more powerful static analysis? If so, what should it look like? What would have to be changed for a useful gradual static type system that allows one to progressively add types to existing code? Should more general properties than classical types and modes be supported? Should static types be combined with dynamic checks? With testing? The logic programming community has been pioneering in these areas with research, solutions, and systems well ahead of other languages, but it has not yet seen widespread use. Ciao's comprehensive approach to this overall topic could shed some light here.

*Reactivity* A rule in Prolog can be viewed as a part of a definition that defines the predicate in the head of that rule in terms of the predicates in the body of the rule. In contrast, most rules in imperative languages are reactive rules that perform actions to change state, as in the case of condition-action rules in production systems and event-condition-action rules in active database systems. Extensions of logic programming to include reactive rules have been developed in CHR (see Chapter 6 in Frühwirth 2009) and in the Logic-based Production System Language LPS (Kowalski and Sadri 2015; Wielemaker *et al*. 2019).

*Module System* As mentioned before, the second part of the ISO standard regarding modules was universally ignored and most Prolog systems settled for a Quintus-inspired module system. However, the implementations incorporate some deviations to support new features. Rules regarding visibility of operators, predicates, and perhaps atoms should be reconsidered. This is addressed to some extent, for example, in the Ciao and SWI module systems but, again, with some differences, which should however not be too difficult to bridge. Systems support some of the legacy code loading methods such as `consult/1` and `ensure_loaded/1` for backwards compatibility, but `use_module/{1,2}` is recommended, specially for large-scale applications. New solutions must address errors and inconsistencies that have already been uncovered by Haemmerlé and Fages (2006) and later by Moura (2009a).

*Objects* An aspect which is closely tied to the module system is that of integrating logic programming with object-oriented features. This has been an elusive goal but, nevertheless, several effective proposals have been put forward which achieve ways of doing whole-program composition more in line with the Logic Programming paradigm. Monteiro and Porto (1989) present the basic concepts of Contextual Logic Programming while Bugliesi *et al.* (1994) offers an extensive discussion on this topic. A more advanced design, in which modules (called *units*) may take parameters and where the context itself becomes a first-class object is provided by GNU Prolog/CX (Abreu and Diaz 2003), which appears as both an object-oriented extension and a dynamic program-structuring mechanism. An application of this system is described by Abreu and Nogueira (2005).

Another example is the O'Ciao model, which implements a source-to-source approach to objects as a conservative *extension* of the Prolog module system (Pineda and Bueno 2002). Logtalk offers a different approach, also based on source-to-source transformations, that adds a separate object-oriented layer on top of many Prolog systems. Thus, it regards objects as a *replacement* for a module system.

*Interfaces* Beyond modules, programming in the large can be supported by mechanisms that allow describing and enforcing separation between interfaces and implementations. Such interfaces are typically sets of signatures, specifying types and other properties, to be matched by all implementations, and thus allow stating what a given implementation must meet, and making implementations interchangeable in principle. They should include at least constructs to express the external connections of a module, not only in terms of the predicates it should expose but also the types, modes, and other characteristics of their arguments, and a compiler/interpreter capable of *enforcing* those interfaces at compile- or loading time. Ciao's assertion language, preprocessor, and interface definitions offer a possible solution in this area.

*Syntactical Support for Data Structures* One of Prolog's strengths is a minimalistic syntax. SWI-Prolog's syntactic support for strings and dictionaries responds to a demand for interfacing with prevalent protocols and exchange formats, for example, XML and YAML. Other systems have other extensions such as feature terms (Aït-Kaci 1993; Aït-Kaci *et al.* 1994).

Many questions need to be answered: Should syntactical support for data types such as associative data structures (feature terms) or strings be standardized? Would the current syntax be affected (e.g., a prevalent syntax for maps, `{"key": "value"}`, might break DCGs)? What are trade-offs the community is willing to take? Should other containers, such as linear access arrays, sets, and character types, be included? How should unification of variable elements work?

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

*Library Infrastructure and Conditional Code* As Moura's efforts concerning Logtalk (cf. Section 3.1) showed, it is possible to support large portable applications across almost all Prolog systems. Yet, a nontrivial amount of conditional code for abstraction libraries is needed to provide support for each new Prolog system. Is the community willing to develop libraries supporting more than a couple of systems? Is a stronger standardized core library required to attract programmer? Is the language that emerges from the portability layer still Prolog?

*Macro System* Many Prolog systems support the source-to-source transformation of terms and goals, via the so-called *term expanders*. It is a powerful feature that is not part of the standard and in some cases can be clunky to use and error-prone since term expanders are often not local to a module, they are order dependent and every term or goal, relevant or not, is fed into the transformation predicate. Ciao shows how module locality and priorities can be used in this context. Or should a more lightweight macro system be made available for code transformation?

*Functional Programming Influences* Curry and Mercury (cf. Section 3.5) are well known for combining the functional and logic programming paradigms. Some Prolog systems such as Ciao have strong support for functional syntax and higher-order. Most Prolog systems provide support for meta-predicates such as `call` and some derivatives such as `maplist`, `filter`, `and reduce`. Should a notation for lambda expressions be introduced as done in, for example, Ciao's predicate abstractions?

*Standard Test Programs Beyond ISO* While test suites for ISO Prolog exist, many features outside the core language are prevalent in a lot of Prolog systems. Unstandardized features, such as modules, constraint programming, or DCGs, are provided by almost all modern Prolog systems, yet implementors rarely share test programs. Obviously, creation of and agreement on tests is challenging, especially since some systems might want to maintain their features as they are. Availability of such shared tests can guarantee that systems behave similarly, allow implementors to test compatibility with other systems, and foster standardization of these features. As a positive example in this line, Logtalk's Prolog test suite is quite comprehensive, covering not only ISO- and de-facto standard features, but many other tests of predicates and notable features such as modules, threads, constraints, coroutining, Unicode support or unbounded integer arithmetic.

### *4.6 Summary and next steps*

Despite differences and incompatibilities between Prolog systems and thus user code, the ensuing divergences are not fundamental. It is our belief that the best initiatives that were put in place in the past, with some tweaks that we hope to have covered, can be re-applied to make much stronger progress in the future.

The necessary implementor involvement seems attainable: most surveyed implementors and users agree that efforts for a common ground should be made, namely by meetings and the sharing of ideas, implementations and even common infrastructure. It is important that the entire community be involved in creating and maintaining the existing core standard as well as debating desired features, in order that upcoming standards take hold and be respected. Since support tools are a must for all modern programming languages, it is important to pool resources. In particular, mature debuggers, as well as developing and testing environments, are often required by Prolog programmers.

The diversity that is unique to Prolog's ecosystem is a strength which should be taken advantage of. In short, Prolog systems need to be useful and usable so they can be more universally employed.

Accordingly, to coordinate converging efforts, an organization, perhaps a Foundation, could be established, either independently or as an initiative within, for instance, the ALP. With converging systems and tool infrastructures, possibilities in community participation and user documentation can vastly improve, rendering Prolog much more attractive to new users both in academia and industry, thereby building a more unified community.

Standard procedures already exist for Prolog improvement, such as the ISO standardization. We feel these would benefit from being complemented by less formal ones, such as the Prolog Commons.

A time- and cost-efficient first step could be to create a web platform to publicly share feature extensions and modifications and propose new ones. It could also provide the community with a forum for public debate and gather pointers to previous and complementary efforts. It could be linked to from the ALP website.

Another productive step would be to define a structured workflow aimed at tracking, supporting, and coordinating the evolution of proposals, efforts, and further initiatives.

Teaching Prolog to children could be most influential: they would be learning logic and computing at the same time, and Prolog literacy would spread much more rapidly through them. There was much activity in this area in the 1980s with the Prolog Education Group (PEG) with its annual workshops. More recently, "Logical English" (Kowalski 2020), although focused primarily on legal applications, including smart contracts, is simply sugared syntax for "pure" Prolog, making it suitable for teaching to children (and others).

Finally, it could be useful for the community to analyze all the relevant Wikipedia pages and other web content in search of misinformation, outdated information, or in general inaccurate content on Prolog or logic programming, in the aim of correcting it.

## 5 Conclusions

We have provided an analysis of the Prolog language and community from the historical, technical, and organizational perspectives.

The first contribution of the paper (Section 2) has been a historical discussion tracking the early evolution of the Prolog language and the genealogy of the many implementations available nowadays. This discussion stemmed from a general definition of what constitutes a "Prolog system" and covered the most notable implementations and milestones in chronological order.

We have seen how Prolog implementations, and even the language itself, have experienced significant evolution and extensions over the years. Despite the common roots, the maintenance of the Prolog name throughout these transformations, and the rich exchange of ideas among system implementers, inevitably some parts of each system have progressed somewhat independently. This has naturally led to some divergence.

In order to assess the current situation, the second contribution of the paper (Section 3) has consisted in a technical survey of the current status of the main Prolog systems available today, comparing them on a per-feature basis with the objective of identifying commonalities and divergences. In addition, we have also tried to identify the core characterizing aspect(s) of each Prolog system, that is, what makes each system unique.

We have observed that there is widespread adherence to the ISO standard and its extensions. More differences understandably appear in the added features that lie beyond ISO. At the same time we have seen that many of these extensions are common to many Prolog systems, even if there are often differences in the interfaces. We have also observed that some Prologs offer unique characteristics which would be very useful to have in other systems. Such differences in general affect portability and make it harder for a programmer to write Prolog programs that work and behave consistently across systems.

However, we have observed that the divergences between Prolog systems are not fundamental. It seems that a good number of the differences could be easily bridged by agreeing on interfaces and/or standard implementations that can be easily ported to all systems.

Wenger *et al.* (2002) suggest that communities reach one or more crucial points where they either fade away, or, alternatively, they find new impetus to start further cycles. We believe that our community has reached such crucial points several times to date, at junctures such as the loss of interest in parallelism in the early 90s (only to come back in full swing, over a decade later), the end of the Fifth Generation project, the advent of Constraints, the long AI winter (also with a mighty comeback), the appearance of the Semantic Web, the advent of ASP, etc. In all previous occasions the community has been able to adapt to these environmental changes and incorporate new scientific and technological advances as extensions to the language and the different implementations, and has continued to produce truly state-of-the-art programming systems, with advanced characteristics that often appeared ahead of many other paradigms.

We believe we are now at one of these crucial points where action is needed. The present Prolog systems, and Prolog as a whole, continue being important and unique within the programming languages landscape, and it is encouraging to see that after all this time new Prolog implementations, new features, and new uses of Prolog appear continuously.

However, at the same time there is some risk of losing unity and strength in the community. This is partly due to its very success (in that it has spawned new communities or successfully complemented existing ones) and partly to community fragmentation and divergences in systems due to new functionalities.

Thus, the last contribution of the paper (Section 4) has been to provide an analysis, including strengths, weaknesses, opportunities and threats, as well as some proposals for the LP community aimed at addressing the most relevant issues concerning the Prolog ecosystem, in a coordinated way. We argue that a joint, inclusive, and coordinated effort involving Prolog implementers and users is necessary. The great initiatives consisting of a small group of experts, such as the ISO working group or the Prolog Commons project, with some tweaks that we hope to have covered, can be re-invigorated to make stronger progress in the future. However, they may not be enough to bring together a user community that contributes to the language and its growth. Accordingly, to apply the lessons learned from the past, we stress the need for both community involvement and stewardship, including perhaps the ALP, for ensuring that the effort is kept moving forward consistently.

Good initiatives are already being undertaken with the declaration of 2022 as the "Year of Prolog"[9]: its celebrations will include an Alain Colmerauer Prize awarded by an international jury for recent application-oriented work of Prolog-inspired technology and its potential for more human-oriented computing in the future, and a "Prolog School Bus" that will travel to reintroduce declarative programming concepts to the younger generation. It is expected that both initiatives will continue into the future, with a yearly prize also forseen for the "nicest" Prolog program written by a student participating in the school caravan.

## Conflicts of interest

The authors declare none.

## References

Abramson, H. and Dahl, V. 1989. *Logic Grammars*. Springer.

Abreu, S. and Diaz, D. 2003. Objective: In minimum context. In *Proceedings ICLP*, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 2916. Springer, 128–147.

Abreu, S. and Nogueira, V. 2005. Using a logic programming language with persistence and contexts. In *Procedings INAP*, M. Umeda, A. Wolf, O. Bartenstein, U. Geske, D. Seipel and O. Takata, Eds. Lecture Notes in Computer Science, vol. 4369. Springer, 38–47.

Adebara, I. and Dahl, V. 2016. Grammar induction as automated transformation between constraint solving models of language. In *Proceedings KnowProS*, R. Bartak, T. L. McCluskey and E. Pontelli, Eds. CEUR Workshop Proceedings, vol. 1648. CEUR.

Aït-Kaci, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press.

Aït-Kaci, H. 1993. An introduction to LIFE – Programming with logic, inheritance, functions and equations. In *Proceedings ILPS*, D. Miller, Ed. MIT Press, 52–68.

Aït-Kaci, H., Dumant, B., Meyer, R., Podelski, A. and Roy, P. V. 1994. *The Wild LIFE Handbook*. Digital Paris Research Laboratory. Prepublication Edition.

Aït-Kaci, H. and Podelski, A. 1991. Towards a meaning of LIFE. In *Proceedings PLILP*, J. Maluszyński and M. Wirsing, Eds. LNCS, vol. 528. Springer, 255–274.

Akers, S. B. 1978. Binary decision diagrams. *IEEE Transactions on Computers 27,* 6, 509–516.

Ali, K. A. and Karlsson, R. 1990. The Muse approach to OR-parallel Prolog. *International Journal of Parallel Programming 19,* 2, 129–162.

Alpuente, M., Feliú, M. A., Joubert, C. and Villanueva, A. 2010. Datalog-based program analysis with BES and RWL. In *Proceedings Datalog 2.0 Workshop*, O. de Moor, G. Gottlob, T. Furche and A. J. Sellers, Eds. Lecture Notes in Computer Science, vol. 6702. Springer, 1–20.

Alviano, M., Dodaro, C., Faber, W., Leone, N. and Ricca, F. 2013. WASP: A native ASP solver based on constraint learning. In *Proceedings LPNMR*, P. Cabalar and T. C. Son, Eds. Lecture Notes in Computer Science, vol. 8148. Springer, 54–66.

Angelopoulos, N., Costa, V. S., Azevedo, J., Wielemaker, J., Camacho, R. and Wessels, L. F. A. 2013. Integrative functional statistics in logic programming. In *Proceedings PADL*, K. Sagonas, Ed. Lecture Notes in Computer Science, vol. 7752. Springer, 190–205.

[9] http://www.prolog-heritage.org/en/Prolog_50.html.

Angelopoulos, N. and Wielemaker, J. 2019. Advances in big data bio analytics. In *Proceedings ICLP (Technical Communications)*, B. Bogaerts, E. Erdem, P. Fodor, A. Formisano, G. Ianni, D. Inclezan, G. Vidal, A. Villanueva, M. D. Vos and F. Yang, Eds. EPTCS, vol. 306. CoRR, 309–322.

Antoy, S. and Hanus, M. 2010. Functional logic programming. *Communications of the ACM 53,* 4, 74–85.

Apt, K. and Wallace, M. 2007. *Constraint Logic Programming Using ECLiPSe*. Cambridge University Press.

Apt, K. R. 2001. Viewpoint: One more revolution to make: Free Scientific Publishing. *Communications of the ACM 44,* 5, 25–28.

Arias, J. and Carro, M. 2016. Description and evaluation of a generic design to integrate CLP and tabled execution. In *Proceedings PPDP*. ACM, 10–23.

Arias, J., Carro, M., Salazar, E., Marple, K. and Gupta, G. 2018. Constraint answer set programming without grounding. *Theory and Practice of Logic Programming 18,* 3-4, 337–354.

Armstrong, J. 2007. A history of Erlang. In *Proceedings HOPL*. ACM, 1–26.

B Prolog Updates Note 2021. URL: http://www.picat-lang.org/bprolog/updates.html. [Accessed on 11 March, 2022].

Bagnara, R. 1999. Is the ISO Prolog standard taken seriously? *Association for Logic Programming Newsletter 12,* 1, 10–12.

Bagnara, R. and Carro, M. 2002. Foreign language interfaces for Prolog: A terse survey. *Association for Logic Programming Newsletter 15,* 2.

Basilio, R., Zaverucha, G. and Carneiro Barbosa, V. 2001. Learning logic programs with neural networks. In *Proceedings ILP*, C. Rouveirol and M. Sebag, Eds. Lecture Notes in Computer Science, vol. 2157. Springer, 15–26.

Baumeister, J. and Seipel, D. 2010. Anomalies in ontologies with rules. *Journal of Web Semantics 8,* 1, 55–68.

Bavarian, M. and Dahl, V. 2006. Constraint based methods for biological sequence analysis. *Journal of Universal Computer Science 12*, 1500–1520.

Beckert, B. and Posegga, J. 1995. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning 15,* 3, 339–358.

Becket, R. and Somogyi, Z. 2008. DCGs + memoing = Packrat parsing but is it worth it? In *Proceedings PADL*, P. Hudak and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 4902. Springer, 182–196.

Beldiceanu, N. and Contejean, E. 1994. Introducing global constraints in CHIP. *Mathematical and computer Modelling 20,* 12, 97–123.

Bender, E. M. and Koller, A. 2020. Climbing towards NLU: On meaning, form, and understanding in the age of data. In *Proceedings ACL*. Association for Computational Linguistics, 5185–5198.

Benichou, M., Beringer, H., Gauthier, J.-M. and Beierle, C. 1992. Prolog at IBM: An advanced and evolving application development technology. *IBM Systems Journal 31,* 4, 755–773.

Birhane, A. 2021. Algorithmic injustice: A relational ethics approach. *Patterns 2,* 2, 100205.

Blackburn, P., Bos, J. and Striegnitz, K. 2006. *Learn Prolog Now!* Vol. 7. College Publications Londres.

Bogaard, T., Wielemaker, J., Hollink, L. and van Ossenbruggen, J. 2016. SWISH DataLab: A web interface for data exploration and analysis. In *Proceedings BNAIC*, T. Bosse and B. Bredeweg, Eds. Communications in Computer and Information Science, vol. 765. Springer, 181–187.

Boizumault, P. 1986. A general model to implement DIF and FREEZE. In *Proceedings ICLP*, E. Shapiro, Ed. Lecture Notes in Computer Science, vol. 225. Springer, 585–592.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

Bolz, C. F., Cuni, A., Fijalkowski, M., Leuschel, M., Pedroni, S. and Rigo, A. 2011. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings PEPM*. ACM, 43–52.

Bonatti, P. A. 2001. Resolution for skeptical stable model semantics. *Journal of Automated Reasoning 27,* 4, 391–421.

Bonatti, P. A. 2010. Datalog for security, privacy and trust. In *Proceedings Datalog 2.0 Workshop*, O. de Moor, G. Gottlob, T. Furche and A. J. Sellers, Eds. Lecture Notes in Computer Science, vol. 6702. Springer, 21–36.

Bonatti, P. A., Pontelli, E. and Son, T. C. 2008. Credulous resolution for answer set programming. In *Proceedings AAAI*, D. Fox and C. P. Gomes, Eds. AAAI Press, 418–423.

Bone, P., Somogyi, Z. and Schachte, P. 2012. Controlling loops in parallel mercury code. In *Proceedings DAMP*. ACM, 11–20.

Börger, E. and Rosenzweig, D. 1995. The WAM - Definition and compiler correctness. In *Logic Programming: Formal Methods and Practical Applications*, C. Beierle and L. Plümer, Eds. Studies in Computer Science and Artificial Intelligence, vol. 11. Elsevier/North-Holland, 20–90.

Bowen, D., Byrd, L. and Clocksin, W. 1983. A portable Prolog compiler. In *Proceedings Logic Programming Workshop*, L. M. Pereira, A. Porto, L. Monteiro and M. Filgueiras, Eds. Universidade NOVA de Lisboa. Núcleo de Intelegência Artificial., 74–83.

Boyer, R. and Moore, J. 1972. The sharing structure in theorem-proving programs. *Machine Intelligence 7*, 101–116.

BProlog Homepage 2021. B-PROLOG. URL: http://www.picat-lang.org/bprolog. [Accessed on 11 March, 2022].

Bratko, I. 2012. *Prolog Programming for Artificial Intelligence*, 4th ed. Addison-Wesley.

Brewka, G., Eiter, T. and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM 54,* 12, 92–103.

Brewka, G., Eiter, T. and Truszczynski, M. 2016. Answer set programming: An introduction to the special issue. *AI Magazine 37,* 3, 5–6.

Bruynooghe, M. 1976. An Interpreter for Predicate Logic Programs. Part I: Basic Principles. Tech. Rep. CW 10, Department of Computer Science, KU Leuven. URL: https://lirias.kuleuven.be/retrieve/522231.

Bruynooghe, M. 2021. Prolog Documents. URL: https://people.cs.kuleuven.be/~Maurice.Bruynooghe/Prolog/Prolog.html. [Accessed 11 March, 2022].

Bueno, F., García de la Banda, M. and Hermenegildo, M. V. 1999. Effectiveness of abstract interpretation in automatic parallelization: A case study in logic programming. *Transactions on Programming Languages and Systems 21,* 2, 189–238.

Bugliesi, M., Lamma, E. and Mello, P. 1994. Modularity in logic programming. *Journal of Logic Programming 19/20*, 443–502.

Byrd, L. 1980. Understanding the control flow of Prolog programs. In *Proceedings Logic Programing Workshop*, S. Åke Tärnlund, Ed., 127–138.

Cabeza, D. and Hermenegildo, M. V. 1996. Implementing distributed concurrent constraint execution in the CIAO system. In *Procedings APPIA-GULP-PRODE*, 67–78.

Cabeza, D. and Hermenegildo, M. V. 2000. A new module system for Prolog. In *Proceedings CL*, J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv and P. J. Stuckey, Eds. Lecture Notes in Computer Science, vol. 1861. Springer, 131–148.

Calegari, R., Ciatto, G., Dellaluce, J. and Omicini, A. 2019. Interpretable narrative explanation for ML predictors with LP: A case study for XAI. In *Proceedings WOA 2019*, F. Bergenti and S. Monica, Eds. CEUR Workshop Proceedings, vol. 2404. CEUR, 105–112.

Calegari, R., Ciatto, G., Denti, E. and Omicini, A. 2020. Logic-based technologies for intelligent systems: State of the art and perspectives. *Information 11,* 3, 1–29.

Calegari, R., Ciatto, G., Mariani, S., Denti, E. and Omicini, A. 2018. LPaaS as micro-intelligence: Enhancing IoT with symbolic reasoning. *Big Data and Cognitive Computing 2,* 3.

Calegari, R., Ciatto, G., Mascardi, V. and Omicini, A. 2021. Logic-based technologies for multi-agent systems: A systematic literature review. *Autonomous Agents and Multi-Agent Systems 35,* 1, 1:1–1:67.

Calegari, R., Ciatto, G. and Omicini, A. 2020. On the integration of symbolic and sub-symbolic techniques for XAI: A survey. *Intelligenza Artificiale 14,* 1, 7–32.

Carlson, B., Carlsson, M. and Diaz, D. 1994. Entailment of finite domain constraints. In *Proceedings ICLP*, P. V. Hentenryck, Ed. MIT Press, 339–353.

Carlsson, M. 1984. On Implementing Prolog in functional programming. *New Generation Computing 2,* 4, 347–359.

Carlsson, M. 1986. SICStus: Preliminary Specification. Draft.

Carlsson, M. and Mildner, P. 2012. SICStus Prolog – the first 25 years. *Theory and Practice of Logic Programming 12,* 1-2, 35–66.

Carro, M. and Hermenegildo, M. 1999. Concurrency in Prolog using threads and a shared database. In *Proceedings ICLP*, D. D. Schreye, Ed. MIT Press, 320–334.

Carro, M., Morales, J., Muller, H., Puebla, G. and Hermenegildo, M. V. 2006. High-level languages for small devices: A case study. In *Proceedings CASES*, K. Flautner and T. Kim, Eds. ACM Press/Sheridan, 271–281.

Chen, W. and Warren, D. S. 1996. Computation of stable models and its integration with logical query processing. *IEEE Transactions on Knowledge and Data Engineering 8,* 5, 742–757.

Chirico, U. 2021. JIProlog–java internet Prolog. URL: http://www.jiprolog.com/. [Accessed 11 March, 2022].

Christiansen, H. 2002. Logical grammars based on constraint handling rules. In *Proceedings ICLP*, P. J. Stuckey, Ed. Lecture Notes in Computer Science, vol. 2401. Springer, 481.

Christiansen, H. and Dahl, V. 2005. HYPROLOG: A new logic programming language with assumptions and abduction. In *Proceedings ICLP*, M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer, 159–173.

Christiansen, H. and Dahl, V. 2018. Natural language processing with (tabled and constraint) logic programming. In *Declarative Logic Programming: Theory, Systems, and Applications*, M. Kifer and Y. A. Liu, Eds. ACM, 477–511.

Ciao Prolog Homepage 2021. The Ciao programming language. URL: https://ciao-lang.org. [Accessed 11 March, 2022].

Ciatto, G., Calegari, R., Mariani, S., Denti, E. and Omicini, A. 2018. From the Blockchain to logic programming and back: Research perspectives. In *Proceedings WOA 2018*, M. Cossentino, L. Sabatucci and V. Seidita, Eds. CEUR Workshop Proceedings, vol. 2215. CEUR, 69–74.

Ciatto, G., Calegari, R. and Omicini, A. 2021a. 2P-Kt: A logic-based ecosystem for symbolic AI. *SoftwareX 16*, 100817, 1–7.

Ciatto, G., Calegari, R. and Omicini, A. 2021b. Lazy stream manipulation in Prolog via backtracking: The case of 2P-Kt. In *Proceedings JELIA 2021*, W. Faber, G. Friedrich, M. Gebser and M. Morak, Eds. Lecture Notes in Computer Science, vol. 12678. Springer, 407–420.

Ciatto, G., Calegari, R., Siboni, E., Denti, E. and Omicini, A. 2020. 2P-Kt: Logic programming with objects & functions in Kotlin. In *Proceedings WOA 2020*, R. Calegari, G. Ciatto, E. Denti, A. Omicini and G. Sartor, Eds. CEUR Workshop Proceedings, vol. 2706. CEUR, 219–236.

Ciatto, G., Maffi, A., Mariani, S. and Omicini, A. 2019. Towards agent-oriented Blockchains: Autonomous smart contracts. In *Proceedings PAAMS*, Y. Demazeau, E. Matson, J. M. Corchado and F. De la Prieta, Eds. Lecture Notes in Computer Science, vol. 11523. Springer, 29–41.

Ciatto, G., Mariani, S., Omicini, A. and Zambonelli, F. 2020. From agents to Blockchain: Stairway to integration. *Applied Sciences 10,* 21, 7460:1–7460:22.

Ciatto, G., Schumacher, M. I., Omicini, A. and Calvaresi, D. 2020. Agent-based explanations in AI: Towards an abstract framework. In *Proceedings EXTRAAMAS*, D. Calvaresi, A. Najjar, M. Winikoff and K. Främling, Eds. Lecture Notes in Computer Science, vol. 12175. Springer, 3–20.

Clark, K. and Gregory, S. 1986. PARLOG: Parallel programming in logic. *Transactions on Programming Languages and Systems 8,* 1, 1–49.

Clark, K. L. 1978. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Springer, 293–322.

Clocksin, W. 1997. *Clause and Effect*. Springer.

Clocksin, W. F. and Mellish, C. 1981. *Programming in Prolog*. Springer.

Clojure Homepage 2021. Clojure. URL: https://clojure.org/. [Accessed 11 March, 2022].

CMU Artificial Intelligence Repository. 1995. SB-Prolog: Stony Brook Prolog v.3.1.01. URL: https://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/impl/prolog/sbprolog/0.html. [Accessed 11 March, 2022].

Codognet, P. and Diaz, D. 1995. `wamcc`: Compiling Prolog to C. In *Proceedings ICLP*, L. Sterling, Ed. MIT Press, 317–331.

Codognet, P. and Diaz, D. 1996. Compiling constraints in clp(FD). *Journal of Logic Programming 27,* 3, 185–226.

Cohen, J. 1979. Non-deterministic algorithms. *Computing Surveys 11,* 2, 79–94.

Cohen, J. 1988. A view of the origins and development of Prolog. *Communications of the ACM 31,* 1, 26–36.

Colmerauer, A. 1970a. Les systèmes Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur. Tech. Rep. Internal publication 43, Département d'Informatique, Université de Montréal.

Colmerauer, A. 1970b. Total precedence relations. *Journal of the ACM 17,* 1, 14–30.

Colmerauer, A. 1975. Les grammaires de métamorphose GIA. Internal publication, Groupe Intelligence artificielle, Faculté des Sciences de Luminy, Université Aix-Marseille II, France, Nov 1975. English version, Metamorphosis grammars. In *Natural Language Communication with Computers*, L. Bolc, Ed. Lecture Notes in Computer Science, vol. 63. Springer, 1978, 133–189.

Colmerauer, A. 1979. Un sous-ensemble intéressant du français. *RAIRO. Informatique théorique et Applications 13,* 4, 309–336.

Colmerauer, A. 1982a. Prolog and infinite trees. In *Proceedings Workshop on Logic Programming 1980*, K. L. Clark and S. Åke Tärnlund, Eds. Academic Press, 231–251.

Colmerauer, A. 1982b. *Prolog-II: Manuel de référence et modèle théorique*. G.I.A. Université Aix-Marseille.

Colmerauer, A. 1984. Equations and inequations on finite and infinite trees. In *Proceedings FGCS*. North-Holland, 85–99.

Colmerauer, A. 1990. An introduction to Prolog III. In *Proceedings Computational Logic*, J. W. Lloyd, Ed. Springer, 37–79.

Colmerauer, A. 1996. Les bases de Prolog IV. Tech. rep., Laboratoire d'Informatique de Marseille.

Colmerauer, A., Kanoui, H., Pasero, R. and Roussel, P. 1973. Un système de communication homme-machine en français. Rapport préliminaire de fin de contrat IRIA. Tech. rep., Faculté des Sciences de Luminy, Université Aix-Marseille II.

Colmerauer, A. and Roussel, P. 1996. The birth of Prolog. In *Proceedings HOPL*, T. J. Bergin and R. G. Gibson, Eds. ACM, 331–367.

Costa, V. S. 1999. Optimising bytecode emulation for Prolog. In *Proceedings PPDP*, G. Nadathur, Ed. Lecture Notes in Computer Science, vol. 1702. Springer, 261–277.

Costa, V. S. 2007. Prolog performance on larger datasets. In *Proceedings PADL*, M. Hanus, Ed. Lecture Notes in Computer Science, vol. 4354. Springer, 185–199.

Costa, V. S., Rocha, R. and Damas, L. 2012. The YAP Prolog system. *Theory and Practice of Logic Programming 12,* 1-2, 5–34.

Costa, V. S., Sagonas, K. and Lopes, R. 2007. Demand-driven indexing of Prolog clauses. In *International Conference on Logic Programming*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, 395–409.

Costa, V. S. and Vaz, D. 2013. BigYAP: Exo-compilation meets UDI. *Theory and Practice of Logic Programming 13,* 4-5, 799–813.

Costa, V. S., Warren, D. H. and Yang, R. 1991. Andorra I: A parallel Prolog system that transparently exploits both and-and or-parallelism. In *Proceedings PPOPP*, R. L. Wexelblat, Ed. ACM, 83–93.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings POPL*. ACM, 238–252.

Covington, M. A., Bagnara, R., O'Keefe, R. A., Wielemaker, J. and Price, S. 2012. Coding guidelines for Prolog. *Theory and Practice of Logic Programming 12,* 6, 889–927.

Cropper, A., Dumančić, S. and Muggleton, S. H. 2020. Turning 30: New ideas in inductive logic programming. In *Proceedings IJCAI*, C. Bessiere, Ed. International Joint Conferences on Artificial Intelligence Organization, 4833–4839.

Cyras, K., Rago, A., Albini, E., Baroni, P. and Toni, F. 2021. Argumentative XAI: A survey. In *Proceedings IJCAI*, Z.-H. Zhou, Ed. International Joint Conferences on Artificial Intelligence Organization, 4392–4399.

Dahl, V. 1977. Un système déductif d'interrogation de banques de données en espagnol. Ph.D. thesis, Université d'Aix-Marseille II.

Dahl, V. 1979. Quantification in a three-valued logic for natural language question-answering systems. In *Proceedings IJCAI*. William Kaufmann, 182–187.

Dahl, V. 1982. On database systems development through logic. *Transactions on Database Systems 7,* 1, 102–123.

Dahl, V. 1986. Gramaticas discontinuas: una herramienta computacional con aplicaciones en la teoria de Reccion y Ligamiento. *Revista Argentina de Lingüística 2,* 2, 375–392.

Dahl, V. 1990. Describing linguistic knowledge about constraints in user-friendly ways. *International Journal of Expert Systems: Research and Applications 3,* 2, 131–146.

Dahl, V. 1992. Comment on implementing government-binding theories. In *Formal Linguistics: Theory and Practice*, R. Levine, Ed. Oxford University Press, 276–289.

Dahl, V., Bel-Enguix, G., Miralles, E. and Tirado, V. to appear. Grammar induction for under-resourced languages: The case of Ch'ol. In *Proceedings AVERTIS 2019. Essays Dedicated to Manuel Hermenegildo on the Occasion of His 60th Birthday*, J. Gallagher, R. Giacobazzi and P. Lopez-Garcia, Eds. Lecture Notes in Computer Science. Springer.

Dahl, V. and McCord, M. 1983. Treating coordination in logic grammars. *Computational Linguistics 9*, 69–91.

Dahl, V. and Miralles, J. E. 2012. Womb grammars: Constraint solving for grammar induction. In *Proceedings CHR*, J. Sneyers and T. Frühwirth, Eds. Report CW, vol. 624. KU Leuven, 32–40.

Dahl, V., Popowich, F. and Rochemont, M. 1993. A principled characterization of dislocated phrases: Capturing barriers with static discontinuity grammars. *Linguistics and Philosophy 16*, 331–352.

Dahl, V. and Sambuc, R. 1976. Un système de banque de données en logique du premier ordre, en vue de sa consultation en langue naturelle. Tech. rep., D.E.A. Report, Université d'Aix-Marseille II.

Dahl, V. and Tarau, P. 1998. From assumptions to meaning. *Canadian Artificial Intelligence 42*, 26–29.

Dahl, V. and Tarau, P. 2004. Assumptive logic programming. In *Proceedings ASAI*.

Dahl, V., Tarau, P. and Li, R. 1997. Assumption grammars for processing natural language. In *Proceedings ICLP*, L. Naish, Ed. MIT Press, 256–270.

Dahl, V., Tarau, P., Moreno, L. and Palomar, M. 1995. Treating Coordination with Datalog Grammars. CoRR cmp-lg/9505006, 1–17.

da Silva, A. F. and Santos Costa, V. 2007. Design, implementation, and evaluation of a dynamic compilation framework for the YAP system. In *Proceedings ICLP*, V. Dahl and I. Niemelä, Eds. LNCS, vol. 4670. Springer, 410–424.

d'Avila Garcez, A. S. and Zaverucha, G. 1999. The connectionist inductive learning and logic programming system. *Applied Intelligence 11,* 1, 59–77.

De Angelis, E., Fioravanti, F., Gallagher, J. P., Hermenegildo, M. V., Pettorossi, A. and Proietti, M. 2021. Analysis and transformation of constrained horn clauses for program verification. *Theory and Practice of Logic Programming, 20th Anniversary Special Issue*.

de Kergommeaux, J. C. and Codognet, P. 1994. Parallel logic programming systems. *Computing Surveys 26,* 3, 295–336.

de Raedt, L., Dumancic, S., Manhaeve, R. and Marra, G. 2020. From statistical relational to neuro-symbolic artificial intelligence. In *Proceedings IJCAI*, C. Bessiere, Ed. International Joint Conferences on Artificial Intelligence Organization, 4943–4950.

de Raedt, L., Kersting, K., Natarajan, S. and Poole, D. 2016. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

de Raedt, L., Kimmig, A. and Toivonen, H. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings IJCAI*, M. M. Veloso, Ed. International Joint Conferences on Artificial Intelligence Organization, 2462–2467.

Denti, E., Omicini, A. and Calegari, R. 2013. tuProlog: Making Prolog ubiquitous. *Association for Logic Programming Newsletter*.

Denti, E., Omicini, A. and Ricci, A. 2001. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In *Proceedings PADL*, I. Ramakrishnan, Ed. Lecture Notes in Computer Science, vol. 1990. Springer, 184–198.

Deransart, P., Ed-Dbali, A. and Cervoni, L. 1996. *Prolog: The Standard*. Springer.

Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings NAACL*. Association for Computational Linguistics, 4171–4186.

Diaz, D., Abreu, S. and Codognet, P. 2012. On the implementation of GNU Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 253–282.

Diaz, D. and Codognet, P. 1993. A minimal extension of the WAM for clp(FD). In *Proceedings ICLP*, D. S. Warren, Ed. MIT Press, 774–790.

Diaz, D. and Codognet, P. 2000. GNU Prolog: Beyond compiling Prolog to C. In *Proceedings PADL*, E. Pontelli and V. S. Costa, Eds. Lecture Notes in Computer Science, vol. 1753. Springer, 81–92.

Dietrich, R. and Hagl, F. 1988. A polymorphic type system with subtypes for Prolog. In *Proceedings ESOP*, H. Ganzinger, Ed. Lecture Notes in Computer Science, vol. 300. Springer, 79–93.

Dincbas, M., Hentenryck, P. V., Simonis, H. and Aggoun, A. 1988. The constraint logic programming language CHIP. In *Proceedings FGCS*. IOS Press, 249–264.

Dovier, A., Formisano, A., Gupta, G., Hermenegildo, M. V., Pontelli, E. and Rocha, R. 2022. Parallel logic programming: A sequel. *Theory and Practice of Logic Programming, 20th Anniversary Special Issue*.

Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM 13,* 2, 94–102.

ECLiPSe Prolog Homepage 2021. The ECLiPSe Constraint Programming System. URL: https://eclipseclp.org. [Accessed 11 March, 2022].

Elcock, E. 1990. Absys: The first logic programming language. *Journal of Logic Programming 9,* 1, 1–17.

Ethnologue 2021. Ethnologue: Languages of the World. URL: https://www.ethnologue.com/. [Accessed 11 March, 2022].

Felleisen, M. 1985. Transliterating Prolog into Scheme. Tech. Rep. 182, Indiana University. Department of Computer Science.

Flanagan, C. 2006. Hybrid type checking. In *Proceedings POPL*. ACM, 245–256.

Floridi, L. and Chiriatti, M. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines 30,* 4, 681–694.

Floyd, R. W. 1967. Nondeterministic algorithms. *Journal of the ACM 14,* 4, 636–644.

Forgy, C. L. 1989. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Readings in Artificial Intelligence and Databases*. Elsevier, 547–559.

França, M. V. M., Zaverucha, G. and d'Avila Garcez, A. S. 2014. Fast relational learning using bottom clause propositionalization with artificial neural networks. *Machine Learning 94,* 1, 81–104.

Frühwirth, T. 1992. Constraint simplification rules. In *Proceedings JICSLP Workshop on Constraint Logic Programming*. (Also available as Technical Report ECRC-92-18, ECRC Munich, Germany, July 1992).

Frühwirth, T. 2009. *Constraint Handling Rules*. Cambridge University Press.

Gackenheimer, C. and Paul, A. 2015. *Introduction to React*, vol. 52. Springer.

Gallagher, J. P. and Henriksen, K. S. 2004. Abstract domains based on regular types. In *International Conference on Logic Programming*. Springer, 27–42.

Gallaire, H., Minker, J. and Nicolas, J. 1984. Logic and databases: A deductive approach. *ACM Computing Surveys 16*, 153–185.

Garcia-Contreras, I., Morales, J. F. and Hermenegildo, M. V. 2016. Semantic code browsing. *Theory and Practice of Logic Programming, ICLP 2016 Special Issue 16,* 5-6, 721–737.

García de la Banda, M., Bueno, F. and Hermenegildo, M. 1996. Towards independent and-parallelism in CLP. In *Proceedings PLILP*, H. Kuchen and S. D. Swierstra, Eds. Lecture Notes in Computer Science, vol. 1140. Springer, 77–91.

García de la Banda, M., Hermenegildo, M. V. and Marriott, K. 2000. Independence in CLP languages. *ACM Transactions on Programming Languages and Systems 22,* 2, 269–339.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T. and Thiele, S. 2008. A User's Guide to gringo, clasp, clingo, and iclingo. URL: https://potassco.org/. [Accessed on 11 March, 2022].

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2014. Clingo = ASP + Control: Preliminary Report. arXiv 1405.3694.

GNU Prolog 2021. The GNU Prolog web site. URL: http://gprolog.org. [Accessed 11 March, 2022].

Green, C. C. 1969a. Application of theorem proving to problem solving. In *Proceedings IJCAI*, D. E. Walker and L. M. Norton, Eds. William Kaufmann, 219–240.

Green, C. C. 1969b. *The Application of Theorem Proving to Question-Answering Systems*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York.

Guidotti, R., Monreale, A., Turini, F., Pedreschi, D. and Giannotti, F. 2019. A survey of methods for explaining Black Box models. *Computing Surveys 51,* 5, 1–42.

Gunning, D. 2016. Explainable Artificial Intelligence (XAI). Funding Program DARPA-BAA-16-53, Defense Advanced Research Projects Agency (DARPA).

Gupta, G., Bansal, A., Min, R., Simon, L. and Mallya, A. 2007. Coinductive logic programming and its applications. In *Proceedings ICLP*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, 27–44.

Gupta, G. and Pontelli, E. 1999. Extended dynamic dependent and-parallelism in ACE. *Journal of Functional and Logic Programming 99,* Special Issue 1.

Gupta, G., Pontelli, E., Ali, K., Carlsson, M. and Hermenegildo, M. V. 2001. Parallel execution of Prolog programs: A survey. *Transactions on Programming Languages and Systems 23,* 4, 472–602.

Gupta, G., Saeedloei, N., DeVries, B. W., Min, R., Marple, K. and Kluzniak, F. 2011. Infinite computation, co-induction and computational logic. In *Proceedings CALCO*, A. Corradini, B. Klin, and C. Cîrstea, Eds. Lecture Notes in Computer Science, vol. 6859. Springer, 40–54.

Gurr, C. A. 1994. Specialising the ground representation in the logic programming language Gödel. In *Proceedings LOPSTR 1993*, Y. Deville, Ed. Workshops in Computing. Springer, 124–140.

Haemmerlé, R. and Fages, F. 2006. Modules for Prolog revisited. In *Proceedings ICLP*, S. Etalle and M. Truszczyński, Eds. Lecture Notes in Computer Science, vol. 4079. Springer, 41–55.

Hankley, W. 1987. Feature analysis of Turbo Prolog. *ACM SIGPLAN Notices 22,* 3, 111–118.

Hanus, M. 1997. Teaching functional and logic programming with a single computation model. In *Proceedings PLILP*, H. Glaser, P. Hartel and H. Kuchen, Eds. Lecture Notes in Computer Science, vol. 1292. Springer, 335–350.

Hanus, M. and Krone, J. 2017. A typeful integration of SQL into curry. In *Proceedings WLP 2015/2016 and WFLP 2016*, S. Schwarz and J. Voigtländer, Eds. EPTCS, vol. 234. CoRR, 104–119.

Hanus, M., Kuchen, H. and Moreno-Navarro, J. 1995. Curry: A truly functional logic language. In *Proceedings Workshop on Visions for the Future of Logic Programming*, 95–107.

Haynes, C. T. 1986. Logic continuations. In *Proceedings ICLP*, E. Shapiro, Ed. Lecture Notes in Computer Science, vol. 225. Springer, 671–685.

Hentenryck, P. V., Saraswat, V. A. and Deville, Y. 1994. Design, implementation, and evaluation of the constraint language cc(FD). In *Proceedings TCS School*, A. Podelski, Ed. Lecture Notes in Computer Science, vol. 910. Springer, 293–316.

Henz, M., Smolka, G. and Würtz, J. 1993. Oz - A programming language for multi-agent systems. In *Proceedings IJCAI*, R. Bajcsy, Ed. Morgan Kaufmann, 404–409.

Hermenegildo, M. 1986. An abstract machine for restricted AND-parallel execution of logic programs. In *Proceedings ICLP*. Lecture Notes in Computer Science, vol. 225. Springer, 25–40.

Hermenegildo, M. 2000. Parallelizing irregular and pointer-based computations automatically: Perspectives from logic and constraint programming. *Parallel Computing 26,* 13–14, 1685–1708.

Hermenegildo, M. and CLIP Group, T. 1993. Towards CIAO-Prolog – A parallel concurrent constraint system. In *Proceedings of the Compulog Net Area Workshop on Parallelism and Implementation Technologies*. FIM/UPM.

Hermenegildo, M., Puebla, G., Bueno, F. and Garcia, P. L. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and The Ciao System Preprocessor). *Science of Computer Programming 58,* 1-2, 115–140.

Hermenegildo, M. V. 1989. High-performance Prolog Implementation: The WAM and Beyond (tutorial). Tech. rep., ICLP.

Hermenegildo, M. V. 2012. Conferences vs. Journals in CS, what to do? Evolutionary ways forward and the ICLP/TPLP model. In *Publication Culture in Computing Research — Position Papers*, K. Mehlhorn, M. Y. Vardi and M. Herbstritt, Eds. Dagstuhl Reports, vol. 12452. Dagstuhl Publishing, 29–30. URL: http://cliplab.org/papers/dagstuhl-tplpmove.pdf.

Hermenegildo, M. V., Bueno, F., Cabeza, D., Carro, M., Garcia de la Banda, M., López-García, P. and Puebla, G. 1996. The CIAO multi-dialect compiler and system: An experimentation workbench for future (C)LP systems (extended abstract). In *Proceedings AGP*, P. Lucio, M. Martelli and M. Navarro, Eds., 105–110.

Hermenegildo, M. V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. F. and Puebla, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming 12,* 1-2, 219–252.

Hermenegildo, M. V., Puebla, G. and Bueno, F. 1999. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*, K. R. Apt, V. W. Marek, M. Truszczynski and D. S. Warren, Eds. Springer, 161–192.

Hermenegildo, M. 1994. Some methodological issues in the design of CIAO - A generic, parallel, concurrent constraint system. In *Proceedings PPCP*, A. Borning, Ed. Lecture Notes in Computer Science, vol. 874. Springer, 123–133.

Hewitt, C. 1969. PLANNER: A language for proving theorems in robots. In *Proceedings IJCAI*, D. E. Walker and L. M. Norton, Eds. William Kaufmann, 295–302.

Hill, P. and Lloyd, J. W. 1994. *The Gödel Programming Language*. MIT press.

Hoarau, S. and Mesnard, F. 1998. Inferring and compiling termination for constraint logic programs. In *Proceedings LOPSTR*, P. Flener, Ed. Lecture Notes in Computer Science, vol. 1559. Springer, 240–254.

Holzbaur, C. 1992. Metastructures vs. attributed variables in the context of extensible unification. In *Proceedings PLILP*, M. Bruynooghe and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 631. Springer, 260–268.

Holzbaur, C. 1995. OFAI CLP(Q,R) Manual, Edition 1.3.3. Tech. Rep. TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna.

Huang, S. S., Green, T. J. and Loo, B. T. 2011. Datalog and emerging applications: An interactive tutorial. In *Proceedings MOD*. ACM, 1213–1216.

Ireson-Paine, J. 2010. The public-domain Prolog library. URL: http://www.j-paine.org/prolog/library.html. [Accessed 11 March, 2022].

ISO/IEC 13211-1. 1995. Information technology – Programming languages – Prolog – Part 1: General core. ISO Standard, International Organization for Standardization.

ISO/IEC 13211-2. 2000. Information technology – Programming languages – Prolog – Part 2: Modules. ISO Standard, International Organization for Standardization.

ISO/IEC 14882. 2020. Programming languages — C++. ISO Standard, International Organization for Standardization.

Jaffar, J. and Lassez, J. 1987. Constraint logic programming. In *Proceedings POPL*. ACM, 111–119.

Jaffar, J. and Maher, M. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

Jaffar, J., Michaylov, S., Stuckey, P. J. and Yap, R. H. C. 1992. The CLP($\mathcal{R}$) language and system. *Transactions on Programming Languages and Systems 14,* 3, 339–395.

Janson, S. and Haridi, S. 1991. Programming Paradigms of the Andorra Kernel Language. Tech. Rep. R-91/8-SE, SICS.

Janssens, G. 1984. WIC — Warren Improved Code. Tech. rep., BIM. Internal Report IR1, URL: https://people.cs.kuleuven.be/~Maurice.Bruynooghe/Prolog/IR01.pdf.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

Johnson, K., Pasquale, F. and Chapman, J. 2019. Artificial intelligence, machine learning, and bias in finance: Toward responsible innovation. *Fordham Law Review 88,* 2, 499–529.

Johnson, K. and Rae, R. 1983. Edinburgh Prolog Tools. Tech. Rep. Note No. 103, Programming Systems Group, AI Applications Institute, Edinburgh University.

Jordan, H., Scholz, B. and Subotic, P. 2016. Soufflé: On synthesis of program analyzers. In *Proceedings CAV, Part II*, S. Chaudhuri and A. Farzan, Eds. Lecture Notes in Computer Science, vol. 9780. Springer, 422–430.

Julia Homepage 2021. Julia. URL: https://julialang.org/. [Accessed 11 March, 2022].

Kacsuk, P. and Wise, M. 1992. *Implementations of Distributed Prolog.* John Wiley & Sons, Inc.

Kaufmann, M. and Boyer, R. S. 1995. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications 29,* 2, 27–62.

Kay, M. 1967. Experiments with a powerful parser. In *Proceedings COLING*, vol. 1, 1–20.

Kowalski, R. 1979. Algorithm = Logic + Control. *Communications of the ACM 22,* 7, 424–436.

Kowalski, R. 2013. Logic programming in the 1970s. In *Proceedings LPNMR*, P. Cabalar and T. C. Son, Eds. Lecture Notes in Computer Science, vol. 8148. Springer, 11–22.

Kowalski, R. 2020. Logical English. In *Proceedings LPOP*, D. Warren, P. V. Roy and Y. A. Liu, Eds. 33–37.

Kowalski, R. and Kuehner, D. 1971. Linear resolution with selection function. *Artificial Intelligence 2,* 3, 227–260.

Kowalski, R. and Sadri, F. 2015. Reactive computing as model generation. *New Generation Computing 33,* 1, 33–67.

Kowalski, R. A. 1974. Predicate logic as programming language. In *Proceedings IFIP*, J. L. Rosenfeld, Ed. North-Holland, 569–574.

Kowalski, R. A. 1988. The early years of logic programming. *Communications of the ACM 31,* 1, 38–43.

Lally, A., Prager, J. M., McCord, M. C., Boguraev, B., Patwardhan, S., Fan, J., Fodor, P. and Chu-Carroll, J. 2012. Question analysis: How Watson reads a clue. *Journal of Research and Development 56,* 3.4, 2:1–2:14.

Lamb, L. C., d'Avila Garcez, A. S., Gori, M., Prates, M. O. R., Avelar, P. H. C. and Vardi, M. Y. 2020. Graph neural networks meet neural-symbolic computing: A survey and perspective. In *Proceedings IJCAI*, C. Bessiere, Ed. International Joint Conferences on Artificial Intelligence Organization, 4877–4884.

λProlog Home Page 2021. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/. [Accessed 11 March, 2022].

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3, 499–562.

Leuschel, M. 2008. Declarative programming for verification: Lessons and outlook. In *Proceedings PPDP*. ACM Press, 1–7.

Leuschel, M. 2020. Fast and effective well-definedness checking. In *Proceedings IFM*, B. Dongol and E. Troubitsyna, Eds. Lecture Notes in Computer Science, vol. 12546. Springer, 63–81.

Leuschel, M. and Bruynooghe, M. 2002. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming 2,* 4 & 5, 461–515.

Leuschel, M. and Butler, M. J. 2008. ProB: An automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer 10,* 2, 185–203.

Lifschitz, V. 1999. Action languages, answer sets, and planning. In *The Logic Programming Paradigm*, K. R. Apt, V. W. Marek, M. Truszczynski and D. S. Warren, Eds. Springer, 357–373.

Lindholm, T., Yellin, F., Bracha, G., Buckley, A. and Smith, D. 2021. *The Java Virtual Machine Specification (Java SE 17 Edition)*. Oracle.

Lipton, Z. C. 2018. The Mythos of model interpretability. *Queue 16,* 3.

Lopes, R., Costa, V. S. and Silva, F. M. A. 2012. A design and implementation of the extended Andorra model. *Theory and Practice of Logic Programming 12,* 3, 319–360.

Lopez-Garcia, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F. and Hermenegildo, M. V. 2018. Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption. *Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification 18,* 2, 167–223.

LPA Ltd. 2021. About LPA. URL: https://www.lpa.co.uk/hom_lpa.htm. [Accessed 11 March, 2022].

Lusk, E., Butler, R., Disz, T., Olson, R., Overbeek, R., Stevens, R., Warren, D. H. D., Calderwood, A., Szeredi, P., Haridi, S., Brand, P., Carlsson, M., Ciepielewski, A. and Hausman, B. 1990. The Aurora or-parallel Prolog system. *New Generation Computing 7,* 2-3, 243–271.

Madsen, M. and Lhoták, O. 2020. Fixpoints for the masses: Programming with first-class datalog constraints. *Proceedings of the ACM on Programming Languages 4,* OOPSLA, 125:1–125:28.

Maher, M. J. 1987. Logic semantics for a class of committed-choice programs. In *Proceedings ICLP*, J.-L. Lassez, Ed. MIT Press, 858–876.

Maier, D., Tekle, K. T., Kifer, M. and Warren, D. S. 2018. Datalog: Concepts, history, and outlook. In *Declarative Logic Programming: Theory, Systems, and Applications*, M. Kifer and Y. A. Liu, Eds. ACM, 3–100.

Manthey, R. and Bry, F. 1988. SATCHMO: A theorem prover implemented in Prolog. In *Proceedings CADE*, R. O. Ewing Lusk, Ed. Lecture Notes in Computer Science, vol. 310. Springer, 415–434.

Marcus, G. and Davis, E. 2020. GPT-3, Bloviator: OpenAI's language generator has no idea what it's talking about. *Technology Review*.

Marek, V. W. and Truszczynski, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm - A 25-Year Perspective*, K. R. Apt, V. W. Marek, M. Truszczynski and D. S. Warren, Eds. Springer, 375–398.

Marple, K., Salazar, E. and Gupta, G. 2017. Computing Stable Models of Normal Logic Programs Without Grounding. arXiv 1709.00501.

Marriott, K. and Stuckey, P. 1998. *Programming with Constraints: An Introduction*. MIT Press.

McCarthy, J. 1962. *LISP 1.5 Programmer's Manual*. MIT Press.

McCord, M. 2006. The Slot Grammar Lexical Formalism. Tech. rep., IBM Research Division, Thomas J. Watson Research Center.

McJones, P. 2021. Prolog and Logic Programming Historical Sources Archive. URL: http://www.softwarepreservation.org/projects/prolog/index.html. [Accessed 11 March, 2022].

Mellish, C. S. 1979. Short Guide to UNIX PROLOG Implementation. Tech. rep., Department of Artificial Intelligence, University of Edinburgh.

Méndez-Lojo, M., Navas, J. and Hermenegildo, M. 2007. A flexible (C)LP-based approach to the analysis of object-oriented programs. In *Proceedings LOPSTR*, A. King, Ed.. LNCS, vol. 4915. Springer, 154–168.

Mera, E., Lopez-Garcia, P. and Hermenegildo, M. V. 2009. Integrating software testing and run-time checking in an assertion verification framework. In *Proceedings ICLP*, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer, 281–295.

Mesnard, F., Payet, É. and Neumerkel, U. 2002. Detecting optimal termination conditions of logic programs. In *Proceedings SAS*, M. V. Hermenegildo and G. Puebla, Eds. Lecture Notes in Computer Science, vol. 2477. Springer, 509–526.

Miller, D. 2021. Reciprocal influences between proof theory and logic programming. *Philosophy & Technology 34*.

Miller, D. and Nadathur, G. 2012. *Programming with Higher-Order Logic*. Cambridge University Press.

Monteiro, L. and Porto, A. 1989. Contextual logic programming. In *Proceedings ICLP*, G. Levi and M. Martelli, Eds. MIT Press, 284–299.

Morales, J., Carro, M. and Hermenegildo, M. V. 2004. Improving the compilation of Prolog to C using moded types and determinism information. In *Proceedings PADL*, B. Jayaraman, Ed. Lecture Notes in Computer Science, vol. 3057. Springer, 86–103.

Morales, J., Carro, M. and Hermenegildo, M. V. 2008. Comparing tag scheme variations using an abstract machine generator. In *Proceedings PPDP*. ACM Press, 32–43.

Morales, J., Carro, M. and Hermenegildo, M. V. 2016. Description and optimization of abstract machines in a dialect of Prolog. *Theory and Practice of Logic Programming 16,* 1, 1–58.

Morales, J., Carro, M., Puebla, G. and Hermenegildo, M. 2005. A generator of efficient abstract machine implementations and its application to emulator minimization. In *Proceedings ICLP*, M. Gabbrielli and G. Gupta, Eds. LNCS, vol. 3668. Springer, 21–36.

Moura, P. 2003. *Design of an Object-Oriented Logic Programming Language*. Ph.D. thesis, Universidade da Beira Interior.

Moura, P. 2005. Prolog portability and standardization. *Association for Logic Programming Newsletter 18,* 3.

Moura, P. 2007. Prolog multi-threading support. URL: https://logtalk.org/plstd/threads.pdf.

Moura, P. 2009a. From plain Prolog to Logtalk objects: Effective code encapsulation and reuse. In *Proceedings ICLP*, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer, 23–23.

Moura, P. 2009b. Uniting the Prolog community: Personal notes. *Association for Logic Programming Newsletter 22,* 1.

Moura, P. 2011. Programming patterns for Logtalk parametric objects. In *Proceedings INAP/WLP*, S. Abreu and D. Seipel, Eds. Lecture Notes in Computer Science, vol. 3392. Springer, 52–69.

Muggleton, S. and de Raedt, L. 1994. Inductive logic programming: Theory and methods. *Journal of Logic Programming 19/20*, 629–679.

Muthukumar, K., Bueno, F., de la Banda, M. G. and Hermenegildo, M. 1999. Automatic compile-time parallelization of logic programs for restricted, goal-level, independent and-parallelism. *Journal of Logic Programming 38,* 2, 165–218.

Muthukumar, K. and Hermenegildo, M. 1989. Determination of variable dependence information at compile-time through abstract interpretation. In *Proceedings North American Conference on Logic Programming*, E. L. Lusk and R. A. Overbeek, Eds. MIT Press, 166–189.

Muthukumar, K. and Hermenegildo, M. 1990. The CDG, UDG, and MEL methods for automatic compile-time parallelization of logic programs for independent and-parallelism. In *Proceedings ICLP*. MIT Press, 221–237.

Mycroft, A. and O'Keefe, R. A. 1984. A polymorphic type system for Prolog. *Artificial intelligence 23,* 3, 295–307.

Nadathur, G. 2001. The metalanguage λprolog and its implementation. In *Proceedings FLOPS*, H. Kuchen and K. Ueda, Eds. Lecture Notes in Computer Science, vol. 2024. Springer, 1–20.

Nadathur, G. and Miller, D. 1988. An Overview of Lambda Prolog. Tech. Rep. 595, Department of Computer & Information Science, University of Pennsylvania.

Naish, L. 1982. An Introduction to MU-Prolog. Tech. Rep. 82/2, University of Melbourne.

Naish, L. 1985. Prolog control rules. In *Proceedings IJCAI*, A. K. Joshi, Ed. Morgan Kaufmann, 720–722.

Naish, L. 1986. *Negation and Control in Prolog*. Lecture Notes in Computer Science, vol. 238. Springer.

Nässén, H., Carlsson, M. and Sagonas, K. 2001. Instruction merging and specialization in the SICStus Prolog virtual machine. In *Proceedings PPDP*. ACM Press, 49–60.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J. and Tack, G. 2007. MiniZinc: Towards a standard CP modelling language. In *Proceedings CP*, C. Bessière, Ed. Lecture Notes in Computer Science, vol. 4741. Springer, 529–543.

Neumerkel, U. 1990. Extensible unification by metastructures. In *Proceedings META*.

Neumerkel, U. 1994. A transformation based on the equality between terms. In *Proceedings LOPSTR*, Y. Deville, Ed. Workshop in Computing. Springer, 162–176.

Neumerkel, U. 2013. SWI7 and ISO Prolog. URL: https://www.complang.tuwien.ac.at/ulrich/iso-prolog/SWI7_and_ISO. [Accessed 11 March, 2022].

Neumerkel, U. and Kral, S. 2002. Declarative program development in Prolog with GUPU. In *Proceedings WLPE*, A. Tessier, Ed. 77–86.

Neumerkel, U. W. 1992. *Specialization of Prolog Programs with Partially Static Goals and Binarization*. Ph.D. thesis, TU Wien.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25*, 241–273.

Noble, S. U. 2018. *Algorithms of Oppression: How Search Engines Reinforce Racism*. New York University Press.

Nogatz, F., Körner, P. and Krings, S. 2019. Prolog coding guidelines: Status and tool support. In *Proceedings ICLP (Technical Communications)*, B. Bogaerts, E. Erdem, P. Fodor, A. Formisano, G. Ianni, D. Inclezan, G. Vidal, A. Villanueva, M. D. Vos and F. Yang, Eds. EPTCS, vol. 306. CoRR, 8–21.

Nori, K. V., Ammann, U., Jensen, K. and Nägeli, H. 1974. The PASCAL 'P' Compiler: Implementation Notes. Tech. rep., Berichte des Instituts für Informatik, ETH Zürich.

O'Keefe, R. A. 1990. *The Craft of Prolog*. MIT Press.

Older, W. and Benhamou, F. 1993. Programming in CLP(BNR). In *Proceedings PPCP*, P. Kanellakis, J.-L. Lassez, and V. Saraswat, Eds. 239–249.

Omicini, A. and Denti, E. 2001. From tuple spaces to tuple centres. *Science of Computer Programming 41,* 3, 277–294.

Omicini, A. and Zambonelli, F. 1998. Co-ordination of mobile information agents in TuCSoN. *Internet Research 8,* 5, 400–413.

Panch, T., Mattie, H. and Atun, R. 2019. Artificial intelligence and algorithmic bias: implications for health systems. *Journal of Global Health 9,* 2.

Pasero, R. 1973. *Représentation du français en logique du premier ordre en vue de dialoguer avec un ordinateur*. Ph.D. thesis, Faculté des Sciences de Luminy, Université Aix-Marseille II.

Paxton, B. 2021. Erlang Companies. URL: https://erlang-companies.org/. [Accessed 11 March, 2022].

Pereira, F. C. 1983. *C-Prolog User's Manual*. EdCaad, University of Edinburgh.

Pereira, F. C. and Warren, D. H. 1980. Definite clause grammars for language analysis — A survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence 13,* 3, 231–278.

Pereira, F. C. N. and Shieber, S. M. 1987. *Prolog and Natural-Language Analysis*. CSLI. CSLI Publications, 185–210.

Pereira, L. M., Pereira, F. C. and Warren, D. H. 1978. *User's Guide to DECsystem-10 Prolog*. Department of Artificial Intelligence, University of Edinburgh.

Pereira, L. M. and Porto, A. 1979. Intelligent backtracking and sidetracking in Horn clause programs-the theory. Tech. Rep. CIUNL 2/79, Universidade Nova de Lisboa. Centro de Informatica.

Piancastelli, G., Benini, A., Omicini, A. and Ricci, A. 2008. The Architecture and design of a malleable object-oriented Prolog engine. In *Proceedings SAC*, R. L. Wainwright, H. M. Haddad, R. Menezes and M. Viroli, Eds., vol. 1. ACM, 191–197.

Pineda, A. and Bueno, F. 2002. The O'Ciao approach to object oriented logic programming. In *Proceedings CICLOPS*, B. Demoen, Ed. Report CW, vol. 344. KU Leuven, 37–48.

Pisano, G., Calegari, R., Omicini, A. and Sartor, G. 2020. Arg-tuProlog: A tuProlog-based argumentation framework. In *Proceedings CILC*, F. Calimeri, S. Perri and E. Zumpano, Eds. CEUR Workshop Proceedings, vol. 2719. CEUR, 51–66.

Pisano, G., Ciatto, G., Calegari, R. and Omicini, A. 2020. Neuro-symbolic computation for XAI: Towards a unified model. In *Proceedings WOA*, R. Calegari, G. Ciatto, E. Denti, A. Omicini and G. Sartor, Eds. CEUR Workshop Proceedings, vol. 2706. CEUR, 101–117.

Poole, D. 1993. Probabilistic horn abduction and Bayesian networks. *Artificial Intelligence 64,* 1, 81–129.

Prolog Commons Working Group 2021. URL: http://prolog-commons.org/. [Accessed 11 March, 2022].

Puebla, G., Bueno, F. and Hermenegildo, M. V. 2000a. A generic preprocessor for program validation and debugging. In *Analysis and Visualization Tools for Constraint Programming*, P. Deransart, M. V. Hermenegildo and J. Małuszynski, Eds. Lecture Notes in Computer Science, vol. 1870. Springer, 63–107.

Puebla, G., Bueno, F. and Hermenegildo, M. V. 2000b. An assertion language for constraint logic programs. In *Analysis and Visualization Tools for Constraint Programming*, P. Deransart, M. V. Hermenegildo and J. Małuszynski, Eds. Lecture Notes in Computer Science, vol. 1870. Springer, 23–61.

Python Homepage 2021. Welcome to python.org. URL: https://www.python.org/. [Accessed 11 March, 2022].

R Homepage 2021. R: The R Project for Statistical Computing. URL: https://www.r-project.org/. [Accessed 11 March, 2022].

RackLog Homepage 2021. The RackLog Language. URL: https://docs.racket-lang.org/racklog/index.html. [Accessed 11 March, 2022].

Ramakrishnan, R., Srivastava, D., Sudarshan, S. and Seshadri, P. 1994. The CORAL deductive system. *The International Journal on Very Large Data Bases 3,* 2, 161–210.

Ramakrishnan, R. and Ullman, J. D. 1993. A survey of research on deductive database systems. *Journal of Logic Programming 23,* 2, 125–149.

Rao, P., Sagonas, K., Swift, T., Warren, D. S. and Freire, J. 1997. XSB: A system for efficiently computing well founded semantics. In *Proceedings LPNMR*, J. Dix, U. Furbach and A. Nerode, Eds. Lecture Notes in Computer Science, vol. 1265. Springer, 430–440.

Rastogi, A., Swamy, N., Fournet, C., Bierman, G. and Vekris, P. 2015. Safe & efficient gradual typing for TypeScript. In *Proceedings POPL*. ACM, 167–180.

Reiter, R. 1978. On closed world data bases. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Springer, 55–76.

Riguzzi, F. 2007. A top down interpreter for LPAD and CP-logic. In *Proceedings AI*IA 2007*, R. Basili and M. T. Pazienza, Eds. Lecture Notes in Computer Science, vol. 4733. Springer, 109–120.

Riguzzi, F. 2018. *Foundations of Probabilistic Logic Programming*. River Publishers.

Roberts, G. M. 1977. *An Implementation of PROLOG*. Ph.D. thesis, University of Waterloo, Waterloo, Ontario, Canada. (Master Thesis).

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM 12,* 1, 23–41.

Rocha, R., Silva, F. M. A. and Costa, V. S. 2005. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming 5,* 1-2, 161–205.

Roussel, P. 1972. *Définition et traitement de l'égalité formelle en démonstration automatique*. Ph.D. thesis, Faculté des Sciences de Luminy, Université Aix-Marseille II.

Roy, P. V., Haridi, S., Schulte, C. and Smolka, G. 2020. A hmstory of the Oz multiparadigm language. In *Proceedings HOPL*. ACM, 83:1–83:56.

Rust Homepage 2021. Rust Programming Language. URL: https://www.rust-lang.org/. [Accessed 11 March, 2022].

Sagonas, K., Swift, T. and Warren, D. S. 1994. XSB as an efficient deductive database engine. *SIGMOD 23,* 2, 442–453.

Sagonas, K. F., Swift, T. and Warren, D. S. 1993. The XSB programming system. In *Proceedings Workshop on Programming with Logic Databases*, R. Ramakrishnan, Ed. Technical Report, vol. #1183. University of Wisconsin, 164.

Sammut, C. and Sammut, R. A. 1983. The implementation of UNSW-PROLOG. *Australian Computer Journal 15,* 2, 58–64.

Sanchez-Ordaz, M., Garcia-Contreras, I., Perez-Carrasco, V., Morales, J. F., Lopez-Garcia, P. and Hermenegildo, M. V. 2021. Verifly: On-the-fly Assertion checking via incrementality. *Theory and Practice of Logic Programming 21,* 6, 768–784.

Santos Costa, V., Sagonas, K. F. and Lopes, R. 2007. Demand-driven indexing of Prolog clauses. In *Proceedings ICLP*, V. Dahl and I. Niemelä, Eds. LNCS, vol. 4670. Springer, 395–409.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *Proceedings ICLP*, L. Sterling, Ed. MIT Press, 715–729.

Sato, T. and Kameya, Y. 1997. PRISM: A language for symbolic-statistical modeling. In *Proceedings IJCAI*. Morgan Kaufmann, 1330–1339.

Schellhorn, G. 1999. *Verification of Abstract State Machines*. Ph.D. thesis, Universität Ulm.

Schellhorn, G. and Ahrendt, W. 1998. The WAM case study: Verifying compiler correctness for Prolog with KIV. In *Automated Deduction—A Basis for Applications*, W. Bibel and P. H. Schmitt, Eds. Applied Logic Series, vol. 10. Springer, 165–194.

Schelog Homepage 2018. The Schelog Language. URL: https://ds26gte.github.io/schelog/index.html. [Accessed 11 March, 2022].

Schimpf, J. and Shen, K. 2012. Ecl$^{i}$ps$^{e}$ - from LP to CLP. *Theory and Practice of Logic Programming 12,* 1-2, 127–156.

Schrijvers, T., Costa, V. S., Wielemaker, J. and Demoen, B. 2008. Towards typed Prolog. In *Proceedings ICLP*, M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer, 693–697.

Schrijvers, T. and Demoen, B. 2008. Uniting the Prolog community. In *Proceedings ICLP*, M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer, 7–8.

Schrijvers, T., Demoen, B., Desouter, B. and Wielemaker, J. 2013. Delimited continuations for Prolog. *Theory and Practice of Logic Programming 13,* 4-5, 533–546.

Schrijvers, T. and Warren, D. S. 2004. Constraint handling rules and tabled execution. In *Proceedings ICLP*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer, 120–136.

Scryer Prolog 2021. Scryer Prolog. URL: https://github.com/mthom/scryer-prolog. [Accessed 11 March, 2022].

Seo, J., Guo, S. and Lam, M. S. 2015. SociaLite: An efficient graph query language based on datalog. *IEEE Transactions on Knowledge and Data Engineering 27,* 7, 1824–1837.

Serafini, L., Donadello, I. and d'Avila Garcez, A. S. 2017. Learning and reasoning in logic tensor networks: Theory and application to semantic image interpretation. In *Proceedings SAC*, A. Seffah, B. Penzenstadler, C. Alves and X. Peng, Eds. ACM, 125–130.

tuProlog Home 2021. tuProlog homepage. URL: http://tuprolog.unibo.it. [Accessed 11 March, 2022].

Shapiro, E., Warren, D. H. D., Fuchi, K., Kowalski, R. A., Furukawa, K., Ueda, K., Kahn, K. M., Chikayama, T. and Tick, E. 1993. The fifth generation project: Personal perspectives. *Communications of the ACM 36,* 3, 46–103.

Shapiro, E. Y. 1983. A Subset of Concurrent Prolog and Its Interpreter. Tech. Rep. TR-003, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan.

Shapiro, E. Y. 1987. *Concurrent PROLOG: Collected Papers*. MIT Press.

Shen, W., Doan, A., Naughton, J. F. and Ramakrishnan, R. 2007. Declarative information extraction using datalog with embedded extraction predicates. In *Proceedings VLDB*, C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C. Kanne, W. Klas and E. J. Neuhold, Eds. ACM, 1033–1044.

SICS. 2021. Quintus Prolog Homepage. URL: https://quintus.sics.se/. [Accessed 11 March, 2022].

SICStus Prolog Homepage 2021. SICStus Prolog — leading Prolog technology. URL: https://sicstus.sics.se. [Accessed 11 March, 2022].

Siek, J. G. and Taha, W. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Papers*. University of Chicago, 81–92. TR-2006-06.

Simon, L., Mallya, A., Bansal, A. and Gupta, G. 2006. Coinductive logic programming. In *Proceedings ICLP*, S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer, 330–345.

Simpkins, N. K. and Hancox, P. 1990. Chart parsing in Prolog. *New Generation Computing 8,* 2, 113–138.

Somogyi, Z., Henderson, F. and Conway, T. C. 1996. The execution algorithm of mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming 29,* 1-3, 17–64.

Sterling, L. and Shapiro, E. Y. 1994. *The Art of Prolog: Advanced Programming Techniques*. MIT press.

Stickel, M. E. 1984. A Prolog technology theorem prover. In *Proceedings SLP*. IEEE, 212–220.

Stickel, M. E. 1992. A Prolog technology theorem prover: a new exposition and implementation in Prolog. *Theoretical Computer Science 104,* 1, 109–128.

Stulova, N., Morales, J. F. and Hermenegildo, M. V. 2018. Exploiting term hiding to reduce run-time checking overhead. In *Proceedings PADL*, F. Calimeri, K. Hamlen and N. Leone, Eds. LNCS, vol. 10702. Springer, 99–115.

Sun, H., Arnold, A. O., Bedrax-Weiss, T., Pereira, F. and Cohen, W. W. 2020. Faithful embeddings for knowledge base queries. In *Proceedings NeurIPS*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan and H. Lin, Eds. NeurIPS.cc.

SWI-Prolog 2021. SWI-Prolog — Manual. URL: https://www.swi-prolog.org/pldoc/man?section=implhistory. [Accessed 11 March, 2022].

SWI Prolog Homepage 2021. SWI Prolog — robust, mature, free. Prolog for the real world. URL: https://www.swi-prolog.org. [Accessed 11 March, 2022].

Swift, T. 2009. Prolog-commons working group report. *Association for Logic Programming Newsletter 22,* 1.

Swift, T. and Warren, D. S. 2012. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming 12,* 1-2, 157–187.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

Symonds, A. J. 1986. Introduction to IBM's knowledge-systems products. *IBM Systems Journal 25,* 2, 134–146.

Syrjänen, T. and Niemelä, I. 2001. The Smodels system. In *Proceedings LPNMR*, T. Eiter, W. Faber and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 2173. Springer, 434–438.

Szabó, P. and Szeredi, P. 2006. Improving the ISO Prolog standard by analyzing compliance test results. In *Proceedings ICLP*, S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer, 257–269.

Szeredi, P. 2004. The early days of Prolog in hungary. *Association for Logic Programming Newsletter 17,* 4.

Tamaki, H. and Sato, T. 1986. OLD resolution with tabulation. In *Proceedings ICLP*, E. Shapiro, Ed. Lecture Notes in Computer Science, vol. 225. Springer, 84–98.

Tarau, P. 1992. BinProlog: A continuation passing style Prolog engine. In *Proceedings PLILP*, M. Bruynooghe and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 631. Springer, 479–480.

Tarau, P. 2012. The BinProlog experience: Architecture and implementation choices for continuation passing Prolog and first-class logic engines. *Theory and Practice of Logic Programming 12,* 1-2, 97–126.

Tarau, P. 2021. Natlog: A lightweight logic programming language with a neuro-symbolic touch. In *Proceedings ICLP (Technical Communications)*, A. Formisano, Y. A. Liu, B. Bogaerts, A. Brik, V. Dahl, C. Dodaro, P. Fodor, G. L. Pozzato, J. Vennekens and N.-F. Zhou, Eds. EPTCS, vol. 345, 141–154.

$\tau$Prolog Homepage 2021. $\tau$Prolog — an open source Prolog interpreter in javascript. URL: http://tau-prolog.org. [Accessed 11 March, 2022].

comp.lang.Prolog FAQ 2021. URL: http://www.logic.at/prolog/faq/. [Accessed 11 March, 2022].

Thom, J. and Zobel, J. 1987. *NU-Prolog Reference Manual.* Department of Computer Science, University of Melbourne.

Tick, E. 1984. Sequential Prolog machine: Image and host architectures. In *Proceedings MICRO*, M. Carter and R. A. Mueller, Eds. ACM/IEEE, 204–216.

Tolan, S., Miron, M., Gómez, E. and Castillo, C. 2019. Why machine learning may lead to unfairness: Evidence from risk assessment for Juvenile Justice in Catalonia. In *Proceedings ICAIL*. ACM, 83–92.

Triska, M. 2012. The finite domain constraint solver of SWI-Prolog. In *Proceedings FLOPS*, T. Schrijvers and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 7294. Springer, 307–316.

Triska, M. 2021. The Power of Prolog. URL: https://www.metalevel.at/prolog. [Accessed 11 March, 2022].

Triska, M., Neumerkel, U. and Wielemaker, J. 2009. Better Termination for Prolog with Constraints. CoRR abs/0903.2168.

Ueda, K. 1985. Guarded horn clauses. In *Proceedings LP*, E. Wada, Ed. Lecture Notes in Computer Science, vol. 221. Springer, 168–179.

Ueda, K. and Chikayama, T. 1990. Design of the kernel language for the parallel inference machine. *The Computer Journal 33,* 6, 494–500.

van Emden, M. 2006. The early days of logic programming: A personal perspective. *The Association of Logic Programming Newsletter 19,* 3.

van Emden, M. and Kowalski, R. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM 23*, 733–742.

van Emden, M. and Lloyd, J. 1984. A logical reconstruction of Prolog II. *The Journal of Logic Programming 1,* 2, 143–149.

Van Gelder, A., Ross, K. A. and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM 38,* 3, 619–649.

Van Roy, P. 1994. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming 19/20*, 385–441.

Van Roy, P. and Despain, A. 1992. High-performance logic programming with the aquarius Prolog compiler. *IEEE Computer Magazine 25,* 1, 54–68.

Vaz, D., Santos Costa, V. and Ferreira, M. 2009. User defined indexing. In *Proceedings ICLP*, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer, 372–386.

Vennekens, J., Verbaeten, S. and Bruynooghe, M. 2004. Logic programs with annotated disjunctions. In *Proceedings ICLP*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer, 431–445.

Wallace, M. and Schimpf, J. 1999. ECLiPSe: Declarative specification and scaleable implementation. In *Proceedings PADL*, G. Gupta, Ed. Lecture Notes in Computer Science, vol. 1551. Springer, 365–366.

Wallace, M. and Veron, A. 1993. Two problems — two solutions: One system ECL$^i$PS$^e$. In *Advanced Software Technologies for Scheduling.* IET, 3/1–3/3.

Wang, Y. and Nadathur, G. 2016. A Higher-Order Abstract Syntax Approach to Verified Transformations on Functional Programs.

Warren, D. H. 1974. Warplan: A System for Generating Plans. Tech. Rep. 76, Dept. of Computational Logic, University of Edinburgh School of Artificial Intelligence.

Warren, D. H. 1975. *PROLOG to DEC 10 Machine Code Compiler.*

Warren, D. H. 1977. *Applied Logic—Its Use and Implementation as Programming Tool*. Ph.D. thesis, University of Edinburgh. Also available as SRI Technical Note 290.

Warren, D. H. 1980. An improved Prolog implementation which optimizes tail recursion. Tech. Rep. 141, Dept. of Computational Logic, University of Edinburgh School of Artificial Intelligence.

Warren, D. H. 1983. An Abstract Prolog Instruction Set. Tech. Rep. Technical Note 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International.

Warren, D. H. 1990. The extended Andorra model with implicit control. Presented at ICLP'90 Workshop on Parallel Logic Programming.

Warren, D. H., Kornfeld, W. and Byrd, L. 1984. Product Specifications. URL: http://www.softwarepreservation.org/projects/prolog/quintus/doc/Warren_et_al-Product_Specifications-1984_07.pdf. [Accessed 11 March, 2022].

Warren, D. H. and Pereira, F. C. 1982. An efficient easily adaptable system for interpreting natural language queries. *Computational Linguistics 8,* 3-4, 110–122.

Warren, D. H., Pereira, L. M. and Pereira, F. C. 1977. Prolog—The language and its implementation compared with lisp. *ACM SIGPLAN Notices 12,* 8, 109–115.

Warren, D. S. 1998. Programming with tabling in XSB. In *Proceedings PROCOMET*, D. Gries and W. P. de Roever, Eds. IFIPAICT, vol. 125. Chapman & Hall, 5–6.

Warren, R., Hermenegildo, M. and Debray, S. K. 1988. On the practicality of global flow analysis of logic programs. In *Proceedings ICLP/SLP*, R. Kowalski and K. A. Bowen, Eds. MIT Press, 684–699.

Wenger, E., McDermott, R. A. and Snyder, W. 2002. *Cultivating Communities of Practice: A Guide to Managing Knowledge*. Harvard Business Press.

Whaley, J., Avots, D., Carbin, M. and Lam, M. S. 2005. Using datalog with binary decision diagrams for program analysis. In *Proceedings APLAS*, K. Yi, Ed. Lecture Notes in Computer Science, vol. 3780. Springer, 97–118.

Wielemaker, J. and Costa, V. S. 2011. On the portability of Prolog applications. In *Proceedings PADL*, R. Rocha and J. Launchbury, Eds. Lecture Notes in Computer Science, vol. 1551. Springer, 69–83.

https://doi.org/10.1017/S1471068422000102 Published online by Cambridge University Press

Wielemaker, J., Hildebrand, M., van Ossenbruggen, J. and Schreiber, G. 2008. Thesaurus-based search in large heterogeneous collections. In *Proceedings ISWC*, A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin and K. Thirunarayan, Eds. Lecture Notes in Computer Science, vol. 5318. Springer, 695–708.

Wielemaker, J., Huang, Z. and van der Meij, L. 2008. SWI-Prolog and the web. *Theory and Practice of Logic Programming 8,* 3, 363–392.

Wielemaker, J., Riguzzi, F., Kowalski, R. A., Lager, T., Sadri, F. and Calejo, M. 2019. Using SWISH to realize interactive web-based tutorials for logic-based languages. *Theory and Practice of Logic Programming 19,* 2, 229–261.

Wielemaker, J., Schrijvers, T., Triska, M. and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 67–96.

XSB Prolog Homepage 2021. Welcome to the home of XSB! URL: http://xsb.sourceforge.net. [Accessed 11 March, 2022].

Zhou, N.-F. 1996. Parameter passing and control stack management in Prolog implementation revisited. *Transactions on Programming Languages and Systems 18,* 6, 752–779.

Zhou, N.-F. 2006. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming 6,* 5, 483–507.

Zhou, N.-F. 2012. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 189–218.

Zhou, N.-F. 2021. Modeling and solving graph synthesis problems using SAT-encoded reachability constraints in Picat. In *Proceedings ICLP (Technical Communications)*, A. Formisano, Y. A. Liu, B. Bogaerts, A. Brik, V. Dahl, C. Dodaro, P. Fodor, G. L. Pozzato, J. Vennekens and N.-F. Zhou, Eds. EPTCS, vol. 345, 165–178.

Zhou, N.-F. and Fruhman, J. 2021. A User's Guide to Picat. URL: http://retina.inf.ufsc.br/picat_guide/. [Accessed 11 March, 2022].

Zhou, N.-F., Kjellerstrand, H. and Fruhman, J. 2015. *Constraint Solving and Planning with Picat.* SpringerBriefs in Intelligent Systems. Springer.