Time for Fun3: Pre-compiling Rules into a High-Level Imperative Language for Top-Down Reasoning

William Van Woensel[1,*], Dörthe Arndt[2,3] and Jos De Roo[4]

[1]Telfer School of Management, University of Ottawa, Ottawa, Canada
[2]Computational Logic Group, Technische Universität Dresden, Dresden, Germany
[3]ScaDS.AI, Dresden/Leipzig, Germany
[4]KNoWS, IDLab, Ghent University, Ghent, Belgium

Abstract

Semantic Web (SW) rule languages, such as Notation3 (N3), add declarative programming abilities to the SW platform. To execute rule-based programs, instead of building complex abstract machines, an option is to translate the program into an imperative language; in that case, the latter's compiler or interpreter can simply be used to execute the program. We propose a challenge to implement top-down reasoners by pre-compiling rules into a modern imperative language. The features of these languages, e.g., functions as first-class citizens and co-routining, and their extensive libraries, can simplify such a translation. Tackling this challenge can yield development simplicity, compared to typical abstract machine implementations; and integrated logic and imperative environments, as translated code can be directly called from other imperative code. We propose one solution to this challenge, called *fun3*, which pre-compiles N3 rules into Python functions. We provide initial performance results that show the feasibility of solving this challenge while not overly sacrificing performance.

Keywords

Notation3, Rules, Semantic Web, Top-down reasoning, Code translation

1. Introduction

Rule-based reasoning in the Semantic Web (SW), i.e., reasoning based on logical implications, adds declarative programming for decision making and problem solving. SW declarative programs follow SW principles for interoperability and exchangeability, and can operate directly on domain ontology concepts. E.g., in health informatics, applications include decision logic for interoperable health data [1], laboratory testing protocols [2], finding clinical pathways [3], and mitigating guideline conflicts [4].

A possible mechanism for executing rule-based programs is backward-chaining, a.k.a. top-down, reasoning. This starts from a given query, and attempts to resolve it using rule conclusions; the conditions of these rules are resolved using facts from data, or conclusions of (other) rules; of which the conditions then need to be resolved, and so on, until all relevant conditions are resolved. It has the benefit that not all possible rule conclusion instances need to be inferred. Top-down reasoning is often implemented using a Warren Abstract Machine (WAM) architecture [5], as it allows for the efficient execution of rule-based programs. WAM is still the standard for Prolog today [6]. However, WAM can be complex to implement, with multiple control structures (e.g., resolution stack, trail stack, choice point stack) and different hardware architectures to optimize for. There exist simplified abstract machines that could be easier to implement, which we discuss in related work. However, such an architecture always leads to an abstract machine that is wholly separate from other accompanying code. Such code is often written in an imperative language, and includes Web servers, database connectors, machine learning components, etc. for which imperative languages may be more suitable or popular.

Alternatively, the rule language can be translated into a high-level imperative language [7].In that case, the execution of the translated program can be delegated to the compiler or interpreter of the

RuleML+RR'25: Companion Proceedings of the 9th International Joint Conference on Rules and Reasoning, September 22–24, 2025, Istanbul, Turkiye
*Corresponding author.
wvanwoen@uottawa.ca (W. Van Woensel); doerthe.arndt@tu-dresden.de (D. Arndt); jos.deroo@ugent.be (J. D. Roo)
0000-0002-7049-8735 (W. Van Woensel); 0000-0002-7401-8487 (D. Arndt); 0000-0001-8862-0666 (J. D. Roo)

imperative language. The declarative rule program, for instance, in the form of translated imperative functions, can then also be directly integrated with imperative programs. This approach was first proposed by Nilsson et al. for Pascal [8], and later implemented by Weiner et al. [7] for C. Since then, modern, high-level languages have been developed with features that can simplify the translation, and support mechanisms such as co-routining (i.e., delaying evaluation until arguments are instantiated).

Challenge. *Implementing a top-down reasoner for a SW rule language, by pre-compiling declarative rules directly into a high-level imperative language, thereby using the target language's compiler or interpreter to execute the translated rules.* To the extent possible, the translated code should only use constructs (e.g., functions, lambdas, classes) and mechanisms (e.g., recursion, co-routining) provided by the target language. E.g., WAM-style structures and mechanisms are to be avoided. Abstract data types can still be utilized, e.g., to more elegantly support SW concepts such as IRIs, literals and blank nodes.

This challenge targets the *development simplicity* of rule-based reasoners, by avoiding the implementation of a custom abstract machine. This could increase the deployment of declarative reasoning on SW, but may come at the expense of execution performance. We provide initial performance results of one challenge solution in this paper. The challenge also targets *integrated logic and imperative environments*, as translated code (e.g., functions) can be directly integrated into imperative programs [9]. The target language tooling can also be exploited to debug the ruleset, i.e., via its translated program; or perform performance / memory analysis. Our exemplar here is the Notation3 (N3) SW rule language [10, 11]. The *eye* reasoner, which was written in Prolog on top of SWI-Prolog [12] and supports both bottom-up and top-down reasoning, has been the state-of-the-art N3 reasoner for over 2 decades[1]. However, *eye* represents only a single platform (Prolog) and is maintained by a single team, which may undermine the longevity of N3 and reduce its breadth of features[2]. It would thus be advantageous if other top-down reasoners could more easily be developed for other platforms and by other teams.

We provide a preliminary solution to this challenge for N3, namely *fun3* (*fun*ctions implementing *N3*). *fun3* pre-compiles N3 rules into functions in the Python modern imperative language. We also used Python to implement the translation process. Top-down reasoning is implemented using continuation passing [7] and unification via comparisons and argument passing. We implemented a small set of N3 builtins as a proof-of-concept; and SW abstractions as Python classes to increase the code's conciseness. To reduce translation complexity and improve performance, Python offers the following:

- *Functions as first-class citizens*: to implement continuation passing, we can simply pass functions directly as continuations, meaning we do not require special constructs (e.g., freeze, melt [7]).
- *AST library*: using this library, we can programmatically construct Python programs during compilation, and directly run the Python programs after their construction.
- *Extensive libraries & tooling*: aside from the AST library, Python has many other libraries for e.g., implementing builtins (e.g., optimization, scheduling, machine learning). It also has extensive tooling available for debugging and analyzing the translated code.
- *Simplicity*: Python is known to be relatively easy to learn and use. Hence, both our translation and translated code may be easier to understand than other imperative languages. This relative simplicity comes at the cost of lower performance compared to e.g., C or Rust.
- *Co-routining and tabling support*: Python functions can be paused and resumed, which can be used to implement co-routining in rule languages. Python decorators [15] can also be used to implement tabling (i.e., memoization). These are considered future work and not tackled here.

Regarding development simplicity: the *fun3* translation code (`gen.py`), not counting utility functions (e.g., simplifying code generation with the AST library) and SW abstractions, takes up less than 700 lines of code. Our implementation and an online demo can be found in our GitHub repository [16]. We evaluated correctness using a custom test suite, and provide initial performance results that show the feasibility of overcoming the challenge without sacrificing too much performance.

We *invite others to take up this challenge*, and design other solutions and address other source (e.g., SHACL rules [17]) and target languages (e.g., Rust, C).

[1]The cwm [13] and jen3 [14] reasoners only implement bottom-up reasoning, and are currently not maintained.
[2]While *eye* is open-source, extending it would require knowledge on Prolog and the underlying *eye* architecture.

2. Background

2.1. Notation3

Notation3 (N3) Logic is a formalism that extends the Resource Description Framework (RDF) with rule-based reasoning. It adds graph terms and collections as first-class citizens, together with rules; and offers a range of math, string, list and logical builtins, with support for Scoped Negation As Failure (SNAF) [11, §3.10.4]. We refer to the W3C Community Group Report [11, 18] for a complete specification.

Let I, B, L, and V denote the sets of Internationalized Resource Identifiers (IRIs), blank nodes, RDF literals, and variables, respectively. The set of *N3 terms*, T_{N3}, is recursively defined as follows:

- $I \cup B \cup L \cup V$ is the set of *atomic terms*;
- $\{\ \{G\} \mid G \in N3GP\}$ is the set of *graph terms*, where G is an N3 graph pattern (see below);
- $\{(t_1, \ldots, t_n) \mid n \geq 0, t_i \in T_{N3}\}$ is the set of *collection terms*.

An *N3 triple* is a tuple $(s, p, o) \in T_{N3} \times T_{N3} \times T_{N3}$, and an *N3 graph pattern* (N3GP) is a set of N3 triples. Variables (or "universals") in N3 are universally quantified, whereas blank nodes are existentially quantified. We call a *Triple Pattern (TP)* an N3 triple that features variables or blank nodes. Regarding syntax, a variable is written with the `?` prefix, graph terms are delineated by curly braces `{}`, and collection terms are written using brackets `()`. The other terms are written the same way as in RDF.

An *N3 rule* is written as an N3 triple, with a subject and object graph term that act as rule clauses; and predicate `log:implies` or `log:impliedBy`. The syntax offers syntactic sugar `=>` and `<=` for these predicates, respectively. For brevity, we will use these symbols from now on. From an operational semantics standpoint, the *eye* and *fun3* reasoners execute rules with `<=` as top-down rules. The "head" of a rule is the conclusion being implied, and the rule "body" constitutes the conditions that imply the rule head. As our paper focuses on top-down reasoning, we only consider top-down rules. In particular, the following subset of rules, where $T_{N3\setminus B}$ is the set of N3 terms without blank nodes (B):

$\{\mathrm{tp}_0\}$ `<=` $\{\mathrm{tp}_1 \ldots \mathrm{tp}_n\}$ with
$\mathrm{tp}_0 \in T_{N3\setminus B} \times (T_{N3\setminus B} \setminus \{\texttt{=>}, \texttt{<=}\}) \times T_{N3\setminus B}$ and
$\mathrm{tp}_i \in T_{N3} \times (T_{N3} \setminus \{\texttt{=>}, \texttt{<=}\}) \times T_{N3}$ for $1 \leq i \leq n$ and $n \in \mathbb{N}$

In the rule head, only 1 TP (tp_0) is allowed (as is the case for *eye* and Prolog), and no blank nodes are allowed. We furthermore assume that all variables occurring in the head of the rule also occur in its body. Neither the rule head or body allow nested rules, as the triple predicates cannot be `=>` or `<=`.

Below, we show N3 rules for determining the `:descendantOf` relation, together with N3 data triples (namespaces omitted for brevity):

Code 1: descendantOf rules with example data.

```
:will :hasParent :vic. :vic :hasParent :ed. :ed :hasParent :pete.

{?x :descendantOf ?y.} <= {?x :hasParent ?y.}.
{?x :descendantOf ?y.} <= {?x :hasParent ?z. ?z :descendantOf ?y.}.
```

The first line shows 3 triples, stating that `:will` has a parent `:vic`, `:vic` has parent `:ed`, and `:ed` has parent `:pete`. The first rule (top-down) states that a resource `?x` having a parent `?y` (object; rule body) implies that `?x` is also a descendant of that parent `?y` (subject; rule head). The second rule states that a resource `?x` having a parent `?z` that is itself a descendant of `?y` (rule body), implies that `?x` is also a descendant of `?y` (rule head). Given an example query `:will :descendantOf ?a.`, top-down reasoning will find ancestors of `:will` using the above rules and data triples.

Structured N3 terms, including graph terms and collections, are called *ground* in case they do not include variables, and *unground* otherwise. We show an example of these structured terms below:

Code 2: Example rule with unground collections and data.

```
:will :top (:lotr :hhgttg :hobbit). :lotr dc:title "Lord of the Rings".
```

```
{ ?person :loves ?title.} <= { ?person :top ( ?book1 ?book2 ?book3 ).
                               ?book1 dc:title ?title.}.
```

The first line shows a data triple with a ground collection, listing the top 3 books of `:will`, and a triple stating the title of the first book. The rule states that a person having a top 3 list of books, with the first book having a title (rule body), implies that the person loves the latter title (rule head). Note that variables nested in the collection are hereby referenced in other rule TPs.

3. Methods

Our approach relies on continuation passing to implement top-down reasoning. We summarize our implementation of this mechanism below (Section 3.1). We discuss our unification mechanisms (Section 3.2) and how builtins can be implemented (Section 3.3). Details will be provided in a subsequent publication. Note that we use the term "function" to refer to functions in imperative languages.

3.1. Continuation Passing

Continuation passing has been described as "threading together" the subgoals in a rule body [7]. We describe this concept in the context of N3 rules, where subgoals correspond to TP resolutions. Below, we discuss the generation of a set of functions per rule (*rule functions*), the runtime linking of these functions using continuation passing (*top-down reasoning*), and finding rule candidates for linking (*clause indexing*). Figure 1 illustrates this process for the data and second rule in Listing 1.

Rule functions. Per rule, a *rule function* is generated for each TP in its body. Figure 1 shows rule functions *fun rule2* and *fun rule2_2* for the second rule in (Listing 1); the corresponding rule TP are shown on the right-hand side. A rule function will attempt to resolve its TP, i.e., find matching values for its variables, by searching for data triples or via top-down reasoning. For each set of matching values, the rule function will call its *continuation*. For all but the last body TP, this continuation is the next body TP's function (e.g., *rule2_2*), also called "internal" (`int ctu`). This way, a rule's functions are "threaded together" at compile time.

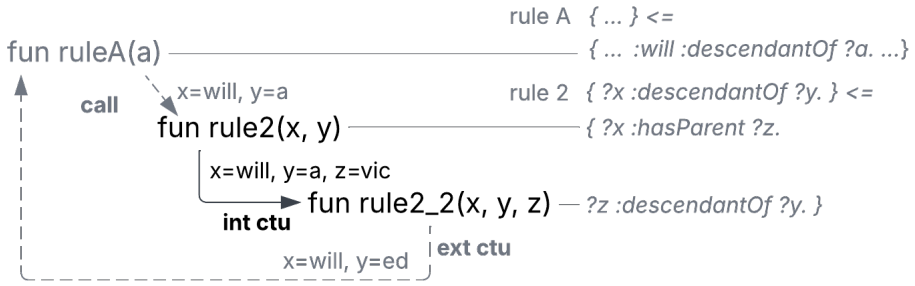


Figure 1: Illustration of continuation passing for Code 1.

Variable values are passed to rule functions using parameters[3]. The first rule function will have parameters for the rule head's variables (e.g., `(x,y)` for *fun rule2*), and subsequent functions have additional parameters for each prior body TP variable (e.g., `(x,y,z)` for *fun rule2_2*). As body variables, we consider both universal and existential (i.e., blank nodes) variables. Since blank nodes cannot occur in the rule head (see Section 2.1), only universals are considered there.

In Figure 1, *fun rule2* will call its continuation *fun rule2_2*, passing arguments `will` for `x`, `a` for `y`, and `vic` for `z`. The first two values were originally passed by *fun ruleA*; the third value was obtained by *fun rule2* by searching for data matches. Functions will use these parameter arguments to restrict the resolution of their TP; this is also referred to as sideways information passing [19].

[3]Variable names are disambiguated so they do not overlap across rule functions.

Top-down reasoning. The body TPs of a rule can be resolved by "invoking" other rules, i.e., top-down reasoning. In Figure 1, a body TP from another rule A, `:will :descendantOf ?a.`, is resolved using the head TP of rule 2, `?x :descendantOf ?y.` Such target rules are selected at compile-time.

In our case, "invocation" means the body TP's rule function (*fun ruleA*) will call the other rule's first function (*fun rule2*). We say that *input is provided* for rule 2: after unification (see below), values for its head variables are passed as arguments (`will` for `x`, `a` for `y`). If there is no value, the special *any* will be passed, which matches anything. The rule's functions will then be executed as described above; internal continuations are called with values for head and body variables (e.g., `vic` for `z`). In case these functions can resolve their TP,the last rule function will call an "external" continuation (`ext ctu`). Here, we say that the *rule's output is provided*: values for the rule head's variables are passed (`will` for `x`, `ed` for `y`).

In more detail, external continuations are implemented as lambda functions, which are also passed to the first rule function (not shown in Figure 1 for brevity). Hence, they represent a "threading" together at runtime. The pseudocode below illustrates the compile- and runtime threading:

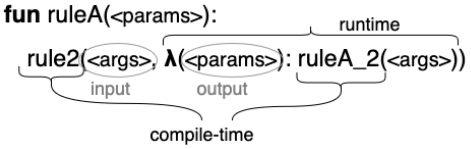


Figure 2: Compile- and runtime threading.

At compile time, *fun ruleA* is compiled to call *fun rule2* to commence top-down reasoning; also, a lambda function (external continuation) is compiled to call the subsequent *fun ruleA_2* (internal continuation). At runtime, the lambda function is passed to *fun rule2*; if successful, the latter will call the lambda function with its output—i.e., values for its head variables. The lambda function will pass on those arguments needed by *fun ruleA_2*. Using runtime external continuations, *fun rule2* can thus be used by any external function, such as *fun ruleA*, to resolve a body TP. Clearly, this approach is only possible in imperative languages where functions are first-class citizens, such as Python.

Clause indexing. To find rule candidates for resolving a TP, we apply clause indexing [7]. Currently, we map concrete predicates[4] to first rule functions, using the predicate in the rule's head TP. We note that, as a "WAM-style structure", this does not contradict the challenge, as the clause index is only used during translation and does not occur in the translated code.

3.2. Unification

We discuss unification in the context of top-down reasoning. If a *body TP* can be successfully unified with a *head TP*, then the body TP may be resolved using the rule associated with the head TP.

TPs are unified based on a pairwise unification of their terms. Term unification checks whether two terms can be considered identical by comparing concrete terms and substituting variables. We describe our implementation of unification below, for ground terms and variables (Section 3.2.1) and for unground structures (Section 3.2.2).

3.2.1. Concrete terms and variables

If both the **head and body TP terms are concrete**, i.e., either ground structures (collections, graph terms) or concrete atomic terms (literals, IRIs), we compare them at compile time. Unification succeeds if they are equal. For the example in Figure 1, the predicates of the body TP of rule A and the head TP of rule 2 will unify, as they are both `descendantOf`.

If the **head TP term is a variable**, an input value can be provided for it, and an output value will be provided by the rule (as described before). Figure 3 shows the pseudocode for *fun ruleA*.

[4]We assume predicates will not often be variables; if so, they are still considered but cannot be found using the index.

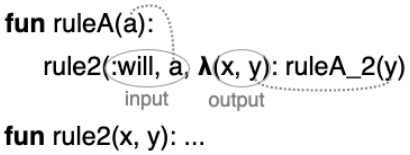


Figure 3: Pseudocode for *fun ruleA*.

Passing input. The rule's first function *fun rule2* will have parameters (e.g., `x`, `y`) for the variables. If the *body term is concrete*, it is simply passed as an argument. E.g., when unifying the subjects of the body TP (`will`) and head TP (`x`), term `will` is passed as an argument for `x`. If the *body term is also a variable* and it occurs as a parameter (e.g., `a`), this means it was found in a previous body TP, and thus has a prior value. So, this value is similarly passed as an argument: e.g., to unify TP objects `?a` and `?y`, `a` is passed as an argument for `y`. These constitute unifications, as the passed values will now be used in rule 2. In case the body variable did not have a prior value, then we pass the special *any* as argument.

Using output. Regardless of the above, we will also unify by using the output value provided by the rule. This covers cases where there was no prior value for the body variable, or this value was *any*. As part of top-down reasoning, the provided external continuation will be called with a value for the head TP variables (i.e., "output"). Here, we pass on the value for `y` to the next internal continuation of rule A (*fun ruleA_2*), thus ensuring the unification of the two variables.

If the **head TP term is concrete**, and the body TP term is a variable, then we cannot pass input as above. Here, we use an example that unifies the objects of body TP `?title :titleOf ?book` (from "rule K") and head TP `"Hobbit" :titleOf ?livre` (from "rule L"). The subjects are unified as described above. Figure 4 shows the pseudocode for rule K.

```
fun ruleK(title, book):
  if title = "Hobbit":
  then: ruleL(book, λ(livre): ruleK_2("Hobbit", livre)
```

Figure 4: Pseudocode for *rule K*.

Runtime comparison. If the body variable (e.g., `?title`) had a prior value, then we can unify by adding a runtime comparison between the body variable and head term.

Using constants. Analogous to before, in order to ensure unification, we will also pass the concrete head term (e.g., `"Hobbit"`) as the body variable's value to the next continuation.

Data matches. The same unification takes place when searching for data matches. In that case, the above mechanism is applied for a "head TP" with variable terms `?s`, `?p` and `?o`.

3.2.2. Unground structures

Ground structured N3 terms, including collections and graph terms (Section 2.1), are unified as described in the prior section. Special consideration is needed for unground structured terms (or structures). We consider the case for collections; unground graph terms are treated analogously.

An unground collection example is illustrated in Listing 2. As shown there, nested variables, such as `?book1`, can be referenced in other body TPs. Hence, we treat them as "regular" variables; they have corresponding function parameters, and values for them are passed to subsequent rule functions. We thus first extend the rule generation described before (Section 3.1). For the example's second TP:

Parameters are added for the collection's nested variables from the first TP. The example also shows a search for data matches (*data.find*), where the same unification as described before takes place.

Regarding unification, two unground collections (with the same length) are unified by performing unification per nested term:

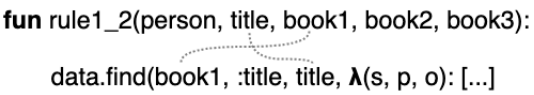


Figure 5: Pseudocode for the unground collections example.

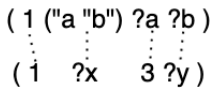


Figure 6: Example ground collections and their unification.

Unifying an unground collection with a single variable is more complex, and requires (a) *instantiating the collection* with function parameters; and (b) *indexing the collection* to get values for nested variables.

In the first case, the **body TP term is an unground collection**. We introduce a new example, where the body TP is `:will :top ( :lotr :hhgttg ?missing )` (from "rule M") and head TP is `?p :top ?books` (from "rule 7"), and we focus on their objects. Unification proceeds similarly to before (Figure 7):

```
fun ruleM(missing):
  rule7(:will, collection(:lotr, :hhgttg, missing), λ(p, books): ruleM_2(books[2])

fun rule7(p, books): ...
fun ruleM_2(missing): ...
```

Figure 7: Pseudocode for rule M.

Passing input. As the head term is a variable (`?books`), the rule function will have a parameter for it. The body collection `(:lotr :hhgttg missing)` is instantiated, if possible (here, with an argument for `missing`), and passed as an argument for `books`. This unifies the collection with the head variable.

Using output. Analogous to before, we ensure unification by using the rule output. The external continuation (lambda function) will be called with a collection[5] for the head variable (`books`). We index this collection to get values for the nested variables (`books[2]`). We then unify these values with the nested variables by passing them to the next continuation (`ruleM_2`).

In the second case, the **head TP term is an unground collection**. Using another example, the body TP is `?p :bottom ?books` (from "rule Q") and head TP is `:will :bottom ( ?other :wheeloftime )` (from "rule 14"), and we again focus on unifying their objects.

```
fun ruleQ(books):
   if books == collection(?other :wheeloftime):
   then:
       rule14(books[0], λ(other): ruleQ_2(:will, collection(other :wheeloftime))

fun rule14(other): ...
fun ruleQ_2(p, books) ...
```

Figure 8: Pseudocode for rule Q.

Runtime comparison. If the body variable (`?books`) had a prior value, we add a runtime comparison to check whether the body variable matches the "compile-time" version of the head collection—i.e., they have the same length and their concrete terms unify.

Passing input. The first rule function will have parameters for the nested variables (`other`) in the head collection. We index the body variable[6] to get values for these nested variables (`books[0]`). As

[5]This value will be a collection due to the prior unification.

[6]Its value will be a collection due to the above comparison. It may also be *any*, which spawns more *any*'s upon indexing.

above, this requires `?books` to have a prior value. This unifies the nested variables with concrete values.

Using output. The external continuation will be called with values for the nested variables. We ensure unification here by using these values (`other`) to instantiate the rule head's collection, and pass the latter as the body variable's value to the next continuation (`ruleQ_2`).

3.3. Builtins

Builtins are predicates with predefined semantics for math, string, list, time, logical and other operations. E.g., this TP uses the `math:sum` builtin to calculate the total cost of a product:

```
( ?price ?shipping ) math:sum ?total .
```

The subject and object of the TP act as input or output for the builtin, depending on its mode [20]. Here, the subject collection constitutes the input; the `math:sum` builtin calculates the sum of its members. The result is unified with the object of the TP (`?total` variable), which thus constitutes the output.

In our approach, builtins are implemented as predefined functions, which accept a subject and object as arguments. If successful, a builtin function calls a continuation with the subject and object; one of them may be substituted with a result, depending on the mode. We show the pseudocode for the `math:greaterThan` builtin below in Figure 9.

```
fun math_greaterThan(s, o, ctu):
  if (s is Literal and value(s) is numeric) and
    (o is Literal and value(o) is numeric):
      if value(s) > value(o):
        ctu(s, o)
```

Figure 9: Pseudocode for `math:greaterThan` builtin.

The builtin function checks whether the subject and object adhere to the expected term types and datatypes (here, literals with a numeric datatype). After, the builtin checks whether the subject term value is larger than the object term value. If so, the continuation is called with the same subject and object. If not, the continuation is not called, and the TP will not yield any results.

Currently, we implemented several math builtins (sum, comparators) and list builtins (iterate, append). Their implementation can be found online [16].

4. Experiments

To evaluate soundness, we validated *fun3* with a custom test suite, including a subset of *eye* tests (in case their builtins were supported by *fun3*). The test suite can be found online [16]. Future work includes extending this test suite with new test cases. Moreover, we target a formal study of soundness and completeness in a subsequent publication.

The remainder of this section reports on an initial performance evaluation of *fun3*, using the setup described in Section 4.1. We provide the results in Section 4.2, and discuss them in Section 4.3.

4.1. Setup

Datasets and rules. We base ourselves on a previously introduced healthcare use case on Zika screening [21]. For this experiment, we re-implemented the described CDC testing guidelines using N3 rules. These rules determine whether a patient should be tested for Zika, based on factors such as pregnancy, symptoms, recent travel to Zika areas, or sexual contacts. We expand on the test case with rules of different complexity:

- **Ruleset 1**: rules only check a single condition for Zika testing, namely whether the person is pregnant. Pregnancy is checked by a separate rule that finds active and confirmed conditions with the clinical code for pregnancy. Utility rules allow for easier navigation of the HL7 FHIR vocabulary. In total, there are 5 rules in this ruleset.
- **Ruleset 2**: rules check three conditions for Zika testing, namely pregnancy, whether the person has a Zika symptom, and whether they had a possible Zika exposure, due to travel to a Zika area or sexual encounters with a possibly exposed person. In total, there are 16 rules in this ruleset.

We represent patient data using the HL7 FHIR [22] vocabulary[7], an EHR interoperability standard that offers an RDF representation (Turtle syntax). We randomly generated datasets of patients with increasing sizes (100, 200, 500, 700, 1000, 2000, 5000 patients) and varying likelihoods of matching rule conditions: in **Dataset 1%** and **Dataset 2%**, patients respectively have a 1% and 2% chance of meeting a condition for Zika testing. We show the detailed sizes of the datasets below:

# patients	dataset 1%	dataset 2%
100	875	1749
200	1882	3312
500	4603	7734
700	6353	11873
1000	9435	16878
2000	18252	34945
5000	44916	83582

Table 1
Datasets: number of triples.

All datasets, rules, and detailed experiment results are in our online GitHub repository [16].

Evaluated systems. We compare *fun3* with eye v11.19.7, the only other N3 reasoner that can perform top-down reasoning. We ran *fun3* using Python v.3.13.5, the latest version at the time of writing. For finding data matches, we currently use a simple datastore implementation that uses *spo*, *pos*, and *osp* indices, and is based on the *rdflib* [23] SimpleMemory store. We use a C++ parser that was generated by ANTLR [24] from the N3 grammar, and integrated into *fun3* using Python language bindings.

Hardware. The experiments were conducted on a MacBook Pro with an Apple M1 Pro processor, 32 GB of RAM, and a 1 TB SSD, running macOS Sonoma 14.6.1. Each experiment was executed 5 times to ensure reliable results, and the performance metrics were averaged over these runs.

4.2. Results

Tables 2 and 3 show the average data and rule load times (*load*), reasoning times (*reason*), and total times (*total*; sum of load and reason times) for *eye* and *fun3*. Table 2 shows the overall times and times per dataset; Table 3 shows the times per ruleset (overall times repeated for ease of reference).

	overall			dataset 1%			dataset 2%		
	load	reason	total	load	reason	total	load	reason	total
eye	1405	7	1412	956	3	959	1854	10	1864
fun3	292	110	402	202	41	243	383	179	562

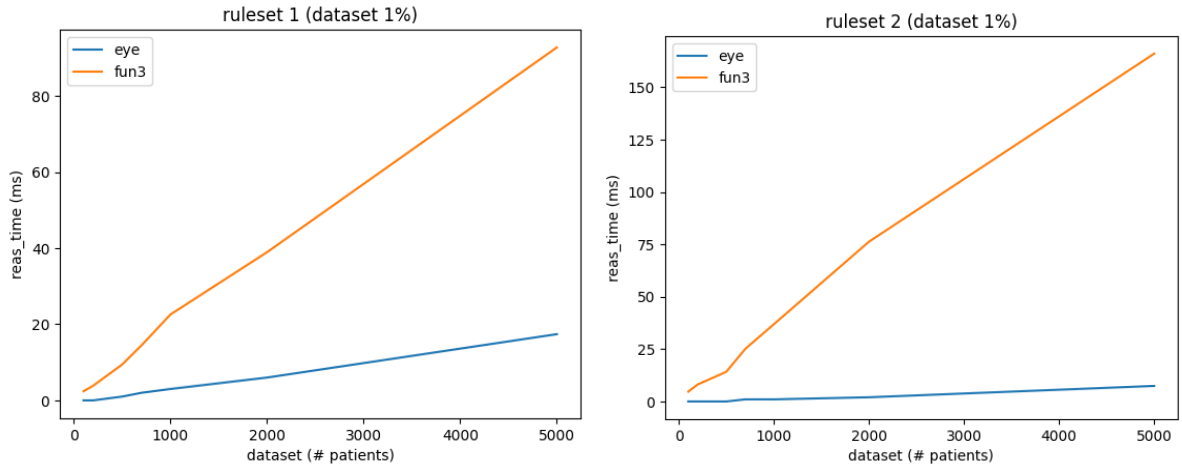
Table 2
Performance (milliseconds): overall and averages per dataset.

The time spent on average by *fun3* to generate the Python code is 3ms for *ruleset 1* and 5ms for *ruleset 2* (part of *reason* times). Figures 10 and 11 plot the reasoning times for *eye* and *fun3* against the dataset sizes, grouped per dataset type.

[7]We use a simplified version of the FHIR vocabulary, which relies less on deeply structured data; it is described online [16].

	overall			ruleset 1			ruleset 2		
	load	reason	total	load	reason	total	load	reason	total
eye	1405	7	1412	1402	7	1409	1408	7	1415
fun3	292	110	402	287	36	323	298	184	482

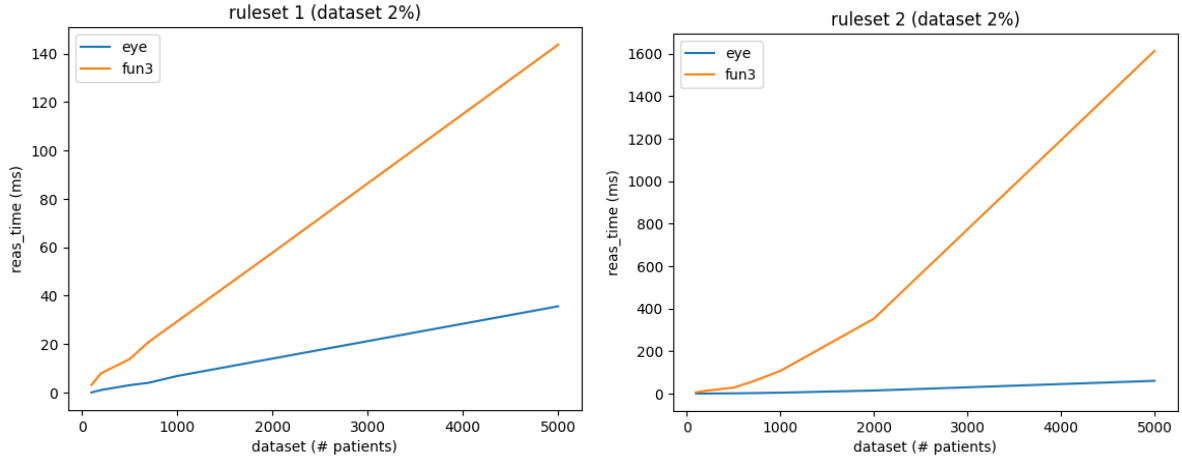
Table 3
Performance (milliseconds): overall and averages per ruleset.



(a) Ruleset 1 + Dataset 1%

(b) Ruleset 2 + Dataset 1%

Figure 10: Comparison of *eye* and *fun3* performance (reasoning time) for *dataset 1%*.



(a) Ruleset 1 + Dataset 2%

(b) Ruleset 2 + Dataset 2%

Figure 11: Comparison of *eye* and *fun3* performance (reasoning time) for *dataset 2%*.

4.3. Discussion

Load times. We observe that average load times for *eye* are much higher than for *fun3*, by almost a factor of 5 (4.8) on average. The *eye* reasoner uses SWI-Prolog Definite Clause Grammars (DCGs) to parse N3[8]. While DCGs are more maintainable, and integrate seamlessly with Prolog reasoning[9], they tend to offer reduced parsing performance—this especially becomes apparent for larger files. SWI-Prolog's clause indexing also trades load-time for execution efficiency, and may thus have a further

[8] *eye* loads Turtle data about 10x faster than N3 data; however, Turtle does not support N3 constructs such as graph terms.
[9] As mentioned, *eye* is built on top of SWI-Prolog.

(albeit likely much smaller) impact on load performance. As mentioned, *fun3* uses an ANTLR C++ parser, which seems to perform much better.

Reasoning times. Inversely, average reasoning times for *eye* are far lower, by almost a factor of 16 (15.7) on average. We also note that, for datasets of increasing size, reasoning times for *fun3* increase by a larger factor compared to *eye*; especially for *ruleset 2*, i.e., the more complex ruleset. In fact, for *ruleset 2* on *dataset 2%*, i.e., the dataset with more matching rule conditions, the increase in reasoning times fits an exponential curve. While *fun3*'s reasoning performance is thus significantly impacted by ruleset type and dataset type and size, *eye*'s performance is not significantly impacted by those factors.

It is not surprising that *eye*'s reasoning performance is far superior to *fun3*. The *eye* reasoner has been under development for over 20 years, and the SWI-Prolog on which it is based for close to 40 years. There are also no optimizations implemented in *fun3*. At the same time, we observe that the Python code generation, an inherent part of our pre-compilation approach, is not a significant overhead.

This initial performance evaluation is clearly limited in scope, and does not provide a comprehensive view of *fun3*'s performance for different rule- and datasets. However, we argue that it shows the feasibility of the approach, at least for relatively small datasets. As outlined in the introduction, this challenge may involve trading execution efficiency for development simplicity. That said, there are several avenues for improving the performance of *fun3*, which we outline in future work. Another consideration here is that, in contrast to *eye*, our approach yields a integrated logic and imperative environment.

5. Related Work

Most related work has taken place in the Prolog realm. We discuss prior work on continuation passing (Section 5.1), simplified abstract machines (Section 5.2), translation into other types of languages (Section 5.3), and other work on compiling N3 into high-level languages (Section 5.4).

5.1. Continuation passing

This section focuses on continuation passing in Prolog, where most prior work has taken place.

The early work by Weiner et al. [7] seems the closest to our approach. The authors outlined 3 overall approaches to running Prolog: (1) compilation of Prolog directly into machine code; (2) WAM-style approaches, which compile Prolog into a sequence of instructions in a custom lower-level language, which is itself implemented by a separate interpreter (e.g., WAM) [5]; and (3) continuation-style approaches, which compile Prolog into a high-level language and rely on continuation passing to implement non-determinism[10]. Like the authors, we pursue option (3) in this work.

The main benefit of option (3) is that the language's existing compiler or interpreter can be leveraged; in their case, the C compiler, and in our case, the Python interpreter. This makes the approach *portable*, when such a compiler or interpreter is already installed on most machines; this is the case for both C and Python. The approach further offers *simplicity*, as availability of a compiler or interpreter obviates the need to write an abstract machine. Moreover, compared to option (2), the authors note that *efficiency* can still be obtained, using a set of optimizing techniques. To implement continuation passing in C, which does not support functions as first-class citizens, the authors implemented two primitives ("freeze" and "melt") in assembly language. No details are given on the core approach to unification, aside from its optimization based on argument modes, i.e., directionality.

We translated N3 rules into a modern high-level imperative language, Python, describing both our continuation passing and unification approach. We further provide our implementation online.

Boyd et al. [9] translate Prolog into native C functions (called PIC, Prolog In C, functions). Such a function will directly invoke other PIC functions to resolve its clause. Hence, as in our case, it relies on the language's call stack, instead of requiring a separate control stack as with WAM. Nevertheless,

[10]Here, this refers to Prolog predicates being able to return multiple values due to backtracking.

WAM-like unification instructions are still created, and WAM control structures (continuation lists, analogous to resolution stacks; and copy, trail, and choice point stacks) are compiled into the PIC functions. The authors mention, as a main benefit of their approach, the creation of a dual logic and imperative programming environment, as PIC functions can be integrated into regular C programs.

Our challenge involves leveraging, to the extent possible, only the constructs and mechanism provided by the target imperative language. However, it may become apparent that additional mechanisms may be needed to optimize performance (e.g., tabling, reducing search space). It will be interesting to see the performance impact of such mechanisms versus the extra complexity they introduce.

5.2. Other Prolog abstract machines

There exist Prolog abstract machines that depart from the "conventional" WAM architectures. *SWI-Prolog* [25] is based on the ZIP [26] abstract machine, which, similarly to WAM, compiles Prolog into an intermediate language. However, it is simplified compared to WAM, having only a minimal set of instructions (7 instructions, without optimization). It interleaves compilation into ZIP instructions with interpreted execution [25, 6]. *BinProlog* [27] replaces WAM with a simplified continuation passing runtime system (the "BinWAM"), a heap-only runtime system that is specialized towards binary programs. Prolog programs are compiled into operationally equivalent binary logic programs for this simplified WAM, a "binarization" process that is similar to our translation into a series of continuations.

While the authors utilize simplified Prolog abstract machines, thus improving development simplicity (one of the goals of our challenge), they do not meet the challenge introduced in this paper. (Note that, as before and below, this is not a reflection on the quality of the approaches.) For more on Prolog implementations based on WAM, as well as alternatives, we refer to Körner et al. [6].

5.3. Translation into other types of languages

Our challenge targets translation into imperative programming languages. Hanus [28] shows a translation into a functional logic language. They note that, while logical programming offers flexibility due to free variables, unification and built-in search, it might yield infinite search spaces. Functional programming offers compactness with nested expressions, and an optimal on-demand evaluation strategy. It can thus be opportune to integrate their features in a single, functional logic language, such as Curry [29]. The author maps logic programs into such functional logic programs, which keep the flexibility of logic programming while reducing infinite search spaces to finite ones.

The language targeted by the translation is the functional logic language Curry, which was developed by the author (and others) of the work. We argue that, by instead targeting a pre-existing (popular) imperative language, the notion of an integrated programming environment becomes more enticing. Such languages are also likely to have more tooling, which can be exploited to debug the original ruleset, i.e., via its translated program; or perform performance / memory analysis.

5.4. Translating N3 into imperative languages

Van Woensel et al. [1] previously pre-compiled N3 rules, together with a domain ontology, into blockchain smart contracts written in the Solidity imperative language [30]. The goal was to shield domain experts, who are best suited to encode high-level smart contract logic, from the details of the blockchain programming language. To support multiple blockchain platforms (e.g., Hyperledger [31], with JavaScript), the N3 rules and ontology were converted into a "bridge" representation, which captures the declarative logic using general imperative programming constructs. These bridge abstractions are then "transpiled" into specific blockchain languages, such as Solidity or JavaScript.

This work represented a first step for translating N3 into imperative programming languages. However, its implementation was informed by the (a) economic rules of blockchain-based systems, where computational work expends cryptocurrency; and (b) limitations of the main target language, Solidity. For that reason, the approach put a number of important restrictions on the input rules:

(1) N3 rules should be structured as a "star-shaped" graph, originating from single term and consisting of sequential paths of nodes.

(2) All predicates should be concrete, and in the N3 graph from (1), variables can occur as intermediary and leaf nodes, and concrete terms can occur as leaf nodes. Each URI term should further correspond to exactly one ontology class declaration.

(3) The set of N3 rules should form a single "chain", where a subsequent rule relies on the output of prior rules, until a final conclusion is inferred.

Hence, the approach did not implement top-down reasoning, in contrast to *fun3*. Continuation passing would have been complex to implement, as Solidity does not support functions as first-class citizens. For examples of translated smart contracts, we refer to its GitHub repository[11].

EyeLet [32] uses Large Language Models (LLMs)—specifically Large Reasoning Models (LRMs) such as ChatGPT o3—to translate N3 into Python programs. Hence, the approach uses both the same source language (N3) and target language (Python) as *fun3*. In ad-hoc experiments, De Roo found that the generated code frequently produces the expected outcomes without further debugging. Nevertheless, such ad-hoc tests do not guaranteee correctness; and, in other cases, the generated code was found to be (highly) inaccurate. It also often takes several minutes to generate a program. Of course, performance and accuracy here can improve as the quality of the foundational model increases. EyeLet also supports the production of goal-directed reasoning traces (proof-oriented explanations).

As *fun3* always applies the same translation mechanism, once the mechanism has been proven sound and complete, correctness can be guaranteed. Compared to EyeLet, the code generation overhead in our experiment is negligible (Section 4.2). *fun3* currently leverages Python's built-in debugging support (`sys.settrace`) to print an execution trace, which can serve as an explanation of the inferences.

6. Conclusion and Future Work

SW rule languages, such as N3, add declarative programming abilities to the SW. However, the complexity of implementing top-down reasoners, often using abstract machines, may be a barrier for their deployment. We propose a challenge that translates the declarative rules into a modern imperative language. Then, the latter's compiler or interpreter can be used to execute the translated program. This challenge thus aims to (a) reduce the development complexity of top-down reasoners; and (b) support integrated logic and imperative environments, as the translated code (e.g., functions) can be integrated into imperative code. Modern imperative languages, featuring functions as first-class citizens and extensive libraries, can simplify such a translation; their tooling can also be leveraged for debugging and code analysis. Initial performance results show the feasibility of *fun3* in solving the challenge.

In future work, we will extend our current test suite with new test cases, and formally study the soundness and completeness of our approach. We further plan the following:

- *Tabling.* As tabling (memoization) is currently not implemented, certain rules will not terminate; and intermediate results may be frequently re-computed, which could have a large impact on performance. In Python, decorators [15] can be used to implement memoization.

- *Co-routining.* Co-routining in rule languages (see before) allows executing rules that would otherwise not be executable (e.g., due to clause ordering and use of certain builtins). This could be implemented using Python co-routining, where the execution of functions can be paused and resumed.

- *Faster datastore.* We currently rely on a simple datastore implementation that uses *spo*, *pos*, and *osp* indices (Section 4.1). Using a more performant datastore implementation (e.g., covering multiple variable access patterns) can reduce the overhead when searching for data matches.

- *N3 builtins.* We aim to implement the remaining N3 builtins, including those supporting Scoped Negation As Failure (SNAF) [11, §3.10.4].

[11] https://github.com/william-vw/blockiot-cds/

Other optimizations will be pursued, including the re-ordering of TP's, based on their shared variables and estimated selectivity; and existing methods to reduce the backtracking search space. We aim to target a more performant imperative language (such as C or Rust). We will look into "unfolding" the continuation passing mechanism into a series of FOR-loops; while this will result in a larger size of the translated code, loops may be more performant. Afterwards, we will perform a more comprehensive evaluation; e.g., based on cases from OpenRuleBench [33, 34]. We also target a more fine-grained performance comparison with *eye* that separately considers its parsing vs. indexing overhead. We note that, when targeting a functional programming language (e.g., Haskell) instead of an imperative language, existing work on backtracking (e.g., using monads [35]) could be leveraged.

Regarding explanations, *fun3* already leverages Python's built-in debugging (`sys.settrace`) to print execution traces. In addition to debugging, these can also serve as an explanation of the inferences.

Finally, another avenue of future work is to support bottom-up (forward) reasoning. When allowing for nested rules, bottom-up N3 rules can generate other, more specific N3 rules. For instance, such rules can be generated based on an ontology's TBox for optimizing reasoning [36].

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] W. Van Woensel, O. Seneviratne, Semantic interoperability on blockchain by generating smart contracts based on knowledge graphs, Blockchain: Research and Applications (2025) 100320. URL: https://www.sciencedirect.com/science/article/pii/S2096720925000478. doi:https://doi.org/10.1016/j.bcra.2025.100320.

[2] W. Van Woensel, M. Elnenaei, S. S. R. Abidi, D. B. Clarke, S. A. Imran, Staged reflexive artificial intelligence driven testing algorithms for early diagnosis of pituitary disorders, Clinical Biochemistry 97 (2021) 48–53. URL: https://www.sciencedirect.com/science/article/pii/S0009912021002265. doi:https://doi.org/10.1016/j.clinbiochem.2021.08.005.

[3] H. Sun, D. Arndt, J. D. Roo, E. Mannens, Predicting future state for adaptive clinical pathway management, J. Biomed. Informatics 117 (2021) 103750. URL: https://doi.org/10.1016/j.jbi.2021.103750. doi:10.1016/J.JBI.2021.103750.

[4] B. Jafarpour, S. Raza Abidi, W. Van Woensel, S. S. Raza Abidi, Execution-time integration of clinical practice guidelines to provide decision support for comorbid conditions, Artificial Intelligence in Medicine 94 (2019) 117–137. URL: https://www.sciencedirect.com/science/article/pii/S0933365717303883. doi:https://doi.org/10.1016/j.artmed.2019.02.003.

[5] D. H. Warren, An abstract prolog instruction set, Technical report (1983).

[6] P. Körner, M. Leuschel, J. Barbosa, V. S. Costa, V. Dahl, M. V. Hermenegildo, J. F. Morales, J. Wielemaker, D. Diaz, S. Abreu, et al., Fifty years of prolog and beyond, Theory and Practice of Logic Programming 22 (2022) 776–858.

[7] J. L. Weiner, S. Ramakrishnan, A piggy-back compiler for prolog, in: R. L. Wexelblat (Ed.), Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988, ACM, 1988, pp. 288–296. URL: https://doi.org/10.1145/53990.54019. doi:10.1145/53990.54019.

[8] J. F. Nilsson, On the compilation of a domain-based prolog, in: R. E. A. Mason (Ed.), Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983, North-Holland/IFIP, 1983, pp. 293–298.

[9] J. L. Boyd, G. M. Karam, Prolog in "c", ACM SIGPLAN Notices 25 (1990) 63–71. URL: https://doi.org/10.1145/382076.382646. doi:10.1145/382076.382646.

[10] T. Berners-Lee, D. Connolly, L. Kagal, Y. Scharf, J. Hendler, N3Logic: A logical framework

for the World Wide Web, Theory and Practice of Logic Programming 8 (2008) 249–269. URL: http://arxiv.org/abs/0711.1533. doi:10.1017/S1471068407003213.
[11] W. Van Woensel, D. Arndt, P.-A. Champin, D. Tomaszuk, G. Kellogg, Notation3 Language, W3C Community Group Report, W3C, 2023. https://w3c.github.io/N3/reports/20230703/.
[12] SWI-Prolog Developers, SWI-prolog (swipl), https://www.swi-prolog.org/, 1987.
[13] T. Berners-Lee, Closed world machine (cwm), https://www.w3.org/2000/10/swap/doc/cwm, 2009.
[14] W. Van Woensel, jen3, https://github.com/william-vw/jen3, 2023.
[15] K. Smith, J. Jewett, S. Montanaro, A. Baxter, Pep 318 – decorators for functions and methods, https://peps.python.org/pep-0318/, 2003.
[16] fun3 online repository, https://github.com/william-vw/fun3, 2025.
[17] Shacl advanced features: Shacl rules, https://www.w3.org/TR/shacl-af/#rules, 2017.
[18] D. Arndt, P.-A. Champin, Notation3 Semantics, W3C Community Group Report, W3C, 2023. https://w3c.github.io/N3/spec/semantics.html.
[19] C. Beeri, R. Ramakrishnan, On the power of magic, in: Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '87, Association for Computing Machinery, New York, NY, USA, 1987, p. 269–284. URL: https://doi.org/10.1145/28659.28689. doi:10.1145/28659.28689.
[20] W. Van Woensel, P. Hochstenbach, Notation3 Builtin Functions, W3C Community Group Report, W3C, 2023. https://w3c.github.io/N3/reports/20230703/builtins.html.
[21] D. Arndt, W. Van Woensel, D. Tomaszuk, Sin3: Scalable inferencing with sparql construct queries., in: ISWC (Posters/Demos/Industry), 2023.
[22] HL7 International, HL7 Fast Health Interop Resources (FHIR), 2025. URL: https://www.hl7.org/index.cfm.
[23] RDFLib Team, Rdflib, 2025. URL: https://rdflib.readthedocs.io/en/stable/.
[24] T. Parr, Another tool for language recognition (antlr), 2025. URL: https://www.antlr.org.
[25] SWI-Prolog developers, SWI-prolog: Reference manual., 2024. URL: https://www.swi-prolog.org/download/stable/doc/SWI-Prolog-9.2.2.pdf.
[26] W. Clocksin, A portable prolog compiler, in: Logic Programming Workshop, Albufeira Portugal, January, 1983.
[27] P. Tarau, Binprolog: a continuation passing style prolog engine, in: International Symposium on Programming Language Implementation and Logic Programming, Springer, 1992, pp. 479–480.
[28] M. Hanus, From logic to functional logic programs, Theory and Practice of Logic Programming 22 (2022) 538–554.
[29] M. Hanus, Curry: An integrated functional logic language, 2025. URL: http://www.curry-lang.org.
[30] Ethereum, Solidity, 2016. URL: https://docs.soliditylang.org/en/v0.8.13/.
[31] The Linux Foundation, LF decentralized trust (previously hyperledger), 2025. URL: https://www.lfdecentralizedtrust.org/.
[32] J. De Roo, Llm generated code to emulate thinking (let), 2025. URL: https://github.com/eyereasoner/eye/tree/master/let.
[33] P. Fodor, Openrulebench - benchmarks for semantic web rule engines, https://www3.cs.stonybrook.edu/ pfodor/openrulebench_web/, 2011.
[34] S. Liang, P. Fodor, H. Wan, M. Kifer, Openrulebench: an analysis of the performance of rule engines, in: J. Quemada, G. León, Y. S. Maarek, W. Nejdl (Eds.), Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009, ACM, 2009, pp. 601–610. URL: https://doi.org/10.1145/1526709.1526790. doi:10.1145/1526709.1526790.
[35] O. Kiselyov, C.-c. Shan, D. P. Friedman, A. Sabry, Backtracking, interleaving, and terminating monad transformers: (functional pearl), SIGPLAN Not. 40 (2005) 192–203. URL: https://doi.org/10.1145/1090189.1086390. doi:10.1145/1090189.1086390.
[36] D. Arndt, B. De Meester, P. Bonte, J. Schaballie, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, R. Van de Walle, et al., Improving owl rl reasoning in n3 by using specialized rules, in: International Experiences and Directions Workshop on OWL, Springer, 2015, pp. 93–104.