CSCI.6962/4962 Software Verification—Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Chapter 5

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

First-Order logic

Goal: to become familiar with *first-order logic* proofs.

- universal quantifications
- existential quantifications
- proof libraries
- methods for quantifier reasoning
- proof heuristics

Quantified sentences

In addition to the sentential logic sentences, we now have *quantified sentences*:

For any variable v and sentence p,

$$\texttt{(forall } v \texttt{ . } p\texttt{)} \tag{1}$$

and

$$\texttt{(exists } v \texttt{ . } p\texttt{)} \tag{2}$$

are also legal first-order sentences. We refer to p as the *body* of (1) and (2).

- Sentences of the form (1) state that *every* object of a certain sort has property p;
- Sentences of the form (2) state that *some* object (of a certain sort) has property p.

Quantified sentences

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).
```

We can quantify over multiple variables with only one quantifier, e.g.:

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

as a shorthand for

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

Quantified sentences

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.

Using universal quantifications

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

$$\texttt{(forall x . x < x + 1)} \tag{3}$$

says that every integer is less than its successor.

Hence, if we know that (3) holds, we should be able to conclude that any *particular* integer is less than its successor, e.g., 5:

$$\texttt{(5 < 5 + 1)} \tag{4}$$

or 78:

$$\texttt{(78 < 78 + 1).} \tag{5}$$

We say that the conclusions (4) and (5) are obtained from (3) by *universal specialization*, or *universal instantiation*.

Using universal quantifications

Universal specialization is performed by the binary method uspec.

- The first argument to uspec is the universal quantification p; which must be in the assumption base.
- 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))
```

Using universal quantifications

The instantiating term does not have to be ground, e.g.:

```
> (!uspec p (2 * x))

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

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.

Using universal quantifications

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.

For example, consider the polymorphic list reverse property:

```
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.
```

Using universal quantifications

The instantiating term might be a list of Boolean terms, or a list of integers, or a polymorphic variable:

```
> (!uspec p (false::nil))
Theorem: (= (reverse (reverse (:: false
                                   nil:(List Boolean))))
                (:: 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))
```

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

  $$\texttt{pick-any}\ \ I\ \ D.$$

Deriving universal quantifications

$$\texttt{pick-any}\ I\ \ D. \tag{6}$$

We refer to D as the *body* of (6).

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 (6).

Deriving universal quantifications

To make things concrete, consider as an example the deduction

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

Recall that `reflex` is a unary primitive method that takes any term t and produces the equality $(t = t)$.

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

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

Deriving universal quantifications

A proof that the equality relation is symmetric follows.

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:

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

Theorem: (forall ?a:'S
           (forall ?b:'S
             (if (= ?a:'S ?b:'S)
                 (= ?b:'S ?a:'S))))
```

Deriving universal quantifications

Proving $(\forall x.(P\ x) \wedge \forall x.(Q\ x)) \implies \forall y.((P\ y) \wedge (Q\ y))$:

```
define [all-P all-Q] := [(forall x . P x) (forall x . Q x)]
> conclude (all-P & all-Q ==> forall y . P y & Q y)
  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))))
```

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`)`. This type of reasoning is known as *existential generalization*.

Deriving existential quantifications

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`)`.

- The second argument is a term t on the basis of which we are to infer `(exists` x `.` p`)`.

Specifically, if $\{x \mapsto t\}(p)$ is in the assumption base, then the call

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

will derive the theorem `(exists` x `.` p`)`.

Deriving existential quantifications

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))
```

Using existential quantifications

- Suppose that we know 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*, a.k.a., *existential specialization* or *existential elimination*.

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, that is, let v be a "witness" for the existential sentence $\exists\ x\ .\ p$, so that $\{x \mapsto v\}(p)$ can be assumed to hold. Then $\cdots D \cdots$

Using existential quantifications

- 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.

Using existential quantifications

Consider proving that for all integers n, *if* $\mathit{even}(n)$ *then* $\mathit{even}(n+2)$, given the following axioms:

$$(\forall\, i \,.\, \mathit{even}(i) \Leftrightarrow \exists\, j \,.\, i = 2 \cdot j) \tag{7}$$

$$(\forall\, x\; y \,.\, x \cdot (y+1) = x \cdot y + x) \tag{8}$$

Proof:

Pick any n and assume $\mathit{even}(n)$. Then, by (7), 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* (8), $(2 \cdot k) + 2 = 2 \cdot (k+1)$, *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* (7), *we conclude* $\mathit{even}(n+2)$.

Using existential quantifications

Existential instantiations are performed by deductions of the form

$$\texttt{pick-witness}\ I\ \texttt{for}\ F\ D$$

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.

Using existential quantifications

$$\texttt{pick-witness}\ I\ \texttt{for}\ F\ D \tag{9}$$

To evaluate (9) in an assumption base β,

1. we check that the existential premise (`exists` $x:S$. p) is in β
2. we generate a fresh variable $v:S$, which will serve as the actual witness variable.
3. we then construct the witness hypothesis, call it p', obtained from p by replacing every free occurrence of $x:S$ by the witness $v:S$.
4. 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 $v:S$.

Using existential quantifications

5. if and when that evaluation produces a conclusion q, we return q as the result of the entire proof (9), provided that q does not contain any free occurrences of $v : S$ (it is an error if it does).

Notes:

- 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.
- I in (9) is not an Athena term variable; it is a name—an identifier—that will come to *denote* a fresh variable (the witness variable $v : S$) in the course of evaluating the body D.

Using existential quantifications

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))))
```

Using existential quantifications

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))))
```

Using existential quantifications

- 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 `(w = w)`.

Using existential quantifications

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))))
```

Example 1

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

```
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)
```

Example 2

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

```
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))
```

Example 2

We can abstract over each case's reasoning with a method:

```
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)))
```

Proof libraries

- The four introduction and elimination mechanisms for quantifiers that we have discussed so far (the methods `uspec` and `egen` and the deduction forms `pick-any` and `pick-witness`), in tandem with the introduction and elimination mechanisms for the sentential connectives presented in Chapter 4, constitute a complete proof system for first-order logic.
- That is, if any sentence p follows logically from an assumption base β, then there is some proof D composed from these mechanisms that can derive p from β.
- However, if we had to limit ourselves to these primitive mechanisms when writing proofs, our job would be much more difficult than it needs to be.

Sentential reasoning library examples

Method	Evaluation sample	Derives
`contra-pos`	`(!contra-pos (`p` ==> `q`))`	`(~ `q` ==> ~ `p`)`
`ex-middle`	`(!ex-middle `p`)`	`(`p` | ~ `p`)`
`comm`	`(!comm (`p` & `q`))`	`(`q` & `p`)`
`bdn`	`(!bdn (~ ~ `p`))`	p
`from-false`	`(!from-false `p`)`	p
`f-complements`	`(!from-complements `p q $\overline{q}$`)`	p
`hsyl`	`(!hsyl (`p_1` ==> `p_2`) (`p_2` ==> `p_3`))`	`(`p_1` ==> `p_3`)`
`dsyl`	`(!dsyl (`p` | `q`) `$\overline{p}$`)`	q
`dm`	`(!dm (~ (`p` & `q`)))`	`(`$\overline{p}$` | `$\overline{q}$`)`
`dm`$'$	`(!dm' (~ (`p` & `q`)))`	`(~ `p` | ~ `q`)`

Sentential reasoning library examples

Method	Evaluation sample	Derives
dist	$(\texttt{!dist}\ (p\ \&\ (q \mid r)))$	$(p\ \&\ q \mid p\ \&\ r)$
dist	$(\texttt{!dist}\ (p \mid (q\ \&\ r)))$	$((p \mid q)\ \&\ (p \mid r))$
cond-def	$(\texttt{!cond-def}\ (p \texttt{ ==> } q))$	$(\sim p \mid q)$
cond-def	$(\texttt{!cond-def}\ (\sim p \texttt{ ==> } q))$	$(p \mid q)$
cond-def$'$	$(\texttt{!cond-def'}\ (\sim p \texttt{ ==> } q))$	$(\sim\ \sim p \mid q)$
neg-cond	$(\texttt{!neg-cond}\ (\sim (p \texttt{ ==> } q)))$	$(p\ \&\ \sim q)$
bicond-def	$(\texttt{!bicond-def}\ (p \texttt{ <==> } q))$	$(p \texttt{ ==> } q)\ \&\ (q \texttt{ ==> } p)$
bicond-def$'$	$(\texttt{!bicond-def'}\ (p \texttt{ <==> } q))$	$(p\ \&\ q \mid \overline{p}\ \&\ \overline{q})$
n-bicond	$(\texttt{!negated-bicond}$ $(p \texttt{ <==> } q))$	$(p\ \&\ \sim q \mid \sim p\ \&\ q)$

Recall sentential logic example

```
assert premise-1 := (A & B | (A ==> C))
assert premise-2 := (C <==> ~ E)

assume -B := (~ B)
  assume A
    conclude -E := (~ E)
      (!cases premise-1
              assume (A & B)
                (!from-complements -E B -B)
              assume A=>C := (A ==> C)
                let {C=>-E := (!left-iff premise-2);
                     C     := (!mp A=>C A)}
                  (!mp C=>-E C))
```

Using sentential reasoning library

```
assert premise-1 := (A & B | (A ==> C))
assert premise-2 := (C <==> ~ E)
assume (~ B)
  assume A
    conclude (~ E)
    let {notA&B := (!neither (~ A) (~ B));
         A=>C   := (!dsyl premise-1 notA&B);
         C=>-E  := (!left-iff premise-2);
         A=>-E  := (!hsyl A=>C C=>-E)}
      (!mp A=>-E A)
```

where `neither` is a new method that infers `(~ (`p` & `q`))` assuming `(~ `p`)` or `(~ `q`)` is in the assumption base β:

```
define (neither notP notQ) :=
  match [notP notQ] {
    [(~ p) (~ q)] => (!dm (!either (~ p) (~ q)))
  }
```

Using sententional reasoning library

Or inlining all internal deductions:

```
assert premise-1 := (A & B | (A ==> C))
assert premise-2 := (C <==> ~ E)

assume (~ B)
  assume A
    conclude (~ E)
    (!mp (!hsyl (!dsyl premise-1
                          (!neither (~ A) (~ B)))   # derives (~ (A & B))
                (!left-iff premise-2))
         A)
```

Methods for quantifier reasoning

Multiple universal specialization and existential generalization:

- Suppose that we have a premise p with $k \geq 0$ universal quantifiers at the front.
- We often want to perform universal specialization on p in one fell swoop, with a *list* of k terms $[t_1 \cdots t_k]$, instead of having to apply uspec k separate times.
- That functionality can be programmed as a recursive method:

```
define uspec* :=
  method (premise terms)
    match terms {
      [] => (!claim premise)
    | (list-of t rest) => (!uspec* (!uspec premise t) rest)
    }
```

Methods for quantifier reasoning

Multiple universal specialization and existential generalization:

An example of `uspec*` in action:

```
assert premise := (forall x y . x = y ==> y = x)

> (!uspec* premise [1 2])
Theorem: (if (= 1 2)
             (= 2 1))
```

The method can accept a list of fewer than k terms, e.g.:

```
assert <-transitivity := (forall x y z . x < y & y < z ==> x < z)

> (!uspec* <-transitivity [1.7 2.9])
Theorem: (forall ?v1805:Real
           (if (and (< 1.7 2.9)
                    (< 2.9 ?v1805:Real))
               (< 1.7 ?v1805:Real)))
```

Note that `uspec*` is also known as `instance`.

Methods for quantifier reasoning

Multiple universal specialization and existential generalization:

- The ability to existentially generalize over multiple terms in one step is likewise possible with the method `egen*`.
- For instance, if we have `(1 < 2)` in the assumption base, we can derive `(exists x y . x < y)` in one step, simply by citing the terms `1` and `2`:

```
(!egen* (exists x y . x < y) [1 2]).
```

- The order of the existential quantifiers corresponds to the order in which the terms are listed, meaning that the generalization over `x` is to be based on `1`, while the generalization over `y` is based on `2`.

Methods for quantifier reasoning

Multiple universal specialization and existential generalization:

In general, for $k > 0$,

`(!egen* (exists` $x_1 \cdots x_k$ `.` p`)` `[`$t_1 \cdots t_k$`])`

derives the conclusion `(exists` $x_1 \cdots x_k$ `.` p`)`, provided that

$$\{x_k \mapsto t_k\}(\cdots \{x_1 \mapsto t_1\}(p) \cdots)$$

is in the assumption base (an error occurs otherwise).

Methods for quantifier reasoning

Forward Horn clause inference:

In practice, the most useful—and common—universal quantifications are of the form

$$\texttt{(forall } x_1 \cdots x_n \texttt{ . } p \texttt{ ==> } q\texttt{)} \tag{10}$$

and

$$\texttt{(forall } x_1 \cdots x_n \texttt{ . } p \texttt{ <==> } q\texttt{).} \tag{11}$$

A few examples:

```
(forall x y . x = y ==> y = x);                      (12)
(forall x y . x parent y ==> x ancestor y);          (13)
(forall x y . x > 0 & y > 0 ==> x - y < x);          (14)
(forall x y . x <= y <==> x = y | x < y).            (15)
```

Methods for quantifier reasoning

Forward Horn clause inference:

- Sentences of form (10) are called *Horn clauses*. We will also refer to them as Horn *rules*, or, when there is no risk of confusion, simply as "rules."
- Note that a sentence of the second form, (11), can be regarded as the conjunction of the following two Horn clauses:

$$\texttt{(forall } x_1 \cdots x_n \texttt{ . } p \texttt{ ==> } q\texttt{)} \tag{16}$$

and

$$\texttt{(forall } x_1 \cdots x_n \texttt{ . } q \texttt{ ==> } p\texttt{).} \tag{17}$$

Methods for quantifier reasoning

Forward Horn clause inference:

- One of the most common things that we want to do with a Horn rule of the form (10) is to apply it (or to "fire" it, in the terminology of rule systems) on some specific terms $t_1, \ldots, t_n$, that is, to derive the appropriate instance of the conclusion q, given that *the corresponding instance of the antecedent* p *holds*.
- For instance, suppose that we know that `peter` is a parent of `mary`, so that the atom

  ```
  (peter parent mary)
  ```

  is in the assumption base. It then becomes evident that rule (13) is applicable, and specifically that we can use it to infer the conclusion `(peter ancestor mary)`.
- This is called "firing" (13) on the terms `peter` and `mary`.

Methods for quantifier reasoning

Forward Horn clause inference:

- This type of inference with Horn rules is also called *forward*, because we proceed from the antecedent of (an instance of) the rule to the consequent.
- Thus, “firing” a Horn rule

$$\texttt{(forall } x_1 \cdots x_n \texttt{ . } p \texttt{ ==> } q\texttt{)}$$

  proceeds in two stages.
  - First, a list of terms $\texttt{[}t_1 \cdots t_n\texttt{]}$ is used to specialize the universal quantification, with each t_i replacing x_i, $i = 1, \ldots, n$.
  - Then, we perform modus ponens on the instantiated rule and its antecedent.

Methods for quantifier reasoning

Forward Horn clause inference:

For instance, the above example can be achieved by:

```
domain Person

declare parent, ancestor: [Person Person] -> Boolean

declare peter, mary: Person

assert ancestor-rule := (forall x y . x parent y ==> x ancestor y)

assert fact := (peter parent mary)

let {rule-instance :=
       conclude (peter parent mary ==> peter ancestor mary)
         (!uspec* ancestor-rule [peter mary])}
  (!mp rule-instance fact)
```

Methods for quantifier reasoning

Forward Horn clause inference:

This is what we do with a single call of the form

(!fire R [$t_1 \cdots t_n$]),

where R is a Horn rule and $t_1 \cdots t_n$ are arbitrary terms of the proper sorts.

In the above example, the call

```
(!fire ancestor-rule [peter mary])
```

derives `(peter ancestor mary)`—provided that the precondition

```
(peter parent mary)
```

is in the assumption base.

Quantifier reasoning library examples

Method	Evaluation sample	Derives
`qn`	`(!qn (~ forall x .` p `x))`	`(exists x .` $\overline{p}$ `x).`
`qn`	`(!qn (~ exists x .` p `x))`	`(forall x .` $\overline{p}$ `x).`
`quant-swap`	`(!quant-swap (forall x y . x` r `y))`	`(forall y x . x` r `y)`
`quant-dist`	`(!quant-dist` $\forall\, x \,.\, p_1 \wedge p_2$`)`	$(\forall\, x \,.\, p_1) \wedge (\forall\, x \,.\, p_2)$
`quant-dist`	`(!quant-dist` $\exists\, x \,.\, p_1 \vee p_2$`)`	$(\exists\, x \,.\, p_1) \vee (\exists\, x \,.\, p_2)$
`quant-dist`	`(!quant-dist` $\exists\, x \,.\, p_1 \wedge p_2$`)`	$(\exists\, x \,.\, p_1) \wedge (\exists\, x \,.\, p_2)$
`quant-dist`	`(!quant-dist` $(\exists\, x \,.\, p_1) \vee (\exists\, x \,.\, p_2)$`)`	$\exists\, x \,.\, p_1 \vee p_2$
`transform`	`(!transform` $\forall\, x \,.\, p \quad \forall\, y \,.\, q$ `[`$M_1 \cdots M_n$`])`	$\forall\, y \,.\, q$

Example of replacement rule with quantifiers

Consider, for instance, the following two sentences:

```
define p :=  (forall x . exists y . x R y ==> ~ (P x & Q y))

define q := (forall z . exists w . ~ ~ z R w ==> ~ P z | ~ Q w)
```

We can infer q from p, and vice versa, simply by citing the two bidirectional methods bdn and dm.

For instance, assuming that p is in the assumption base,

```
(!transform p q [bdn dm])
```

should produce q.

Conversely, if q is in the assumption base,

```
(!transform q p [bdn dm])
```

should produce p.

Proof heuristics for first-order logic

- We will now expand the collection of tactics for sentential logic with some new entries:
  - two backward tactics for introducing quantifiers and
  - some forward tactics for eliminating them.
- We will also slightly extend the notion of a proof spec. Instead of "Derive p from β," the new format for a proof spec will be

  $$\text{Derive } p \text{ from } (\beta, \boldsymbol{T})$$

  where $\boldsymbol{T}$ is a finite set of terms, which we call the *proof terms*.
- Writing first-order proofs often requires choosing appropriate terms with which to specialize universal quantifications (via uspec), or from which to existentially generalize (via egen).

Proof heuristics for first-order logic

We choose for proof terms the set of all terms t such that:

1. t occurs in the assumption base or in the goal; unless the said occurrence of t also contains bound variable occurrences.
2. t is a fresh variable introduced by a universal generalization (a `pick-any`), or a witness variable introduced by an existential instantiation (a `pick-witness`).

Backward tactics for quantifiers

The following is the backward tactic for universal quantifications:

$\boxed{\text{Derive } (\texttt{forall } v \texttt{ . } p) \text{ from } (\beta, \boldsymbol{T})}$

pick-any I

(!**force2** $\{v \mapsto I\}(p)$ $\boldsymbol{T} \cup \{I\}$) [forall<-]

- Informally, this tactic can be read as follows:
  - To derive a universal quantification (forall v . p),
  - pick a fresh variable named I and
  - attempt to find a proof that derives $\{v \mapsto I\}(p)$.
- Note that the variable denoted by I becomes a member of the set of available proof terms in the new subgoal.

Backward tactics for quantifiers

The following is the backward tactic for existential quantifications:

$\boxed{\text{Derive } \texttt{(exists } v \texttt{ . } p\texttt{)} \text{ from } (\beta, \boldsymbol{T} = \{\ldots, t, \ldots\})}$

let $\{$_ := (!**force2** $\{v \mapsto t\}(p)$ $\boldsymbol{T}$)$\}$ [exists<-]
  (!egen (exists v . p) t)

To derive an existentially quantified sentence

$$\texttt{(exists } v \texttt{ . } p\texttt{)}, \qquad (18)$$

- choose a proof term t and
- try to show that p holds for t, i.e., try to derive $\{v \mapsto t\}(p)$.
- If successful, we can infer (18) by existential generalization.

There may be several proof terms available (nondeterministic choice).

Forward tactics for quantifiers

We start with a tactic for eliminating existential quantifiers:

$$\boxed{\text{Derive } r \text{ from } (\beta = \{\ldots, (\cdots\,\texttt{(exists}\ v\ \texttt{.}\ q\texttt{)}^{+}_{*}\cdots),\ldots\}, \boldsymbol{T})}$$

let $\{$_ := (!**force2** (exists v . q) $\boldsymbol{T}$)$\}$ [exists->]
  pick-witness I for (exists v . q)
    (!**force2** r $\boldsymbol{T} \cup \{I\}$)

One of the highest-priority tactics, it advises us to eliminate existential quantifiers:

- If we see an existential quantification positively embedded in a universal position in the assumption base,
- derive it and then
- “unpack” it—eliminate the existential quantifier (via `pick-witness`).

Forward tactics for quantifiers

Universal quantifications that are negatively embedded in the assumption base are essentially existential quantifications, and should also be unpacked as soon as possible.

Thus, in a sense, the following tactic is the dual of `[exists->]` :

$$\boxed{\text{Derive } r \text{ from } (\beta = \{\ldots, (\cdots\texttt{(forall } v \texttt{ . } q\texttt{)}^{-}_{*}\cdots), \ldots, \boldsymbol{T})}$$

$$
\begin{array}{ll}
\texttt{let } \{\texttt{p1 := (!}\textbf{force2}\texttt{ (\~{} forall } v \texttt{ . } q\texttt{) } \boldsymbol{T}\texttt{);} & \texttt{[exists2->]} \\
\quad\quad \texttt{p2 := (!qn p1)}\} & \\
\quad \texttt{pick-witness } I \texttt{ for p2} & \\
\quad\quad \texttt{(!}\textbf{force2}\ r\ \boldsymbol{T} \cup \{I\}\texttt{))} &
\end{array}
$$

Forward tactics for quantifiers

- Finally, we introduce a forward tactic involving universal quantifiers.
- It recommends specializing a universal quantification in the assumption base with some available proof term:

$$\boxed{\text{Derive } r \text{ from } (\beta = \{\ldots, (\cdots \texttt{(forall } v \texttt{ . } q\texttt{)}^{+}_{*} \cdots), \ldots\}, \boldsymbol{T} = \{\ldots, t, \ldots\})}$$

let $\{$p := (!**force2** (forall v . q) $\boldsymbol{T}$); [forall->]
_ := (!uspec p t)$\}$
(!**force2** r $\boldsymbol{T}$)

Forward tactics for quantifiers

- The dual tactic is applicable when we have an existential quantification negatively embedded in the assumption base.
  - We then try to derive the negation of the existential sentence,
  - transform it into a universal quantification by moving the negation sign inward, and
  - specialize it:

$$\boxed{\text{Derive } r \text{ from } (\{\ldots, (\cdots \texttt{(exists } v \texttt{ . } q\texttt{)}^{-}_{*} \cdots), \ldots\}, \boldsymbol{T} = \{\ldots, t, \ldots\})}$$

```
let {p1 := (!force2 (~ exists v . q) T);          [forall2->]
     p2 := (!qn p1);
     _ := (!uspec p2 t)}
  (!force2 r T)
```

Forward tactics for quantifiers

One new extraction tactic dealing with universal quantifiers:

$\boxed{\text{Derive } r \text{ from } (\beta = \{\ldots, p = (\cdots \texttt{(forall } v \texttt{ . } q\texttt{)}^{+}_{*} \cdots), \ldots\}, \boldsymbol{T})}$

let $\{$lemma := (!**force2** $\theta($(forall v . q)$)$ $\boldsymbol{T}$);

 # where r properly matches q in p under θ

(!uspec lemma $\theta(v)$) [forall3->]

This new tactic treats any universal quantification (forall v . q) as a generalized parent of all substitution instances of its body q. Its dual is:

$\boxed{\text{Derive } r \text{ from } (\beta = \{\ldots, (\cdots \texttt{(exists } v \texttt{ . } q\texttt{)}^{-}_{*} \cdots), \ldots\}, \boldsymbol{T})}$

let $\{$p1 := (!**force2** $\theta($(~ exists v . q)$)$ $\boldsymbol{T}$); [forall4->]

p2 := (!qn p1)$\}$

 # where r properly matches $\overline{q}$ in p under θ

(!uspec $\theta($(forall v . $\overline{q}$)$)$ $\theta(v)$)

Proof strategy for first-order logic

The overall strategy for deploying tactics carries over unchanged from sentential logic, with two minor additions.

- First, we now try to eliminate existential quantifiers if no extraction tactics are applicable.
- Second, we try the unrestricted universal instantiation tactic `[forall->]` and its dual `[forall2->]` before we try proof by contradiction.

Proof strategy for first-order logic

Thus, the general ranking is now as follows:

1. reiteration (`[claim->]`) and constant tactics (`[true<-]` and `[false<-]`);
2. the complement tactic (`[cft]`);
3. extraction tactics, including `[forall3->]` and `[forall4->]` ;
4. existential instantiation tactics (`[exists->]` and `[exists2->]`);
5. replacement tactics;
6. backward tactics (including `[forall<-]`), with the exception of `[not<-]`;
7. generalized disjunction tactic;
8. indirect tactic, `[not<-]`, or the negation heuristic.