Prudens: An Argumentation-Based Language for Cognitive Assistants

Article · December 2022

CITATIONS
0

READS
78

2 authors:

Vasileios Theodoros Markos
Open University of Cyprus
8 PUBLICATIONS **10** CITATIONS
SEE PROFILE

Loizos Michael
126 PUBLICATIONS **903** CITATIONS
SEE PROFILE

Prudens: An Argumentation-Based Language for Cognitive Assistants

Vassilis Markos[1] and Loizos Michael[1,2]

[1] Open University of Cyprus, Nicosia, Cyprus
vasileios.markos@st.ouc.ac.cy, loizos@ouc.ac.cy
[2] CYENS Center of Excellence, Nicosia, Cyprus

Abstract. In this short system paper, we present our implementation of a prioritized rule-based language for representing actionable policies, in the context of developing cognitive assistants. The language is associated with a provably efficient deduction process, and owing it to its interpretation under an argumentative semantics it can naturally offer ante-hoc explanations on its drawn inferences. Relatedly, the language is associated with a knowledge acquisition process based on the paradigm of machine coaching, guaranteeing the probable approximate correctness of the acquired knowledge against a target policy. The paper focuses on demonstrating the implemented features of the representation language and its exposed APIs and libraries, and discusses some of its more advanced features that allow the calling of procedural code, and the computation of in-line operations when evaluating rules.

Keywords: Logical programming · Non-monotonic reasoning · Cognitive assistants.

1 Introduction

The widespread adoption of Artificial Intelligence (AI) in everyday applications has led to an upsurging interest in the design of *cognitive assistants*, i.e., of systems "augmenting human intelligence", as put by Engelbart [2]. Naturally, such systems are required to be cognitively compatible with humans, in an effort to facilitate human-machine interaction, which makes, explainability and understandability natural prerequisites as well [3,12]. Considering the above desiderata, argumentation-based designs seem as a proper choice that can at the same time accommodate cognitive compatibility, while providing substantial potential for interpretability and explainability for such systems [10,6].

Having in mind the above, with this work we present a declarative programming language, Prudens, aiming to facilitate the design of cognitive assistants. Prudens is an argumentation-based language that can fully support efficient deduction as described in [11]. Moreover, Prudens is also compatible with machine coaching, a provably efficient human-in-the-loop machine learning methodology, under the Probably Approximately Correct (PAC) semantics [14,11]. Given its reliance on arguments, Prudens can also support the design of assistants that

can explain their decisions, by providing the internal arguments that have led the system to draw a conclusion as an explanation.

The rest of this paper is structured as follows: (i) in Section 2 we present the basic syntax of Prudens; (ii) in Section 3 we present extended features of the language; (iii) in Section 4 we discuss currently ongoing and future works related to Prudens and; (iv) in Section 5 we conclude. All online resources regarding Prudens may be found at http://cognition.ouc.ac.cy/prudens/.

2 Basic Syntax and Semantics

First we discuss the basic syntax of Prudens, by describing the language's basic rule syntax and their prioritization as well as the underlying reasoning process.

2.1 Rule Syntax

The core ("vanilla") version of Prudens basically implements the knowledge representation language described in [11]. It provides ***constants***, which correspond to entities of the universe of discourse, as well as ***variables***, which serve as placeholders for constants. Moreover, first-order ***predicates*** (with variables and/or constants as arguments) as well as propositional ones are provided, encoding relations and conditions about the universe of discourse, respectively. ***Literals*** are either predicates themselves or negated, with negation being understood as classical negation within the scope of Prudens. Two literals corresponding to the same predicate but with opposite signs are ***conflicting***. The language, building on the above, allows for if-then ***rules***, which connect a set of premises, the rules' ***body***, with a single literal, the rules' ***head***. As with literals, two rules with conflicting heads are ***conflicting*** as well. Lastly, a list of rules alongside a priority relation defined over all pairs of conflicting rules comprise a ***policy***. By default, priorities in a policy are determined by the rules' order of appearance. That is, the later a rule appears in the policy, the higher its priority is over conflicting ones. Also, a ***context*** is a set of *pairwise non-conflicting* literals, corresponding to a set of facts being known at the beginning of the reasoning process — see Section 2.2 for more. The language's "vanilla" constructs are shown in Table 1.

2.2 Reasoning

Reasoning in Prudens is performed utilizing prioritized forward-chaining semantics, by exhaustively inferring all possible facts through all policy rules, respecting priorities between them, given each time a set of currently known facts — initially, the *context*. For instance, consider the example policy shown in Figure 1. In this case, a context containing `isMonday` and `bobCalls` would lead us infer `atWork` and `-answerCall`, as follows: (i) At first, knowing `isMonday` and `bobCalls`, `R1` and `R2` fire, allowing us to infer `atWork` and `answerCall`; (ii) Knowing `atWork`, rule `R3` fires, leading to `-answerCall`, which conflicts with `answerCall`, which is resolved by preferring `-answerCall` over `answerCall`,

Item	Syntax	Example
constant	Any non empty string containing alphanumeric characters or underscores (`a-zA-Z0-9_`), starting with a lowercase latter (`a-z`).	`alice`, `office_2`
variable	Any non empty string containing alphanumeric characters or underscores (`a-zA-Z0-9_`), starting with an uppercase latter (`A-Z`).	`User`, `Place_23`
predicate	Any non empty string starting with a lowercase latter (`a-z`) and possibly followed by an arbitrary number of alphanumeric characters or underscores (`a-zA-Z0-9_`). In case of first-order predicates, a comma separated list of variables and/or constants should follow, enclosed in parentheses.	`atHome`, `at(X, bob)`
literal	A predicate, possibly preceded by a dash (`-`), indicating negation	`-atWork`, `friends(X, Y)`
rule	A string starting with a non empty sequence of alphanumeric characters or underscores (`a-zA-Z0-9_`), followed by `::`, followed by a comma-separated list of literals (body), followed by the keyword `implies`, followed by a single literal (head).	`R1 :: a, b implies z` `r2 :: f(X, 3) implies g(X)`

policy	A semicolon-separated list of rules, preceded by the `@Knowledge` keyword.	`@Knowledge`
		`R1 :: a implies z;`
		`R2 :: a, b implies -z;`

context	A semicolon-separated list of pairwise non-conflicting literals.	`a; b; -at(work);`

Table 1. The syntax of Prudens.

```
@Knowledge
R1 :: isMonday implies atWork;
R2 :: bobCalls implies answerCall;
R3 :: atWork implies -answerCall;
```

Fig. 1. A simple propositional policy regarding phone call management.

since the former was inferred by `R3`, which is of higher priority than `R2`. (iii) Having inferred `atWork` and `-answerCall`, nothing more may be inferred, so the process terminates. For more on Prudens's reasoning algorithm, see [11].

2.3 Custom Priorities and Dilemmas

Apart from order-induced implicit rule prioritization, one may define custom rule priorities in essentially two ways: programmatically, by providing a priority function as an optional argument to the Prudens's core reasoning function

```
@Knowledge
R1 :: bobCalls implies answerCall | 1;
R2 :: atWork implies -answerCall | 0;

@Knowledge
R1 :: bobCalls implies answerCall | 1;
R2 :: atWork implies -answerCall | 1;
```

Fig. 2. Two policies with custom priorities.

and; explicitly, by providing priorities within each rule's declaration. Regarding explicit priority manipulation, the language's "vanilla" rule syntax is extended to allow for priorities to be declared through integers following a rule's head, separated by a `|`, as in the policies shown in Figure 2. There, numbers indicate priority, with negative values being allowed as well. So, given a context containing `bobCalls; atWork;`, the top policy in Figure 2 would yield `answerCall`, since `R1` is preferred over `R2`.

Naturally, allowing for custom priorities, there might be cases where two rules are incomparable, either because no priority between them has been explicitly determined, or because they are of the same priority, as in Figure 2 (bottom). Such cases are called ***dilemmas***. Since there is no clear way to resolve a dilemma, the reasoning engine abstains from making a decision, ignoring both rules and proceeding with the reasoning process. Any dilemmas encountered throughout a reasoning cycle are noted and returned separately from the rest inferences.

3 Extended Syntax and Semantics

Apart from the language's core features presented in Section 2, there are several additional features that are offered by Prudens, as discussed below.

3.1 The Unification Predicate

Prudens comes with a built-in multipurpose binary predicate, denoted by `?=(·,·)`. The unification predicate holds true provided that its two arguments are unifiable. So, for instance, given a rule like: `R1 :: f(X), ?=(X, Y) implies g(Y);` and a context containing `f(2)`, we would get `g(2)` as an inference. In general, (function-free) unification is conceptualized as with most logical programming interfaces; so two constants unify if they are equal, a variable unifies with any constant and two (unassigned) variables always unify. We shall note at this point that the unification predicate might not be used as a head literal in any rule.

The very same predicate also allows for numerical operations within its arguments, provided that they do not invoke any variables that remain unassigned *once all other body predicates are grounded*. So, a rule like `R1 :: f(X), ?=(Y, X+3) implies g(Y);` with a context containing `f(2)` would infer `g(5)` but `R2 :: f(X), ?=(Y-3, X) implies g(Y);` with the same context would not, since

```
@Knowledge
R1 :: calls(X), friend(X) implies answer(X);
R2 :: calls(X), time(H, M), ?lessThan(H, 17) implies -answer(X);

@Procedures
function lessThan(a, b) {
    return parseFloat(a) < parseFloat(b);
}
```

Fig. 3. A policy indicating that any calls before 17:00 should be rejected, with the help of a procedural predicate (`lessThan`), evaluating numerical comparisons.

there is no value assigned to `Y` by the time the predicate is evaluated. Any mathematical expressions within `?=(·,·)` should adhere to ECMAScript 6 syntax.

3.2 On-the-fly Math Operations

Prudens also allows for math operations to be executed within any predicate, given the restrictions mentioned above, about unassigned variables within the unification predicate. Also, similarly to the unification predicate, numerical operations may not be used in head literals. For instance, the following rule: `R1 :: f(X, 2*X) implies double;` given a context containing `f(2,4)` infers `double`. An equivalent rule, avoiding within-predicate operations, would be `R2 :: f(X, Y), ?=(Y, 2*X) implies double;`, which, however, introduces an additional variable, `Y`, and leads to slightly slower processing time. Thus, whenever possible, within-predicate math operations should be preferred against `?=(·,·)`.

3.3 Procedural Predicates

Prudens allows for users to determine their own procedural predicates through procedural code. Namely, one may provide general Boolean functions as a predicate's "definition". For instance, expanding our running call management example, let us consider the following scenario: we would like to answer all friends calls, on condition that it is past 17:00. Assuming that `time(H,M)` represents the time of the call, a policy that captures this functionality is shown in Figure 3. There, the procedural binary predicate `?lessThan`, which compares its two arguments and holds true whenever its first argument is less than its second, facilitates an efficient execution of numerical comparisons. So, a context containing `calls(alice)`, `friend(alice)` and `time(16,43)` would result to `-answer(alice)` using the above policy. However, substituting `time(16,43)` with `time(18,12)` would result to `answer(alice)`, as expected.

When working with procedural predicates there are several things one should keep in mind: (i) the `@Knowledge` keyword should *always* come before the `@Procedures`

```
@Knowledge
R1 :: f(A), g(X) implies h(X,A);
```

```
@Knowledge
R1 :: f(A), g(X) implies h(X,A);
R2 :: h(X,b) implies z(X);
```

```
@Knowledge
R1 :: bobCalls implies answer;
R2 :: atWork implies reject;

C1 :: answer # reject;
```

Fig. 4. Two first-order policies (left) and one with a compatibility constraint (right).

keyword; (ii) predicate names, when referenced in a rule's body, should be preceded by a `?`; (iii) procedural predicates are not allowed as rule heads; (iv) predicate declarations should adhere to ECMAScript 6 standards; (v) no function calls are allowed within a procedural predicate other than built-in functions of JavaScript; (vi) every argument of a procedural predicate is by default considered to be a string, so in case they are supposed to be treated as integers or floats, the built-in JavaScript `parseInt` and `parseFloat` functions should be used, respectively.

3.4 Partially Grounded Contexts

Literals in a context, in contrast to what is demanded by the language's "vanilla" version, may be partially or even totally ungrounded. That is, a context may well contain literals like `f(Y)`, even if `Y` is a free variable. In any such case, variables propagate throughout inferences, unifying with other variables whenever it makes sense. For instance, consider the policies shown left in Figure 4. Using the top left one with a context containing `f(Y); g(3);` one infers `h(3,Y)`. Using a context containing `f(Y)` with the policy shown bottom left in Figure 4 this time, we infer `z(3)`. Note here that fresh variables, not used elsewhere in the underlying policy, should be preferred in contexts.

3.5 Extended Conflict Semantics

In the vanilla version of the language, two literals are considered conflicting in case they stem from the same predicate but have opposite signs. Prudens, however, also allows for rules that determine broader conflicts between arbitrary predicates. Such rules are called ***compatibility constraints*** and adhere to the following syntax:

```
ruleName :: pred1 # pred2;
```

So, for instance, using the policy shown in Figure 4 (right) and a context containing `bobCalls; atWork;`, one infers `reject`, since `C1` declares `reject` and `answer` as conflicting literals. Note at this point that there are no assumed priorities between compatibility constraints, in contrast to what is the case with the rest rules in a policy.

4 Ongoing and Future Work

Below we briefly present some of the most prominent works in progress invoking Prudens as well as discuss possible future directions.

4.1 Deduction, Induction & Abduction

So far, Prudens has been utilized as the underlying deductive engine for machine coaching [11], an interactive human-in-the-loop methodology that allows a human coach to train a machine on a certain task by providing advice to it, echoing ideas from McCarthy's advice taking machines [9]. Furthermore, Prudens's semantics allow for abductive reasoning as well. Hence, a candidate domain of application is, among others, Neural-Symbolic Integration [4], where machine coaching could be used as an induction mechanism to train the symbolic module and Prudens could serve as the underlying knowledge representation language for both deduction as well as abduction, extending ideas found in [13].

4.2 Natural Language Interfaces

While cognitively easier than imperative programming, declarative programming still requires from the programmer to be accustomed to some sort of coding formalism. Consequently, interfaces that allow users to program using natural language provide a user-friendly and cognitively simple alternative to sheer coding. We are currently working towards two independent Natural Language Interfaces (NLIs). The first one relies on building a NL-to-Prudens translator utilizing machine coaching [5] to learn the underlying translation grammar. Here, Prudens itself is used at the meta-level as the interaction language between the human coach and the machine. The second one is the design of a Controlled NLI for Prudens, in the spirit of other works in the field of Logical Programming, like Logical English [7]. Apart from the aforementioned ongoing projects, Large Language Models are also under consideration as appropriate NL-to-Prudens translators, with the potential of also capturing Prudens's reasoning semantics [1].

4.3 Applications

At the time of writing this, there are two applications being developed utilizing Prudens in the background. The first one, a mobile call assistant, intends to use machine coaching so as to elicit a user's preferences regarding their calls and notifications management, with Prudens serving as the knowledge representation language in the background. Inspired by previous work on chess coaching [8], where again Prudens had been used in the background for deductive and inductive purposes, our second ongoing project is related to another strategy game: Minesweeper. There, users are asked to explain to an agent how to play Minesweeper successfully, again utilizing machine coaching as the learning methodology, with Prudens facilitating human-machine interaction.

5 Conclusions

We have presented Prudens, an argumentation-based language for the design of cognitive assistants. We have discussed its syntax as well as any additional features it provides. Moreover, we have presented several currently developed and future applications as well as extensions of Prudens, aiming to further facilitate the design of cognitive assistants by non-experts.

Apart from the aforementioned ongoing projects, we are also working on the direction of generating comprehensive visualizations about Prudens and its internal processes. Namely, effort is being put on visualizing the reasoning process of Prudens in a step-by-step manner, so as to facilitate user understanding and, consequently, build more trust with the end-user. At the same time, we are also working on knowledge-graph based explanation representations, again, as an attempt to make Prudens more accessible to a less expert audience. Ultimately, our goal is to build an easy-to-use ecosystem for Machine Coaching, allowing for an efficient and thorough *in situ* assessment of Machine Coaching.

Acknowledgements This work was supported by funding from the European Regional Development Fund and the Government of the Republic of Cyprus through the Research and Innovation Foundation under grant agreement no. INTEGRATED/0918/0032, from the EU's Horizon 2020 Research and Innovation Programme under grant agreements no. 739578 and no. 823783, and from the Government of the Republic of Cyprus through the Deputy Ministry of Research, Innovation, and Digital Policy.

References

1. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating Large Language Models Trained on Code (2021). https://doi.org/10.48550/ARXIV.2107.03374, https://arxiv.org/abs/2107.03374
2. Engelbart, D.C.: Augmenting Human Intellect: A Conceptual Framework. In: SRI Summary Report AFOSR-3223 (1962)
3. de Graaf, M., Malle, B.F.: How People Explain Action (and Autonomous Intelligent Systems Should Too). In: AAAI Fall Symposia (2017)
4. Hammer, B., Hitzler, P.: Perspectives of Neural-Symbolic Integration, vol. 77 (01 2007). https://doi.org/10.1007/978-3-540-73954-8
5. Ioannou, C., Michael, L.: Knowledge-Based Translation of Natural Language into Symbolic Form. In: Proceedings of the 7th Linguistic and Cognitive Approaches

To Dialog Agents Workshop - LaCATODA 2021. pp. 24–32. Montreal, Canada (2021), http://ceur-ws.org/Vol-2935/#paper3
6. Kakas, A., Michael, L.: Cognitive Systems: Argument and Cognition. IEEE Intelligent Informatics Bulletin **17**, 14–20 (12 2016)
7. Kowalski, R.: English as a logic programming language. New Generation Computing **8**(2), 91–93 (Aug 1990). https://doi.org/10.1007/BF03037468, https://doi.org/10.1007/BF03037468
8. Markos, V.: Application of the Machine Coaching Paradigm on Chess Coaching. Master's thesis, School of Pure & Applied Sciences, Open University of Cyprus (2020)
9. McCarthy, J.: Programs with Common Sense. In: Proceedings of Teddington Conference on the Mechanization of Thought Processes (1958)
10. Mercier, H., Sperber, D.: Why Do Humans Reason? Arguments for an Argumentative Theory. Behavioral and Brain Sciences **34**(2), 57–74 (2011). https://doi.org/10.1017/S0140525X10000968
11. Michael, L.: Machine Coaching. In: IJCAI 2019 Workshop on Explainable Artificial Intelligence. pp. 80–86. Macau, China (2019), https://www.researchgate.net/publication/334989337_Machine_Coaching
12. Miller, T.: Explanation in Artificial Intelligence: Insights from the Social Sciences. Artificial Intelligence **267**, 1–38 (2019). https://doi.org/https://doi.org/10.1016/j.artint.2018.07.007, https://www.sciencedirect.com/science/article/pii/S0004370218305988
13. Tsamoura, E., Hospedales, T., Michael, L.: Neural-Symbolic Integration: A Compositional Perspective. Proceedings of the AAAI Conference on Artificial Intelligence **35**(6), 5051–5060 (May 2021), https://ojs.aaai.org/index.php/AAAI/article/view/16639
14. Valiant, L.G.: A Theory of the Learnable. In: STOC '84: Symposium on Theory of Computing. pp. 1134–1142 (1984)