D Automated Theorem Proving

In the preface we pointed out that for pedagogical reasons the proofs in this book would not use external automated theorem provers (ATPs) as inference black boxes. We did discuss and use SAT solvers, but mostly as tools for solving hard combinatorial problems, not for proofs. Indeed, for sentential logic we developed our own theorem prover, `prop-taut`, which was subsequently used inside `chain` applications to handle tedious steps. We believe that the material presented in this book demonstrates that in a proper natural deduction setting it is possible to write fairly high-level proofs without using any external theorem provers as oracles, simply by starting with primitive introduction and elimination constructs and then using trusted abstraction mechanisms to introduce increasingly powerful and higher-level reasoning methods. And in fact we have only scratched the surface of what is possible here. Many more sophisticated methods can be implemented, from congruence closure to general first-order reasoning, that could lift the level of proof detail even higher while remaining within Athena's semantics.

That said, theorem proving technology has made remarkable progress over the last 10–15 years, and it is unlikely that hand-written methods will attain the level of automation and efficiency achieved by state-of-the-art ATP systems. In large verification projects, having access to that kind of automation can be a considerable boon to productivity. For that reason, Athena is integrated with cutting-edge ATPs that can be seamlessly invoked as if they were primitive methods or procedures.

In this chapter we briefly survey some of the mechanisms available in Athena for automated reasoning through external ATPs, describing their interface and illustrating their use with examples. We specifically discuss: (a) automated theorem proving for general (polymorphic many-sorted) first-order logic, and (b) SMT (satisfiability-modulo-theories) solving. In the SMT section we also illustrate solving instances of the weighted Max-SMT problem, the SMT analogue of the weighted Max-SAT problem that allows one to attach numeric weights to clauses and search for models that maximize the total weight. We also discuss the use of SMT solving in conjunction with binary search for the purpose of solving optimization problems over linear cost functions with very rich logical structure. Such problems are typically easier to express in a sorted logical framework such as Athena than in ILP (integer linear programming). Several examples of using SMT solving in Athena (particularly for solving such highly constrained optimization problems) are available in the literature; some references are given at the end of the chapter.

D.1 General automated theorem proving

ATPs can be used in Athena in two ways, either via a primitive method, `derive-from`, or via the following native syntax form:

$$p \ \texttt{from} \ J_1, \ldots, J_n \tag{1}$$

where each J_i is either a sentence or a list of sentences. We discuss these in turn.

The following is a very simple example of the use of form (1):

```
declare A, B, C, D, E: Boolean

> assume hyp := (A & B) {
    B from hyp;
    A from hyp;
    (B & A) from A, B
  }

Theorem: (if (and A B)
             (and B A))
```

As each J_i can be either a sentence or a list of sentences, one is able to write, e.g.,

$$p \ \texttt{from} \ \texttt{hyp}, \texttt{reflect-def}, \texttt{axiom-1}, \texttt{lemma-2}$$

where `hyp`, `axiom-1`, and `lemma-2` are individual sentences and `reflect-def` is a list of sentences (presumably defining a function symbol `reflect`; see the example below). Note that (1) may only appear inside a proof block, as in the body of the **assume** shown above. If it is to serve as a stand-alone deduction, it must be enclosed within curly braces.

The semantics of (1) are simple: an external ATP is invoked and given the task of deriving the goal sentence p from the listed premises (the sentences enumerated to the right of **from**). All of these sentences must be in the assumption base at the time when this derivation is attempted. If the ATP succeeds (within a default amount of time, typically 100 seconds), then the goal sentence is successfully returned as a theorem; otherwise an error occurs. The **from** construct can be understood as syntax sugar for an application of the ternary method `derive-from`, which provides a more flexible interface to ATPs, and to which we now turn.

A call to `derive-from` is of the following form:

`(!derive-from` p L *options*`)`.

The idea here is that L is a list of premises (sentences in the assumption base) and we want to derive p from them. The third argument, *options*, is a map that specifies values for certain parameters that will be described shortly. If you do not want to bother with specifying options, use the method `prove` instead, like this:

(!prove p L).

This is essentially a call to derive-from with some default option values, i.e., (!prove p L) is really a shorthand for

(!derive-from p L *default-option-map*).

Some simple examples of using prove are given below, first from sentential and then from full first-order logic:

```
> assume h1 := (A & B)
    assume h2 := (~ C ==> ~ B)
      (!prove C [h1 h2])

Theorem: (if (and A B)
             (if (if (not C)
                         (not B))
                     C))
```

The resulting theorem is a tautology, so it could also be derived from the empty list of premises:

```
clear-assumption-base

define [hyp-1 hyp-2] := [(A & B) (~ C ==> ~ B)]

> conclude theorem-1 := (hyp-1 & hyp-2 ==> C)
    (!prove theorem-1 [])

Theorem: (if (and (and A B)
                  (if (not C)
                      (not B)))
             C)
```

If the list L given as the second argument to prove is nonempty, then every sentence in L must be in the assumption base when the method is applied, otherwise an error is reported. An error will also occur if the first argument is not a sentence, or if the second argument is not a list L. However, each individual element of L may be either a sentence or it can itself be a list of sentences, i.e., L is of the form [$V_1 \cdots V_n$] where each V_i is either a sentence or a list of sentences. If we write $\phi(V_i)$ for the set of sentences represented by V_i,[1] then the ultimate set of sentences that will serve as the premises from which the goal is to be derived can be written as:

$$\bigcup_{i=1}^{n} \phi(V_i).$$

1 So that if V_i is a single sentence p, $\phi(V_i)$ is the singleton $\{p\}$; and if V_i is a list of sentences [$p_1 \cdots p_k$], then $\phi(V_i)$ is the set $\{p_1,\ldots,p_k\}$.

For example:

```
datatype (BTree S) := null | (node S (BTree S) (BTree S))

define [t t' t1 t2] := [?t ?t' ?t1 ?t2]

declare reflect: (S) [(BTree S)] -> (BTree S)

assert* reflect-def :=
   [(reflect null = null)
    (reflect (node x t1 t2) = (node x (reflect t2) (reflect t1)))]

define (leaf x) := (node x null null)  # A shorthand for making leaves

> (eval reflect (node 1 (leaf 2) (leaf 3)))

Term: (node 1
            (node 3
                  null:(BTree Int)
                  null:(BTree Int))
            (node 2
                  null:(BTree Int)
                  null:(BTree Int)))

define r := reflect # A shorthand for reflect

conclude reflect-twice := (forall t . r r t = t)
  by-induction reflect-twice {
    (t as null) => (!prove (r r t = t) reflect-def)
  | (t as (node _ t1 t2)) =>
     let {[ih1 ih2] := [(r r t1 = t1) (r r t2 = t2)]}
       (!prove (r r t = t) [ih1 ih2 reflect-def])
  }
```

Here, in the second call to prove, the list argument contained as elements both the individual sentences ih1 and ih2 as well as reflect-def, the list of sentences comprising the definition of reflect.

If we wish, we may pass the entire assumption base as the second argument to prove. For example, the above proof could also be written as:

```
conclude reflect-twice := (forall t . r r t = t)
  by-induction reflect-twice {
    (t as null) => (!prove (r r t = t) (ab))
  | (t as (node _ _ _)) => (!prove (r r t = t) (ab))
  }
```

We will have more to say on automating inductive proofs in section D.1.3.

Keep in mind, however, that the more premises we give to the ATP, the harder the problem becomes, as the underlying search space becomes increasingly large and intractable.

In fact, even with only 15–20 sentences in the assumption base, some of the best ATPs are unable, when used with their default settings, to derive the desired conclusions shown above from the entire assumption base within a reasonable time period (less than a couple of minutes). In other words, if the assumption base is even moderately large, the proof as given above will fail in the default amount of time allotted to an ATP call.[2] In fact, oftentimes, especially when equational reasoning is involved, `chain` might be able to succeed at automation better than ATPs, even when it needs to contend with the entire assumption base. For instance, even with many sentences in the assumption base, the automatic `chain` calls below succeed quickly:

```
conclude reflect-twice := (forall t . reflect reflect t = t)
  by-induction reflect-twice {
    (t as null) => (!chain [(reflect reflect t) = t])
  | (t as (node _ _ _)) => (!chain [(reflect reflect t) = t])
  }
```

(Recall that when no justification items are specified for a given `chain` step, the entire assumption base is used by default.) Further, when an automated equational chain succeeds, a derivation is easy to obtain with `find-eqn-proof` (see footnote 21). In this case, for example, if we inserted a call to `find-eqn-proof` before the inductive `chain` application, say, like this:

```
  | (t as (node _ _ _)) => let {[L R] := [(r r t) t]
                                _ := (print (find-eqn-proof L R (ab)))}
                             (!chain [L = R])
```

then we would essentially get as output the following detailed chain, along with elements of the assumption base justifying each step, where `?v1`, `?v2`, and `?v3` are fresh variables corresponding to the three wildcards in the pattern `(t as (node _ _ _))`:

```
L = (reflect reflect (node ?v1 ?v2 ?v3))

  = (reflect (node ?v1 (reflect ?v3)
                       (reflect ?v2)))  # from reflect-def

  = (node ?v1 (reflect (reflect ?v2))
              (reflect (reflect ?v3)))  # from reflect-def

  = (node ?v1 ?v2
              (reflect (reflect ?v3))   # from the left i.h.

  = (node ?v1 ?v2 ?v3)                  # from the right i.h

  = R
```

2 It might succeed if we are willing to wait longer.

Returning to external ATPs and the issue of large sets of premises, we note that the problem of making ATPs ignore potentially useless or redundant information is an active research area, especially in connection with reasoning in very large knowledge bases. One approach is to use various syntactic or semantic criteria to eliminate from consideration those premises which do not appear to be sufficiently *related* to the goal at hand. One such simple technique is the SINE algorithm of Hoder and Voronkov [49]. SINE is implemented in Athena. To use SINE-selected premises instead of the entire assumption base, first **load** `"sine"` and then instead of `(!derive-from` *p* `(ab)` *options*`)` use

`(!derive-from` *p* `(SINE.all-relevant-sentences` *p*`)` *options*`)`.

More recent—and more sophisticated—approaches include machine-learning-based algorithms for inducing premise selection criteria from large proof corpora; see, e.g., *Premise Selection for Mathematics by Corpus Analysis and Kernel Methods* [2].

What happens under the hood during a call to `derive-from` or `prove`? Athena negates the goal and translates that negation along with the given premises into vanilla first-order logic (see more on that step in section D.1.1), then those sentences are converted into CNF (conjunctive normal form), and finally the ATP specified in the *options* map is invoked on the resulting set of sentences (we will describe that map shortly). The translations are fast, so the bottleneck is typically the proof search—unsurprisingly, given that that is the really hard problem.

Athena is integrated with two main ATPs (although any TPTP-compliant ATP can be easily added): Spass and Vampire. Both systems can be downloaded from their websites.[3] The default ATP is Spass. The user can specify which ATP to use in the *options* argument to `derive-from`, as the value of the `'atp` key. Another useful parameter whose value can be specified in *options* is `'max-time`, which specifies the maximum number of seconds that Athena will wait for the ATP to find a proof. The default is 100 seconds. Thus, for instance, the following calls Vampire with a maximum wait time of 20 seconds:

```
define goal := (forall x y . x = y <==> y = x)

define options := |{'atp := 'vampire , 'max-time := 20}|

> (!derive-from goal [] options)

Theorem: (forall ?x:'S
           (forall ?y:'S
             (iff (= ?x:'S ?y:'S)
                  (= ?y:'S ?x:'S))))
```

3 After downloading them, the Vampire and Spass executables must be placed in the same directory as the Athena executable and must be named `vampire.exe` and `spass.exe` in Windows environments or `vampire` and `spass` in non-Windows environments.

There are a few other parameters that can be set inside *options*, but they need not be described here. The infix **from** construct (1) uses these default option values.

There are no restrictions on what sentential connectives or quantifiers may appear in the goal or premises supplied to the primitive ATP methods, and there is no restriction on the structure of these sentences. Free variables may appear in any of them, and even sentences that are not legal first-order sentences in the conventional (unsorted) sense may be used as inputs; Athena will automatically translate them into an equivalent form. For instance, the following is a perfectly legal sentence in Athena's sorted language, but would not be so in conventional first-order logic:

```
(= (= ?x ?y) (= ?y ?x)).
```

Nevertheless, it is correctly handled by the translation process:

```
> (!prove ((?x = ?y) = (?y = ?x)) [])

Theorem: (= (= ?x:'S ?y:'S)
            (= ?y:'S ?x:'S))
```

We close this section with some less trivial illustrations of the abilities of ATPs. For sentential reasoning, consider the n*th* De Bruijn sentence, constructed by `(make-db` *n*`)` (see Exercise 4.36):

```
define db-3 := (make-db 3)

> (!prove db-3 [])

Theorem: (if (and (if (iff ?A1:Boolean ?A2:Boolean)
                      (and ?A1:Boolean
                           (and ?A2:Boolean ?A3:Boolean)))
                  (and (if (iff ?A2:Boolean ?A3:Boolean)
                           (and ?A1:Boolean
                                (and ?A2:Boolean ?A3:Boolean)))
                       (if (iff ?A3:Boolean ?A1:Boolean)
                           (and ?A1:Boolean
                                (and ?A2:Boolean ?A3:Boolean)))))
             (and ?A1:Boolean
                  (and ?A2:Boolean ?A3:Boolean)))

> (!prove (make-db 4) [])

... Error: ...
Unable to derive the conclusion ...
from the given hypotheses.

> (size (!prove (make-db 99) []))

Term: 20195
```

Before moving on to full first-order logic examples, we note that when Vampire is used, it is possible to obtain as output those premises that *were actually used* in the discovered proof. This can be useful in a variety of situations. For instance, we can check whether the assumption base is inconsistent by attempting to derive `false` from it:

```
define (inconsistent-ab?) :=
  match (!prove false (ab)) {
    false => true
  | _ => false
  }

> (inconsistent-ab?)

··· Error: ···
Unable to derive the conclusion false from the given hypotheses.
```

When the prover is unable to show an inconsistency, as above, all is good and fine. (Though if the assumption base is large enough we might want to increase the time limit significantly.) However, if the prover does succeed in deriving `false` from the assumption base, thereby establishing that the latter is inconsistent, all we would know is that *some* subset of the assumption base is inconsistent. Typically, however, we would like to know precisely *which* subset is inconsistent (i.e., which sentences exactly are implicated in the derivation of a contradiction). This is just one example where an output list of the sentences that were actually used in the discovered derivation is useful. To obtain such output, pass a cell as the value of the `'used-premises` key in the *options* map. If a proof is found, then the said cell will afterward contain the premises used in that proof. To take a very simple example:

```
define prems := (cell [])

assert  (A & B)
retract (B & A)

> (length (ab))

Term: 18

> (!derive (B & A) (ab) |{'atp := 'vampire,
                          'max := 60,
                          'used-premises := prems}|)

Theorem: (and B A)

> (ref prems)

List: [(and A B)]
```

We close with a couple of first-order logic examples taken from Pelletier's collection of "seventy-five problems for testing automatic theorem provers" [81]. We first introduce the following domain and symbols:

```
domain Object

declare P, Q: [Object] -> Boolean

declare a: Object

declare f: [Object Object] -> Object
```

We begin with *Andrew's challenge*, which is solved instantaneously:

```
define left := ((exists x . forall y . P x <==> P y) <==>
                ((exists x . Q x) <==> (forall y . P y)))

define right := ((exists x . forall y . Q x <==> Q y) <==>
                 (exists x . P x) <==> (forall y . Q y))

> (!prove (left <==> right) [])

Theorem: (iff (iff (exists ?x:Object
                     (forall ?y:Object
                       (iff (P ?x:Object)
                            (P ?y:Object))))
                   (iff (exists ?x:Object
                          (Q ?x:Object))
                        (forall ?y:Object
                          (P ?y:Object))))
              (iff (exists ?x:Object
                     (forall ?y:Object
                       (iff (Q ?x:Object)
                            (Q ?y:Object))))
                   (iff (exists ?x:Object
                          (P ?x:Object))
                        (forall ?y:Object
                          (Q ?y:Object)))))
```

The second problem is taken from group theory (problem 65 in the Pelletier set). The primitive Athena procedure `associative` takes any binary symbol *f* and produces a sentence to the effect that *f* is associative:

```
assert p1 := (associative f)

assert p2 := (forall x . a f x = x)

define goal := ((forall x . x f x = a) ==> forall x y . x f y = y f x)

> (!prove goal [p1 p2])
```

```
Theorem: (if (forall ?x:Object
               (= (f ?x:Object ?x:Object)
                  a))
             (forall ?x:Object
               (forall ?y:Object
                 (= (f ?x:Object ?y:Object)
                    (f ?y:Object ?x:Object)))))
```

D.1.1 Many-sortedness and polymorphism

The most competitive extant ATPs are based on unsorted first-order logic.[4] Therefore, a translation is necessary from the polymorphic many-sorted logic of Athena to classical unsorted first-order logic. The translation need not preserve logical equivalence, but it must preserve satisfiability.

Let us discuss simple (ground) sorts before we turn to polymorphism. Without a sort-respecting translation, unsound results would be very easy to obtain. Consider, for instance, the following perfectly innocuous assertions:

```
datatype Color := red | green | blue

assert p1 := (red =/= green & green =/= blue & red =/= blue)


> assert p2 := (forall x . x = true | x = false)

The sentence
(forall ?x:Boolean
  (or (= ?x:Boolean true)
      (= ?x:Boolean false)))
has been added to the assumption base.
```

If we disregarded sorts, then at this point we would already have an inconsistent assumption base, because the sort-less translations of p1 and p2 are as follows:

$$\mathit{red} \neq \mathit{green} \wedge \mathit{green} \neq \mathit{blue} \wedge \mathit{red} \neq \mathit{blue}$$

and

$$(\forall x \,.\, x = \mathit{true} \vee x = \mathit{false}),$$

which are mutually inconsistent, as the first implies that there are at least three things in the universe, whereas the second implies that there are no more than two things.

A proper translation of p2 would be something along the lines of:

4 Spass is an exception, as it can accept many-sorted first-order logic formulas as input.

$$(\forall x \,.\, Boolean(x) \Rightarrow x = true \vee x = false),$$

which can be understood as saying that everything *that is Boolean* is either *true* or *false*. That approach, first introduced by Wang in 1952 [110], requires the introduction of a new unary predicate symbol S for each monomorphic sort S. Then, writing $\mathbf{T}[p]$ for the single-sorted translation of a many-sorted sentence p, we have

$$\mathbf{T}[(\forall\, x : S \,.\, p)] = (\forall x \,.\, S(x) \Rightarrow \mathbf{T}[p])$$

and

$$\mathbf{T}[(\exists\, x : S \,.\, p)] = (\exists x \,.\, S(x) \wedge \mathbf{T}[p]).$$

A similar but somewhat more complicated technique could accommodate polymorphism. Essentially, we could introduce a binary predicate `has-sort` relating Athena terms to their possibly polymorphic sorts. Constraints built with `has-sort` could then be generated during the translation process as needed to capture the sort information attached to Athena terms. For instance, a constraint like

```
(has-sort x (Pair S (List S)))
```
(2)

would represent the polymorphic sort information associated with a variable such as `?x:(Pair 'S (List 'S))`. (We would also need to introduce new unsorted function symbols for every sort constructor, e.g., `Pair` would become a new binary function symbol in our unsorted vocabulary.) The variable `S` would be universally quantified in any sentence containing (2). Indeed, all sort variables $S_1, \ldots, S_n$ corresponding to sort parameters appearing in an Athena sentence would be universally quantified upfront in the translated result. Sort constraints like (2) would be required for every polymorphic variable and every polymorphic constant occurring in an Athena sentence. These constraints would be conjoined to existential quantifications over a polymorphic sort, and would be conditional for universal quantifications.

However, this approach would have the drawback of significantly altering the boolean structure of the original sentence(s), which could have a negative impact on ATP performance. An alternative approach is to leave the boolean structure intact and instead (recursively) tag every term occurrence

$$t = (f \;\cdots)$$

in the sentence being translated with its sort information, so that every previous term occurrence of t now becomes an occurrence like `(sorted-term` t s`)` where s is a (single-sorted) term representing the sort of t, and `sorted-term` is a new and unique binary function symbol. This is similar to the technique proposed by Couchot and Lescuyer [22] and is the approach taken in Athena. The basic details, along with soundness proofs, can be found in that paper. The presence of polymorphic subsorting requires some additional machinery but since we do not discuss subsorting here we need not delve into those details.

D.1.2 ATP-powered chaining

We can combine the structure and clarity of the tabular notation of `chain` with the power of ATPs by turning on a settings flag, `atps-with-chain`. This flag is off by default. Accordingly, for instance, if we tried to solve the above group theory problem (Pelletier problem 65) by chaining, we would fail:

```
assert p1 := (associative f)
assert p2 := (forall x . a f x = x)

> assume hyp := (forall x . x f x = a)
    pick-any x y
      (!chain [(x f y) = (y f x)  [p1 p2 hyp]])

standard input:3:7: Error: Equational chaining error
on the 1st step of the chain, in going from: ···
```

Likewise for implication and/or equivalence chaining. For instance:

```
define left := ((exists x . forall y . P x <==> P y) <==>
                ((exists x . Q x) <==> (forall y . P y)))

define right := ((exists x . forall y . Q x <==> Q y) <==>
                 (exists x . P x) <==> (forall y . Q y))

> (!chain [left ==> right  []])

standard input:1:2: Error: Implicational chaining error
on the 1st step of the chain, in going from: ···
```

However, if we turn the flag on, then each chaining step will be handed off to ATPs if the regular chaining algorithm fails:

```
> set-flag atps-with-chain "on"

OK.

> assume hyp := (forall x . x f x = a)
    pick-any x y
      (!chain [(x f y) = (y f x)  [p1 p2 hyp]])

Theorem: (if (forall ?x:Object
               (= (f ?x:Object ?x:Object)
                  a))
             (forall ?x:Object
               (forall ?y:Object
                 (= (f ?x:Object ?y:Object)
                    (f ?y:Object ?x:Object)))))

> (!chain [left ==> right  []])
```

```
Theorem: (if (iff (exists ?x:Object
                    (forall ?y:Object
                      (iff (P ?x:Object)
                           (P ?y:Object))))
                  (iff (exists ?x:Object
                         (Q ?x:Object))
                       (forall ?y:Object
                         (P ?y:Object))))
             (iff (exists ?x:Object
                    (forall ?y:Object
                      (iff (Q ?x:Object)
                           (Q ?y:Object))))
                  (iff (exists ?x:Object
                         (P ?x:Object))
                       (forall ?y:Object
                         (Q ?y:Object)))))

> (!chain [left <==> right  []])

Theorem: (iff (iff (exists ?x:Object
                     (forall ?y:Object
                       (iff (P ?x:Object)
                            (P ?y:Object))))
                   (iff (exists ?x:Object
                          (Q ?x:Object))
                        (forall ?y:Object
                          (P ?y:Object))))
              (iff (exists ?x:Object
                     (forall ?y:Object
                       (iff (Q ?x:Object)
                            (Q ?y:Object))))
                   (iff (exists ?x:Object
                          (P ?x:Object))
                        (forall ?y:Object
                          (Q ?y:Object)))))
```

D.1.3 Automated induction

Every time a datatype or structure by the name of *S* is introduced, Athena automatically defines a method named *s*`-induction`, where *s* is the downcased version of *S*. For example, when the polymorphic datatype `List` is introduced, Athena automatically defines a method named `list-induction`. Likewise, when `Pair` is introduced, Athena automatically defines a method `pair-induction`.

These are all unary methods that take an arbitrary goal p and attempt to derive p automatically by induction on S, where p is a universally quantified sentence over S:

$$\texttt{(forall } v\texttt{:}S\ q\texttt{).} \qquad (3)$$

The structure of the overall inductive proof is set up by Athena using the **`by-induction`** schema automatically extracted from the definition of S. So in the case of `List`, for example, the overall proof that would be attempted by a call of the form

`(!list-induction` p`)`

would be of the form:

```
1 by-induction p {
2   nil => (!prove p1 (ab))
3 | (:: h t) => (!prove p2 (ab))
4 }
```

where p_1 and p_2 are the appropriate instances of p. In particular, p_1 is obtained from q by replacing all free occurrences of v:S with `nil`, and p_2 is likewise obtained from q by replacing all free occurrences of v:S with `(:: h t)`. Because Athena automatically inserts inductive hypotheses in the assumption base before going on to evaluate the deduction D_i of each clause π_i `=>` D_i in a **`by-induction`** proof, the call `(ab)` in line 3 will pick up the appropriate inductive hypothesis for the tail `t`.

One drawback of the s`-induction` methods is that each inductive subproof will be a call to `prove` with the contents of the entire assumption base as the value of the second argument (the list of premises). There are two issues with that. First, the fact that we are using `prove` means that this will be a default ATP call, so we cannot choose which theorem prover to use, how long to wait in each case, and so on. And second, the fact that we are always passing the entire assumption base to the ATP to use as premises is overkill. As we discussed earlier, the larger the assumption base, the more difficult the job of the ATP becomes.

An alternative interface to automated induction that gives us a much finer degree of control is provided by methods named s`-induction-with`, which are also automatically defined every time a datatype or structure S is introduced. Unlike s`-induction`, a method of the form s`-induction-with` is binary. The first argument is again the desired goal, a universally quantified sentence of the form (3). The second argument is what gives us the greater control. It can be either a unary ATP method M or a map from the names of S's constructors to unary prover methods. Let's start with the first alternative, when the second argument to s`-induction-with` is a unary ATP method M. Thanks to static scoping, we can pass as the value of this argument a method that takes the desired goal and tries to derive it using ATPs but with whatever options we wish to specify, for instance:

`(!list-induction-with` p `method (goal) (!derive-from goal (ab)` *options*`))` where *options* specify which theorem prover to use or how long to wait for the answer. In fact this approach also gives us a knob for adjusting the list of premises given to the ATP. For instance, we might write:

```
(!list-induction-with p method (goal) (!derive-from goal (prems goal) options))
```

where `(prems goal)` is a procedure call that determines what premises to use depending on the form of the goal.

Even greater flexibility is afforded by the second alternative, whereby the second argument to *s*`-induction-with` is a map from the names of *S*'s constructors to unary ATP methods:

```
(!list-induction-with p
  |{'nil := method (goal)
              ...
    ':: :=  method (goal)
              ...
   }|)
```

allowing us to tailor the ATP calls to the individual constructors involved in each case.

Using these sort-specific automated-induction methods as a basis, the Athena library defines a couple of more generic automated-induction methods that can be deployed on *any* datatype or structure: `induction*` and `induction*-with`. The first is unary, whose single argument is the goal to be inductively proved; and the second is binary, whose first argument is the goal to be proved and whose second argument is a unary ATP method.

The first obvious difference of these two methods from the methods that were previously discussed is that there is no *s*`-` prefix needed, that is, it is not necessary to write `list-induction` in one case vs. `N-induction` in the other, and so on. One simply writes `induction*` (or `induction*-with`). Another significant difference is that the goal given to these methods does not even need to be of the form (3). The goal does need to be universally quantified, but there may be several universal quantifiers upfront:

$$\forall\, x_1 \, \forall\, x_2 \cdots \forall\, x_n \, . \, \cdots$$

and the very first one, $\forall\, x_1$, does not even need to range over a datatype or structure. It may instead range over a conventional sort, such as `Int` or some user-introduced domain, as long as some inner quantifier $\forall\, x_i$, $i > 1$, does range over a datatype or structure. In that case Athena will pull that quantifier upfront (by swapping x_1 and x_i) before attempting an inductive proof, and after a successful derivation it will put the result back in its original form. A second difference is that there may be multiple universal quantifiers ranging over datatypes or structures, calling for nested inductive proofs. Athena will handle that automatically as well.

The difference between `induction*` and `induction*-with` is akin to the difference between *s*`-induction` and *s*`-induction-with`. The first form, `induction*`, uses default ATP settings and uses the entire assumption base as the premises for each ATP call. The latter form, `induction*-with`, gives us greater control over ATP options and over which subset of the assumption base to use, depending on the given goal.

Readers are encouraged to recast the development of one of the longer chapters, such as Chapter 18, using ATPs.

D.2 SMT solving

Satisfiability-modulo-theories (SMT) [29] [30] is a fairly recent technology that can be seen as a generalization of classic propositional satisfiability (see Section 4.13). An SMT solver accepts as input a quantifier-free sentence with various interpreted and uninterpreted function symbols. The interpreted atoms come from background theories such as linear (integer or real) arithmetic, inductive datatypes, uninterpreted functions with equality, etc. The satisfiability of an input sentence p is then determined with respect to these background theories, along with the boolean structure of p. For instance, even though `(1 < 0)` is a perfectly satisfiable sentence when regarded as an uninterpreted sentence of first-order logic, it is clearly unsatisfiable when viewed specifically in the context of integer arithmetic. Most SMT solvers will not simply determine whether p is satisfiable, but if it is, they will also provide a satisfying model. Owing to their ability to natively handle useful background theories, particularly arithmetic, SMT solvers have found a wide array of uses in a short amount of time. For a general introduction to SMT solving and its applications, refer to *Satisfiability Modulo Theories: Introduction and Applications* [29]. Applications of SMT solving specifically in connection with Athena can be found in a number of publications [6], [5], [7].

Athena is integrated with two SMT solvers, Yices and CVC4, both of which can be freely downloaded.[5] SMT functionality in Athena is built into the top-level module `SMT`. The most generic interface to that functionality is via the primitive binary procedure `SMT.smt-solve`. The first argument is either a single sentence or a list of sentences, representing the constraint(s) to be solved. The second argument is a map providing values for various options that control the SMT solving. The most important of these are:

- The value assigned to the key `'solver` specifies which SMT solver to use. Current possible values for this key are `'yices` and `'cvc`.
- The value assigned to the key `'timeout` is the maximum number of seconds that Athena will wait for an answer from the solver.
- The value assigned to the key `'results` is a hash table that will hold the output identities produced by the solver when the input constraints are satisfiable. These identities will

5 They must be placed in the same directory as the Athena executable and named `yices.exe` and `cvcopt.exe` in Windows environments or `yices` and `cvc` in non-Windows environments.

therefore collectively determine a *model* for the given constraints. Specifically, the hash table will map terms (not necessarily just variables) to lists of terms. That a term might be mapped to several terms (a list thereof) is due to the fact that oftentimes SMT solvers produce multiple equations of the form

$$t = t_1, t = t_2, \ldots, t = t_n.$$

In the API we are describing, the hash table will map t to a list comprising $t_1, \ldots, t_n$.

- The value assigned to the key `'stats` is a cell that will contain a map providing information (mostly running times) about various stages of the process, such as the translation from Athena to the language accepted by the relevant SMT solver, the size of the translated sentence, the SMT solving itself, etc.

The result of a call to `SMT.smt-solve` is either `'Satisfiable` or `'Unsatisfiable`. In the former case, the model produced as evidence of satisfiability will be contained in the hash table that was specified in the call to `SMT.smt-solve`. Here is a simple example:

```
define ht := (HashTable.table)

define options := |{'solver := 'yices, 'results := ht}|

> (SMT.smt-solve (?x < 3) options)

Term: 'Satisfiable

> ht

Table: ?x:Int := [0]
```

Thus, we see that in this case the constraint `(< ?x:Int 3)` is determined to be satisfiable, and the corresponding model is stored in `ht`. Of course, in this case the model is very simple, consisting essentially of a single value assigned to the variable `?x`.

A simpler interface is available when we are confident that the structure of the constraints is such that a satisfying model will only assign values to variables, and at most one value to each variable at that. In that case we can apply the unary procedure `SMT.solve` on the given constraint (or list of constraints). The result will be either the term `'Unsatisfiable` or a *substitution* from variables to appropriate values. For example:

```
> (SMT.solve  x < 3)

Substitution: {?x:Int --> 0}

> (SMT.solve x < 3 & x > 0)

Substitution: {?x:Int --> 1}

> (SMT.solve x < 3 & x > 2)
```

```
Term: 'Unsatisfiable

> (SMT.solve x < 3.0 & x > 2.0)

Substitution: {?x:Real --> 2.5}
```

`SMT.solve` uses Yices by default.

SMT solvers understand data types natively:

```
datatype Day := Mon | Tue | Wed | Thu | Fri | Sat | Sun

define (weekday d) := (d = Mon | d = Tue | d = Wed | d = Thu | d = Fri)

define (weekend d) := (d = Sat | d = Sun)

define d := ?d:Day

> (SMT.solve weekday d & d =/= Fri)

Substitution: {?d:Day --> Thu}

> (SMT.solve weekday d & weekend d)

Term: 'Unsatisfiable
```

Reasoning with more complicated (e.g., recursive) datatypes is similar:

```
define [l1 l2] := [?l1:(List 'S1) ?l2:(List 'S2)]

> (SMT.solve l1 = l2 & l1 = 1::nil & l2 = 2::nil)

Term: 'Unsatisfiable

> (SMT.solve x = hd tl 1::2::nil)

Substitution: {?x:Int --> 2}

> (SMT.solve x = pair-left 1 @ 2)

Substitution: {?x:Int --> 1}

> (SMT.solve x = pair-right 'a @ 'b)

Substitution: {?x:Ide --> 'b}
```

(Recall that `hd` and `tl` are the selectors of the "consing" constructor of `List`, while `pair-left` and `pair-right` are the selectors of the `pair` constructor.)

Polymorphism is handled automatically by the translation from Athena to the input language of the respective SMT solver, though in a different way than the translation described in Section D.1.1. For SMT solving, Athena simply grounds sort parameters to concrete sorts handled by the SMT solvers. This tends to produce the most digestible results. Here is an example of reasoning with a polymorphic uninterpreted function:

```
declare f: (S) [S] -> S

define constraint1 := (f f f f f f f x = x)

define constraint2 := (f f f f f x = x)

(SMT.holds? constraint1 & constraint2 ==> f x = x)

Term: true
```

Note that `SMT.holds?` simply takes the given constraint, negates it, and tests for satisfiability. If the negation is unsatisfiable, `SMT.holds` returns `true`, otherwise it returns `false`. Thus, in this example, the result means that `constraint-1` and `constraint-2` logically entail `(f x = x)`.

The humble theory of uninterpreted functions from which the preceding example is taken is in fact exceedingly useful, particularly in hardware verification. For instance, we can often optimize a pipelined circuit to make it run faster by removing or rearranging certain components of it, such as multiplexers. We must then prove that the new circuit is in fact equivalent to the original one. That problem can often be posed as a satisfiability problem in the theory of uninterpreted functions, where we use (e.g., boolean) functions to model circuit components. The following example is taken from Section 3.5.1 of *Decision Procedures: An Algorithmic Point of View* [62]. The first circuit stores its result in latch L_5, and the optimized circuit stores its result in L'_5.

```
declare L1, L2, L3, L4, L5, L1', L2', L3', L4', L5', input: Boolean
declare C,D,F,G,H,K: [Boolean] -> Boolean

define circuit-1 :=
  (and (L1 = F input)
       (L2 = L1)
       (L3 = K G L1)
       (L4 = H L1)
       (L5 = (ite (C L2) L3 (D L4))))

define circuit-2 :=
  (and (L1' = F input)
       (L2' = C L1')
       (L3' = (ite (C L1') (G L1') (H L1')))
       (L5' = (ite L2' (K L3') (D L3'))))

define correctness := (circuit-1 & circuit-2 ==> L5 <==> L5')

> (SMT.holds? correctness)
```

```
Term: true
```

SMT problems formulated in Athena may include fragments from the theory of arrays, which is represented here via the default maps discussed in Section 10.4.3, in module `DMap`. At present only Yices can solve such problems. Some simple examples:

```
define [zero-map at] := [(DMap.empty-map 0) DMap.at];;

define updated-with :=
  lambda (mapping p)
    match p {
     [key value] => (DMap.update (pair key value) mapping)
   | [key --> value] => (DMap.update (pair key value) mapping)};;

define query-1 := (?result = zero-map at x)

define query-2 := (?map = _ updated-with [7 --> 99] &
                   ?map at 7 =/= 2 * 50 - 1)

define query-3 := (x < 2 & ?my-map at x = z & z = y + 8 & y > 99)

> (SMT.solve query-1)

Substitution: {?result:Int --> 0}

> (SMT.solve query-2)

Term: 'Unsatisfiable

> (SMT.solve query-3)

Substitution:
{?z:Int --> 108
?y:Int --> 100
?x:Int --> 0}
```

We continue with an example that uses SMT solving to solve any instance of the N-*queens problem*. The example is a fairly typical illustration of using Athena to fluidly express complex constraints which are then outsourced for solution to an external solver. Assuming we have N queens to place on an $N \times N$ chess board, the row and column of the i^{th} queen will be respectively expressed by `(row i)` and `(col i)`:

```
declare row, col: [Int] -> Int [150]
```

We define a convenient shorthand `in` for expressing the constraint that a numeric variable *x* should be in a given *range* defined by a low and a high endpoint, which are here placed inside a two-element list:

```
define (in x L) :=
  match L {
    [l h] => (l <= x & x <= h)
  }

> (x in [2 5])

Sentence: (and (<= 2 ?x:Int)
               (<= ?x:Int 5))
```

We will also need an absolute-value function, which we can easily define as follows:

```
declare abs: [Int] -> Int

assert* abs-def := [(abs x = (ite (x < 0) (- x) x))]
```

When does a queen *threaten* another queen on the same board? That happens iff the two queens are on the same row, or on the same column, or else on the same diagonal. The latter holds iff the absolute value of the row difference for the two queens is identical to the absolute value of their column difference. We can readily express this with an Athena procedure that takes the coordinates of two queens and produces a sentence that holds iff the queens threaten each other:

```
define (threatens r1 c1 r2 c2) :=
   (r1 = r2 | c1 = c2 | abs (r1 - r2) = abs (c1 - c2))
```

For example:

```
> (threatens 1 2 1 5)

Sentence: (or (= 1 1)
              (or (= 2 5)
                  (= (abs (- 1 1))
                     (abs (- 2 5)))))
```

We can now write a procedure that produces all the relevant constraints for a given instance of the *N*-queens problem (i.e., for a given value of *N*) as follows:

```
define (make-constraints N) :=
  let {all := (1 to N);
       every-queen-somewhere :=
         (map lambda (i)
                (row i in [1 N] & col i in [1 N])
              all);
       no-threats := (map
                        lambda (p)
                          match p {
                            [i j] => (i =/= j ==> ~ threatens (row i) (col i)
                                                                (row j) (col j))
                          }
                        (cprod all all))}
    (join every-queen-somewhere no-threats abs-def)
```

Note that cprod is a binary library procedure that forms the Cartesian product of two lists, e.g.:

```
> (cprod [1 2] [3 4])

List: [[1 3] [1 4] [2 3] [2 4]]
```

That's it. Observe that there is zero procedural knowledge expressed anywhere. The generated constraints are a purely declarative description of an instance of N-queens.

We can now readily arrive at a top-level solution that takes an input N and either outputs 'Unsatisfiable if there is no solution or else produces a list of row-column pairs for the N queens that constitutes a solution:

$$[[r_1 \;\; c_1] \;\cdots\; [r_N \;\; c_N]],$$

where $[r_i \;\; c_i]$ gives the row and column of the i^{th} queen:

```
define (solve-N-queens N) :=
 let {constraints := (make-constraints N);
      ht := (HashTable.table);
      get-answer := lambda (f i) (first (HashTable.lookup ht (f i)))}
   match (SMT.smt-solve constraints |{'solver := 'yices, 'results := ht}|) {
     'Satisfiable => (map lambda (i) [(get-answer row i) (get-answer col i)]
                          (1 to N))
   | res => res}
```

And in action:

```
> (solve-N-queens 2)

Term: 'Unsatisfiable

> (solve-N-queens 3)

Term: 'Unsatisfiable

> (solve-N-queens 4)

List: [[4 2] [3 4] [2 1] [1 3]]

> (solve-N-queens 6)
```

```
List: [[5 4] [3 1] [6 2] [4 6] [2 3] [1 5]]

> (solve-N-queens 10)

List: [[6 5] [4 1] [7 7] [5 3] [8 2] [1 8] [3 9] [2 4] [10 6] [9 10]]
```

Note that in this formalization one of the constraints was the actual definition of abs, which was a universally quantified sentence. While that might work sometimes, depending on the heuristics that SMT solvers might have for dealing with quantifiers, it is generally not recommended to have any quantified formulas in the SMT input, since the solver might not be able to handle them, and even if it does, performance might suffer. It is generally better to *ground* all quantified sentences to a depth dictated by the size of the problem instance and replace them with a Boolean combination of the resulting instances. In this case we do not in fact need the general definition of abs; we only need the values of the abs function on the input range that is possible for the given problem instance, namely (- N) to N. So we can actually replace abs-def with what is essentially a look-up table for abs in the range of interest:

```
define (abs-values N) :=
  (map lambda (i)
         (abs i = check {(i less? 0) => (times (- 1) i) | else => i})
       ((- N) to N))

> (abs-values 2)

List: [
(= (abs (- 2))
   2)

(= (abs (- 1))
   1)

(= (abs 0)
   0)

(= (abs 1)
   1)

(= (abs 2)
   2)
]
```

Changing the last line of make-constraints to make it use abs-values instead of abs-def as follows:

```
(join every-queen-somewhere no-threats (abs-values N))
```

has a dramatic impact on performance, making the solving over 50 times faster on average (for large values of N).

We end this section with a discussion of the SMT version of Max-SAT. Max-SAT is a well-studied generalization of satisfiability that allows for the solution of difficult optimization problems. Max-SAT is just like the SAT problem described in Section 4.13, except that it seeks to maximize the number of satisfied clauses. A more practically useful variant is the *weighted* Max-SAT problem, where weights are attached to clauses and the objective is to find an interpretation that maximizes (or equivalently, minimizes) the total weight of the satisfied clauses. Max-SAT has a very large number of practical applications, including probabilistic inference in Bayesian networks and general inference in Markov logic [24], and even learning Bayesian networks directly from data [23]. Yices solves the weighted Max-SAT problem in the context of SMT, so it is a further generalization of weighted Max-SAT, which we may call Max-SMT. Athena exposes that functionality via the procedure `SMT.solve-max`. The procedure is unary and takes as input a list of weighted constraints, each of these being a pair of the form `[`c w`]` where c is an Athena sentence (quantifier-less) and w is the integer weight attached to c. Weights can be arbitrary integers or a special "infinite" token, `'inf`, indicating a hard constraint that must be satisfied no matter what.

Occasionally we deal with problems that are easier to model and solve if we can perform general optimization by minimizing some (typically linear) objective function. Most SMT solvers at present do not perform optimization (apart from Max-SAT, in the case of Yices), but Athena efficiently implements an integer optimizer on top of SMT solving. The idea is to use binary search to discover an optimal solution with as few calls to the SMT solver as possible: at most $O(\log n)$ calls, where n is the maximum value that the objective function can attain. Specifically, let c be an arbitrary constraint that we wish to satisfy in such a way that the value of some "cost" term t is minimized, where max is the maximum value that can be attained by the cost function (represented by t).[6] The algorithm is the following: We first try to satisfy c conjoined with the constraint that the cost term t is between 0 and half of the maximum possible value: $0 \leq t \leq (max \ \texttt{div} \ 2)$. If we fail, we recursively call the algorithm and try to satisfy c augmented with the constraint $(max \ \texttt{div} \ 2) + 1 \leq t \leq max$. If we succeed, we recursively call the algorithm and try to satisfy c augmented with the constraint $0 \leq t \leq (max \ \texttt{div} \ 4)$; and so on. This is guaranteed to converge to the minimum value of t for which c is satisfied, provided that the original constraint c is satisfiable for some value of t. The algorithm is implemented by `SMT.solve-and-minimize`, so that

```
(SMT.solve-and-minimize c t max)
```

returns a satisfying assignment for c that minimizes t (whose maximum value is max).

For example, suppose that `x`, `y`, and `z` are integer variables to be solved for (the role of `d-x`, `d-y` and `d-z` will be explained shortly):

6 If this value is not known a priori, it can be taken to be the greatest positive integer that can be represented on the computer.

```
define [x y z d-x d-y d-z] :=
       [?x:Int ?y:Int ?z:Int ?d-x:Int ?d-y:Int ?d-z:Int]
```

subject to the following disjunctive constraint:

```
define c := (x in [10 20]   & y in [1 20]  & z in [720 800] |
             x in [500 600] & y in [30 40] & z in [920 925])
```

Suppose also that the desired values for these variables are `x = 13`, `y = 15`, `z = 922`. Clearly these values cannot be attained subject to the given constraints. However, we would like to find values for them that come *as close as possible* to the desired values while respecting the constraints. This sort of problem has many practical applications (e.g., optimal repair in access control requests [5]). We can readily model it in a form that is amenable to `solve-and-minimize` as follows. First we define the objective-function term t as the sum of the three differences:

```
define t := (d-x + d-y + d-z)
```

with the individual difference terms defined as follows:

```
define d-x-def := (ite (x > 13)
                       (d-x = x - 13)
                       (d-x = 13 - x))

define d-y-def := (ite (y > 15)
                       (d-y = y - 15)
                       (d-y = 15 - y))

define d-z-def := (ite (z > 922)
                       (d-z = z - 922)
                       (d-z = 922 - z))
```

Thus, the “definition” of `d-x` states that if `x` is greater than 13 then `d-x` is equal to `(x - 13)`, otherwise `d-x` is equal to `(13 - x)`. Accordingly, `d-x` captures the absolute value of the difference of `x` from 13. The definitions of the other two difference terms are similar. Assume that we do not know the exact maximum value that `t` can attain, but we know that it cannot be more than 10^6. (In this case the maximum value of t is clearly smaller, but let us pretend otherwise.) We can then solve the problem with the following call:

```
define diff-clauses := (d-x-def & d-y-def & d-z-def)

define query := (c & diff-clauses)

> (SMT.solve-and-minimize query t 1000000)

Substitution:
{?d-z:Int --> 122
?d-y:Int --> 0
?d-x:Int --> 0
?z:Int --> 800
?y:Int --> 15
?x:Int --> 13}
```

This solution was found by a total of 8 calls to the SMT solver. Note that `?x` and `?y` are identical to the desired values, while `?z` is as close as possible to the desired value, namely, 800. For comparison purposes, here is the result that we would get if we solved the query without minimizing t:

```
> (SMT.solve query)

Substitution:
{?d-z:Int --> 202
?d-y:Int --> 14
?d-x:Int --> 3
?z:Int --> 720
?y:Int --> 1
?x:Int --> 10}
```

The total distance of this solution from the desired values is $487+15+1=503$, as opposed to 122, the distance returned by `SMT.solve-and-minimize`, which is the smallest possible distance allowed by the given constraints.

Why were only 8 calls required when we started the binary search with a maximum of 10^6? One would expect about $\log 10^6$ calls to the SMT solver, i.e., roughly 20 such calls. However, the implementation uses the information returned by each call to the SMT solver to speed up the search. That often results in significant shortcuts, cutting down the total number of iterations by more than a factor of 2.

This procedure allows for the optimization of any linear integer quantity. Moreover, unlike independent branch-and-bound algorithms for integer programming, it allows not just for numeric constraints, but for arbitrary boolean structure as well, along with constraints from other theories such as reals, lists and other algebraic datatypes, arrays, bit vectors, etc. A drawback is that the entire core constraint plus the bound constraint is solved anew each time we halve the range, so typically this method is not as fast as a native optimizer. But in many cases the added modeling expressivity makes up for that.