Benchmarking for Integrating Logic Rules with Everything Else

Yanhong A. Liu   Scott D. Stoller   Yi Tong   K. Tuncay Tekle
Computer Science Department, Stony Brook University, Stony Brook, New York, USA
{liu,stoller,yittong,tuncay}@cs.stonybrook.edu

Integrating logic rules with other language features is increasingly sought after for advanced applications that require knowledge-base capabilities. To address this demand, increasingly more languages and extensions for such integration have been developed. How to evaluate such languages?

This paper describes a set of programming and performance benchmarks for evaluating languages supporting integrated use of rules and other features, and the results of evaluating such an integrated language together with logic languages and languages not supporting logic rules.

1 Introduction

As knowledge-base capabilities are increasingly needed for advanced applications, especially applications that require sophisticated reasoning, logic rules are increasingly needed together with other language features. This leads to increasingly more languages and extensions that provide such programming support, for using rules through libraries or directly as built-ins, in many different ways. To evaluate such language support for integrated uses of rules with other language features, a set of benchmarking problems is needed.

Many benchmarking suites have been developed for evaluating logic languages and rule engines, and even drastically more for imperative and other languages. In contrast, benchmarks and evaluation for languages that support integrated use of rules together with other language features have been lacking.

This paper describes a set of programming and performance benchmarks for such evaluation, and the uses of these benchmarks in evaluating an integrated language together with logic languages and languages not supporting logic rules. The benchmarks are developed and organized into three sets:

OpenRuleBench—for problems focused on using rules and evaluating rule systems, including all problems from OpenRuleBench [15], but modified and organized to use more appropriate language features for the benchmarking itself, especially imperative scripts for running and taking measurements.

RBAC—for problems that require (1) frequent imperative updates interleaved with expensive queries and (2) objects and classes with inheritance to organize system components, subsuming full Role-Based Access Control (RBAC) [1, 17], with queries expressed in different ways of using rules and not using rules.

PA—for problems expressed using rules together with set queries and aggregate queries and recursive functions, from problems in program analysis (PA), because aggregate queries are particularly important for many database, data mining, and machine learning applications, but are not in OpenRuleBench [15].

The input data and workload for these benchmarks are designed for runs with different problem scales—(1) large input data for queries using rules, (2) large query results from inference using rules, (3) large number of rules, (4) frequent switches between using rules and using other features, and (5) frequent invocations of inference using rules.

To appear in EPTCS.

The evaluation is for benchmark problems written in (1) Alda, (2) XSB, (3) Python, and (4) DistAlgo. XSB [28, 32] is a top logic rule engine for queries using rules, selected for reasons stated in Section 3.1. Python [27] is most used for its rich features including high-level queries but lacks logic rules. DistAlgo [20, 16] extends Python for distributed programming with higher-level and faster queries but still lacks logic rules. Alda [21] is an integrated language that supports logic rules as built-ins by building on DistAlgo and employing XSB. The evaluation yields the following main results:

- Queries using rules in Alda and XSB are similar, and are (1) simpler than using `while` loops in Python and DistAlgo, even when these loops use high-level set queries, and (2) drastically, asymptotically faster than `while` loops with high-level set queries in Python and DistAlgo.
- Benchmarking using more generic imperative scripts in Alda is drastically simpler than using rules in XSB in OpenRuleBench; and integrated use of rules with other features for RBAC and PA problems yields simpler programs than not using rules or only using rules.
- The performance of Alda programs is the accumulated performance of queries in XSB, other features in Python and DistAlgo, and interface between XSB and Python. Thus it is expected to increase accordingly when any part used is improved, including reducing the current interface overhead to 1% of it.
- Even with the current interface overhead, Alda programs are faster or even drastically faster than half or more of the rule engines tested in OpenRuleBench [15] for all but one benchmark, because the overhead is only linear in sizes of data and results and number of queries, and thanks to the XSB query performance.

Our analysis supports that similar results can be obtained by using other efficient rule engines, including ASP systems, and other languages not supporting logic rules, including lower-level languages. Our implementation has been available by request and will be made public pending improved documentation.

2 Language

XSB and Python are well established, and DistAlgo is subsumed by Alda. So we first introduce Alda, an integrated language that supports rules with all of sets, functions, updates, and objects, all as built-ins, seamlessly integrated without extra interface code. Figure 1 shows an example program in Alda.

Rules are defined using rule sets (e.g. lines 15–17, with rule set name `trans_rs`), and queried using calls to an inference function, `infer` (e.g. on line 19, using rule set `trans_rs`). The two rules in `trans_rs` (lines 16–17) define predicate `path` using predicate `edge`, where the first rule is the base case, and the second rule is the recursive case. The call to `infer` (line 19) returns the result of querying `path`, i.e., the set of pairs for which `path` holds, given that `edge` equals `RH`, i.e., `edge` holds for the set of pairs in `RH`.

In a rule set, predicates not in any conclusion are called base predicates; the other predicates are called derived predicates. The key idea is that predicates are simply set-valued variables and vice versa. So predicates can be defined and used directly as other variables, in any scope—global, object field, or local to a rule set. The only exception is that derived predicates can only be updated by `infer`, and are automatically maintained by an implicit call to `infer` when any non-local base predicates are updated. The exception ensures the declarative semantics of rules.

All other features—class, inheritance, method (function and procedure), update, and set query—are the same as in object-oriented languages. We mostly use Python syntax except for a few conventions from Java (`extends` for inheritance, `new` for object creation, and omission of `self` when there is no ambiguity) for ease of reading, and a few ideal syntax (`:=` for assignment, `{}` for empty set, and `+` for set union), and pattern matching in queries (in examples explained later).

For the example in Figure 1 but without using rules, computing the transitive closure (i.e., result of `infer` in function `transRH`) in a variable `T`, by adding a parameter `E` to function `transRH` and passing the value of `RH` to `E` when calling `transRH`, can use a `while` loop in Python, as used for [17]:

```
class CoreRBAC:                    # class for Core RBAC component/object
  def setup():                     # method to set up the object, with no arguments
    self.USERS, self.ROLES, self.UR := {},{},{}
                                   # set users, roles, user-role pairs to empty sets
  def AddRole(role):               # method to add a role
    ROLES.add(role)                # add the role to ROLES
  def AssignedUsers(role):         # method to return assigned users of a role
    return {u: u in USERS | (u.role) in UR}  # return set of users having the role
  ...
class HierRBAC extends CoreRBAC: # Hierarchical RBAC extending Core RBAC
  def setup():
    super().setup()                # call setup of CoreRBAC, to set sets as in there
    self.RH := {}                  # set ascendant-descendant role pairs to empty set
  def AddInheritance(a,d):         # to add inherit. of an ascendant by a descendant
    RH.add((a,d))                  # add pair (a,d) to RH
  rules trans_rs:                  # rule set defining transitive closure
    path(x,y) if edge(x,y)         # path holds for (x,y) if edge holds for (x,y)
    path(x,y) if edge(x,z), path(z,y)  # ... if edge holds for (x,z) and for (z,y)
  def transRH():                   # to return transitive RH and reflexive role pairs
    return infer(path, edge=RH, rules=trans_rs) + {(r,r): r in ROLES}
  def AuthorizedUsers(role):       # to return users having a role transitively
    return {u: u in USERS. r in ROLES | (u.r) in UR and (r.role) in transRH()}
  ...
h = new(HierRBAC, [])              # create HierRBAC object h, with no args to setup
h.AddRole('chair')                 # call AddRole of h with role 'chair'
...
h.AuthorizedUsers('chair')         # call AuthorizedUsers of h with role ‘chair’
...
```

Figure 1: An example program in Alda, for Role-Based Access Control (RBAC), demonstrating logic rules used with sets, functions, updates, and objects.

```
T = E.copy()
W = {(x,y) for (x,z) in T for (z2,y) in E if z2==z} - T
while W:
  T.add(W.pop())
  W = {(x,y) for (x,z) in T for (z2,y) in E if z2==z} - T
```

or a `while` loop with higher-level set queries in DistAlgo:

```
T := E.copy()
while some (x,z) in T, (z,y) in E | (x,y) not in T:
  T.add((x,y))
```

Using rules is clearly simpler, both conceptually and in the amount of code.

Alda is implemented by extending the DistAlgo compiler [20, 16] and invoking XSB [28, 32] for inference. DistAlgo is compiled to Python. Function `infer` translates data from Python to Prolog facts, translates results back, and invokes XSB in between using command line and file passing. The obvious overhead of this external interface can be removed with an in-memory interface between Python to XSB, which is actively being developed by the XSB team.[1] However, this has not affected Alda programs from having generally good performance, thanks to the XSB query performance.

3 Programming and performance benchmarks

[1]A version working for Unix, not yet Windows, has been released, and passing lists of length 100 million in memory took about 30 nanoseconds per element [32, release notes]. So even for the largest data in our experiments, of size a few millions, it would take 0.1-0.2 seconds to pass in memory, instead of 10-20 seconds with the current external interface.

Name	Description	Prog size	Data size
Join1	non-recursive tree of binary joins as inference rules	225	*
Join2	join from IRIS system producing large intermediate result	41	*
JoinDup	join of separate results of five copies of Join1	163	*
LUBM	university database adapted from LUBM benchmark	377	*
Mondial	geographical database derived from CIA Factbook	36	59,733
DBLP	well-known bibliography database on the Web	20	2,437,867
TC	classical transitive closure of a binary relation	75	*
SG	well-known same-generation siblings problem	90	*
WordNet	natural language processing queries based on WordNet	298	465,703
Wine	well-known OWL wine ontology as rules	1103	654
ModSG	modified SG to exclude ancestor-descendant relationships	38	*
Win	well-known win-not-win game with non-stratified negation	24	*
MagicSet	non-stratified rules from magic-set transformation	34	*

The three groups (of 6, 4, 3) in order are called large join tests, Datalog recursion, and default negation, respectively. Prog size is the XSB program size in lines of code without comments and empty lines. Data size is the input data size in number of facts; * means that scripts are used to generate input data of desired sizes.

Table 1: Benchmarks from OpenRuleBench.

This section presents three sets of benchmarks that we developed. They are written to express each given problem in the most direct way we can think of. This helps make the benchmarks clear and easy to read. This also avoids being unfair to any particular language or system being compared with.

3.1 OpenRuleBench—a wide variety of rule-based applications

OpenRuleBench [15] contains a wide variety of database, knowledge base, and semantic web application problems, written using rules in 11 well-known rule systems from 5 different categories, as well as large data sets and a large number of test scripts for running and measuring the performance. Among 14 benchmarks described in [15], we consider all except for one that tests interfaces of rule systems with databases (which is a non-issue for Python and its extensions DistAlgo and Alda, because Python has standard and widely-used database interfaces).

Table 1 summarizes the benchmarks. We compare with the benchmark programs in XSB, for three reasons: (1) XSB has been the most advanced rule system supporting well-founded semantics for non-stratified negation and tabling techniques for efficient query evaluation, and has been actively developed for over three decades, to this day; (2) among all systems reported in [15], XSB was one of the fastest, if not the fastest, and the single most consistent across all benchmarks; and (3) among all measurements reported, only XSB, OntoBroker, and DLV could run all benchmarks, but OntoBroker went bankrupt, and measurements for DLV were almost all slower, often by orders of magnitude.

We easily translated all 13 benchmarks into Alda, automatically for all except for three cases where the original rules used features beyond Datalog, which became two cases after we added support for negation. In all cases, it was straightforward to express the desired functionality in Alda, producing a program that is very similar to XSB or even simpler. Additionally, the code for reading data, running tests, timing, and writing results is drastically simpler in Alda than in XSB, because Alda extends Python. The three exception cases and additional findings are described below.

Result set. In most logic languages, including Prolog and many variants, a query returns only the first result that matches the query. To return the set of all results, some well-known tricks are used. The LUBM benchmark includes the following extra rules to return all answers of `query9_1`:

```
query9 :- query9_1(X,Y,Z), fail.
query9 :- writeln('========query9.======').
```

The first rule first queries `query9_1` to find an answer (a triple of values for `X`,`Y`,`Z`) and then uses `fail` to trick the inference into thinking that it failed to find an answer and so continuing to search for an answer; and it does this repeatedly, until `query9_1` does fail to find an answer after exhausting all answers. The second rule is necessary, even if with an empty right side, to trick the inference into thinking that it succeeded, because the first rule always ends in failing; this is so that the execution can continue to do the remaining work instead of stopping from failing.

In fact, this trick is used in all benchmarks, but other uses are buried inside the code for running, timing, etc., specialized for each benchmark, not as part of the rules for the application logic.

In Alda, such rules and tricks are never needed. A call to `infer` with query `query9_1` returns the set of all query results as desired.

Function symbols. Logic rules may use function symbols to form structured data that can be used as arguments to predicates. Uses of function symbols can be translated away. The Mondial benchmark uses a function symbol `prov` in several intermediate conclusions and hypotheses of the form `isa(prov(Y,X),provi)` or `att(prov(Y,X),number,A)`. They can simply be translated to `isa('prov',Y,X,provi)` and `att('prov',Y,X,number,A)`, respectively.

Negation. Logic languages may use negation applied to hypotheses in rules. Most logic languages only support stratified negation, where there is no negation involved in cyclic dependencies among predicates. Such negation can be done by set differences. The ModSG benchmark has such a negation, as follows, where `sg` is defined by the rules in the SG benchmark, and `nonsg` is defined by two new rules.

```
sg2(X,Y) :- sg(X,Y), not nonsg(X,Y).
```

In Alda, this can be written as

```
sg2 = infer(sg, rules=sg_rs) - infer(nonsg, rules=nonsg_rs)
```

where `sg_rs` and `nonsg_rs` are the rule sets defining `sg` and `nonsg`, respectively.

Alda also supports negation in rules. Its current implementation translates negation to tabled negation `tnot` in XSB. This handles even non-stratified negation by computing well-founded semantics using XSB [3], contrasting Prolog's negation as failure. The Win and MagicSet benchmarks have non-stratified negation. Both of them, as well as ModSG, can be expressed directly in Alda rule sets by using `not` for negation.

Benchmarking and organization. In OpenRuleBench benchmarks, even though the rules to be benchmarked are declarative and succinct, the benchmarking code for reading input, running tests, timing, and writing results are generally much larger. For example, the Join1 benchmark has 4 small rules and 9 small queries similar in size to those in the transitive closure example, plus a manually added tabling directive for optimization. However, for each of the 9 queries, 19 more lines for an import and two much larger rules are used to do the reading, running, timing, and writing.

In general, because benchmarking executes a bundle of commands, scripting those directly is simplest. Furthermore, organizing benchmarking code using procedures, objects, etc., allows easy reuse without duplicated code. These features are much better supported in languages like Python than rule systems, for both ease of programming and performance,

In fact, OpenRuleBench uses a large number of many different files, in several languages (language of the system being tested, XSB, shell script, Python, makefile) for such scripting. For example for Join1, the 4 rules, tabling directive, and benchmarking code are also duplicated in each of the 9 XSB files, one for each query; a 46-line shell script and a 9-line makefile are also used.

In contrast, our benchmarking code is in Alda, which uses Python functions for scripting. A single 45-line Alda program is used for timing any of the benchmarks, and for pickling (i.e., object serialization in Python, for fast data reading after the first reading) and timing of pickling.

Name	Features used for computing transitive role hierarchy
RBACnonloc	rule set `transRH_rs` with implicit `infer`, without `transRH()`, Sec. 3.2
RBACallloc	rule set `trans_role_rs` and `transRH()` that has only `infer`, Sec. 3.2
RBACunion	rule set `trans_rs` and `transRH()` that has `infer` and union, Sec. 2
RBACda	`while` loop and high-level set queries in DistAlgo, Sec. 2
RBACpy	`while` loop and high-level set comprehensions in Python, Sec. 2

Each benchmark performs a combination of updates to sets and relations `USERS`, `ROLES`, `UR`, and `RH` and queries with function `AuthorizedUsers(role)`, where the transitive role hierarchy is computed with a different way of using rules, or not using rules. In `AuthorizedUsers(role)` of all five programs, the call to `transRH()`, or reference to field `transRH`, is lifted out of the set query, by assigning its value to a local variable and using that variable in the query.

Table 2: Benchmarks for RBAC updates and queries.

Aggregation. Despite the wide variety of benchmarks in OpenRuleBench, no benchmark uses aggregate queries. Aggregate queries are essential for many database, data mining, and machine learning applications. We discuss them and compare with aggregate queries in a rule language like XSB in Section 3.3.

3.2 RBAC—rules with objects, updates, and set queries

The ANSI standard for Role-Based Access Control (RBAC [7, 1] involves many sets and query and update functions in a total of 9 components. To program the transitive role hierarchy at a high level, a complex and inefficient `while` loop was used before [17], because an efficient algorithm would be drastically even more complex.

With support for rules, the entire RBAC standard is easily written in Alda, similar as in Python [17], except with rules for computing the transitive role hierarchy, as shown in Figure 1, yielding a simpler yet more efficient program.

Below, we specify more ways of using rules to compute the transitive role hierarchy and function `AuthorizedUsers(role)` in Figure 1. All these ways are declarative and differ in size by only 1-2 lines. Table 2 summarizes the benchmarks for RBAC that include all RBAC classes with their inheritance relationships and perform update operations and these query functions in different ways.

In particular, in the first way below, a field, `transRH`, is used and maintained automatically; it avoids calling `transRH()` repeatedly, as desired in the RBAC standard, and it does so without the extra maintenance code in the RBAC standard for handling updates.

Rules with only non-local predicates. Using rules with only non-local predicates, one can add a field `transRH`, and use `transRH` in place of calls to `transRH()`, e.g., in function `AuthorizedUsers(role)`, and use the following rule set instead of `trans_rs` in class `HierRBAC`:[2]

```
rules transRH_rs:  # no need to use infer explicitly
  transRH(x,y) if RH(x,y)
  transRH(x,y) if RH(x,z), transRH(z,y)
  transRH(x,x) if ROLES(x)
```

Field `transRH` is automatically maintained at updates to `RH` and `ROLES` by implicit calls to `infer`; no explicit calls to `infer` are needed. This eliminates the need of function `transRH()` and repeated expensive calls to it even when its result is not changed most of the time. Overall, this simplifies the program, ensures correctness, and improves efficiency.

[2]The first rule could actually be omitted, because the second argument of `RH` is always in `ROLES` and thus the second rule when joining `RH` with reflexive pairs in `transRH` from the third rule subsumes the first rule.

Part	Analysis	Features used
1 Ext	classes, extension relation	rules (recursive if refined name analysis is used)
2 Stat	statistics, roots	aggregate and set queries
3 Hgt	max height, roots of max height	recursive functions, aggregate and set queries
4 Desc	max desc, roots of max desc	recursive rules, functions, aggregate and set queries

In Parts 1 and 4 that use rules, not using rules (esp. for recursive analysis, with tabling) would be drastically worse (i.e., harder to program and less efficient). In Parts 2-4 that use aggregate and set queries, using rules or recursive functions would be clearly worse. In Parts 3 and 4 that use functions, not using functions (with tabling, also called caching) would be much worse.

Table 3: Benchmark PA for program analysis, integrating different kinds.

Rules with only local predicates. Using rules with only local predicates, `infer` must be called explicitly. One can simply use the function `transRH()` in Figure 1, which calls `infer` using rule set `trans_rs` in the running example and then unions with reflexive role pairs. Alternatively, one can use the rules in `trans_rs` plus a rule that uses a local predicate `role` to infer reflexive role pairs:

```
rules trans_role_rs:  # as trans_rs plus the added last rule
  path(x,y) if edge(x,y)
  path(x,y) if edge(x,z), path(z,y)
  path(x,x) if role(x)
def transRH():        # use infer only, pass in also ROLES
  return infer(path, edge=RH, role=ROLES, rules=trans_role_rs)
```

Both ways show the ease of using rules by simply calling `infer`. Despite possible inefficiency in some cases, using only local predicates has the advantage of full reusability of rules and full control of calls to `infer`.

3.3 Program analysis—rules with aggregate queries and recursive functions

We designed a benchmark for program analysis (PA) problems, especially to show integrated use of rules with aggregate queries and recursive functions. Aggregate queries help quantify and characterize the analysis results, and recursive functions help do these on recursive structures. We describe programs written in Alda and then in XSB.

The benchmark is for analysis of class hierarchy of Python programs. It uses logic rules to extract class names and construct the class extension relation; aggregate queries and set queries to characterize the results and find special cases of interest; recursive functions as well as aggregate and set queries to analyze the special cases; and more logic rules, functions, and set and aggregate queries to further analyze the special cases.

Table 3 summarizes different parts of this benchmark, called PA. A variant, called PAopt, is the same as PA except that, in the recursive rule for defining transitive descendant relationship, the two hypotheses are reversed, following previously studied optimizations [18, 33].

Because the focus is on evaluating the integrated use of different features, each part that uses a single feature, such as rules, is designed to be small. Compared with making each part larger, which exercises individual features more, this design highlights the overhead of connecting different parts, in terms of both ease of use and efficiency of execution.

The benchmark program takes as input the abstract syntax tree (AST) of a Python program (a module or an entire package), represented as a set of facts. Each AST node of type T with k children corresponds to a fact for predicate T with k+1 arguments: id of the node, and ids of the k children. Lists are represented using `Member` facts. A `Member(`*lst*`,`*elem*`,`*i*`)` fact denotes that list *lst* has element *elem* at the *i*th position.

Part 1: Classes and class extension relation. This part examines all `ClassDef` nodes in the AST. A `ClassDef` node has 5 children: class name, list of base-class expressions, and three nodes not used for this analysis. The following rules can be used to find all defined class names and build a class extension relation using base-class expressions that are `Name` nodes. A `Name` node has two children: name and context.

```
rules class_extends_rs:
  defined(c) if ClassDef(_,c,_, _,_,_)
  extending(c,b) if ClassDef(_,c,baselist, _,_,_),
                    Member(baselist,base,_), Name(base,b,_)
```

For a dynamic language like Python, analysis involving names can be refined in many ways to give more precise results, e.g., [9]. We do not do those here, but Datalog rules are particularly good for such analysis of bindings and aliases for names, e.g., [30].

Part 2: Characterizing results and finding special cases. This part computes statistics for defined classes and the class extension relation and finds root classes (class with subclass but not super class). These use aggregate queries and set queries, where `(_,c)` and `(=c,_)` are tuple patterns, `_` matches anything, `c` is bound to the matched value, and `=c` matches the value of `c`.

```
num_defined := count(defined)
num_extending := count(extending)
avg_extending := num_extending/num_defined
roots := {c: (_,c) in extending, not some (=c,_) in extending}
```

Similar queries can compute many other statistics and cases: maximum number of classes that any class extends, leaf classes, histograms, etc.

Part 3: Analysis of special cases. This part computes the maximum height of the extension relation, which is the maximum height of the root classes, and finds root classes of the maximum height. These use a recursive function as well as aggregate and set queries.

```
def height(c):
  return 0 if not some (_,=c) in extending
         else 1 + max{height(d): (d,=c) in extending}
max_height := max{height(r): r in roots}
roots_max_height := {r: r in roots, height(r) = max_height}
```

For efficiency when a subclass can extend multiple base classes, caching of results of function calls is used. In Python, one can simply add `import functools` to import module `functools`, and add `@functools.cache` just above the definition of `height` to cache the results of that function.

Part 4: Further analysis of special cases. This part computes the maximum number of descendant classes following the extension relation from a root class, and finds root classes of the maximum number. Recursive functions and aggregate queries similar to finding maximum height do not suffice here, due to shared subclasses that may be at any depth. Instead, the following rules can infer all `desc(c,r)` facts where class `c` is a descendant following the extension relation from root class `r`, and aggregate and set queries with function `num_desc` then compute the desired results.

```
rules desc_rs:
  desc(c,r) if roots(r), extending(c,r)
  desc(c,r) if desc(b,r), extending(c,b)
def num_desc(r):
  return count{c: (c,=r) in desc}
max_desc := max{num_desc(r): r in roots}
roots_max_desc := {r: r in roots, num_desc(r) = max_desc}
```

For efficiency of the last query, caching is also used for function `num_desc`. If the last query is omitted, function `num_desc` can also be inlined in the `max_desc` query.

Comparing with aggregate queries and functions in rule languages. While rules in Alda correspond directly to rules in rule languages, expressing aggregate queries and functions using rules require translations that formulate computations as hypotheses and introduce additional variables to relate these hypotheses.

Aggregate queries are used extensively in database and machine learning applications, and are essential for analyzing large data or uncertain information. These queries are easy to express directly in database languages and scripting languages, but are less so in rule languages like Prolog; most rule languages also do not support general aggregation with recursion due to their subtle semantics [19]. For example, the simple query `num_defined := count(defined)` in Alda, when written in XSB, becomes:

```
num_defined(N) :- setof(C, defined(C), S), length(S, N).
```

Recursive functions are used extensively in list and tree processing and in solving divide-and-conquer problems. They are natural for computing certain information about parse trees, nested scopes, etc. However, in rule languages, they are expressed in a way that mixes function arguments and return values, and require sophisticated mode analysis to differentiate arguments from returns. For example, the `height` query, when written in XSB, becomes:

```
height(C,0) :- not(extending(_,C)).
height(C,H) :- findall(H1, (extending(D,C), height(D,H1)), L),
               max_list(L,H2), H is H2+1.
```

4 Experimental results

We present results about running times, program sizes, and data sizes. All measurements were taken on a machine with an Intel Xeon X5690 3.47 GHz CPU, 94 GB RAM, running 64-bit Ubuntu 16.04.7, Python 3.9.9, and XSB 4.0.0. For each experiment, the reported running times are CPU times averaged over 10 runs. Garbage collection in Python was disabled for smoother running times when calling XSB. Program sizes are numbers of lines excluding comments and empty lines. Data sizes are number of facts.

Compilation times and program sizes. Table 4 shows Alda compilation times and related XSB, Alda, DistAlgo, and Python program sizes before and after compilation. They are for all benchmarks described in Section 3, plus three variants of TC, explained below in the paragraph on performance of classical queries using rules. The Alda programs are 4–970 lines for OpenRuleBench benchmarks, 385–423 lines for RBAC benchmarks, and 33 lines for PA benchmarks. For each group of benchmarks, there is a single shared Alda file of benchmarking code, shown in the last row of each group.

The compilation times for all programs are 0.6 seconds or less, and for all but RBAC benchmarks and Wine in OpenRuleBench are about 0.1 seconds or less.

For Alda programs that have corresponding XSB programs (OpenRuleBench in Table 1 and PA), Alda programs are all much smaller, almost all by dozens or even hundreds of lines, and by an order of magnitude for Join1 and TC in OpenRuleBench, because we have all the benchmarking code in the shared benchmarking file.

Performance of classical queries using rules. We experimented with four programs for computing transitive closure: TC, the TC benchmark from OpenRuleBench, which is the same as `trans_rs` except with renamed predicates; TCrev, a well-known variant with the two predicates reversed in the recursive rule; and TCpy and TCda, which use the Python and DistAlgo `while` loops, respectively, in Section 2. For comparison, we also directly run the XSB program for TC from OpenRuleBench, and its corresponding

Benchmark name	Original XSB size	Alda size	Compilation time (ms)	Generated Python size	Generated XSB size
Join1	225	23	33.0	32	5
Join2	41	11	18.5	16	9
JoinDup	163	42	45.6	20	36
LUBM	377	125	116.4	29	110
Mondial	36	8	16.2	16	6
DBLP	20	4	16.3	16	2
TC	75	5	7.9	16	3
TCrev	*75	5	7.7	16	3
TCda	–	5	4.9	18	-
TCpy	–	7	6.5	11	-
SG	90	13	17.9	20	7
WordNet	298	58	76.2	44	28
Wine	1103	970	605.6	16	968
ModSG	38	14	14.8	16	12
Win	24	4	9.5	16	2
MagicSet	34	9	18.2	16	7
ORBtimer	–	45	35.2	56	-
RBACnonloc	–	423	346.4	538	4
RBACallloc	–	387	318.4	481	4
RBACunion	–	386	316.6	481	3
RBACda	–	385	312.8	483	-
RBACpy	–	387	314.6	476	-
RBACtimer	–	44	43.3	67	-
PA	*55	33	49.7	93	6
PAopt	*55	33	40.8	93	6
PAtimer	–	40	32.6	56	-

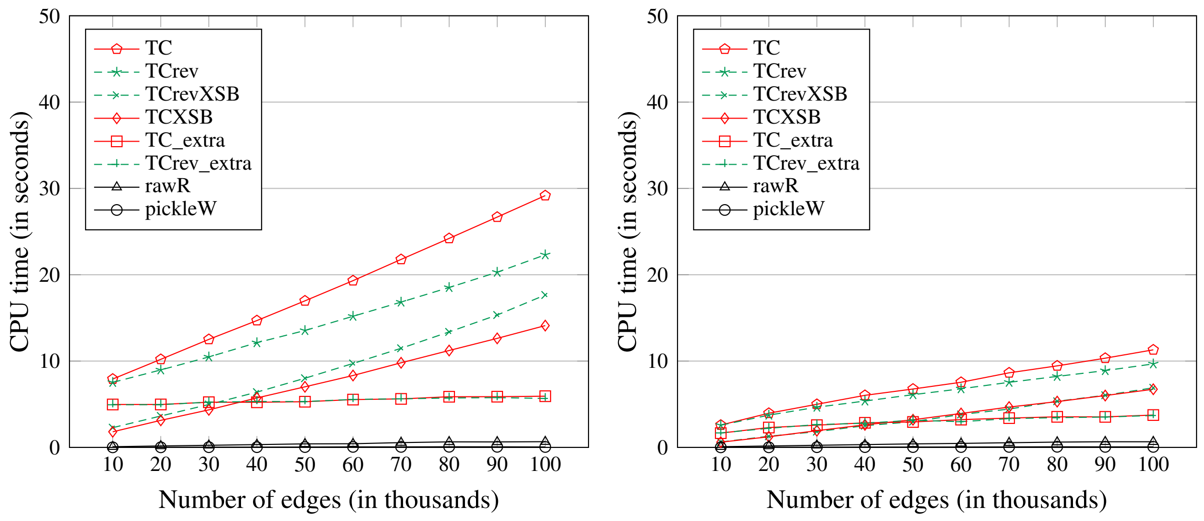
For Original XSB size, entries without * are from OpenRuleBench, as in Table 1; * indicates XSB programs we added; – means there is no corresponding XSB program. For Generated XSB size, - means no XSB code is generated.

Table 4: Compilation times and program sizes before and after compilation.

version for TCrev, except we change `load_dyn` to `load_dync`, for much faster reading of facts in XSB's canonical form; we call these programs TCXSB and TCrevXSB, respectively. The same data generation as OpenRuleBench is used to return large query results—almost complete graphs.

Figure 2 shows the running times of the TC benchmarks. RawR and PickleW are times for reading facts in XSB/Prolog form as used in OpenRuleBench and writing them in pickled form for use in Alda, respectively. Pickling is done only once; the pickled data is read in all repeated runs and all of TC, TCrev, TCda, and TCpy. The running time of Alda programs includes not only (1) reading data, (2) executing queries, and (3) returning results, but also (2pre) preparing data, queries, and commands and writing data to files for XSB before (2), and (2post) reading results from files written by XSB after (2). TC_extra and TCrev_extra are the part of TC and TCrev, respectively, for extra work interfacing with XSB, i.e., for 2pre and 2post and for XSB to read data (xsbRdata) and write results (xsbWres). A breakdown showing the time needed for each step in the extra work is in [22].

The results are as expected: the two Alda programs that use rules are asymptotically, drastically faster than Python and DistAlgo `while` loops, and they exhibit known notable performance differences [33, 34]. Most notably but as expected, passing the query results back from XSB has a high overhead, up to 5.9



TCpy and TCda are not in the charts because they are asymptotically slower and took drastically longer: on 100 vertices, with cyclic data of 200 edges (2% of smallest data point in the charts), TCpy took 624.6 seconds, and TCda took 249.9 seconds; and with acyclic data of 600 edges, TCpy took 160.4 seconds, and TCda took 65.2 seconds.

Figure 2: Running times of TC benchmarks on cyclic (left) and acyclic (right) graphs.

seconds, out of 29.2 seconds total, for graphs of 100K edges, but this overhead is expected to be reduced to 1% of it when the in-memory Python-XSB interface is used. Note that reversing the two predicates in the recursive rule does give a linear-factor asymptotically different running time, but that barely shows because OpenRuleBench uses almost complete graphs.

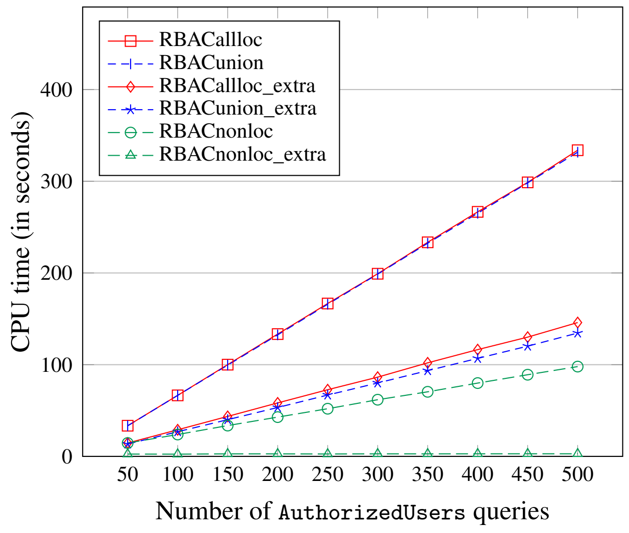
Integrating with objects, updates, and set queries. We use RBAC benchmarks in Table 2, Section 3.2, for this evaluation, especially with frequent queries and updates and intensively frequent restart of XSB for queries randomly mixed with updates of the queried data: 5000 users, 500 roles, 5000 UR relation, 550 RH relation, up to 500 queries, and 230 updates of various kinds.

Figure 3 shows the running times of the RBAC benchmarks, all scaling linearly in the number of queries, as expected. Labels with suffix _extra indicate the part of the running time of the corresponding program for extra work interfacing with XSB. A breakdown of the time for extra work is in [22].

The results are as expected as well: RBACunion and RBACalloc are very close, and are much slower than RBACnonloc—up to 331.7 and 333.9 seconds, respectively, vs. 97.9 seconds. Most notably but as expected, the overhead of repeated queries using XSB is high for RBACunion and RBACalloc, but low for RBACnonloc, up to 134.5 and 145.9 seconds, respectively, vs. 2.8 seconds. The highest overhead is from restarting XSB 500 times, which will be totally eliminated when in-memory interface is used.

Integrating with aggregate queries and recursive functions. We use PA and PAopt benchmarks and their corresponding programs in XSB, as described in Section 3.3, for this evaluation, and we focus on applying the analysis to large programs as input data. The programs analyzed include 9 widely-used open-source Python packages for a wide range of application domains: NumPy, SciPy, MatPlotLib, Pandas, SymPy, Blender, Django, Scikit-learn, and PyTorch—with 641K–5.1M input facts total and 252K–2.2M facts used by the analysis.

Table 5 shows data sizes, analysis results, and running times of the analysis. The columns are sorted by the total number of facts used. A breakdown of the running time into steps interfacing with XSB as well as the remainder of the running time is in [22].



RBACpy	
#queries	time (s)
50	688.7
100	1381.5
150	> 30 min

RBACda	
#queries	time (s)
50	384.6
100	768.1
150	1141.7
200	1517.5
250	> 30 min

RBACpy and RBACda are not in the chart because they are drastically slower.

Figure 3: Running times of RBAC benchmarks, for a workload of updates and queries over 5000 users, 500 roles, 5500 user-role assignments, and 550 role hierarchy pairs.

The results are not as expected: we found the corresponding XSB programs to be highly inefficient, being all slower and even drastically slower than Alda programs, even 120 times slower for PyTorch. Significant effort was spent on performance debugging and manual optimization, and we eventually created a version that is faster than Alda—5.1 seconds vs. 15.2 seconds on the largest input, SymPy—by using additional directives for targeted tabling that also subsumes some indexing.

As expected, the Alda programs here have a high overhead of passing the data to XSB, up to 13.1 seconds on SymPy, which again is expected to be reduced to 1% of it with in-memory Python-XSB interface. This means that the resulting Alda programs would be faster than even the manually optimized XSB, showing that computations not using rules, e.g., aggregations and functions, are not only simpler and easier in Alda/Python than in XSB but also faster.

Scaling with data and rules. We use the two largest benchmarks from OpenRuleBench: DBLP, with over 2.4 million facts, the largest real-world data set among all in OpenRuleBench; and Wine, with 961 rules, the largest rule set among all. Table 6 shows the running times for both benchmarks, for both the Alda programs and the XSB programs. _extra is the part of the total time on 2pre, 2post, xsbRdata, and xsbWres. OrigTotal is the Total time for the original program from OpenRuleBench, which uses `load_dyn` instead of `load_dync`.

The results are again as expected. For DBLP, XSB is more than three times as fast as Alda, 9.5 seconds vs. 30.6 seconds, but as for PA benchmarks, the overhead of passing the large data to XSB is large, here 26.9 seconds, and is expected to be reduced to 1% of that; note that the Alda program has faster reading from pickled data.

For Wine, XSB is more than eight times as fast as Alda, 3.8 seconds vs. 31.0 seconds. This is due to the use of `auto_table` in Alda generated code, which does variant tabling, whereas this program, through manual debugging and optimization, was found to need subsumptive tabling [34]. Optimizations [18, 33, 34] can be added to the Alda compiler to match this efficiency automatically. Note that this slowest Alda program is still faster than half of the systems tested in OpenRuleBench, which took up to 140 seconds and three systems gave errors, and where XSB was the fastest at 4.47 seconds [15].

Measure	Item/Name	numpy	django	sklearn	blender	pandas	matplot	scipy	pytorch	sympy
Data	Total	640,715	815,551	862,031	909,600	942,315	1,064,859	1,092,466	5,142,905	5,115,105
size	`ClassDef`	587	1,835	535	2,146	849	994	898	6,467	1,830
	`Name`	96,076	119,077	137,066	107,638	153,664	152,357	178,754	797,072	1,063,842
	`Member`	155,207	199,416	210,410	242,531	227,766	268,736	260,848	1,270,917	1,112,296
	Total used	251,870	320,328	348,011	352,315	382,279	422,087	440,500	2,074,456	2,177,968
Ratio	Used/Total	39.3%	39.3%	40.4%	38.7%	40.6%	39.6%	40.3%	40.3%	42.6%
Result	`#defined`	519	1610	533	2118	804	935	882	4323	1786
size	`#extending`	419	1457	710	2951	407	610	719	2207	1816
	`#roots`	79	225	51	133	88	104	60	137	92
	`max_height`	8	7	5	4	7	5	3	5	12
	`#roots_max_h`	1	2	2	1	2	1	4	1	1
	`#desc`	427	2329	822	4376	436	605	721	2174	2413
	`max_desc`	84	309	256	1,638	65	47	353	1,045	1,078
	`#roots_max_d`	1	1	1	1	1	1	1	1	1
Running	PA	2.542	3.573	3.263	5.134	3.342	3.733	3.646	14.652	15.243
time	PAopt	2.631	3.520	3.235	4.661	3.341	3.676	3.633	14.706	15.132
(in	PAXSB	6.297	112.091	10.795	243.765	6.378	14.400	22.221	969.228	65.382
seconds)	PAoptXSB	13.170	343.428	17.066	326.629	18.863	40.871	29.675	1773.374	181.961
	PAXSBopt	0.804	1.499	1.071	2.149	1.118	1.317	1.300	5.158	5.051
Ratio	PAopt	103.5%	98.5%	99.1%	90.8%	100.0%	98.5%	99.6%	100.4%	99.3%
over	PAXSB	247.7%	3137.2%	330.8%	4748.0%	190.8%	385.7%	609.5%	6615.0%	428.9%
PA	PAoptXSB	518.1%	9611.7%	523.0%	6362.1%	564.4%	1094.9%	813.9%	12103.3%	1193.7%
	PAXSBopt	31.6%	42.0%	32.8%	41.9%	33.5%	35.3%	35.6%	35.2%	33.1%

Total is the total number of facts about each package. Total used is the sum of numbers of `ClassDef`, `Name`, and `Member` facts.

Table 5: Data size, analysis results, and running times for program analysis benchmarks.

	Alda								XSB	
Name	RawR	PickleW	2pre	xsbRdata	xsbWres	2post	_extra	Total	Total	OrigTotal
DBLP	12.187	3.131	15.722	11.197	0.054	0.020	26.993	30.573	9.492	63.494
Wine	0.008	0.000	0.037	0.219	0.000	0.001	0.257	30.960	3.754	3.826

Table 6: Running times (in seconds) of DBLP and Wine benchmarks.

5 Related work and conclusion

Many benchmarking suites have been developed for evaluating the performance of queries in logic languages and rule engines, including the carefully constructed OpenRuleBench for comprehensively evaluating a diverse set of problems in a wide range of systems [15]. In particular, Prolog has had benchmarks for comparing performance of different implementations [5, 4]. Some are focused on a special class of problems, e.g., interpreters written in Prolog [13]. Some are for evaluating the performance of a particular implementation, e.g., SWI Prolog [31]. There are also works that evaluate queries in more general rule engines and database systems, e.g., the LUBM benchmark [11] and its extensions [24, 29] for OWL on a range of systems, and evaluations including SQL with rules [12, 2]. There are also works focused on evaluating interfaces, e.g., Java Prolog Interface [23]. These works study only queries.

Even drastically more benchmarking suites were developed for evaluating other systems. Works range from systematic studies, e.g., [14, 10, 25], to a large variety of specific benchmarks, e.g., the SPEC benchmarks for computer systems in general [25], the TPC benchmarks for transaction processing [26], the LINPACK benchmarks [6] on solving linear equations, and many more. These benchmarks exercise updates and many other language features but not logic rules.

In contrast, our work develops benchmarks that exercise integrated use of rules together with other language features. We improve OpenRuleBench benchmarks with significantly simplified benchmarking code, and design new benchmarks that exercise tightly integrated uses of different features on problems that test different problem scales in different ways. We also compare different ways of using rules vs. not using rules.

In conclusion, this work presents a set of programming and performance benchmarks for evaluating languages supporting integrated use of rules and other features, and the results of using these benchmarks in an evaluation. Future work can perform evaluations with additional languages and systems, especially efficient ASP systems such as Clingo [8].

Acknowledgments. We thank Bo Lin for his robust DistAlgo compiler implementation, and his initial help in implementing the Alda compiler, and we thank David S. Warren for his initial help in implementing an interface to XSB. We also thank Thang Bui for additional applications in program analysis and optimization, and students in undergraduate and graduate courses for using Alda and its earlier versions, called DA-rules. This work was supported in part by NSF under grants CCF-1954837, CCF-1414078, and IIS-1447549 and ONR under grants N00014-21-1-2719, N00014-20-1-2751, and N00014-15-1-2208.

References

[1] ANSI INCITS (2004): *Role-Based Access Control*. ANSI INCITS 359-2004, American National Standards Institute, International Committee for Information Technology Standards.

[2] Stefan Brass & Mario Wenzel (2019): *Performance Analysis and Comparison of Deductive Systems and SQL Databases*. In: Proceedings of the 3rd International Workshop on the Resurgence of Datalog in Academia and Industry, CEUR-WS.org, pp. 27–38.

[3] Weidong Chen & David S. Warren (1996): *Tabled Evaluation with Delaying for General Logic Programs*. Journal of the ACM 43(1), pp. 20–74, doi:10.1145/227595.227597.

[4] (2001): *The CHINA Benchmark Suite*. `http://www.cs.unipr.it/China/Benchmarks`. Accessed Feb. 1, 2023.

[5] (1985): *Prolog Benchmarking Suites*. `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/code/bench/0.html`. Accessed Feb. 1, 2023.

[6] Jack J Dongarra, Piotr Luszczek & Antoine Petitet (2003): *The LINPACK benchmark: past, present and future*. Concurrency and Computation: practice and experience 15(9), pp. 803–820, doi:10.1002/cpe.728.

[7] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn & Ramaswamy Chandramouli (2001): *Proposed NIST Standard for Role-Based Access Control*. ACM Transactions on Information and Systems Security 4(3), pp. 224–274, doi:10.1145/501978.501980.

[8] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2019): *Multi-shot ASP solving with clingo*. Theory and Practice of Logic Programming 19(1), pp. 27–82, doi:10.1017/S1471068418000054.

[9] Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Tom Rothamel & Tuncay Tekle (2010): *Alias Analysis for Optimization of Dynamic Languages*. In: Proceedings of the 6th Symposium on Dynamic Languages, ACM Press, pp. 27–42, doi:10.1145/1869631.1869635.

[10] Jim Gray, editor (1993): *The Benchmark Handbook for Database and Transaction Processing Systems*, 2nd edition. Morgan Kaufmann Publishers.

[11] Yuanbo Guo, Zhengxiang Pan & Jeff Heflin (2005): *LUBM: A benchmark for OWL knowledge base systems*. Journal of Web Semantics 3(2-3), pp. 158–182, doi:10.1016/j.websem.2005.06.005.

[12] Ulrich John, Petra Hofstedt, Armin Wolf et al. (2019): *A New Benchmark Database and An Analysis of Transitive Closure Runtimes*. Deklarative Ansätze zur Künstlichen Intelligenz–punktuelle Beiträge, p. 3.

[13] Philipp Körner, David Schneider & Michael Leuschel (2020): *On the performance of bytecode interpreters in Prolog*. In: International Workshop on Functional and Constraint Logic Programming, Springer, pp. 41–56, doi:10.1007/978-3-030-75333-7_3.

[14] Byron C. Lewis & Albert E. Crews (1985): *The evolution of benchmarking as a computer performance evaluation technique*. MIS Quarterly, pp. 7–16, doi:10.2307/249270.

[15] Senlin Liang, Paul Fodor, Hui Wan & Michael Kifer (2009): *OpenRuleBench: An Analysis of the Performance of Rule Engines*. In: Proceedings of the 18th International Conference on World Wide Web, ACM Press, pp. 601–610, doi:10.1145/1526709.1526790.

[16] Bo Lin & Yanhong A. Liu (2014 (Latest update January 30, 2022)): *DistAlgo: A Language for Distributed Algorithms*. `http://github.com/DistAlgo`. Accessed May 25, 2023.

[17] Yanhong A. Liu & Scott D. Stoller (2007): *Role-Based Access Control: A Corrected and Simplified Specification*. In: Department of Defense Sponsored Information Security Research: New Methods for Protecting Against Cyber Threats, Wiley, pp. 425–439.

[18] Yanhong A. Liu & Scott D. Stoller (2009): *From Datalog Rules to Efficient Programs with Time and Space Guarantees*. ACM Transactions on Programming Languages and Systems 31(6), pp. 1–38, doi:10.1145/1552309.1552311.

[19] Yanhong A. Liu & Scott D. Stoller (2022): *Recursive Rules with Aggregation: A Simple Unified Semantics*. Journal of Logic and Computation 32(8), pp. 1659–1693, doi:10.1093/logcom/exac072. Also `http://arxiv.org/abs/2007.13053`.

[20] Yanhong A. Liu, Scott D. Stoller & Bo Lin (2017): *From Clarity to Efficiency for Distributed Algorithms*. ACM Transactions on Programming Languages and Systems 39(3), pp. 12:1–12:41, doi:10.1145/2994595. Also `http://arxiv.org/abs/1412.8461`.

[21] Yanhong A. Liu, Scott D. Stoller, Yi Tong & Bo Lin (2023): *Integrating logic rules with everything else, seamlessly*. Theory and Practice of Logic Programming. Special issue for selected papers from ICLP 2023. To appear. Also `http://arXiv.org/abs/2305.19202`.

[22] Yanhong A. Liu, Scott D. Stoller, Yi Tong, Bo Lin & K. Tuncay Tekle (2022): *Programming with Rules and Everything Else, Seamlessly*. Computing Research Repository arXiv:2205.15204 [cs.PL], doi:10.48550/arXiv.2205.15204.

[23] Jose E Zalacain Llanes (2022): *Java Prolog Interface*. arXiv preprint arXiv:2203.17134, doi:10.48550/arXiv.2203.17134.

[24] Li Ma, Yang Yang, Zhaoming Qiu, Guotong Xie, Yue Pan & Shengping Liu (2006): *Towards a complete OWL ontology benchmark*. In: European Semantic Web Conference, Springer, pp. 125–139, doi:10.1007/11762256_12.

[25] Raghunath Nambiar & Meikel Poess, editors (2009): *Performance Evaluation and Benchmarking*. Springer, doi:10.1007/978-3-642-10424-4.

[26] Raghunath Nambiar, Nicholas Wakou, Forrest Carman & Michael Majdalany (2010): *Transaction Processing Performance Council (TPC): State of the council 2010*. In: Technology Conference on Performance Evaluation and Benchmarking, Springer, pp. 1–9, doi:10.1007/978-3-642-18206-8_1.

[27] Python Software Foundation (2023): *Python*. `http://python.org/`. Accessed Jun 7, 2023.

[28] Konstantinos Sagonas, Terrance Swift & David S. Warren (1994): *XSB as an Efficient Deductive Database Engine*. In: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, ACM Press, pp. 442–453, doi:10.1145/191839.191927.

[29] Gunjan Singh, Sumit Bhatia & Raghava Mutharaju (2020): *OWL2Bench: a benchmark for OWL 2 reasoners*. In: International semantic web conference, Springer, pp. 81–96, doi:10.1007/978-3-030-62466-8_6.

[30] Yannis Smaragdakis & George Balatsouras (2015): *Pointer Analysis*. Foundations and Trends in Programming Languages 2(1), pp. 1–69, doi:10.1561/2500000014.

[31] (2022): *SWI-Prolog benchmark suite*. `https://github.com/SWI-Prolog/bench`. Accessed Feb. 1, 2023.

[32] Theresa Swift, David S. Warren, Konstantinos Sagonas, Juliana Freire, Prasad Rao, Baoqiu Cui, Ernie Johnson, Luis de Castro, Rui F. Marques, Diptikalyan Saha, Steve Dawson & Michael Kifer (2022): *The XSB System Version 5.0,x*. `http://xsb.sourceforge.net`. Latest release May 12, 2022.

[33] K. Tuncay Tekle & Yanhong A. Liu (2010): *Precise Complexity Analysis for Efficient Datalog Queries*. In: Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, pp. 35–44, doi:10.1145/1836089.1836094.

[34] K. Tuncay Tekle & Yanhong A. Liu (2011): *More Efficient Datalog Queries: Subsumptive Tabling Beats Magic Sets*. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp. 661–672, doi:10.1145/1989323.1989393.