Alda: Integrating Logic Rules with Everything Else, Seamlessly

Yanhong A. Liu
Computer Science Department, Stony Brook University, Stony Brook, NY
liu@cs.stonybrook.edu

Sets and rules have been used for easier programming since the late 1960s. While sets are central to database programming with SQL and are also supported as built-ins in high-level languages like Python, logic rules have been supported as libraries or in rule-based languages with limited extensions for other features. However, rules are central to deductive database and knowledge base programming, and better support is needed.

This position paper highlights the design of a powerful language, Alda [LST+22], that supports logic rules together with sets, functions, updates, and objects, all as seamlessly integrated built-ins, including concurrent and distributed processes. The key idea is to allow sets of rules to be defined in any scope, support predicates in rules as set-valued variables that can be used and updated directly, and support queries using rules as either explicit or implicit automatic calls to an inference function.

Alda has a formal semantics, is implemented in a prototype compiler that builds on an object-oriented language (DistAlgo [LSL17, Dis21] extending Python [Pyt22]) and an efficient logic rule system (XSB [SW12, SWS+22]), and has been used successfully on benchmarks and problems from a wide variety of application domains—including those in OpenRuleBench [LFWK09], role-based access control (RBAC) [ANS04, FSG+01], and program analysis—with generally good performance.

An example. Figure 1 shows an example program in Alda. It is for a small portion of the ANSI standard for role-based access control (RBAC) [ANS04, FSG+01]. It shows the uses (with line numbers in parentheses) of

- classes (1-8, 9-21) with inheritance (9, 11), and object creation (22) with setup (2-3, 10-12);
- sets, including relations (3, 12);
- methods, including procedures (5-6, 13-14) and functions (7-8, 18-19, 20-21), and calls (23, 24);
- updates, including initialization (3, 12) and membership changes (6, 14); and
- queries, including set queries (8, 19 after union "+", 21) and queries using rules (19 before "+");

where the rules are defined in a rule set (15-17), explained in the next part.

Note that queries using set comprehensions (e.g., on lines 8, 19, 21) can also be expressed by using rules and inference, even though comprehensions are more widely used. However, only some queries using rules and inference can be expressed by using comprehensions; queries using recursive rules (e.g., on lines 16-17) cannot be expressed using comprehensions.

Rules with sets, functions, updates, and objects. In Alda, rules are defined in rule sets, each with a name and optional declarations for the predicates in the rules.

ruleset ::= `rules` *name* `(`*declarations*`):` *rule*+
rule ::= p`(`arg_1`, ..., `arg_a`)` `if` p_1`(`arg_{11}`, ..., `arg_{1a_1}`), ..., `p_k`(`arg_{k1}`, ..., `arg_{ka_k}`)`

In the rule form, p, p_1, ..., p_k denote predicates, $p(arg_1, ..., arg_a)$ denotes that p holds for its tuple of arguments, and if denotes that its left-side conclusion holds if its right-side conditions all hold. In a rule set, predicates not in any conclusion are called base predicates; the other predicates are called derived predicates.

```
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). In rules `trans_rs`, the first rule says there is a path from `x` to `y` if there is an edge from `x` to `y`, and the second rule says there is a path from `x` to `y` if there is an edge from `x` to `z` and there is an edge from `z` to `y`. The call to infer queries and returns the set of pairs for which `path` holds given that `edge` holds for exactly the pairs in set `RH`, by doing inference using rules in `trans_rs`.

The key ideas of seamless integration of rules with sets, functions, updates, and objects are:

1. a predicate is a set-valued variable that holds the set of tuples for which the predicate is true;
2. queries using rules are calls to an inference function, infer, that computes desired values of derived predicates using given values of base predicates;
3. values of base predicates can be updated directly as for other variables, whereas values of derived predicates can only be updated by infer; and
4. predicates and rule sets can be object attributes as well as global and local names, just as variables and functions can.

Declarative semantics of rules are ensured by automatically maintaining values of derived predicates when values of base predicates are updated, by appropriate implicit calls to infer.

For example, in Figure 1, one could use an object field `transRH` in place of calls to `transRH()` in `AuthorizedUsers(role)`, use the following rule set instead of `trans_rs`, and remove `transRH()`.

```
  rules transRH_rs:                # no need to call 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.

Discussion. Note that predicates in rules as set-valued variables, e.g., `edge`, and calling infer to take or return values of set variables, e.g., `RH` in `edge=RH`, avoids the need of high-order predicates or other sophisticated features, e.g., [CKW93], to reuse rules for different predicates in logic languages.

Alda also supports tuple patterns for set elements in set queries (as in DistAlgo [LS09]) and in queries using rules, e.g., `(1,=x,y) in p` matches any triple in set `p` whose first element is 1 and whose second element equals the value of `x`, and binds `y` to the third element if such a triple exists.

Of course, through DistAlgo, Alda also supports distributed programming, e.g., for distributed RBAC [Liu18, LS18], also called trust management [LMW02], in decentralized systems.

Declarations in rules could specify predicate types and scopes, but are designed more importantly for specifying assumptions about predicates being certain, complete, closed, or not [LS20, LS21, LS22]. This is to give respective desired semantics for rules with unrestricted negation and aggregation.

Note that the examples discussed use an ideal syntax, while the Alda implementation supports the Python syntax. For example, `x := {}` is written as `x = set()` in Python syntax.

The Alda implementation compiles rule sets in rules and queries using infer to XSB rules and queries, and compiles the rest to Python, which calls XSB to do the inference. The current implementation supports primarily Datalog rules, but also handles unrestricted negation by using XSB's computation of the well-founded semantics [VRS91]. More general forms of rules and queries can be compiled to rules and queries in XSB or other rule systems using the same approach. In general, any efficient inference algorithm and implementation method can be used to compute the semantics of rules and infer.

Future work includes (1) support for easy use of different desired semantics, especially with modular use of rules, similar to knowledge units in DA-logic [LS21]; and (2) efficient implementation with complexity guarantees [LS09, TL10, TL11] for computing different desired semantics.

Acknowledge. 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.

Thanks to many people for help and discussions, including: Scott Stoller, for complete formal semantics for Alda and experimental results; Yi Tong, for implementation extending DistAlgo implementation and initial experiments; Bo Lin, for initial help with implementation and his robust DistAlgo implementation; David Warren, for an initial 28-line XSB program for interface to XSB; Tuncay Tekle, for additional initial experiments and his significant results on efficient Datalog queries; Thang Bui, for additional applications in program analysis and optimization; and students in undergraduate and graduate courses, for using earlier versions of Alda, called DA-rules.

References

[ANS04] ANSI INCITS. Role-Based Access Control. ANSI INCITS 359-2004, American National Standards Institute, International Committee for Information Technology Standards, Feb. 2004.

[CKW93] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[Dis21] DistAlgo. `distalgo.cs.stonybrook.edu`, 2021. Accessed September 14, 2022.

[FSG+01] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and Systems Security*, 4(3):224–274, 2001.

[LFWK09] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. OpenRuleBench: An analysis of the performance of rule engines. In *Proceedings of the 18th International Conference on World Wide Web*, pages 601–610. ACM Press, 2009.

[Liu18] Yanhong A. Liu. Role-based access control as a programming challenge. In *Proceedings of the Workshop on Logic and Practice of Programming*, Oxford, U.K., 2018.

[LMW02] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[LS09] Yanhong A. Liu and Scott D. Stoller. From Datalog rules to efficient programs with time and space guarantees. *ACM Transactions on Programming Languages and Systems*, 31(6):1–38, 2009.

[LS18] Yanhong A. Liu and Scott D. Stoller. Easier rules and constraints for programming. In *Proceedings of the Workshop on Logic and Practice of Programming*, Oxford, U.K., 2018.

[LS20] Yanhong A. Liu and Scott D. Stoller. Founded semantics and constraint semantics of logic rules. *Journal of Logic and Computation*, 30(8):1609–1638, Dec. 2020. Also `http://arxiv.org/abs/1606.06269`.

[LS21] Yanhong A. Liu and Scott D. Stoller. Knowledge of uncertain worlds: Programming with logical constraints. *Journal of Logic and Computation*, 31(1):193–212, Jan. 2021. Also `https://arxiv.org/abs/1910.10346`.

[LS22] Yanhong A. Liu and Scott D. Stoller. Recursive rules with aggregation: A simple unified semantics. *Journal of Logic and Computation*, 2022. To appear. Also `http://arxiv.org/abs/2007.13053`.

[LSL17] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. From clarity to efficiency for distributed algorithms. *ACM Transactions on Programming Languages and Systems*, 39(3):12:1–12:41, May 2017.

[LST+22] Yanhong A. Liu, Scott D. Stoller, Yi Tong, Bo Lin, and K. Tuncay Tekle. Programming with rules and everything else, seamlessly. *Computing Research Repository*, arXiv:2205.15204 [cs.PL], May 2022. `http://arxiv.org/abs/2205.15204`.

[Pyt22] Python Software Foundation. Python. `http://python.org/`, Accessed September 14, 2022.

[SW12] Terrance Swift and David S Warren. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012.

[SWS+22] 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, and Michael Kifer. *The XSB System Version 5.0,x*, May 2022. `http://xsb.sourceforge.net`. Latest release May 12, 2022.

[TL10] K. Tuncay Tekle and Yanhong A. Liu. Precise complexity analysis for efficient Datalog queries. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 35–44, 2010.

[TL11] K. Tuncay Tekle and Yanhong A. Liu. More efficient Datalog queries: Subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 661–672, 2011.

[VRS91] Allen Van Gelder, Kenneth Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.