5 First-Order Logic

IN THIS CHAPTER we turn our attention to predicate (or *first-order*) logic. Our main objective is to describe the various Athena mechanisms for constructing proofs in predicate logic. In addition, we present heuristics for developing such proofs, we build a library of generically useful proof methods, and we also discuss the semantics of (polymorphic, many-sorted) predicate logic.

Recall from Section 2.4 that the syntax of predicate logic extends the syntax of sentential logic. So, just as before, we have atomic sentences (atoms), negations, conjunctions, disjunctions, conditionals, and biconditionals, where an atom can be an arbitrary term of sort `Boolean`. Thus, assuming that `x`, `y`, and `z` are term variables, all of the following are legal first-order sentences:

Sentence	Type
`(x < x + 1)`	*Atom*
`(~ even 3)`	*Negation*
`(x < x + 1 & 7 =/= 8)`	*Conjunction*
`(x < y | x = y | y < x)`	*Disjunction*
`(x < y & y < z ==> x < z)`	*Conditional*
`(x + 1 < y + 1 <==> x < y)`	*Biconditional*

We assume for the remainder of this chapter that the following domains have been introduced:

```
domains Object, Element, Set
```

along with the following symbols:

```
declare even, odd, prime: [Int] -> Boolean

declare subset: [Set Set] -> Boolean

declare P, Q, S: [Object] -> Boolean

declare R, T: [Object Object] -> Boolean

declare a, b, c, d: Object
```

We also assume that x, y, z, and w have been defined as the variables ?x, ?y, ?z, and ?w:

```
define [x y z w] := [?x ?y ?z ?w]
```

In addition to the kinds of sentences we are already familiar with from sentential logic, we now have *quantified sentences* (or simply *quantifications*): For any variable *v* and sentence *p*,

(forall *v* . *p*) (5.1)

and

(exists *v* . *p*) (5.2)

are also legal first-order sentences. We refer to *p* as the *body* of (5.1) and (5.2). One can also quantify over multiple variables with only one quantifier, for example, writing

```
(forall x y . x + y = y + x)
```

as a shorthand for

```
(forall x . forall y . x + y = y + x).
```

Intuitively, sentences of the form (5.1) state that *every* object of a certain sort has such-and-such property; while sentences of the form (5.2) state that *some* object (of a certain sort) has such-and-such property. For instance, the statement that every prime number greater than two is odd can be expressed as follows:

```
(forall x . prime x & x > 2 ==> odd x),
```

while the statement that there is some even prime number can be expressed as:

```
(exists x . prime x & even x).
```

Quantifiers can be combined to form more complex sentences. For instance:

```
(forall x . exists y . x subset y)
```

states that every set has some superset, while

```
(exists x . forall y . x subset y)
```

says that there is a set that is a subset of every set. Note that it is not necessary to explicitly annotate occurrences of quantified variables with their respective sorts. For instance, in the above examples it is not necessary to tell Athena that x and y range over sets. That is inferred automatically:

```
> (exists x . forall y . x subset y)

Sentence: (exists ?x:Set
            (forall ?y:Set
              (subset ?x:Set ?y:Set)))
```

When we wish to make the sort S of a quantified variable explicit, we will write quantifications as

$$\texttt{(forall } v:S \texttt{ . } p\texttt{)}$$

and

$$\texttt{(exists } v:S \texttt{ . } p\texttt{)}.$$

In the next two sections we will present Athena's mechanisms for introducing and eliminating each type of quantified sentence. But first we briefly turn our attention to an exercise that develops more fully the idea that terms and sentences are trees, and gives computationally precise definitions of concepts such as *variable occurrences* in a sentence (both *free* and *bound* variable occurrences), free variable replacements, and alpha-equivalence. These concepts have already been introduced, so those who feel sufficiently comfortable with them can move on to the next section. But for those who don't mind some coding, this exercise is a good opportunity to consolidate these ideas in greater algorithmic detail.

Exercise 5.1: As we explained in Section 3.3 by way of examples, terms and sentences are essentially trees. We will now make this and other related ideas more precise.

(a) Define two unary procedures `term->tree` and `sent->tree` that transform a given term or sentence into an explicit tree representation. Each node in the tree will be a map m of the form

$$\texttt{|\{ 'data := } x\texttt{, 'pos := } L\texttt{, 'children := [}m_1 \cdots m_n\texttt{] \}|}, \tag{5.3}$$

where the `'data` value x is the function symbol, variable, quantifier, or sentential constructor that appears at that node; the `'pos` value L is the Dewey path of the node in the tree, as a list of positive integers; and $m_1, \ldots, m_n$ are the maps that (recursively) correspond to the children of the node (hence leaf nodes will have $n = 0$), where each m_i is itself of the form (5.3). Thus, for instance, we should get:

```
> define t := (1 + x)

Term t defined.

> (term->tree t)

Map:
|{
'data := +
'pos := []
'children := [ |{'data := 1, 'pos := [1], 'children := []}|
               |{'data := ?x:Int, 'pos := [2], 'children := []}|
             ]
}|

> (term->tree true)
```

```
Map: |{'data := true, 'pos := [], 'children := []}|

> (sent->tree (~ x < 1))

Map:
|{
'data := not,
'pos := [],
'children := [|{'data := <,
                'pos := [1],
                'children := [|{'data := ?x:Int,
                                'pos := [1 1],
                                'children := []}|
                              |{'data := 1,
                                'pos := [1 2],
                                'children := []}|]
              }|]
}|
```

(b) Define a procedure `tree-leaves` that takes a tree T produced by (either) one of the above procedures and returns a list of all the leaves of T, where each leaf is represented as a record (map) of the form `|{'data :=` x`, 'pos :=` L`}|`. For instance:

```
> (tree-leaves term->tree t)

List: [|{'data := 1, 'pos := [1]}|
       |{'data := ?x:Int, 'pos := [2]}|]
```

(c) Define a procedure `var-occs` that takes a term t or sentence p and produces a map m whose keys are all and only the variables that occur in t (or p). The value that m assigns to such a variable v is a list of the positions at which v occurs in t (or p). For example:

```
> (var-occs forall x . exists y . x = y)

Map: |{?x:'T3097 := [[1] [2 2 1]], ?y:'T3097 := [[2 1] [2 2 2]]}|
```

(d) Define a procedure `bound-var-occs` that takes a sentence p and produces a map in the form described above, except that the keys now consist of all and only the variables that have bound occurrences in p. Also define a procedure `free-var-occs` that does the same for the variables that have free occurrences in the input p.

(e) Define a procedure `alpha-equiv?` that determines whether two sentences are alpha-equivalent.[1]

1 Of course Athena already provides an implementation of alpha-equivalence—one can simply apply `equal?` to two sentences. The objective here, as in the following two parts of this exercise, is to implement this functionality from the ground up for instructive purposes.

(f) Define a procedure `sent-rename` that alpha-renames a given sentence.

(g) Define a procedure `(replace-var-by-term x t p)` that replaces every free occurrence of variable x in a given sentence p by term t, performing renaming as needed so as to avoid variable capture. Assume that the sort of t is an instance of the most general sort of x in p. □

5.1 Working with universal quantifications

5.1.1 Using universal quantifications

A universal quantification makes a general statement, about *every* object of some sort. For instance,

```
(forall x . x < x + 1)
```
(5.4)

says that every integer is less than its successor. Hence, if we know that (5.4) holds, we should be able to conclude that any *particular* integer is less than its successor, say, that 5 is less than its successor:

```
(5 < 5 + 1)
```
(5.5)

or that 78 is less than its successor:

```
(78 < 78 + 1).
```
(5.6)

We say that the conclusions (5.5) and (5.6) are obtained from (5.4) by *universal specialization*, or *universal instantiation.*

In Athena, universal specialization is performed by the binary method `uspec`. The first argument to `uspec` is the universal quantification p that we want to instantiate; p must be in the assumption base, otherwise the application will fail. The second argument is the term with which we want to specialize p:

```
assert p := (forall x . x < x + 1)

> (!uspec p 5)

Theorem: (< 5
            (+ 5 1))

> (!uspec p 78)

Theorem: (< 78
            (+ 78 1))

> (!uspec p (2 * x))

Theorem: (< (* 2 ?x:Int)
```

```
            (+ (* 2 ?x:Int)
               1))
```

The last example shows that the instantiating term does not have to be ground.

More precisely, if p is a universal quantification `(forall` v `.` q`)` in the assumption base and t is a term, then the method call

`(!uspec` p t`)`

will produce the conclusion $\{v \mapsto t\}(q)$, where $\{v \mapsto t\}(q)$ is the sentence obtained from q by replacing every free occurrence of v by t. An error will occur if the result of this replacement is ill-sorted, as would happen, for instance, if we tried to specialize (5.4) with the Boolean term `true`. Note also that the sort S of the quantified variable v may be polymorphic. The application of `uspec` will work fine as long as the sort S_t of the instantiating term t is unifiable with S (recall the definition of unifiable sorts from page 51). For example, the quantified sentence might state that the double reversal of any list L of any sort is identical to L, and the instantiating term might be a list of Boolean terms, or a list of integers, or a polymorphic variable:

```
declare reverse: (T) [(List T)] -> (List T)

> assert p := (forall x . reverse reverse x = x)

The sentence
(forall ?x:(List 'S)
  (= (reverse (reverse ?x:(List 'S)))
     ?x:(List 'S)))
has been added to the assumption base.

> (!uspec p (true::false::nil))

Theorem: (= (reverse (reverse (:: true
                                  (:: false
                                      nil:(List Boolean)))))
            (:: true
                (:: false
                    nil:(List Boolean))))

> (!uspec p (78::nil))

Theorem: (= (reverse (reverse (:: 78
                                  nil:(List Int))))
            (:: 78
                                  nil:(List Int)))

> (!uspec p ?L)

Theorem: (= (reverse (reverse ?L:(List 'S)))
```

```
            ?L:(List 'S))
```

In proof systems in which alpha-convertible sentences are not viewed as identical,[2] care must be taken when performing universal specialization to avoid variable capture. In particular, the instantiating term should not contain any variables that might get accidentally bound as a result of the substitution. Violating this proviso generates an error in such systems. For instance, consider the sentence

$$\forall x \,.\, \exists y \,.\, x \neq y, \tag{5.7}$$

which essentially says that there are at least two distinct individuals. If, in such a system, we were allowed to specialize (5.7) with the variable y, we would obtain the nonsensical conclusion

$$\exists y \,.\, y \neq y.$$

The problem here is variable capture: The instantiating term (the variable y) becomes improperly bound by the existential quantifier $\exists y$.

One way of dealing with this issue is to disallow such universal specializations. Systems like Athena, in which the choice of a quantified variable name is immaterial, offer a more flexible alternative: They automatically avoid variable capture simply by alpha-renaming the sentence being specialized, using fresh variables. Doing so is safe because, for deductive purposes, the renamed sentence is identical to the original. For instance, here is what happens if we try to reproduce the above example in Athena:

```
> assert p := (forall ?x . exists ?y . ?x =/= ?y)

The sentence
(forall ?x:'S
  (exists ?y:'S
    (not (= ?x:'S ?y:'S))))
has been added to the assumption base.

> (!uspec p ?y)

Theorem: (exists ?v1915:'S
           (not (= ?y:'S ?v1915:'S)))
```

Athena accepted the attempt to instantiate

$$p = \texttt{(forall ?x . exists ?y . x =/= ?y)} \tag{5.8}$$

with ?y, but averted variable capture by first renaming p to something like

$$q = \texttt{(forall ?v1914 . exists ?v1915 . ?v1914 =/= ?v1915).} \tag{5.9}$$

2 So that, e.g., $\forall x \,.\, P(x)$ and $\forall y \,.\, P(y)$ are treated as two distinct sentences.

Substituting `?y` for the free occurrence of `?x` in the body of p now becomes the same as substituting `?y` for the free occurrence of `?v1914` in the body of (5.9), a replacement that is harmless and results in the theorem displayed above. The upshot is that there is no need to be concerned about variable capture when performing universal instantiation.

Finally, it is noteworthy that in some respects a universal quantification can be understood as a large—potentially infinite—conjunction. For instance, if `B` is a unary predicate on `Boolean`, then the sentence

```
(forall ?x:Boolean . B ?x:Boolean)
```

means the exact same thing as

```
(B true & B false),
```

given that `true` and `false` are the only `Boolean` values. Likewise, the sentence

```
(forall ?x:Int . ?x:Int < ?x:Int + 1)
```

can be informally viewed as the infinite conjunction of all atoms of the form (`t < t + 1`), for every possible integer numeral t (positive, negative, and zero). With this analogy in mind, universal instantiation can be thought of as a generalized version of conjunction elimination.

5.1.2 Deriving universal quantifications

How do we go about proving that every object of some sort S has a property P? That is, how do we derive a goal of the form $(\forall\, v : S\,.\, P(v))$? Typically, mathematicians prove such statements by reasoning as follows:

$$\text{Consider any } I \text{ of sort } S. \text{ Then } \cdots D \cdots$$

where the name (identifier) I occurs free inside the proof D. We can regard this construction as a unary method with parameter I and body D that will take any term t of sort S and will attempt to prove $P(t)$. Clearly, if we have a method which can prove that *any* given object in the relevant domain has the property P, then we are entitled to conclude the desired $(\forall\, v : S\,.\, P(v))$. But how can we make sure that this "method" is indeed capable of doing that?

Well, we can try applying the method to some random term and see if it succeeds, that is, we can try evaluating the body D with some random term t of sort S in place of the parameter I, and see if we succeed in deriving $P(t)$. But what if the success is a fluke? What if D exploits some special assumptions about t that wouldn't be valid for some other term t'? Suppose, for instance, that the domain in question is that of the natural numbers, and when we apply the method to, say, `zero`, we do get the theorem P(`zero`) as output. But what if D relied on some special assumptions about `zero` in the assumption base? Even if

it did not, how can we be sure that the method truly generalizes, meaning that applying the same reasoning to *any* other term t of sort `N` will also derive $P(t)$?

The trick is to choose our test term t judiciously. It must be a term with no baggage, so to speak. That is, a term about which there are definitely no special assumptions in effect. A *fresh variable* x of sort S is just such a term; its freshness ensures that no special assumptions about it can possibly be in effect, since the assumption base will not contain any occurrences of x. Accordingly, we may regard x as representing a truly arbitrary element of the domain at hand. Hence, if the evaluation of D in an environment in which I refers to a fresh variable x succeeds in deriving the theorem $P(x)$, we can conclude that P holds for *every* object of the relevant sort. That is, we can safely conclude $(\forall\, v : S\,.\, P(v))$.

Reasoning of this kind is expressed in Athena with deductions of the form

```
pick-any I D.
```
(5.10)

We refer to D as the *body* of (5.10). To evaluate a deduction of this form in an assumption base β, we first generate a fresh variable x of sort S, where S is itself a fresh sort variable (representing a completely unconstrained sort), say, `?v135:'S47`. This ensures that x is a variable that has never been used before in the current Athena session. We then evaluate the body D in β and, importantly, in an environment in which the name I refers to the fresh variable x. We say that D represents the *scope* of that variable. If and when that evaluation results in a conclusion p, we return the quantification `(forall` x `.` p`)` as the final result of (5.10).

To make things concrete, consider as an example the deduction

```
pick-any x (!reflex x).
```
(5.11)

Recall that `reflex` is a unary primitive method that takes any term t and produces the equality `(`t `=` t`)`. To evaluate (5.11) in some assumption base β, Athena will first generate a fresh variable of a completely general and fresh sort, say `?x18:'S35`. It will then evaluate the body of the **`pick-any`** deduction, namely `(!reflex x)`, in an environment in which `x` refers to `?x18:'S35`. Thus, Athena will essentially be evaluating the deduction `(!reflex ?x18:'S35)`. According to the semantics of `reflex`, that will produce the equality `(?x18:'S35 = ?x18:'S35)`. Finally, Athena will generalize over the fresh variable and return the conclusion

```
(forall ?x18:'S35 . ?x18 = ?x18)
```
(5.12)

as the result of the entire **`pick-any`**. For readability purposes, however, Athena will present this conclusion as

```
(forall ?x:'S . ?x = ?x)
```
(5.13)

instead of (5.12). That is harmless because the two sentences are alpha-equivalent, and Athena treats alpha-equivalent sentences as identical for deductive purposes. Hence, if all

goes well, the user will never actually see or have to deal with the particular fresh variable that Athena generated, `?x18:'S35`.[3] The user simply enters the deduction (5.11) and the theorem (5.13) is produced:

```
> pick-any x (!reflex x)

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

This example demonstrates that **pick-any** can universally generalize not just over all the elements of a specific (ground) sort, such as the natural numbers or the booleans, but over all the elements of infinitely many sorts. Such a polymorphic generalization can then be specialized with any terms of appropriate sorts, say, with natural numbers or with booleans:

```
> define p := pick-any x (!reflex x)

Theorem: (forall ?x:'S
           (= ?x:'S ?x:'S))

Sentence p defined.

> (!uspec p true)

Theorem: (= true true)

> (!uspec p (2 * y))

Theorem: (= (* 2 ?y:Int)
            (* 2 ?y:Int))
```

As another example, here is a proof that the equality relation is symmetric. Recall that `sym` is a unary primitive method that takes an equality $(s = t)$ and returns $(t = s)$, provided that $(s = t)$ is in the assumption base:

```
1 > pick-any a
2     pick-any b
3       assume h := (a = b)
4         (!sym h)
5
6 Theorem: (forall ?a:'S
7              (forall ?b:'S
8                (if (= ?a:'S ?b:'S)
9                    (= ?b:'S ?a:'S))))
```

3 But Athena does choose a fresh variable such as $?I\_n$, where I is the original **pick-any** name and n is a number, so that if the proof fails with an error message, it is easier to see where that particular fresh variable originated.

How was this proof evaluated? It first generated a fresh variable of a brand new and completely unconstrained sort, say `?a74:'S95`, and then proceeded to line 2 to evaluate the body of the outer **`pick-any`** in an environment in which the name `a` refers to `?a74:'S95`. Now the body of the **`pick-any`** happens to be another **`pick-any`**, so another fresh variable (of unconstrained sort again) is generated, say, `?b75:'S96`, and Athena proceeds to evaluate the body of the inner **`pick-any`** in an environment in which `b` denotes `?b75:'S96` (and `a` denotes `?a74:'S95`). The body of the inner **`pick-any`** is an **`assume`**, so Athena adds the hypothesis `h := (?a74:'S95 = ?b75:'S95)` to the assumption base (note that the two sorts `'S95` and `'S96` have been unified at this point), and goes on to evaluate the body of the **`assume`**, namely, the application of `sym` to the identity `h`. Since `h` is in the assumption base, `sym` succeeds and returns the conclusion `(?b75:'S95 = ?a74:'S95)`, so, backing up one level, the **`assume`** returns the conclusion

```
(?a74 = ?b75 ==> ?b75 = ?a74)
```

(we have omitted the variable sorts for brevity). Hence, backing up one more level, the inner **`pick-any`** returns the generalization

```
(forall ?b75 . ?a74 = ?b75 ==> ?b75 = ?a74),
```

or, equivalently,

```
(forall ?b . ?a74 = ?b ==> ?b = ?a74).
```

Finally, backing up one more level, the outer **`pick-any`** returns the generalization

```
(forall ?a74 . forall ?b . ?a74 = ?b ==> ?b = ?a74),
```

or, equivalently,

```
(forall ?a:'S . forall ?b:'S . ?a = ?b ==> ?b = ?a).
```

This analysis included a lot of low-level details. It is rarely necessary to descend to such a level when one is working with Athena proofs. After a little practice one can understand what a proof does quite well at the higher and more intuitive level of its source text.

Observe that both quantified variables `?a` and `?b` in the conclusion of the above proof have the same sort `'S`. Again, `'S` is a sort variable, ranging over the collection of all available sorts. The only constraint in the above sentence is that `?a` and `?b` range over the *same* sort. That sort could be anything, but it has to be the same for both variables. This constraint was discovered by sort inference in the course of evaluating the deduction.

Although not usually necessary, users may, if they wish, provide explicit sort annotations for **`pick-any`** identifiers, and indeed this practice sometimes makes the code more readable, and can also simplify sort inference, especially with nested **`pick-any`** deductions. For instance, one may write:

```
pick-any e:Element
   pick-any s:Set
     assume h := (e in s)
        ...
```

In that case the generated fresh variable will be of the specified sort.

Nested **pick-any** deductions, incidentally, of the form

pick-any I_1 **pick-any** I_2 **pick-any** I_3 $\cdots$

may be abbreviated as **pick-any** I_1 I_2 I_3 $\cdots$. Thus, for example, the previous proof could also be written as follows:

```
pick-any a b
  assume h := (a = b)
    (!sym h)
```

As another example, let us prove the tautology

```
((forall x . P x) & (forall y . Q y) ==> forall y . P y & Q y)
```

(For example, if everything is green and everything is large, then everything is both green and large.)

```
define [all-P all-Q] := [(forall x . P x) (forall x . Q x)]

> assume hyp := (all-P & all-Q)
    pick-any y:Object
      let {P-y := conclude (P y)
                    (!uspec all-P y);
           Q-y := conclude (Q y)
                    (!uspec all-Q y)}
        (!both P-y Q-y)

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

Many more examples will appear later in the text and in the exercises.

We close this section by mentioning an alternative mechanism for deriving universal quantifications, the syntax form

generalize-over E D. (5.14)

Here E is an arbitrary Athena expression whose evaluation must produce a variable v. The idea is that v appears in D, and we will generalize over v whatever conclusion we obtain from D. That is, we evaluate D in the given assumption base β, and whatever conclusion p we get from that evaluation, we generalize it over v, thereby arriving at `(forall` v `.` p`)` as the overall result of (5.14). There is a proviso here, namely that the variable v should not have any free occurrences in the assumption base β. This ensures that D cannot exploit any special assumptions about v that might happen to be in effect when D is evaluated, and which could lead to an invalid generalization. As an example, evaluating the deduction

```
generalize-over ?foo:Int (!reflex ?foo:Int)
```

in some assumption base β will result in `(forall ?foo:Int . ?foo = ?foo)`, provided that `?foo:Int` does not occur free in β. If it does occur free, an error will be reported.

The two forms **`pick-any`** and **`generalize-over`** are closely related. The latter is more primitive. The former can actually be defined as syntax sugar on top of **`generalize-over`**. Specifically, we can define **`pick-any`** I D as

```
let {I := (fresh-var)}
  generalize-over I D
```

Thus, **`pick-any`** ensures that the aforementioned proviso is satisfied by generalizing over a *fresh* variable. Either form can be used to derive universal quantifications, but **`pick-any`** is almost always more convenient. The **`generalize-over`** construct is useful primarily when writing methods for discovering proofs automatically.

5.2 Working with existential quantifications

5.2.1 Deriving existential quantifications

If we know that 2 is an even number, then clearly we may conclude that *there exists* an even number. Likewise, if we know—or have assumed—that box `b` is red, we may conclude that there exists a red box. In general, if we have $\{x \mapsto t\}(p)$, we may conclude `(exists` x `.` p`)`.[4] This type of reasoning is known as *existential generalization*. In Athena, existential generalization is performed by the binary method `egen`. The first argument to `egen` is the existential quantification that we want to derive, say

`(exists` x `.` p`).` (5.15)

The second argument is a term t on the basis of which we are to infer (5.15). Specifically, if $\{x \mapsto t\}(p)$ is in the assumption base, then the call

4 Recall that $\{x \mapsto t\}(p)$ is the sentence obtained from p by replacing every free occurrence of x by t (renaming as necessary to avoid variable capture).

```
(!egen (exists x . p) t)
```

will derive the theorem `(exists x . p)`. The idea here is that because p holds for the object named by t, that is, because $\{x \mapsto t\}(p)$ is in the assumption base, we are entitled to conclude that there is *some* object for which p holds. We might view the term t as evidence for the existential claim at hand.

For instance, suppose that `(even 2)` is in the assumption base. Since we know that `2` is even, we are entitled to conclude that there exists an even integer:

```
assert (even 2)

> (!egen (exists x . even x) 2)

Theorem: (exists ?x:Int
           (even ?x:Int))
```

It is an error if the required evidence is not in the assumption base:

```
> clear-assumption-base

Assumption base cleared.

> (!egen (exists x . even x) 2)

standard input:1:1: Error: Failed existential
generalization---the required witness sentence
(even 2)
is not in the assumption base.
```

5.2.2 Using existential quantifications

Suppose that we know—or that we have simply assumed—that an existential quantification is true, so that some sentence of the form `(exists x . p)` is in the assumption base. How can we put such a sentence to use, that is, how can we derive further conclusions with the help of such a premise?

The answer is the technique of *existential instantiation*.[5] It is very commonly used in mathematics, in the following general form:

> We have it as a given that $\exists\, x\,.\, p$, so that p holds for *some* object. Let v be a name for such an object (i.e., let v be a "witness" for the existential sentence $\exists\, x\,.\, p$, so that $\{x \mapsto v\}(p)$ can be assumed to hold). Then $\cdots D \cdots$ (5.16)

5 Common alternative names for it are *existential specialization* and *existential elimination* .

where D is a deduction that proceeds to derive some conclusion q with the aid of the assumption $\{x \mapsto v\}(p)$, along with whatever other assumptions are currently operative. We refer to

$$\exists\, x\, .\, p$$

as the *existential premise*; v is called the *witness* variable; and the sentence $\{x \mapsto v\}(p)$ is called the *witness hypothesis*. We call D the *body* of the existential instantiation. It represents the *scope* of the witness hypothesis, as well as the scope of v. The conclusion q derived by the body D becomes the result of the entire proof (5.16).

The following proof is a simple and fairly typical example of how existential instantiation is used in practice. (We use conventional notation for this example, since this is a generic, language-agnostic illustration of the technique.) The proposition of interest here is that for all integers n, *if* $even(n)$ *then* $even(n+2)$ (i.e., if n is even, then so is $n+2$). Let us say that the unary predicate *even* is defined as follows:

$$(\forall\, i\, .\, even(i) \Leftrightarrow \exists\, j\, .\, i = 2 \cdot j). \tag{5.17}$$

The proof relies on the lemma

$$(\forall\, x\, y\, .\, x \cdot (y+1) = x \cdot y + x) \tag{5.18}$$

and proceeds as follows:

> Pick any n and assume $even(n)$. Then, by (5.17), we infer $(\exists\, j\, .\, n = 2 \cdot j)$, that is, there is some number, which, when multiplied by 2, yields n. *Let k stand for such a number, so that $n = 2 \cdot k$. Then, by congruence, $n+2 = (2 \cdot k) + 2$. But, by* (5.18)*, $(2 \cdot k) + 2 = 2 \cdot (k+1)$, hence, by the transitivity of equality, $n+2 = 2 \cdot (k+1)$. Therefore, by existential generalization, we obtain $(\exists\, m\, .\, n+2 = 2 \cdot m)$, and so, from* (5.17)*, we conclude $even(n+2)$.*

We have italicized the existential elimination argument in the above proof. The existential premise here is $(\exists\, j\, .\, n = 2 \cdot j)$; the witness is the variable k; and the witness premise is the equality $n = 2 \cdot k$. The conclusion of the existential instantiation is $even(n+2)$.

There are some important caveats that must be observed to ensure that this type of reasoning, as outlined in (5.16), will not lead us astray. The witness v must serve as a dummy placeholder—no special assumptions about it should be used. In particular, the body D must not rely on any previous working assumptions that might happen to contain free occurrences of v. Things can easily go wrong otherwise. Suppose, for example, that the current assumption base contains the atom $even(k)$, for some variable k, along with the sentence $(\exists\, n\, .\, odd(n))$. Now if we unwisely choose k as a witness for this existential premise, we will obtain the witness hypothesis $odd(k)$, and hence, in tandem with $even(k)$, we will be able to conclude that there is a number that is both even and odd. We will shortly see how Athena manages to enforce this proviso automatically, so that users do not need

to be concerned with it. Further, to ensure that the witness v is only used as a temporary dummy, the final conclusion q should not depend on v in any essential way. Specifically, q should not contain any free occurrences of v.

Existential instantiations in Athena are performed by deductions of the form

```
pick-witness I for F D
```
(5.19)

where I is a name that will be bound to the witness variable, F is a phrase that evaluates to an existential premise (`exists` $x : S$ `.` p), and D is the body.[6] To evaluate (5.19) in an assumption base β, we first make sure that the existential premise (`exists` $x : S$ `.` p) is indeed in β; if not, evaluation halts and a relevant error is reported. Assuming that the existential premise is in β, we begin by generating a fresh variable $v{:}S$, which will serve as the actual witness variable. We then construct the witness hypothesis, call it p', obtained from p by replacing every free occurrence of $x{:}S$ by the witness $v{:}S$. Finally, we evaluate the body D in the augmented assumption base $\beta \cup \{p'\}$ and in an environment in which the name I is bound to the witness variable $v{:}S$. If and when that evaluation produces a conclusion q, we return q as the result of the entire proof (5.19), provided that q does not contain any free occurrences of $v{:}S$ (it is an error if it does). The fact that the witness variable $v{:}S$ is freshly generated is what guarantees that the body D will not be able to rely on any special assumptions about it. The freshness of $v{:}S$ along with the explicit proviso that it must not occur in the conclusion q ensures that the witness is used only as a temporary placeholder. Bear in mind that I itself in (5.19) is not an Athena term variable; it is a name—an identifier—that will come to *denote* a fresh Athena variable (the witness variable $v{:}S$) in the course of evaluating the body D.

As an example, let us use existential instantiation to derive the tautology

```
((exists x . ~ prime x) ==> ~ forall x . prime x)
```

```
> assume hyp := (exists x . ~ prime x)
    pick-witness w for hyp  # We now have (~ prime w)
      (!by-contradiction (~ forall x . prime x)
         assume all-prime := (forall x . prime x)
           let {prime-w := (!uspec all-prime w)}
             (!absurd prime-w (~ prime w)))

Theorem: (if (exists ?x:Int
               (not (prime ?x:Int)))
             (not (forall ?x:Int
                    (prime ?x:Int))))
```

As an example of an incorrect use of `pick-witness`, the following violates the proviso that the witness variable must not appear in the conclusion:

6 It is an error if the evaluation of F produces any value other than an existentially quantified sentence.

```
> assume hyp := (exists x . prime x)
    pick-witness w for hyp
      (!claim (prime w))

input prompt:2:5: Error: Failed existential instantiation---the
witness variable occurs free in the resulting sentence.
```

Another potential error is that the existential premise is not in the assumption base:

```
> pick-witness w for (exists x . x =/= x)
    (!true-intro)

input prompt:1:22: Error: Existential sentence to be instantiated
is not in the assumption base:
(exists ?x:'S
  (not (= ?x:'S ?x:'S))).
```

When the existential premise has multiple consecutive existentially quantified variables, we can abbreviate nested **pick-witness** deductions with the form

pick-witnesses $I_1 \cdots I_n$ **for** F D

where F is a phrase that evaluates to an existential premise (`exists` $x_1 \cdots x_m$ `.` p) with $m \geq n$. The semantics of **pick-witnesses** is given by desugaring to **pick-witness**. For instance,

pick-witnesses I_1 I_2 **for** (`exists` x_1 x_2 `.` p) D

is an abbreviation for

pick-witness I_1 **for** (`exists` x_1 x_2 `.` p)
  pick-witness I_2 **for** (`exists` x_2 `.` $\{x_1 \mapsto I_1\}(p)$)
    D

Here is a sample proof that (`exists x y . x < y`) implies (`exists y x . x < y`):

```
> assume hyp := (exists x y . x < y)
    pick-witnesses w1 w2 for hyp # This gives (w1 < w2)
      let {_ := (!egen (exists x . x < w2) w1)}
        (!egen (exists y x . x < y) w2);;

Theorem: (if (exists ?x:Real
               (exists ?y:Real
                 (< ?x:Real ?y:Real)))
             (exists ?y:Real
               (exists ?x:Real
                 (< ?x:Real ?y:Real))))
```

Sometimes it is convenient to give a name to the witness hypothesis and then refer to it by that name inside the body of the **pick-witness**. This can be done by inserting a name (an identifier) before the body D of the **pick-witness**. That identifier will then refer to the witness premise inside D. For example, the proof

```
pick-witness w (exists x . x = x) wp D
```

will give the name `wp` to the witness premise, so that every free occurrence of `wp` within D will refer to the witness premise. Thus, for instance, one of our earlier proofs could be written as follows:

```
> assume hyp := (exists x . ~ prime x)
    pick-witness w for hyp -prime-w
    # We now have -prime-w := (~ P w) in the a.b.
      (!by-contradiction (~ forall x . prime x)
        assume all-prime := (forall x . prime x)
          let {prime-w := (!uspec all-prime w)}
            (!absurd prime-w -prime-w))

Theorem: (if (exists ?x:Int
                (not (prime ?x:Int)))
             (not (forall ?x:Int
                    (prime ?x:Int))))
```

This can also be done for **pick-witnesses** deductions; an identifier appearing right before the body will denote the witness premise. For example:

```
pick-witnesses w1 w2 for (exists x y . x =/= y) wp D.
```

Note that, intuitively, existential quantifications correspond to (potentially infinite) disjunctions. For instance, let `B` be a unary predicate on `Boolean`:

```
declare B: [Boolean] -> Boolean
```

To say that there is some boolean value for which `B` holds:

```
(exists ?x:Boolean . B ?x:Boolean))
```
(5.20)

is simply to say that `B` holds for `true` *or* `B` holds for `false`:

```
(B true | B false).
```
(5.21)

Sentences (5.20) and (5.21) have the exact same content. Likewise, to say that there is some integer that is prime:

```
(exists ?x:Int . prime ?x:Int)
```
(5.22)

is really the same as saying that 0 is prime *or* 1 is prime, *or* −1 is prime, *or* 2 is prime, *or* −2 is prime, and so on:

```
(prime 0 | prime 1 | prime (- 1) | prime 2 | ···).
```
(5.23)

Accordingly, we should expect existential elimination and introduction to intuitively correspond to disjunction elimination and introduction, respectively. Consider, for example, an instantiation of the existential claim that `B` holds for *some* boolean value:

```
pick-witness I for (exists x . B x)
  D
```

Such a proof could also be expressed as a disjunction elimination:

```
(!cases (B true | B false)
  assume (B true)
    D1
  assume (B false)
    D2)
```

The chief difference is that the first proof is more efficient because it does not consider every possible case separately. Rather, it abstracts what reasoning is common to D_1 and D_2 into what is essentially one single *method* parameterized over I. Also, the first proof does not use an explicit **`assume`**, as the insertion of the witness hypothesis into the assumption base is performed automatically. Thus, we can explain the evaluations of D_1 and D_2 as the evaluation of one and the same deduction, D, but under two different lexical bindings: with I denoting `true` in the first case and with I denoting `false` in the second. Indeed, if we only needed to work with *finite* domains, then we could formally define the evaluation semantics of **`pick-witness`** proofs by desugaring such proofs into `cases` applications. Such desugaring is not possible in the general case, which includes infinite domains, but nevertheless it is still instructive to keep the analogy between existential instantiation and disjunction elimination in mind.

We close by mentioning an alternative mechanism for existential instantiation, the syntax form

```
(with-witness E F D).
```

The expression E must produce an Athena variable v, which will be used as the witness variable of the existential instantiation. Accordingly, this variable must not have any free occurrences in the assumption base. The phrase F must produce an existential quantification `(exists` x `.` p`)`, which will serve as the existential premise and must therefore be in the assumption base. Finally, D is the body of the instantiation, which, when evaluated in the given assumption base augmented with the witness hypothesis $\{x \mapsto v\}(p)$, must yield a conclusion q that does not contain any free occurrences of v. The sentence q then becomes the conclusion of the entire existential instantiation.

This is, essentially, a more primitive version of **pick-witness**. It stands to **pick-witness** in the same relationship that **generalize-over** stands to **pick-any**. Specifically, we can desugar **pick-witness** *I* **for** *F D* as

```
let {I := (fresh-var)}
  (with-witness I F D)
```

5.3 Some examples

In this section we present some simple examples of quantifier reasoning illustrating the mechanisms introduced in the previous section. We first write each derived sentence in conventional notation, and then we show how to write and, more importantly, how to prove that sentence in Athena.

$$(\forall x \,.\, P(x) \wedge Q(x)) \Rightarrow (\forall y \,.\, P(y)) \wedge (\forall y \,.\, Q(y))$$

```
assume hyp := (forall x . P x & Q x)
  let {all-P := pick-any y:Object
                  conclude (P y)
                    (!left-and (!uspec hyp y));
       all-Q := pick-any y:Object
                  conclude (Q y)
                    (!right-and (!uspec hyp y))}
    (!both all-P all-Q)
```

$$(\exists x \,.\, P(x) \wedge Q(x)) \Rightarrow (\exists y \,.\, P(y)) \wedge (\exists y \,.\, Q(y))$$

```
assume hyp := (exists x . P x & Q x)
  pick-witness w for hyp wp # we now have wp := (P w & Q w) in the a.b.
    let {Pw := (!left-and wp);
         Qw := (!right-and wp);
         some-P := (!egen (exists y . P y) w);
         some-Q := (!egen (exists y . Q y) w)}
      (!both some-P some-Q)
```

$$((\forall x \,.\, P(x)) \vee (\forall x \,.\, Q(x))) \Rightarrow (\forall y \,.\, P(y) \vee Q(y))$$

```
assume hyp := ((forall x . P x) | (forall x . Q x))
  pick-any y
    (!cases hyp
            assume case-1 := (forall x . P x)
              conclude (P y | Q y)
                (!either (!uspec case-1 y) (Q y))
            assume case-2 := (forall x . Q x)
```

```
            conclude (P y | Q y)
              (!either (P y) (!uspec case-2 y)))
```

$$(\exists x \,.\, P(x) \vee Q(x)) \Rightarrow (\exists x \,.\, P(x)) \vee (\exists x \,.\, Q(x))$$

```
assume hyp := (exists x . P x | Q x)
  pick-witness w for hyp wp
  # we now have wp := (P w | Q w) in the a.b.
    (!cases (P w | Q w)
            assume (P w)
              let {some-P := (!egen (exists x . P x) w)}
                (!either some-P (exists x . Q x))
            assume (Q w)
              let {some-Q := (!egen (exists x . Q x) w)}
                (!either (exists x . P x) some-Q))
```

$$(\forall x \,.\, P(x) \Rightarrow Q(x)) \Rightarrow ((\forall x \,.\, P(x)) \Rightarrow (\forall x \,.\, Q(x)))$$

```
assume hyp1 := (forall x . P x ==> Q x)
  assume hyp2 := (forall x . P x)
    pick-any a
      conclude (Q a)
        let {Pa=>Qa := (!uspec hyp1 a);
             Pa := (!uspec hyp2 a)}
          (!mp Pa=>Qa Pa)
```

Note that, strictly speaking, the theorem produced by this proof is printed as

$$(\forall x \,.\, P(x) \Rightarrow Q(x)) \Rightarrow ((\forall x \,.\, P(x)) \Rightarrow (\forall a \,.\, Q(a)))$$

but this is of course equivalent (in fact identical, for deductive purposes) to:

$$(\forall x \,.\, P(x) \Rightarrow Q(x)) \Rightarrow ((\forall x \,.\, P(x)) \Rightarrow (\forall x \,.\, Q(x)))$$

We could easily produce the latter representation if we so prefer, either by using `x` in place of `a` in the above proof, or else by wrapping the present **`pick-any`** proof inside a **`conclude`** that is quantified over `x`:

```
assume hyp1 := (forall x . P x ==> Q x)
  assume hyp2 := (forall x . P x)
    conclude (forall x . Q x)
      pick-any a
        conclude (Q a)
          let {Pa=>Qa := (!uspec hyp1 a);
               Pa := (!uspec hyp2 a)}
            (!mp Pa=>Qa Pa)
```

$$((\exists x\,.\,P(x)) \vee (\exists x\,.\,Q(x))) \Rightarrow (\exists x\,.\,P(x) \vee Q(x))$$

```
assume hyp := ((exists x . P x) | (exists x . Q x))
  let {goal := (exists x . P x | Q x)}
    (!cases
      hyp
      assume case-1 := (exists x . P x)
        pick-witness w for case-1 # we now have (P w) in the a.b.
          let {Pw|Qw := (!either (P w) (Q w))}
            (!egen goal w)
      assume case-2 := (exists x . Q x)
        pick-witness w for case-2 # we now have (Q w) in the a.b.
          let {Pw|Qw := (!either (P w) (Q w))}
            (!egen goal w))
```

A closer look at this proof reveals some duplication of effort: The reasoning is essentially identical for both cases, yet it is repeated almost verbatim for each case separately. We can easily factor out the commonalities in a single method that we can then reuse accordingly. The savings afforded by such abstraction are minimal in this case (as the proof is so small to begin with), but it is nevertheless instructive to carry out the factoring anyway. In larger proofs the benefits can be more substantial:

```
assume hyp := ((exists x . P x) | (exists x . Q x))
  let {goal := (exists x . P x | Q x);
       M := method (ex-premise)
              assume ex-premise
                pick-witness w for ex-premise
                  let {Pw|Qw := (!either (P w) (Q w))}
                    (!egen goal w)}
    (!cases hyp (!M (exists x . P x))
                (!M (exists x . Q x)))
```

The next section presents several more examples of custom-written first-order methods.

Exercise 5.2: Each of the following listings asserts a number (possibly zero) of premises and defines a goal:

Listing 5.3.1

```
assert premise-1 := (forall x y . x R y)

define goal := (forall x . x R x)
```

Listing 5.3.2

```
define goal := (forall x . exists y . x = y)
```

Listing 5.3.3

```
assert premise-1 := (forall x . P x | Q x ==> S x)
assert premise-2 := (exists y . Q y)

define goal := (exists y . S y)
```

Listing 5.3.4

```
assert premise-1 := (exists x . P x & Q x)
assert premise-2 := (forall y . P y ==> S y)

define goal := (exists x . S x & Q x)
```

Listing 5.3.5

```
assert premise-1 := (~ exists x . Q x)
assert premise-2 := (forall x . P x ==> Q x)

define goal := (~ exists x . P x)
```

Listing 5.3.6

```
assert premise-1 := (forall x . P x ==> Q x)
assert premise-2 := (exists x . S x & ~ Q x)

define goal := (exists x . S x & ~ P x)
```

Listing 5.3.7

```
assert premise-1 := (forall x . x R x ==> P x)
assert premise-2 := (exists x . P x ==> ~ exists y . Q y)

define goal := ((forall x . Q x) ==> ~ exists z . z R z)
```

Listing 5.3.8

```
define goal :=
 ((exists x . P x | Q x) <==> (exists x . P x) | (exists x . Q x))
```

Derive each goal from the corresponding premises. □