Declarative rules and rule-based systems

Maxim Tarasov
maxeeem@gmail.com

Temple AGI Team

Abstract. Starting with the premise that rule-based systems still have their place in AI toolkit, we explore different ways of implementing such systems. We find that some of the criticisms towards rule-based systems, namely their large number of rules and high maintenance cost can be addressed by using declarative style programming and a shift to a higher level of abstraction from rules to meta-rules, operating on the structure of data rather than its contents. We use OpenNARS 4 as example implementation to demonstrate advantages of this approach but the results are generally applicable to other rule-based systems.

Keywords: Non-Axiomatic Logic · AGI · Relational Programming

1 Introduction

Despite widespread beliefs in academia [3] that traditional rule-based systems [4] have been surpassed by their neural counterparts and are no longer a viable approach to building intelligent computer systems, they still have their place in medical [2] and legal [1] applications as well as other domains. It is also an intriguing area of research whether the two approaches are not mutually exclusive and could potentially be combined in a hybrid system [1, 5].

So if we accept the premise that rule-based systems still have their place in AI toolkit, it is worth exploring different ways of implementing such systems. Typically, rules are thought of as *if-then* pairs i.e. `if it's raining, take an umbrella`. When implemented in a computer system, this usually takes a form similar to `if (state.is_raining): umbrella.take()`. Some of the major criticisms of rule-based systems are that there are many such rules required for any useful application and maintaining such a large rule-set becomes a prohibitive task as the system grows. We will now briefly discuss an alternative way of representing rules using NARS as an example but the general approach can be adapted for other rule-based systems.

In computer programming, it is said that the code is imperative if it specifies a sequence of steps the computer should take to accomplish a task. So in the example above this would be writing out explicit steps like "check current weather", "compare to rule condition", "execute action" and so on. Declarative programming specifies "what" needs to be done instead of "how", which is left to the computer to figure out. All we want to do is tell the computer in some structured way that "if it's raining, take an umbrella".

2 NARS is Not A Rule-based System

NARS is a reasoning system developed by Pei Wang [6] and it uses a language called Narsese to express knowledge as well as rules. Because the system has rules, it is tempting to call it a "rule-based system", however we will attempt to show that Narsese rules are of a different kind, describing more the structure of the data and *rules of operation* on that structure, than the contents of the data itself. NARS uses what is called a Non-Axiomatic Logic (NAL) and an example "rule" in Narsese may look like "`{A =/> B, A}` $\vdash$ `B`" which means *"if A implies B and A is observed, then B"*. Note that we are not talking about any particular A or B here, but rather abstract terms, describing what should happen to data if it matches a certain pattern without explicitly mentioning any specific data items. In this way, the inference rules of NARS which are fixed for any given system, can be described as *meta*-rules, while the *empirical* rules, encoded as knowledge provided to the system, i.e. `is_raining =/> take(umbrella)`, can be dynamically updated at run time — an ability most expert systems lack.

We will now turn our attention back to imperative vs. declarative approaches of representing rules in computer systems. The meta-rules of NARS have been typically implemented [10] in an imperative way which presents maintainability issues. Additionally, while the fixed rule set makes the problem much smaller than in traditional rule-based systems, the issue of customization remains where creating a custom system with some subset of rules is not as straightforward. In this paper we propose using miniKanren to address these limitations and improve both maintainability and customization of the system by making it more declarative. While this paper deals primarily with NARS, the approach described here is general and can be applied to other rule-based systems.

3 Inference Engine

At a high level of abstraction there is a component of the system, typically called the Inference Engine, whose primary function is to take inputs and apply rules to them in a data-driven manner. In the new design of OpenNARS 4, inference rules are stored in a text file for readability and allow for a convenient way to customize the system and create just the right combination for a particular use case. The choice of miniKanren for inference also means that other implementations could adopt this design with minimal effort, regardless of their programming language. Viewed this way, OpenNARS 4 represents a reference implementation optimizing for correctness rather than speed. See code [11] and video [9] for details.

Sample inference rules from `nal-rules.yml`:

```
{<M --> P>. <S --> M>} <S --> P>
{(C ^ M) ==> P. M ==> S} (C ^ S) ==> P
<(T1 * T2) --> R> <=> <T1 --> (/, R, _, T2)>
```

Similar to the earlier examples, these inference rules feature abstract terms like `M`, `P` and `T1`, as well as different copulas and connectors like `-->`, `^`, `*` which

are covered in detail in [6]. To process these inference rules, we use miniKanren, "a family of Domain Specific Languages for logic programming [that] has been implemented in a growing number of host languages" [7]. It has certain similarities with Prolog and in OpenNARS 4 we use a Python implementation of miniKanren [8] with its basic usage illustrated below:

```
# kanren uses unification to match forms within expression trees.
# This code asks for values of x such that (1, 2) == (1, x):
>>> run(1, x, eq((1, 2), (1, x)))
(2,)
```

In a more concrete case, when used for inference, this is the line of code that does most of the heavy lifting. Here `c` is the conclusion, `t1, t2` are the input tasks and `p1, p2` are the two premises from the rule:

```
>>> run(1, c, eq((t1, t2), (p1, p2)), *constraints)
```

With miniKanren doing most of the work, we just need a way to convert between Narsese and logic representations. OpenNARS 4 uses an off-the-shelf parser to convert text to Narsese, and two additional custom functions to convert between Narsese and logic representations. During the working cycle, incoming tasks are checked against all of the inference rules and the results produced by miniKanren are converted back to Narsese statements (see Figure 1).

4 Results and Discussion

Utilizing off-the-shelf components for parsing and inference allowed the team to focus on the specifics of NARS. At present, the Inference Engine is able to execute *∼300-400 inference cycles per second* which is about an order of magnitude slower than existing implementation but still sufficient. For specific applications, it should be possible to treat this implementation as a template and utilize other programming languages like Rust or C to achieve even higher performance while maintaining design parity.

In terms of maintainability and customization, this declarative rule system is much easier to understand and debug but it has its drawbacks. Chief among them is miniKanren's limited support for parallel processing, and while there is work happening in this area [12], it represents a design trade off between readability and maximum performance.

Ultimately, particular implementation choices always come down to weighing different alternatives. The aim of this paper is to demonstrate a way of utilizing miniKanren to create an inference engine for rule processing. It offers some clear benefits but is not free of compromises. We will continue to explore its potential and invite others to contribute and improve on our initial effort.

Acknowledgements

The author would like to thank Pei Wang for the valuable discussions and his comments and suggestions on the initial draft.

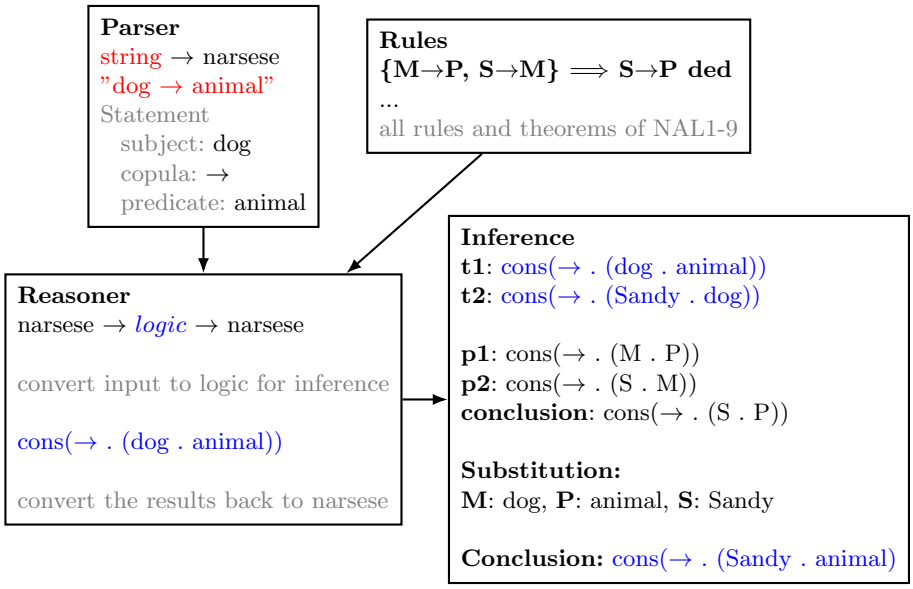

Fig. 1. *Parser* converts Narsese text such as "dog → animal" to Python classes. *Rules* are stored in a text file as "{M→P. S→M} ⊢ S→P .ded". *Reasoner* converts inputs (t1, t2) and rules (p1, p2) to logic form i.e. cons(→.(dog.animal) and uses unification to derive a conclusion if the inputs match the rule.

References

1. Billi et. al. (2023). Large Language Models and Explainable Law: a Hybrid Methodology. arXiv:2311.11811
2. Braja et. al. (2024) Extracting Social Support and Social Isolation Information from Clinical Psychiatry Notes: Comparing a Rule-based NLP System and a Large Language Model. arXiv:2403.17199
3. Chiticariu et. al. (2013) Rule-Based Information Extraction is Dead! Long Live Rule-Based Information Extraction Systems!. In Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, pp. 827-830
4. Grosan, C., Abraham, A. (2011). Intelligent Systems: A Modern Approach. Germany: Springer Berlin Heidelberg. pp. 149-179
5. Vacareanu et. al. (2024) Best of Both Worlds: A Pliable and Generalizable Neuro-Symbolic Approach for Relation Classification. arXiv:2403.03305
6. Wang, P.: Non-Axiomatic Logic: A Model Of Intelligent Reasoning. World Scientific Publishing Co. Pte. Ltd. (2013)
7. miniKanren language, http://minikanren.org (accessed 06/01/24)
8. miniKanren Python, https://github.com/pythological/kanren (accessed 06/01/24)
9. miniKanren inference engine, https://youtu.be/y3pUwgOso9A (accessed 06/01/24)
10. OpenNARS, https://github.com/opennars (accessed 06/01/24)
11. PyNARS, https://github.com/bowen-xu/PyNARS (accessed 06/01/24)
12. https://www.mail-archive.com/minikanren@googlegroups.com/msg00402.html (accessed 06/01/24)