Integrating Model Checking and Theorem Proving for Relational Reasoning

Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin Rinard

MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
{arkoudas,khurshid,marinov,rinard}@lcs.mit.edu

Abstract. We present `Prioni`, a tool that integrates model checking and theorem proving for relational reasoning. `Prioni` takes as input formulas written in Alloy, a declarative language based on relations. `Prioni` uses the Alloy Analyzer to check the validity of Alloy formulas for a given scope that bounds the universe of discourse. The Alloy Analyzer can refute a formula if a counterexample exists within the given scope, but cannot prove that the formula holds for all scopes. For proofs, `Prioni` uses Athena, a denotational proof language. `Prioni` translates Alloy formulas into Athena proof obligations and uses the Athena tool for proof discovery and checking.

1 Introduction

`Prioni` is a tool that integrates model checking and theorem proving for relational reasoning. `Prioni` takes as input formulas written in the Alloy language [3]. We chose Alloy because it is an increasingly popular notation for the calculus of relations. Alloy is a first-order, declarative language. It was initially developed for expressing and analyzing high-level designs of software systems. It has been successfully applied to several systems, exposing bugs in Microsoft COM [5] and a naming architecture for dynamic networks [6]. It has also been used for software testing [8], as a basis of an annotation language [7], and for checking code conformance [13]. Alloy is gaining popularity mainly for two reasons: it is based on relations, which makes it easy to write specifications about many systems; and properties of Alloy specifications can be automatically analyzed using the Alloy Analyzer (AA) [4].

`Prioni` leverages AA to model-check Alloy specifications. AA finds *instances* of Alloy specifications, i.e., assignments to relations in a specification that make the specification true. AA requires users to provide only a *scope* that bounds the universe of discourse. AA then automatically translates Alloy specifications into boolean satisfiability formulas and uses off-the-shelf SAT solvers to find satisfying assignments to the formulas. A satisfying assignment to a formula that expresses the negation of a property provides a counterexample that illustrates a violation of the property. AA is restricted to finite refutation: if AA does not find a counterexample within some scope, there is no guarantee that no counterexample exists in a larger scope. Users can increase their confidence by re-running AA for a larger scope, as long as AA completes its checking in a reasonable amount of time.

It is worth noting that a successful exploration of a finite scope may lead to a false sense of security. There is anecdotal evidence of experienced AA users who developed Alloy specifications, checked them for a certain scope, and believed the specifications to hold when in fact they were false. (In particular, this happened to the second author in his earlier work [6].) In some cases, the fallacy is revealed when AA can handle a larger scope, due to advances in hardware, SAT solver technology, or translation of Alloy specifications. In some cases, the fallacy is revealed by a failed attempt to carefully argue the correctness of the specification, even if the goal is not to produce a formal proof of correctness.

`Prioni` integrates AA with a theorem prover that enables the users to prove that their Alloy specifications hold for all scopes. `Prioni` uses Athena for proof representation, discovery, and checking. Athena is a type-ω denotational proof language [1] for polymorphic multi-sorted first-order logic. We chose Athena for several reasons: 1) It uses a natural-deduction style of reasoning that makes it easier to read and write proofs. 2) It offers a high degree of automation through the use of *methods*, which are akin to the tactics and tacticals of HOL [2] and Isabelle [10]. In addition, Athena uses Otter [14] for proof search. Otter is a cutting-edge theorem prover for first-order logic, and its use allows one to skip many tedious steps, focusing instead on the interesting parts of the proof. 3) It offers a strong soundness guarantee. 4) It has a flexible polymorphic sort system with built-in support for structural induction.

`Prioni` provides two key technologies that enable the effective use of Athena to prove Alloy specifications. First, `Prioni` provides an axiomatization of the calculus of relations in Athena and a library of commonly used lemmas for this calculus. Since this calculus is the foundation of Alloy, the axiomatization and the lemmas together eliminate much of the formalization burden that normally confronts users of theorem provers. Second, `Prioni` provides an automatic translation from Alloy to the Athena relational calculus. This translation eliminates the coding effort and transcription errors that complicate the direct manual use of theorem provers. Finally, we note that since Athena has a formal semantics, the translation also gives a precise semantics to Alloy.

`Prioni` supports the following usage scenario. The user starts from an Alloy specification, model-checks it and potentially changes it until it holds for as big a scope as AA can handle. After eliminating the most obvious errors in this manner, the user may proceed to prove the specification. This attempt may introduce new proof obligations, such as an inductive step. The user can then again use AA to model-check these new formulas to be proved. This way, model checking aids proof engineering. But proving can also help model checking. Even when the user cannot prove that the whole specification is correct, the user may be able to prove that a part of it is. This can make the specification smaller, and AA can then check the new specification in a larger scope than the original specification. Machine-verifiable proofs of key properties greatly increase our trust in the reliability of the system. An additional benefit of having *readable* formal proofs lies in improved documentation: such proofs not only show that the desired properties hold, but also *why* they hold.

2 Model Checking

We next illustrate the use of our Prioni prototype on a recursive function that returns the set of all elements in a list. We establish that the result of the function is the same as a simple relational expression that uses transitive closure. The following Alloy specification introduces lists and the function of interest:

```
module List
sig Object {}
sig Node {
  next: option Node,   // next is a partial function from Node to Node
  data: Object }       // data is a total function from Node to Object
det fun elms(n: Node): set Object {
  if (no n.next) then result = n.data
  else result = n.data + elms(n.next) }
assert Equivalence { all n: Node | elms(n) = n.*next.data }
check Equivalence for 5
```

The declaration `module` names the specification. The keyword `sig` introduces a *signature*, i.e., a set of indivisible atoms. Each signature can have *field* declarations that introduce relations. By default, fields are total functions; the modifiers `option` and `set` are used for partial functions and general relations, respectively.

The keyword `fun` introduces an Alloy "*function*", i.e., a parametrized formula that can be invoked elsewhere in the specification. In general, an Alloy function denotes a relation between its arguments and the result; the modifier `det` specifies an actual function. The function `elms` has one argument, `n`. Semantically, all variables in Alloy are relations (i.e., sets). Thus, `n` is not a scalar from the set `Node`; `n` is a singleton subset of `Node`. (A general subset is declared with `set`.) In the function body, `result` refers to the result of the function. The intended meaning of `elms` is to return the set of objects in all nodes reachable from `n`. The operator '`.`' represents relational composition; `n.next` is the set of nodes that the relation `next` maps `n` to. Note that the recursive invocation type-checks even when this set is empty, because the type of `n` is essentially a set of `Node`s.

The keyword `assert` introduces an *assertion*, i.e., a formula to be checked. The prefix operator '`*`' denotes reflexive transitive closure. The expression `n.*next` denotes the set of all nodes reachable from `n`, and `n.*next.data` denotes the set of objects in these nodes. `Equivalence` states that the result of `elms` is exactly the set of all those objects. The command `check` instructs AA to check this for the given *scope*, in this example for all lists with at most five nodes and five objects. AA produces a counterexample, where a list has a cycle. Operationally, `elms` would not terminate if there is a cycle reachable from its argument. In programming language semantics, the least fixed point is taken as the meaning of a recursive function definition. Since Alloy is a declarative, relational language, AA instead considers all functions that satisfy the recursive definition of `elms`.

We can rule out cyclic lists by adding to the above Alloy specification the following: `fact AllAcyclic { all n: Node | n !in n.^next }`. A *fact* is a formula that is assumed to hold, i.e., AA checks if the assertion follows from the conjunction of all facts in the specification. `AllAcyclic` states that there is no node `n` reachable from itself, i.e., no node `n` is in the set `n.^next`; '`^`' denotes transitive closure. We again use AA to check `Equivalence`, and this time AA produces no counterexample.

3 Axiomatization

We next introduce certain key parts of our axiomatization of the calculus of relations in Athena. (Athena's syntax is based on Lisp-like s-expressions; refer to the Appendix for a brief overview of Athena.) The axiomatization represents relations as sets of tuples in a typed first-order finite-set theory. Tuples of binary relations (i.e., ordered pairs) are represented with the following polymorphic Athena structure: `(structure (Pair-Of S T) (pair S T))`. `Prioni` introduces similar structures for tuples of greater lengths as needed.

Sets are polymorphic, their sort being given by a domain constructor: `(domain (Set-Of S))`, and with the membership relation `in` typed as follows: `(declare in ((S) -> (S (Set-Of S)) Boolean))`. Set equality is captured by an extensionality axiom `set-ext`, and set operations are defined as usual. We also introduce a singleton-forming operator:

```
(declare singleton ((T) -> (T) (Set-Of T)))
(define singleton-def
  (forall ?x ?y (iff (in ?x (singleton ?y)) (= ?x ?y))))
```

Relation operations are defined set-theoretically, e.g.:

```
(declare transpose ((T) -> ((Set-Of (Pair-Of T T))) (Set-Of (Pair-Of T T))))
(define transpose-def
  (forall ?R ?x ?y (iff (in (pair ?x ?y) (transpose ?R))
                        (in (pair ?y ?x) ?R))))
```

Alloy has one general composition operator '`.`' that can be applied to two arbitrary relations at least one of which has arity greater than one. Such a general operator could not be typed precisely in a Hindley-Milner-like type system such as that of Athena, and even if it could, it would have a fairly involved definition that would unduly complicate theorem proving. So what our translation does instead is introduce a small number of specialized composition operators `comp-n-m` that compose relations of types $S_1 \times \cdots \times S_n$ and $T_1 \times \cdots \times T_m$, with $S_n = T_1$. Such operators are typed precisely and have straightforward definitions; for instance:

```
(declare comp-2-2 ((S T U) -> ((Set-Of (Pair-Of S T)) (Set-Of (Pair-Of T U)))
                              (Set-Of (Pair-Of S U))))
(forall ?R1 ?R2 ?x ?y
  (iff (in (pair ?x ?y) (comp-2-2 ?R1 ?R2))
       (exists ?z
         (and (in (pair ?x ?z) ?R1)
              (in (pair ?z ?y) ?R2)))))
```

Many Alloy specifications use only `comp-1-2` and `comp-2-2`. In the less common cases, `Prioni` determines the arities at hand and automatically declares and axiomatizes the corresponding composition operators.

Transitive closure is defined in terms of exponentiation. For the latter, we need a minimal theory of natural numbers: their definition as an inductive structure and the primitive recursive definition of addition, in order to be able to prove statements such as $(\forall\, R, n, m)\, R^{n+m} = R^n \,.\, R^m$.

4 Translation

`Prioni` automatically translates any Alloy specification into a corresponding Athena theory. A key aspect of this translation is that it preserves the meaning of the Alloy specification. We next show how `Prioni` translates our example Alloy specification into Athena. Each Alloy signature introduces an Athena domain:

```
(domain Object-Dom)
(domain Node-Dom)
```

Additionally, each Alloy signature or field introduces a constant set of tuples whose elements are drawn from appropriate Athena domains:

```
(declare Object (Set-Of Object-Dom))
(declare Node (Set-Of Node-Dom))
(declare next (Set-Of (Pair-Of Node-Dom Node-Dom)))
(declare data (Set-Of (Pair-Of Node-Dom Object-Dom)))
```

In our example, Alloy field declarations put additional constraints on the relations. The translation adds these constraints into the assumption base (i.e., a set of propositions that are assumed to hold, as explained in the Appendix):

```
(assert (is-fun next))
(assert (is-total-fun Node data))
```

where `is-fun` and `is-total-fun` are defined as expected. Each Alloy "function" introduces an Athena function symbol (which can be actually a relation symbol, i.e., a function to the Athena predefined sort `Boolean`):

```
(declare elms (-> ((Set-Of Node-Dom)) (Set-Of Object-Dom)))
(define elms-def
  (forall ?n ?result
    (iff (= (elms ?n) ?result)
      (and (and (singleton? ?n) (subset ?n Node))
        (and (if (empty? (comp-1-2 ?n next))
                 (= ?result (comp-1-2 ?n data)))
             (if (not (empty? (comp-1-2 ?n next)))
                 (= ?result (union (comp-1-2 ?n data) (elms (comp-1-2 ?n next)))))))))
(assert elms-def)
```

where `empty?-def` is as expected. Alloy facts are simply translated as formulas and added to the assumption base:

```
(define AllAcyclic
  (forall ?n (not (subset (singleton ?n) (comp-1-2 (singleton ?n) (tc next))))))
(assert AllAcyclic)
```

Finally, the assertion is translated into a proof obligation:

```
(define Equivalence
 (forall ?n (= (elms (singleton ?n)) (comp-1-2 (comp-1-2 (singleton ?n) (rtc next)) data))))
```

Recall that all values in Alloy are relations; in particular, Alloy blurs the type distinction between scalars and singletons. In our Athena formalization, however, this distinction is explicitly present and can be onerous for the Alloy user. To alleviate this, `Prioni` allows users to intersperse Athena text with expressions and formulas written in Alloy notation and enclosed within double quotes. (We will follow that practice in the sequel.) Every piece of Alloy notation within Athena is translated into native Athena text by a parser that resolves the typing issue by automatically injecting scalars into singletons.

5 Proof

The assertion `Equivalence` states the correctness of `elms` as an equality between sets. To prove this equality, we show that `elms` is sound: `all n: Node | elms(n) in n.*next.data`, and complete: `all n: Node | n.*next.data in elms(n)`. The desired equality will then follow from set extensionality. This section presents the proof of soundness, which is more interesting than the completeness proof.

The proof uses a few simple lemmas from `Prioni`'s library of results frequently used in relational reasoning:

```
(define comp-mono "all x, y, R | y in x.*R => y.*R in x.*R")
(define first-power-lemma "all R | R in *R")
(define comp-lemma "all s1, s2, R | s1 in s2 => s1.R in s2.R")
(define scalar-lemma "all x, y, R | y in x.R <=> [x y] in R")
(define subset-rtc-lemma "all x, R | x in x.*R")
(define fun-lemma "all x, y, R | [x y] in R && (is-fun R) => y = x.R")
```

and a couple of trivial set-theory lemmas:

```
(define union-lemma "all s1, s2, s | s1 in s && s2 in s => s1+s2 in s")
(define subset-trans "all s1, s2, s3 | s1 in s2 && s2 in s3 => s1 in s3")
```

We also need the following lemma about `next` and `data`:

```
(define lemma-one "all n: Node | n.data in n.*next.data")
```

This immediately follows from `comp-lemma` and `subset-rtc-lemma` using the method `prove` (explained below).

The proof also needs an induction principle for the Alloy lists. Athena supports inductive reasoning for domains that are generated by a set of free constructors. But Alloy structures are represented here as constant sets of tuples, so we must find an alternative way to perform induction on them. In our list example, an appropriate induction principle is:

$$\frac{\begin{array}{c}(\forall\, n)\,(\neg(\exists\, m)\, m \texttt{ in } n.\texttt{next}) \Rightarrow P(n)\\ (\forall\, n)\,(\forall\, m)\,[m \texttt{ in } n.\texttt{next} \Rightarrow P(m) \Rightarrow P(n)]\end{array}}{(\forall\, n)\, P(n)}$$
$$\text{provided } (\forall\, n)\, n \notin n.\texttt{next}^{+}$$

The rule is best read backward: to prove that a property P holds for every node n, we must prove: 1) the left premise, which is the base case: if n does not have a successor, then P must hold for n; and 2) the right premise, which is the inductive step: $P(n)$ must follow from the assumption $P(m)$ whenever m is a successor of n. The proviso $(\forall\, n)\, n \notin n.\texttt{next}^{+}$ rules out cycles, which would render the rule unsound.

Athena makes it possible to introduce arbitrary inference rules via *primitive methods*. Unlike regular methods, whose bodies must be deductions, primitive methods are defined by expressions. (The distinction between expressions and deductions plays a key role in type-ω DPLs [1].) A primitive method is thus free to generate any conclusion it wishes by performing an arbitrary computation on its inputs. Since no guarantees can be made about soundness, primitive methods are part of one's trusted base and must be used sparingly.

We have implemented the above induction rule with a primitive method `list-induction` parameterized over the goal property P. P is implemented as an Athena function `goal` that constructs the desired proposition for a given argument. In this case, we have: `(define (elms-goal n) "elms(n) in n.*next.data")`. The primitive method `list-induction` takes a `goal` as an argument, constructs the two premises from it, checks that they are in the assumption base along with the acyclicity constraint, and if successful, outputs `(forall ?n (goal ?n))`.

The base step is proved automatically:

```
(define base-step
  (!prove "all n | (no m | m in n.next) => goal(n)"
          [elms-def empty-def scalar-lemma lemma-one]))
```

where `prove` is a binary method.[1] (All method calls in Athena are prefixed with '`!`', which distinguishes them from Athena function calls [1].) A method call `(!prove` P $[P_1 \cdots P_n]$`)` attempts to derive the conclusion P from the premises $P_1, \ldots, P_n$, which must be in the current assumption base. If a proof is found, the conclusion P is returned. Currently, Otter is used for the proof search. Where deductive forms such as `assume` (and others explained in the Appendix) are used to guide the deduction, `prove` is used to skip tedious steps. A call `(!prove` P $[P_1 \cdots P_n]$`)` essentially says to Athena: "P follows from $P_1, \ldots, P_n$ by standard logical manipulations: universal specializations, modus ponens, etc. There is nothing interesting or deep here—you work out the details." If we are wrong, either because P does not in fact follow from $P_1, \ldots, P_n$ or because it is a non-trivial consequence of them, the method call will fail within a preset maximum time limit (currently 1 min). Otherwise, a proof will invariably be found almost instantaneously and P will be successfully returned.

The proof of the inductive step is more interesting:

```
(define inductive-step
  (pick-any n m
    (assume-let ((hyp "m in n.next")
                 (ind-hyp (elms-goal m)))
      (dlet ((P1 (!prove "elms(n) = n.data + elms(m)" [elms-def hyp set-ext empty?-def]))
             (P2 (!prove "m.*next in n.*next" [hyp comp-mono first-power-lemma]))
             (P3 (!prove "m.*next.data in n.*next.data" [P2 comp-lemma]))
             (P4 (!prove "elms(m) in n.*next.data" [P3 ind-hyp subset-trans])))
        (!prove "elms(n) in n.*next.data" [P1 lemma-one P4 union-lemma id-axiom]))))
```

The key constructs of the proof (`pick-any`, `assume-let`, and `dlet`) are explained in the Appendix. At this point, both the base and the inductive step have been proved and are in the assumption base, so we can now apply `list-induction` to obtain the desired conclusion: `(!list-induction elms-goal)`. Checking both directions (soundness and completeness) of the correctness proof takes about 1 sec in our current implementation.

6 Conclusions

`Prioni` is a tool that integrates model checking and theorem proving for relational reasoning. Other such tools have focused on reactive systems and modal logics [11, 12] or general first-order logic [9], while `Prioni` focuses on structural system properties. A key issue in the usability of such a tool is the difficulty of finding proofs. We have addressed this issue by lightening the formalization burden through our automatic translation and by providing a lemma library that captures commonly used patterns in relational reasoning. Athena makes it easy to guide the proof, focusing on its interesting parts, while Otter automatically fills in the gaps.

[1] Currently, `prove` is a primitive method and thus Otter is part of our trusted base. However, it is not difficult to implement Otter's inference rules (paramodulation, hyper-resolution, etc.) as Athena methods and then use these methods to define `prove` as a regular method.

References

[1] K. Arkoudas. Type-ω DPLs. MIT AI Memo 2001-27, 2001.

[2] M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a theorem proving environment for higher-order logic.* Cambridge University Press, Cambridge, England, 1993.

[3] D. Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. Available online: http://sdg.lcs.mit.edu/alloy/book.pdf.

[4] D. Jackson, I. Schechter, and I. Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

[5] D. Jackson and K. Sullivan. COM revisited: Tool-assisted modeling of an architectural framework. In *Proc. 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, San Diego, CA, 2000.

[6] S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, Sep 2000.

[7] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *Proc. ACM SIGPLAN 2002 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Seattle, WA, Nov 2002.

[8] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, Nov. 2001.

[9] W. McCune. Mace: Models and counter-examples. `http://www-unix.mcs.anl.gov/AR/mace/`, 2001.

[10] L. Paulson. *Isabelle, A Generic Theorem Prover.* Lecture Notes in Computer Science. Springer-Verlag, 1994.

[11] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, Liege, Belgium, 1995. Springer Verlag.

[12] N. Shankar. Combining theorem proving and model checking through symbolic analysis. *Lecture Notes in Computer Science*, 1877, 2000.

[13] M. Vaziri and D. Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Proc. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, Apr. 2003.

[14] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning, Introduction and Applications.* McGraw-Hill, Inc., 1992.

A Brief Athena Overview

Athena is a type-ω denotational proof language [1] for polymorphic multi-sorted first-order logic. This Appendix presents parts of Athena relevant to the understanding of the example. In Athena, an arbitrary universe of discourse (sort) is introduced with a `domain` declaration, for example:

```
(domain Real)
(domain Person)
```

Function symbols and constants can then be declared on the given domains, e.g.:

```
(declare + (-> (Real Real) Real))
(declare joe Person)
(declare pi Real)
```

Relations are functions whose range is the predefined sort `Boolean`, e.g.,

```
(declare less? (-> (Real Real) Boolean))
```

Domains can be polymorphic, e.g.,

```
(domain (Set-Of T))
```

and then function symbols declared on such domains can also be polymorphic:

```
(declare insert ((T) -> (T (Set-Of T)) (Set-Of T)))
```

Note that in the declaration of a polymorphic symbol, the relevant sort parameters have to be listed within parentheses immediately before the arrow `->`. The equality symbol `=` is a predefined relation symbol with sort `((T) -> (T T) Boolean)`.

Inductively generated domains are introduced as "structures", e.g.,

```
(structure Nat
  zero
  (succ Nat))
```

Here `Nat` is freely generated by the *constructors* `zero` and `succ`. This is equivalent to issuing the declarations `(domain Nat)`, `(declare zero Nat)`, `(declare succ (-> (Nat) Nat))`, and additionally postulating a number of axioms stating that `Nat` is freely generated by `zero` and `succ`. Those axioms along with an appropriate induction principle are automatically generated when the user defines the structure. Structures can also be polymorphic, e.g.,

```
(structure (List-Of T)
  nil
  (cons T (List-Of T)))
```

The fundamental data values in Athena are terms and propositions. Terms are s-expressions built from declared function symbols such as `+` and `pi`, and from *variables*, written as `?`I for any identifier I. Thus `?x`, `(+ ?foo pi)`, `(+ (+ ?x ?y) ?z)`, are all terms. The (most general) sort of a term is inferred automatically; the user does not have to annotate variables with their sorts. A proposition P is either a term of sort Boolean (say, `(less? pi (+ pi pi))`); or an expression of the form `(not` P`)` or `(`$\odot$ P_1 P_2`)` for $\odot \in \{\texttt{and}, \texttt{or}, \texttt{if}, \texttt{iff}\}$; or `(`$Q$ $x_1 \cdots x_n$ P`)` where $Q \in \{\texttt{forall}, \texttt{exists}\}$ and each x_i a variable. Athena also checks the sorts of propositions automatically using a Hindley-Milner-like type inference algorithm.

The user interacts with Athena via a read-eval-print loop. Athena displays a prompt `>`, the user enters some input (either a phrase to be evaluated or a top-level directive such as `define`, `assert`, `declare`, etc.), Athena processes the user's input, displays the result, and the loop starts anew. The most fundamental concept in Athena is the *assumption base*—a finite set of propositions that are assumed to hold, representing our "axiom set" or "knowledge base". Athena starts out with the empty assumption base, which then gets incrementally augmented with the conclusions of the deductions that the user successfully evaluates at the top level of the read-eval-print loop. A proposition can also be explicitly added into the global assumption base with the top-level directive `assert`. (Note that in Athena the keyword `assert` introduces a formula that is supposed to hold, whereas in Alloy `assert` introduces a formula that is to be checked.)

`Prioni` starts by adding relational calculus axioms and already proved lemmas to the empty assumption base. It then translates the Alloy specification and adds to the assumption base all translated constraints and definitions. Only the translated Alloy assertion is not added to the assumption base; rather, it constitutes the proof obligation.

An Athena deduction D is always evaluated in a given assumption base β. Evaluating D in β will either produce a proposition P (the "conclusion" of D in β), or else it will generate an error or will diverge. If D does produce a conclusion P, Athena's semantics guarantee $\beta \models P$, i.e., that P is a logical consequence of β. There are several syntactic forms that can be used for deductions.

The form `pick-any` introduces universal generalizations: `(pick-any` $I_1 \cdots I_n$ D`)` binds the names $I_1 \cdots I_n$ to fresh variables $v_1, \ldots, v_n$ and evaluates D. If D yields a conclusion P, the result returned by the entire `pick-any` is $(\forall\, v_1, \ldots, v_n)\, P$.

The form `assume` introduces conditionals: to evaluate `(assume` P D`)` in an assumption base β, we evaluate D in $\beta \cup \{P\}$. If that produces a conclusion Q, the conditional $P \Rightarrow Q$ is returned as the result of the entire `assume`. The form `(assume-let ((`I P`))` D`)` works like `assume`, but also lexically binds the name I to the hypothesis P within D.

The form `(dlet ((`I_1 D_1`)` $\cdots$ `(`I_n D_n`))` D`)` is used for sequencing and naming deductions. To evaluate such a deduction in β, we first evaluate D_1 in β to obtain a conclusion P_1. We then bind I_1 to P_1, insert P_1 into β, and continue with D_2. The conclusions P_i of the various D_i are thus incrementally added to the assumption base, becoming available as lemmas for subsequent use. The body D is then evaluated in $\beta \cup \{P_1, \ldots, P_n\}$, and its conclusion becomes the conclusion of the entire `dlet`.