6 Implication Chaining

IN THIS chapter we extend the chaining style of proof from equational logic to full first-order logic. Proofs expressed in this style tend to be particularly readable and will be used widely throughout the rest of the book.

6.1 Implication chains

Recall the general form of a proof by equational chaining:

$$\begin{array}{lll} \texttt{(!chain [}t_1 \texttt{ =} & t_2 & J_1 \\ \texttt{ =} & t_3 & J_2 \\ & \vdots & \\ \texttt{ =} & t_{n+1} & J_n\texttt{])}. \end{array}$$

The goal here is to "connect" the starting term t_1 with the final term t_{n+1} through the identity relation, that is, to derive

$$t_1 = t_{n+1}. \tag{6.1}$$

This is done in a number of steps $n > 0$, where each step derives an intermediate identity $t_i = t_{i+1}$, $i = 1, \ldots, n$, by citing J_i as its justification (we do not need to be concerned here with the exact nature of that justification). Provided that each step in the chain goes through, the desired result (6.1) finally follows from the transitivity of the identity relation. Ultimately, it is this transitivity that makes equational chaining work, and it is this transitivity, along with the inherently tabular format of `chain`, that makes these proofs so perspicuous.

But identity is not the only important logical relation that is transitive. Implication and equivalence are also transitive. Capitalizing on that observation, we can extend the chaining style of proof from identities to implications and equivalences, reaping similar benefits in notational clarity.

Let us consider implications first. The general format here is very similar to equational chaining, except that the chain links are now sentences p_i rather than terms t_i, and the symbol `==>` takes the place of `=`:

$$\begin{array}{lll} \texttt{(!chain [}p_1 \texttt{ ==>} & p_2 & J_1 \\ \texttt{ ==>} & p_3 & J_2 \\ & \vdots & \\ \texttt{ ==>} & p_{n+1} & J_n\texttt{])}. \end{array}$$

The larger idea remains the same: the goal is to "connect" the starting point p_1 with the end point p_{n+1}, this time through the implication relation, that is, to derive

$$p_1 \Rightarrow p_{n+1}. \tag{6.2}$$

This is done in a number of steps $n > 0$, where each step derives an intermediate implication $p_i \Rightarrow p_{i+1}$, $i = 1, \ldots, n$, by citing J_i as its justification (we will discuss shortly exactly what kinds of justification are appropriate here; for now we only need to keep in mind that each step in the chain must include some justification item, whatever that might be, just like equational chaining). Provided that each step in the chain goes through, the desired conditional (6.2) ultimately follows from the transitivity of implication.

The justification J_i for a step p_i `==>` p_{i+1} typically consists of a unary method M that can be applied to p_i in order to produce p_{i+1}. That is, M is a method that can derive the right-hand side of the step from the left-hand side. More precisely, M is such that *if* p_i is in the assumption base, then (`!`M p_i) will produce p_{i+1}. Here is an example:

```
> (!chain [(A & B) ==> A left-and])

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

As with equational chaining, we prefer to enclose justification items inside lists, so we would typically write the above as follows:

```
> (!chain [(A & B) ==> A [left-and]])

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

The reason for the list notation is that a justification item may consist of more than one method (and it may also include sentences, as we will soon see), so in the general case we need to group all of the given entries—methods and/or sentences—in a list.

Here is a more interesting example:

```
> (!chain [(A & ~~ B) ==> (~~ B) [right-and]
                      ==> B      [dn]])

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

This chain has two steps. On the first step we use `right-and` to derive `(~ ~ B)` from

`(A & ~ ~ B)`,

on the assumption that the latter holds; and on the second step we derive `B` from `(~ ~ B)` by `dn`.

Note that even though an implication `chain` produces a conditional conclusion, there is no explicit use of **`assume`** anywhere. It is the implementation of `chain` that uses **`assume`**, not its clients. Specifically, the implementation starts by assuming the first sentence of the chain, p_1, and proceeds to derive the second element of the chain, p_2, by using the given method(s). If successful, this produces the conclusion $p_1 \Rightarrow p_2$. Then the same process is repeated for the next pair of the chain: The conclusion $p_2 \Rightarrow p_3$ is derived, by assuming p_2 and inferring p_3 through the given methods. We continue in that fashion, until finally transitivity is used to produce the output $p_1 \Rightarrow p_{n+1}$.

Anonymous methods can appear inline in the justification list of a given step:

```
> (!chain [(A & ~~ B) ==> B [method (p) (!dn (!right-and p))]])

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

> (!chain [(forall ?x . ?x = ?x) ==> (1 = 1) [method (p) (!uspec p 1)]])

Theorem: (if (forall ?x:'S
               (= ?x:'S ?x:'S))
             (= 1 1))
```

In the first chain, **`method`** `(p) (!dn (!right-and p))` was applied to the hypothesis `(A & ~ ~ B)` to produce the conclusion `B` in one step. Inlining very small anonymous methods inside a chain step like that may be acceptable, but readability is compromised if the method is more complicated. Even the first of the two examples above would arguably be cleaner if the definition of the method were pulled out of the chain:

```
> let {M := method (p) (!dn (!right-and p))}
    (!chain [(A & ~~ B) ==> B [M]])

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

The justification list of a given step p_i `==>` p_{i+1} may contain extraneous methods that fail to derive p_{i+1} from p_i. The implementation of `chain` will try every given method until one succeeds, disregarding those that fail. For instance, if we add superfluous information to each step of the previous example, the proof will still work :

```
> (!chain [(A & ~~ B) ==> (~~ B) [right-and dn mp]
                      ==> B      [left-and dn]])

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

This is a useful feature because oftentimes we know that a certain *set* of resources (methods and/or premises) justify a given step, but it would be onerous to specify *exactly which* elements of that set are needed for the step; we prefer to have Athena make that determination for us.[1]

A justifying method for a step p_i `==>` p_{i+1} need not be unary, taking p_i as its only argument and deriving p_{i+1}. Occasionally it makes sense to feed both the left-hand side premise p_i *and* the goal p_{i+1} as two distinct arguments to a justifying method. Methods of either type (unary or binary) are acceptable as justifications for an implication step. When a binary method is used, `chain` passes it the premise p_i as its first argument and the goal p_{i+1} as its second argument. For instance:

```
> let {M := method (premise goal) (!right-and premise)}
    (!chain [(A & ~~ B) ==> (~~ B) [M]
                        ==> B      [dn]])

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

In this simple example the binary method `M` ignores its second argument, the goal, because we already (statically) know what that goal is—it's the second conjunct of the premise! But in more complex scenarios we may not know the goal ahead of time, and in such cases it can be helpful to make the justifying method binary, giving it access to the premise as well as to the goal.

For example, we often want to take steps p_i `==>` p_{i+1} that pass from p_i to an arbitrarily complicated conjunction p_{i+1} that contains p_i as one of its conjuncts, while all the other conjuncts of p_{i+1} are already known or assumed to hold. We want a general-purpose method, call it `augment`, that can justify such steps, so that we can write, for instance:

```
assert A, B

> (!chain [C ==> (A & B & C) [augment]])

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

> (!chain [C ==> (C & B) [augment]])

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

1 In fact this feature is not just useful but necessary, because, as we will see shortly, sometimes one justifier is necessary for one part of the implication step and another justifier is needed for another part of the same step (such cases arise often in structural implication steps; see Section 6.3). So the implementation of `chain` must be able to accept an arbitrarily large list of justifiers and automatically determine which of these may be used where.

```
> (!chain [C ==> (and A A B C A B) [augment]])

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

The natural way to define `augment` is as a binary method that takes p_i as its first argument and the conjunction p_{i+1} as its second argument and then derives p_{i+1} by `conj-intro`:[2]

```
define augment :=
  method (premise conjunctive-goal)
    (!conj-intro conjunctive-goal)
```

Another example is the method `existence`, which allows us to pass from any sentence p to an existential generalization of p, that is, to a sentence of the form `(exists` $v_1 \cdots v_n$ `.` q`)` such that p can be obtained from q by substituting appropriate terms for the free occurrences of the variables $v_1 \cdots v_n$. For instance:

```
domain Person
declare likes: [Person Person] -> Boolean
declare Mary, Peter: Person

> (!chain [(Mary likes Peter) ==>
           (exists x . x likes Peter) [existence]])

Theorem: (if (likes Mary Peter)
             (exists ?x:Person
               (likes ?x:Person Peter)))

> (!chain [(zero < S zero) ==> (exists x y . x < y) [existence]])

Theorem: (if (< zero
                (S zero))
             (exists ?x:N
               (exists ?y:N
                 (< ?x:N ?y:N))))
```

The most straightforward way to implement `existence` is as a binary method that takes the premise p as its first argument and the desired existential generalization as its second argument:

```
define existence :=
  method (premise eg-goal)
    match (match-sentences premise (quant-body eg-goal)) {
```

2 Recall (footnote 16) that `conj-intro` is a unary method that takes as input an arbitrarily complicated conjunction p and derives p, provided that all of p's conjuncts are in the assumption base. It can be regarded as a generalization of both.

```
    (some-sub sub) => (!egen* eg-goal (sub (qvars-of eg-goal)))
  }
```

(The procedure `quant-body` returns the body of a quantified sentence, while `qvars-of` takes a quantified sentence of the form $(Q\ v_1 \cdots v_n\ .\ p)$, where p does not start with the quantifier Q, and returns the list of variables $[v_1 \cdots v_n]$. Both are defined in Athena's library. Finally, recall Exercise 5.3 for the definition of `egen*`.) Users can implement their own methods for implication steps, with either unary or binary interfaces.

Sometimes an implication step p_i `==>` p_{i+1} can only go through if we combine the premise p_i with additional, earlier information, namely, with sentences that were previously asserted, assumed, or derived.[3] Suppose, for example, that we know `(A ==> B)` to be the case, say, because we have asserted it. Then, intuitively, we should be able to write something like the following:

```
assert (A ==> B)

(!chain [(~ B) ==> (~ A) M])
```

The question now is what sort of justification method M could work here. It is clear that modus tollens (`mt`) is involved, but the issue seems to be that M only has access, via its arguments, to local information, namely, to the left-hand side of the step (or, if we express M as a binary method, to the right-hand side of the step as well). So how can we make M take `(A ==> B)` into account?

The answer lies in a sort of partially evaluated application of modus tollens, whereby we define M so that `(A ==> B)` already appears inside its body:

```
1 assert A=>B := (A ==> B)
2
3 > (!chain [(~ B) ==> (~ A) [method (p) (!mt A=>B p)]])
4
5 Theorem: (if (not B)
6              (not A))
```

It should be clear that

```
method (p) (!mt A=>B p)
```

is a unary method which, when applied to the left-hand side of the implication step on line 3, namely `(~ B)`, will successfully derive the right-hand side, `(~ A)`, as required by the specification of `chain`.

3 Uses of `augment` are actually of this sort, but the key difference of a step p_i `==>` p_{i+1} `[augment]` from the cases we are about to discuss is that the "earlier information" is all embedded in and retrievable from the right-hand side, p_{i+1}, and is therefore local to the step, unlike the forthcoming examples.

The following example uses the same technique to combine the left-hand side of a chain step with *two* previous pieces of information via the ternary method cases:

```
assert A=>C := (A ==> C)
assert B=>C := (B ==> C)

> (!chain [(A | B) ==> C [method (p) (!cases p A=>C B=>C)]])

Theorem: (if (or A B)
             C)
```

However, it is tedious to write justifying methods in this long form every time we want to combine a left-hand side with previous information through a method M of $k > 1$ arguments (such as mt, cases, etc.). It is better to have a single generic mechanism that lets us specify M and the $k - 1$ nonlocal arguments, and constructs the appropriate method automatically. The binary procedure with is such a mechanism. It takes the k-ary method M as its first argument and a list of the $k - 1$ nonlocal arguments as its second argument, and produces the appropriate method required by the implementation of chain. Using with in infix notation,[4] the preceding example involving mt can be written as follows:

```
> (!chain [(~ B) ==> (~ A) [(mt with [A=>B])]])

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

Informally, this step says: "we derive the goal (~ A) by applying mt to the left-hand premise (~ B) *and* to the nonlocal premise (A ==> B), in some appropriate order." Or, somewhat more precisely, "we derive the right-hand side (~ A) from the left-hand side, (~ B), through a method that is obtained from mt by fixing its other argument to be (A ==> B)."

The cases example above can be expressed as follows:

```
assert A=>C := (A ==> C)
assert B=>C := (B ==> C)

> (!chain [(A | B) ==> C [(cases with [A=>C B=>C])]])

Theorem: (if (or A B)
             C)
```

This chaining step can likewise be understood as follows: Derive C by applying cases to the left-hand side (A | B) along with the two (nonlocal) sentences A=>C and B=>C.[5]

Another advantage of this approach is that the nonlocal arguments can be listed in an arbitrary order. The implementation of with will try different permutations to discover one

4 Recall that binary procedures can be used in infix by default.

5 Again, by "nonlocal" we mean a sentence that appears on neither side of the step.

that succeeds. For instance, in the above chain, we could list the two nonlocal arguments in reverse order and the proof would still go through:

```
> (!chain [(A | B) ==> C [(cases with [B=>C A=>C])]])

Theorem: (if (or A B)
             C)
```

By contrast, earlier, when we expressed the justifying method of this example in long form, the arguments to cases had to be given in the exact right order:

```
(!chain [(A | B) ==> C [method (p) (!cases (A | B)
                                           A=>C
                                           B=>C)]])
```

Also, the data values in the list argument of with can be of arbitrary type, not just sentences. For instance:

```
> (!chain [(forall x . x = x) ==> (1 = 1) [(uspec with [1])]])

Theorem: (if (forall ?x:'S
               (= ?x:'S ?x:'S))
             (= 1 1))
```

When the list has only one element, we can drop the square brackets altogether and simply write the element by itself:

```
assert A=>B := (A ==> B)

> (!chain [(~ B) ==> (~ A) [(mt with A=>B)]])

Theorem: (if (not B)
             (not A))

> (!chain [(forall x . x = x) ==> (1 = 1) [(uspec with 1)]])

Theorem: (if (forall ?x:'S
               (= ?x:'S ?x:'S))
             (= 1 1))
```

6.2 Using sentences as justifiers

It is natural to allow sentences to appear as justifications of implication steps, particularly sentences that we will call *rules*, namely, sentences of the following form:

$$\texttt{(forall } v_1 \cdots v_k \texttt{ . } p_1 \texttt{ \& } \cdots \texttt{ \& } p_n \Rightarrow q_1 \texttt{ \& } \cdots \texttt{ \& } q_m \texttt{)} \tag{6.3}$$

where $k, n \geq 0$, $m > 0$.[6] Consider, for instance, a universally quantified premise that expresses the symmetry of marriage:

```
declare married-to: [Person Person] -> Boolean

assert* marriage-symmetry := (x married-to y ==> y married-to x)
```

Then we should be able to proceed from a sentence of the form (*s* `married-to` *t*) to the conclusion (*t* `married-to` *s*) simply by citing `marriage-symmetry`. Inference steps of this form are extremely common in computer science and mathematics. The implementation of `chain` allows for such steps, as the following example demonstrates:

```
> (!chain [(Ann married-to Tom)
       ==> (Tom married-to Ann) [marriage-symmetry]])

Theorem: (if (married-to Ann Tom)
             (married-to Tom Ann))
```

The implementation of `chain` realizes that the starting premise `(Ann married-to Tom)` *matches* the antecedent of the cited rule, `marriage-symmetry`, under the substitution

`?x:Person --> Ann, ?y:Person --> Tom`

and proceeds to instantiate the rule with these bindings and perform modus ponens on the result of the instantiation and the starting premise. This sequence of actions, wherein a starting premise is matched against the antecedent of a rule, resulting in a substitution, and then the rule is instantiated under that substitution and "fired" via modus ponens, is a fundamental mode of reasoning,[7] and steps of that form are very common in implication chains.[8] Here is another example:

```
assert* <-tran := (x < y & y < z ==> x < z)

> (!chain [(x < 3.14 & 3.14 < 5.2) ==> (x < 5.2) [<-tran]])

Theorem: (if (and (< ?x:Real 3.14)
                  (< 3.14 5.2))
             (< ?x:Real 5.2))
```

6 The form of (6.3) suggests that rule conjunctions must be binary (with longer conjunctions expressed as right-associative compositions of binary conjunctions), but in fact polyadic conjunctions are also allowed.

7 Recall the relevant discussion of forward Horn clause inference from Section 5.4.

8 Strictly speaking, `chain` will match both the left-hand and right-hand sides of the step against the antecedent and consequent of the cited rule, respectively. This is necessary because in some cases the antecedent might not have enough information to give us the proper substitution. The forthcoming example on page 411, with the step `true ==> (is-male father Ann)`, illustrates that situation.

When the antecedent of the rule is a binary conjunction, the instantiated conjuncts can be listed in any order on the left-hand side of the step. Thus, for example, the following works just as well:

```
> (!chain  [(3.14 < 5.2 & x < 3.14) ==> (x < 5.2) [<-tran]])

Theorem: (if (and (< 3.14 5.2)
                  (< ?x:Real 3.14))
             (< ?x:Real 5.2))
```

The implementation of `chain` will notice that the left-hand side matches the antecedent of the rule modulo the commutativity of conjunction, and will rearrange the conjuncts as needed.

Biconditionals can also be used as rules. Athena will automatically extract the appropriate conditional for the step at hand:

```
declare empty: [Set] -> Boolean

define [s s1 s2] := [?s:Set ?s1:Set ?s2:Set]

assert* empty-def := (empty s <==> forall x . ~ x in s)

> (!chain [(empty s) ==> (forall x . ~ x in s) [empty-def]])

Theorem: (if (empty ?s:Set)
             (forall ?x:Element
               (not (in ?x:Element ?s:Set))))

> (!chain [(forall x . ~ x in s) ==> (empty s) [empty-def]])

Theorem: (if (forall ?x:Element
               (not (in ?x:Element ?s:Set)))
             (empty ?s:Set))
```

In addition, `chain` allows for rules of the following form:

$$\texttt{(forall } v_1 \cdots v_k \texttt{ . } p_1 \texttt{ | } \cdots \texttt{ | } p_n \Rightarrow q_1 \texttt{ \& } \cdots \texttt{ \& } q_m\texttt{)}, \tag{6.4}$$

where $k, n \geq 0$ and $m > 0$.[9] For example:

```
assert* R := (s1 = null | s2 = null ==> s1 intersection s2 = null)

> (!chain [(x = null | y = null) ==> (x intersection y = null) [R]])

Theorem: (if (or (= ?x:Set null)
                 (= ?y:Set null))
```

9 Again, more flexibility is actually allowed in that polyadic disjunctions may also be used.

```
            (= (intersection ?x:Set ?y:Set)
               null))
```

A step of this form will go through as long as the antecedent matches *some* disjunct in the antecedent:

```
> (!chain [(x = null) ==> (x intersection y = null) [R]])

Theorem: (if (= ?x:Set null)
             (= (intersection ?x:Set ?y:Set)
                null))
```

The implementation of `chain` actually allows for some flexibility in using a rule of the form (6.3) as a justifier for a step p `==>` q. Specifically, if the left-hand side p matches the antecedent of the rule and the right-hand side q matches *some* conjunct of the consequent, the step will go through. For example, consider a rule whose consequent has three conjuncts:

```
declare child, parent: [Person Person] -> Boolean

assert* R := (father x = y ==> male y & y parent x & x child y)
```

Then all of the following chain steps are successful:

```
> (!chain [(father Ann = Tom) ==> (Tom parent Ann) [R]])

Theorem: (if (= (father Ann)
                Tom)
             (is-parent-of Tom Ann))

> (!chain [(father Ann = Tom) ==> (male Tom) [R]])

Theorem: (if (= (father Ann)
                Tom)
             (male Tom))

> (!chain [(father Ann = Tom) ==> (Ann child Tom) [R]])

Theorem: (if (= (father Ann)
                Tom)
             (child Ann Tom))
```

This is just a convenience that can save us the effort of detaching the conjunct later in a separate step. More interestingly, `chain` also allows us to use the rule in the contrapositive direction: If p matches the *complement* of *some* consequent conjunct, and q matches either the complement of the antecedent or else

$$(p'_1 \mid \cdots \mid p'_n)$$

where p'_i is the complement of p_i, then the step goes through. For instance:

```
1  declare A, B, C, D, E, F: Boolean
2
3  assert R := (A & B & C ==> D & E & F)
4
5  > (!chain [(~ (D & E & F)) ==> (~ (A & B & C)) [R]])
6
7  Theorem: (if (not (and D
8                         (and E F)))
9               (not (and A
10                        (and B C))))
11
12 > (!chain [(~ F) ==> (~ (A & B & C)) [R]])
13
14 Theorem: (if (not F)
15              (not (and A
16                        (and B C))))
17
18 > (!chain [(~ E) ==> (~A | ~B | ~C) [R]])
19
20 Theorem: (if (not E)
21              (or (not A)
22                  (or (not B)
23                      (not C))))
```

The first application of `chain`, on line 5, is a typical use of the contrapositive. The second application, on line 12, saves some effort in that it first tacitly infers the negated consequent, `(~ (D & E & F))`, from the given `(~ F)`. The third application, on line 18, goes further in that it follows up with an application of De Morgan's law.

Similar remarks apply to rules of the form (6.4). First, the right-hand side of the step may only match one of the consequent's conjuncts, not all of them. Again, this allows for tacit conjunction simplification:

```
assert* R := (A | B | C ==> D & E & F)

> (!chain [A ==> E [R]])

Theorem: (if A E)
```

Second, in the contrapositive direction, it is possible for the left-hand side to match the complement of *some* consequent conjunct and for the right-hand side to match either the complement of the antecedent or the conjunction of the complements of some of the antecedent's disjuncts:

```
assert* R := (A | B | C ==> D & E & F)

> (!chain [(~ E) ==> (~ B) [R]])

Theorem: (if (not E)
             (not B))

> (!chain [(~ D) ==> (~B & ~C) [R]])

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

It is possible to use "rules" without an explicit antecedent, so that $n = 0$ in the general form (6.3). The implementation of `chain` will treat such a degenerate rule as a conditional whose antecedent is `true`:

```
assert R := (forall x . male father x)

> (!chain [true ==> (male father Ann) [R]])

Theorem: (if true
             (male (father Ann)))
```

This example works because Athena transforms the sentence

```
(forall x . male father x)
```

into the logically equivalent

```
(forall x . true ==> male father x).
```

We will see that implication chains starting with `true` are particularly handy with a variant of `chain` called `chain->` that returns the last element of the chain as its conclusion.

We have said that `chain` accepts sentences ("rules") as justifiers for implication steps, in addition to methods, but internally it is all methods; `chain` will actually "compile" a given rule such as (6.3) into a method that can take any instance of the left-hand side of the rule as input and will derive the appropriately instantiated right-hand side of the rule as its conclusion.

6.2.1 Nested rules

Sometimes rules of the form (6.3) are buried inside larger, enclosing rules. A typical example is provided by definitions of relation symbols, which are often of the form

$$\texttt{(forall } x_1 \cdots x_n \texttt{ . (R } x_1 \cdots x_n \texttt{) <==> } C \texttt{)} \tag{6.5}$$

where C is itself a rule of the form (6.3). For instance:

```
assert* subset-definition :=
   (s1 subset s2 <==> forall x . x in s1 ==> x in s2)
```

Here the body of the definition is

```
(forall x . x in s1 ==> x in s2).
```

When `s1` and `s2` take on particular values, this becomes a rule that can be used in chaining steps. For example, suppose we know that a certain set `A` is a subset of some set `B`. Then we can derive the following specialized, nested rule from `subset-definition`:

```
(forall x . x in A ==> x in B).
```

We would like the ability to use this specialized rule in an implication step without having to explicitly derive it first, simply by citing the *enclosing rule* as our justifier—in this case, `subset-definition`. Thus, if the assumption base contains `(A subset B)`, we would like the following chain to produce the conditional stating that if `e` is in `A` then `e` is in `B` (where `e` is some term of sort `Element`):

```
(!chain [(e in A) ==> (e in B) [subset-definition]]).
```
(6.6)

If implication steps could *only* use rules of the form (6.3) as justifiers, then steps such as the above could not work. We would first have to derive the nested rule separately, by specializing `subset-definition` with `A` and `B`, and then use that rule in the above step:

```
let {nested-rule :=
       (!chain-> [(A subset B)
             ==>  (forall x . x in A ==> x in B) [subset-definition]])}
  (!chain [(e in A) ==> (e in B) [nested-rule]])
```

(See Section 6.4 for a description of `chain->`.) Alternatively, using nested chains (Section 6.8), we could write:

```
(!chain [(e in A) ==> (e in B)
                          [(forall x . x in A ==> x in B) <==
                             (A subset B) [subset-definition]]])
```

Either way, we would have to explicitly derive the inner, specialized rule first, and *then* use that as a justifier for the desired implication step.

This should not be necessary, however. We should be able to take chaining steps such as (6.6) directly, without having to explicitly derive the specialized inner rule first, and Athena allows this, for example:

```
declare A,B: Set

assert (A subset B)

pick-any element
```

```
(!chain [(element in A) ==> (element in B) [subset-definition]])

Theorem: (forall ?element:Element
           (if (in ?element:Element A)
               (in ?element:Element B)))
```

In general, Athena will accept as a justifier for an implication step any nested rule of the form

$$\texttt{(forall } x_1 \cdots x_n \texttt{ . } p \texttt{ <==> (forall } y_1 \cdots y_m \texttt{ . } p_1 \texttt{ ==> } p_2\texttt{))}, \tag{6.7}$$

as well as

$$\texttt{(forall } x_1 \cdots x_n \texttt{ . } p \texttt{ ==> (forall } y_1 \cdots y_m \texttt{ . } p_1 \texttt{ ==> } p_2\texttt{))}, \tag{6.8}$$

and will automatically derive and use a properly specialized instance of the inner rule,

$$\texttt{(forall } y_1 \cdots y_m \texttt{ . } p_1 \texttt{ ==> } p_2\texttt{)},$$

provided that the corresponding instance of the outer antecedent, p, is in the assumption base.[10]

6.3 Implication chaining through sentential structure

There is another noteworthy way of enabling an implication chain step from p to q: by using the structure of p and q. The simplest case occurs when p and q are identical, in which case the step will succeed even without any justifiers. The next simplest case occurs when both p and q are atomic sentences of the form $\texttt{(}R\ s_1 \cdots s_n\texttt{)}$ and $\texttt{(}R\ t_1 \cdots t_n\texttt{)}$, respectively. Then, if J is a justifier (e.g., a list of identities and/or conditional identities) licensing the conclusions $\texttt{(}s_i\ \texttt{=}\ t_i\texttt{)}$ for $i = 1, \ldots, n$, then the step

$$\texttt{(}R\ s_1 \cdots s_n\texttt{)}\ \texttt{==>}\ \texttt{(}R\ t_1 \cdots t_n\texttt{)}\ J \tag{6.9}$$

will succeed (in some appropriate assumption base β).[11] For example:

```
define [\/ /\] := [union intersection]
assert* R1 := (x \/ y = y \/ x)
assert* R2 := (x /\ null = null)

> (!chain [(s1 /\ null subset s2 \/ s3)
       ==> (null subset s3 \/ s2)       [R1 R2]])

Theorem: (if (subset (intersection ?s1:Set null)
                     (union ?s2:Set ?s3:Set))
```

10 Of course the outer rule—(6.7) or (6.8)—must itself be in the assumption base.

11 More precisely, what we mean by J "licensing the conclusions $s_i = t_i$ for $i = 1, \ldots, n$" is that, for each such i, the equational step $\texttt{(!chain [}s_i\ \texttt{=}\ t_i\ J\texttt{])}$ should go through in β (where, again, β is the assumption base in which the step (6.9) is taking place).

```
            (subset null
                    (union ?s3:Set ?s2:Set)))
```

By using the given identities, R1 and R2, the system was able to equate corresponding terms on each side:

1. the first subterm of the left-hand side, (s1 /\ null), was equated with the first subterm of the right-hand side, null, via R2; and
2. the second subterm of the left-hand side, (s2 \/ s3), was equated with the second subterm of the right-hand side, (s3 \/ s2), via R1.

This is just a form of relational congruence. The same result could be obtained through rcong, but we would first need to establish the identities of the respective terms explicitly. This shorthand is more convenient.[12]

Two other structural cases occur when p and q are of the form

$$(\odot\ p_1 \cdots p_n) \text{ and } (\odot\ q_1 \cdots q_n)$$

respectively, where $\odot$ is either the conjunction or disjunction constructor. Such cases are handled recursively: If the justifier J can enable each step $(p_i$ ==> $q_i)$ for $i = 1, \ldots, n$, then the step $(p$ ==> $q)$ goes through. For instance:

```
assert* marriage-symmetry := (x married-to y ==> y married-to x)
assert* union-comm := (x \/ y = y \/ x)

> (!chain [(Tom married-to Ann | s1 \/ s2 = s3)
       ==> (Ann married-to Tom | s2 \/ s1 = s3)   [marriage-symmetry
                                                   union-comm]])

Theorem: (if (or (married-to Tom Ann)
                 (= (union ?s1:Set ?s2:Set)
                    ?s3:Set))
             (or (married-to Ann Tom)
                 (= (union ?s2:Set ?s1:Set)
                    ?s3:Set)))
```

Here the implementation of chain will realize that the two sides of the implication step are disjunctions, and will attempt to use the given justifiers to recursively relate the corresponding disjuncts: (Tom married-to Ann) with (Ann married-to Tom), and

```
(s1 \/ s2 = s3)
```

with (s2 \/ s1 = s3). Both of these will succeed with the given information. The example would work just as well if it involved conjunctions instead. Ultimately, steps of this form succeed owing to the validity of the following inference rule, where $\odot \in \{\wedge, \vee\}$:

12 The implementation of chain does use rcong for steps of this form.

$$\frac{(p_1 \texttt{ ==> } q_1) \quad \cdots \quad (p_n \texttt{ ==> } q_n)}{((\odot\ p_1 \cdots p_n) \texttt{ ==> } (\odot\ q_1 \cdots q_n))} \quad [SC]$$

Note that this rule is valid only for $\odot \in \{\texttt{and}, \texttt{or}\}$. It is *not* valid for $\odot \in \{\texttt{not}, \texttt{if}, \texttt{iff}\}$.

The two remaining structural cases occur when p and q are both quantified sentences, respectively of the form

$$(Q\ x\ p') \text{ and } (Q\ y\ q')$$

for $Q \in \{\texttt{forall}, \texttt{exists}\}$, in which case `chain` will continue its structural work recursively, with the given justifier, on appropriately renamed variants of p' and q'. For example:

```
> (!chain [(forall s1 . s1 \/ null = null)
       ==> (forall s2 . null \/ s2 = null)  [union-comm]])

Theorem: (if (forall ?s1:Set
               (= (union ?s1:Set null)
                  null))
             (forall ?s2:Set
               (= (union null ?s2:Set)
                  null)))
```

6.4 Using chains with chain-last

Sometimes when we put together an implication chain of the form

$$\begin{array}{lll} \texttt{(!chain [}p_1 \texttt{ ==> } p_2 & J_1 \\ \texttt{==> } p_3 & J_2 \\ \vdots & \\ \texttt{==> } p_{n+1} & J_n\texttt{])} \end{array} \tag{6.10}$$

we are operating in an assumption base that contains the initial sentence, p_1. In such cases we are usually not content merely with showing that p_1 implies p_{n+1}, which is the result that would be returned by (6.10). Instead, we want to derive the final element of the chain, p_{n+1}. That can be done just as above, but with a method named `chain-last` rather than `chain`. An alternative (and shorter) name that is often used for `chain-last` is `chain->`. This method works just as `chain` does, except that after the implication $p_1 \Rightarrow p_{n+1}$ is produced, modus ponens is used on it and p_1 to derive p_{n+1}.[13] For instance, the `chain-last` call below will produce the conclusion `B`:

13 If you view the chain of implications as a line of dominoes, you can think of the effect of `chain-last` as tapping the first domino, p_1, in order to topple (derive) the last piece, p_{n+1}.

```
> assume hyp := (A & ~~ B)
    (!both (!chain-last [hyp ==> (~~ B) [right-and]
                             ==> B      [dn]])
           (!left-and hyp))

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

Thus, a call of the form

$$\texttt{(!chain-last [}p_1\ \texttt{==>}\ p_2\ J_1 \quad \cdots \quad p_n\ \texttt{==>}\ p_{n+1}\ J_n\texttt{])}$$

is equivalent to:

$$\texttt{(!mp (!chain [}p_1\ \texttt{==>}\ p_2\ J_1 \quad \cdots \quad p_n\ \texttt{==>}\ p_{n+1}\ J_n\texttt{]) }p_1\texttt{)}.$$

And conversely,

$$\texttt{(!chain [}p_1\ \texttt{==>}\ p_2\ J_1 \quad \cdots \quad p_n\ \texttt{==>}\ p_{n+1}\ J_n\texttt{])}$$

is equivalent to

$$\texttt{assume }p_1\ \texttt{(!chain-last [}p_1\ \texttt{==>}\ p_2\ J_1 \quad \cdots \quad p_n\ \texttt{==>}\ p_{n+1}\ J_n\texttt{])}.$$

Note that, unless it is exceptionally long, a `chain-last` application rarely needs a conclusion annotation, since its conclusion is immediately apparent: It is always the last link of the chain.

The rest of the book contains many other examples similar to the preceding proof of

```
(A & ~ ~ B ==> B & A)
```

where applications of `chain-last` (and other `chain` variants) are mixed in with other Athena constructs: appearing as arguments to enclosing method calls, inside **`assume`** bodies, and so on. Generally speaking, the more chaining we use, the more readable the proof. If we can express an entire proof as a chain, we probably should. And it is often surprising how far we can get with chaining alone. The preceding proof, for example, can be expressed as a single `chain`, and the result is shorter and clearer:

```
> (!chain [(A & ~~ B) ==> (~~ B & A) [comm]
                      ==> (B & A)    [dn]])

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

By default, `chain-last` can be used on an implication chain that starts with `true` even if the assumption base does not contain `true` explicitly. This is useful in tandem with the aforementioned convention, whereby, for chaining purposes, a nonconditional rule is treated as a conditional with `true` as its antecedent.

```
assert* R := (male father x)

> (!chain-> [true ==> (male father Ann) [R]])

Theorem: (male (father Ann))
```

6.5 Backward chains and chain-first

Implication chains can be written in reverse, by using the symbol <== instead of ==>. This can be useful in showing how a goal decomposes into something different (and hopefully simpler). As an example, suppose we have the following properties in the assumption base, where `Mod` denotes the remainder function on natural numbers:

```
declare pos: [N] -> Boolean
declare less: [N N] -> Boolean [<]
declare Mod: [N N] -> N [%]
define [x y z] := [?x:N ?y:N ?z:N]

assert* mod-< := (pos y ==> x % y < y)
assert* less-asymmetric := (x < y ==> ~ y < x)
assert* <-S-2 := (x < y ==> x < S y)
```

Suppose now we want to prove that for any two natural numbers `a` and `b`, the successor of `b` is not less than `(a % b)`: `(~ S b < a % b)`. The following chain demonstrates how this somewhat complex goal reduces to the simple atom `(pos b)`:

```
pick-any a:N b:N
  (!chain [(~ S b < a % b) <== (a % b < S b)    [less-asymmetric]
                           <== (a % b < b)      [<-S-2]
                           <== (pos b)          [mod-<]])
```

This is operationally equivalent to reversing the links of the chain and using forward rather than backward implications:

```
(!chain [(pos b) ==> (a % b < b)     [mod-<]
                 ==> (a % b < S b)   [<-S-2]
                 ==> (~ S b < a % b) [less-asymmetric]])
```

For the purposes of proof development, however, the two are not the same. To write the second (forward) chain, we have to know the key starting point ahead of time: `(pos b)`.

Essentially, we must have already figured out the reasoning in its entirety and then simply written it down. That rarely happens in practice. In this case the backward version may be a closer reflection of how the proof would actually be built, because we can *start with the goal*, which we do know from the beginning, and then incrementally transform it by inspecting the assumption base for appropriate information. In this case, presumably, we would notice that the goal matches the consequent of `less-asymmetric`, and that would lead us to take the backward step of transforming the goal to the appropriate instance of the corresponding antecedent: `(a % b < S b)`. Then we might notice that this new goal matches the consequent of `<-S-2`, and that would lead us to transform that goal to the relevant instance of the corresponding antecedent: `(a % b < b)`. Finally, we would observe that this goal matches the consequent of `mod-<`, and that would lead us, with one last backward step, to the goal `(pos b)`, which may already be known to hold. This process is known as *backward chaining*. We have already seen it in connection with the backward tactics we discussed when we studied proof heuristics, and we will explore it again in the context of logic programming (in Appendix B).

In analogy with `chain-last`, there is a `chain-first` method that can be used to derive the first element of a backward chain, provided that the last one is in the assumption base. An alternative name for `chain-first` is `chain<-`. And also as before, when the last sentence is `true`, `chain-first` will succeed even if `true` is not in the assumption base. For example:

```
1  define [x y z] := [?x ?y ?z]
2
3  assert* p-def := (x parent y <==> x = father y | x = mother y)
4  assert* gp-def := (x grandparent z <== x parent y & y parent z)
5
6  assert* fact1 := (Mary = mother Bob)
7  assert* fact2 := (Peter = father Mary)
8
9  > (!chain-first
10      [(Peter grandparent Bob)
11    <== (Peter parent Mary & Mary parent Bob)        [gp-def]
12    <== (Peter = father Mary & Mary = mother Bob)    [p-def]
13    <== (true & true)                               [fact1 fact2]
14    <== true                                        [augment]])
15
16 Theorem: (grandparent Peter Bob)
```

(The second and third steps of this example, on lines 11 and 12, demonstrate the structural implication chaining discussed in Section 6.3.) Essentially the same proof could also be written in a somewhat shorter form by avoiding any reference to `true` and starting from the given facts:

```
 (!chain-first
   [(Peter grandparent Bob)
<== (Peter parent Mary & Mary parent Bob) [gp-def]
```

```
<== (Peter = father Mary & Mary = mother Bob)])
```

Athena will realize that the first element of the chain is a conjunction of facts and will derive that conjunction automatically.

6.6 Equivalence chains

We can use `chain` to put together equivalence chains just as well, by using the symbol `<==>` instead of `==>`. A `chain` call of the form

$$\begin{array}{rl} \texttt{(!chain [}p_1 \texttt{ <==> } p_2 & J_1 \\ \texttt{<==> } p_3 & J_2 \\ \vdots & \\ \texttt{<==> } p_{n+1} & J_n\texttt{])} \end{array} \tag{6.11}$$

will derive the biconditional (p_1 `<==>` p_{n+1}), provided that each step p_i `<==>` p_{i+1} J_i goes through, $i = 1, \ldots, n$. Everything that we have said so far about implication steps applies here as well, with one additional caveat: The relevant justifying methods in J_i must be bidirectional, that is, they must not only be able to derive the right-hand side from the left-hand side, but conversely as well. Here is an example:

```
> (!chain [(~~ A & (B ==> C)) <==> ((B ==> C) & ~~ A)  [comm]
                               <==> ((~C ==> ~B) & A)  [contra-pos bdn]])

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

Both of these steps went through because the justifying methods are bidirectional. An error would occur if, say, we replaced the bidirectional version of double negation, `bdn`, with the regular unidirectional version, `dn`, since we would then be unable to derive `(~ ~ A)` from `A`.

As suggested by the second step of the previous example, everything that we have said about structural implication steps carries over to equivalence steps. Another example:

```
assert* R1 := (x \/ y = y \/ x)
assert* R2 := (x /\ null = null)

> (!chain [(s /\ null subset s1 \/ s2)
      <==> (null subset s2 \/ s1)       [R1 R2]])

Theorem: (iff (subset (intersection ?S:Set null)
```

```
                    (union ?S1:Set ?S2:Set))
            (subset null
                    (union ?S2:Set ?S1:Set)))
```

Structural equivalence chaining is actually more flexible, as the analogue of rule [*SC*] is more widely applicable:

$$\frac{(p_1 \texttt{ <==> } q_1) \quad \cdots \quad (p_n \texttt{ <==> } q_n)}{((\odot\ p_1 \cdots p_n) \texttt{ <==> } (\odot\ q_1 \cdots q_n))} \tag{6.12}$$

Rule (6.12) holds for $\odot \in \{\texttt{not}, \texttt{and}, \texttt{or}, \texttt{if}, \texttt{iff}\}$, not just for $\odot \in \{\texttt{and}, \texttt{or}\}$, as was the case with [*SC*]. Thus, for example:

```
> (!chain [(~ (A & B)) <==> (~ (B & A)) [comm]])

Theorem: (iff (not (and A B))
              (not (and B A)))
```

The quantified analogues of the rule hold as well:

$$\frac{(p \texttt{ <==> } q)}{(Q\ v\ .\ p \texttt{ <==> } Q\ v\ .\ q)} \tag{6.13}$$

for $Q \in \{\texttt{forall}, \texttt{exists}\}$. For example:

```
assert* R := (s1 \/ s2 = s2 \/ s1)

> (!chain [(forall x y z . x subset y \/ z)
      <==> (forall x y z . x subset z \/ y)  [R]])

Theorem: (iff (forall ?x:Set
                (forall ?y:Set
                  (forall ?z:Set
                    (subset ?x:Set
                            (union ?y:Set ?z:Set)))))
              (forall ?x:Set
                (forall ?y:Set
                  (forall ?z:Set
                    (subset ?x:Set
                            (union ?z:Set ?y:Set))))))
```

In fact, `chain` implements a somewhat stronger formulation of (6.13) that allows for alpha-equivalence. For example, the following variant of the preceding example works just as well:

```
> (!chain [(forall x y z . x subset y \/ z)
      <==> (forall u v w . u subset w \/ v)  [R]])

Theorem: (iff (forall ?x:Set
```

```
              (forall ?y:Set
                (forall ?z:Set
                  (subset ?x:Set
                          (union ?y:Set ?z:Set)))))
            (forall ?u:Set
              (forall ?v:Set
                (forall ?w:Set
                  (subset ?u:Set
                          (union ?w:Set ?v:Set))))))
```

6.7 Mixing equational, implication, and equivalence steps

Equivalence and implication steps can be mixed in a chain. In particular, it is possible to switch from an equivalence step to an implication step:

$$\begin{array}{lll} \texttt{(!chain [}p_1 & \texttt{<==>} & p_2 \quad J_1 \\ & \vdots & \\ & \texttt{<==>} & p_{i+1} \quad J_i \\ & \texttt{==>} & p_{i+2} \quad J_{i+1} \\ & \vdots & \\ & \texttt{==>} & p_{n+1} \quad J_n\texttt{]).} \end{array}$$

For example:

```
> (!chain [(A & B | A & C) <==> (A & (B | C)) [dist]
                           ==> A              [left-and]])

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

We can also switch from implication steps to equivalence steps:

```
> (!chain [(A & B | A & C) <==> (A & (B | C)) [dist]
                           ==> A              [left-and]
                          <==> (~~ A)         [bdn]])

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

If there is even one implication step, however, the chain of equivalences is broken, and the conclusion of the `chain` cannot be a biconditional; at best, it will be a conditional. Hence,

strictly speaking, there is little reason to mix equivalence and implication steps. The above proof, for instance, could just as well be written with implication steps only:

```
(!chain [(A & B | A & C) ==> (A & (B | C)) [dist]
                         ==> A             [left-and]
                         ==> (~~ A)        [bdn]])
```

Nevertheless, the former version conveys more information to the reader: it makes it clear that the first two elements of the chain are equivalent (owing to `dist`), as are the last two elements of the chain (owing to `bdn`).

Backward implication steps can also be mixed with equivalence steps. Athena will adjust the "direction" of the result accordingly. In the presence of backward implication steps, the antecedent and consequent will be the last and first elements of the chain, respectively:

```
> (!chain [(~~ A) <==>  A                [bdn]
                  <==   (A & (B | C))    [left-and]
                  <==>  (A & B | A & C)  [dist]])

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

More interestingly, equational steps can be mixed with implication and/or equivalence steps. In one direction, we can switch from equational steps to implication steps:

$$\begin{array}{lll} \texttt{(!chain->}\ [t_1 = t_2 & & J_1 \\ & \vdots & \\ & = t_{n+1} & J_n \\ & \texttt{==>}\, p_1 & J_{n+1} \\ & \vdots & \\ & \texttt{==>}\, p_{m+1} & J_{n+m+1}\texttt{]).} \end{array} \tag{6.14}$$

This can be useful when we first establish an identity $t_1 = t_{n+1}$ by rewriting and then that identity becomes transformed to some conclusion p_1, either via some appropriate method(s) or because it matches the antecedent of some rule and can therefore be used to derive the rule's consequent. After that point, implication chaining proceeds normally. The final result of (6.14) will be the last element of the chain, p_{m+1}. For example, suppose that we have defined the `divides` relation as shown below; and suppose we know that multiplication distributes over addition. We can then prove that a divides $(a \cdot b) + (a \cdot c)$, for all natural numbers a, b, and c, with the following mixed chain:

```
declare Plus:  [N N] -> N [+]
declare Times: [N N] -> N [*]
declare divides: [N N] -> Boolean
```

```
define [x y z k] := [?x:N ?y:N ?z:N ?k:N]

assert* divides-def := (x divides y <==> exists z . x * z = y)

assert* times-dist := (x * (y + z) = x * y + x * z)

pick-any a b c
  (!chain-> [(a * (b + c)) = (a * b + a * c)      [times-dist]
         ==> (exists k . a * k = a * b + a * c)   [existence]
         ==> (a divides a * b + a * c)            [divides-def]])
```

In the reverse direction, we can switch from implication/equivalence steps to equational steps. For example, a call of the following form will derive the conclusion $s = t_m$, provided that the initial sentence p_1 is in the assumption base.

$$\begin{array}{lll} \texttt{(!chain-> [}p_1 \texttt{ ==> } p_2 & J_1 \\ \vdots & \\ \texttt{==> (}s \texttt{ = } t_1\texttt{)} & J_{n+1} \\ \texttt{ = } t_2 & J_{n+2} \\ \vdots & \\ \texttt{ = } t_m & J_{n+m}\texttt{]).} \end{array} \tag{6.15}$$

If `chain` is used instead of `chain->` in (6.15), then the conditional $p_1 \Rightarrow s = t_m$ is returned.[14] For example:

```
declare Minus:  [N N] -> N [-]
declare pos:  [N] -> Boolean

assert* R1 := (x * zero = zero)
assert* R2 := (x - x = zero)

pick-any a b c d
 (!chain [(pos a & b = c * (d - d)) ==> (b = c * (d - d)) [right-and]
                                        = (c * zero)   [R2]
                                        = zero         [R1]])

Theorem: (forall ?a:N
           (forall ?b:N
             (forall ?c:N
               (forall ?d:N
                 (if (and (pos ?a:N)
```

14 If `chain` were used instead of `chain->` in the case of (6.14), the conditional $t_1 = t_{n+1} \Rightarrow p_{m+1}$ would be returned. While this is fine as far as it goes, it means that all the work of the equational part of the chain would be in vain, since the same result could be produced by skipping the equational steps and simply starting the implication chain with the hypothesis $t_1 = t_{n+1}$. In general, `chain` applications that start with equational steps and then switch to implication steps are unnecessary.

```
                        (= ?b:N
                           (Times ?c:N
                                  (Minus ?d:N ?d:N))))
                    (= ?b:N zero))))))
```

Multiple switches can occur in the same chain, from implication steps to equational steps, back to implication (and/or equivalence) steps, and so on, arbitrarily many times:

```
> pick-any a b c d
    (!chain [(pos a & b = c * (d - d)) ==> (b = c * (d - d)) [right-and]
                                             = (c * zero)   [R2]
                                             = zero         [R1]
                                           ==> (zero = b)   [sym]])

Theorem: (forall ?a:N
           (forall ?b:N
             (forall ?c:N
               (forall ?d:N
                 (if (and (pos ?a:N)
                          (= ?b:N
                             (Times ?c:N
                                    (Minus ?d:N ?d:N))))
                     (= zero ?b:N))))))
```

Athena allows for switching from implication to identity steps even when the final conclusion is an atomic sentence other than an identity. For instance:

```
1  declare less: [N N] -> Boolean [<]
2
3  pick-any a b c d
4    (!chain [(pos a & b < c * (d - d)) ==> (b < c * (d - d)) [right-and]
5                                             = (c * zero)   [R2]
6                                             = zero         [R1]])
7
8  Theorem: (forall ?a:N
9             (forall ?b:N
10              (forall ?c:N
11                (forall ?d:N
12                  (if (and (pos ?a:N)
13                           (less ?b:N
14                                 (Times ?c:N
15                                        (Minus ?d:N ?d:N))))
16                      (less ?b:N zero))))))
```

In this example, rewriting starts on line 4 with the atom `(b < c * (d - d))`. The rightmost child of this atom, `c * (d - d)`, is rewritten in two steps into the term `zero`, and the conclusion `(b < zero)` is thereby generated by congruence. Instead of `(b < c * (d - d))`,

we could have started with an arbitrary atom $R(t_1, \ldots, t_n)$, and we could have then proceeded in the same fashion to rewrite the last term, t_n, into some other term t'_n in a number of equational steps, eventually producing the result $R(t_1, \ldots, t'_n)$.

6.8 Chain nesting

It is possible to insert a chain inside the justification list of another chain, to an arbitrary depth. The rationale for such nesting arises as follows. Sometimes, a step deep inside a long chain must rely for its justification on a result p that is not in the assumption base in which the chain itself is evaluated:

$$\begin{array}{lll} \texttt{(!chain [}t_1 & = t_2 & J_1 \\ & \vdots & \\ & = t_{i+1} & \texttt{[}p\texttt{]} \\ & \vdots & \\ & = t_{n+1} & J_n\texttt{])}. \end{array} \tag{6.16}$$

(Here the chain is equational, but similar remarks apply to implication chains or mixed chains.) In this case, the passage from t_i to t_{i+1} relies on p, but p is not in the assumption base in which (6.16) is to be evaluated. One way to get around this difficulty is to establish p *prior* to the chain, by some proof D, so that it can then become available as a lemma to (6.16), say, by putting (6.16) inside a **let**:

```
let {_ := conclude p
           D}
  (6.16)
```

This approach works, but its drawback is that it might obscure the structure of the argument: p is established first, but it does not get used until considerably later in the proof, in justifying the step from t_i to t_{i+1} inside (6.16). It would be preferable if we could somehow inline the derivation of p right where it is needed, in the justification of the said step.[15] We would then have no need for an enclosing **let**; we could instead express the chain directly

15 Assuming that p is only needed at that one point. If p is used at multiple locations inside the `chain`, then it would be preferable to derive it first and make it available as a lemma inside the chain.

in the following form:

$$\begin{array}{lll} \texttt{(!chain [}t_1 \texttt{ =} & t_2 & J_1 \\ & \vdots & \\ \texttt{ =} & t_{i+1} & \texttt{[}\textit{Derive p here}\texttt{]} \\ & \vdots & \\ \texttt{ =} & t_{n+1} & J_n\texttt{])} \end{array}$$

While the implementation of `chain` does not allow for arbitrary deductions inside justification lists, it does allow for abbreviated subchains. So if p can be obtained by a chain, there is probably a way to inline that chain above, placing it right where it says "*Derive p here*." By "abbreviated" we mean that the inlined chain will not be a complete method call of the form `(!chain ···)` or `(!chain-> ···)`, etc.; only the steps of the chain will be inlined.

As a concrete example from a real proof, consider the following goal:

```
declare leq: [N N] -> Boolean [<=]

define plus-minus-cancel :=
  (forall x y . y <= x ==> x = (x - y) + y)
```

This is a useful result, proven by induction on `x`. For the basis, we need to prove

```
(forall y . y <= zero ==> zero = (zero - y) + y)
```

The available premises from which we are to derive this result are as follows:

```
assert* R1 := (zero - x = zero)
assert* R2 := (x + zero = x)
assert* R3 := (x <= zero ==> x = zero)
```

Now here is one possible proof of the basis step:

```
conclude (forall y . y <= zero ==> zero = (zero - y) + y)
  pick-any y
    assume hyp := (y <= zero)
      let {lemma := (!chain-> [hyp ==> (y = zero) [R3]])}
        (!chain-> [((zero - y) + y)
                   = ((zero - y) + zero)     [lemma]
                   = (zero + zero)           [R1]
                   = zero                    [R2]
                  ==> (zero = (zero - y) + y) [sym]])
```

An alternative is to inline the chain derivation of `lemma` right where it is needed:

```
conclude (forall y . y <= zero ==> zero = (zero - y) + y)
  pick-any y
    assume hyp := (y <= zero)
      (!chain-last [((zero - y) + y)
```

```
5                  = ((zero - y) + zero)     [(y = zero) <== hyp [R3]]
6                  = (zero + zero)           [R1]
7                  = zero                    [R2]
8                ==> (zero = (zero - y) + y) [sym]])
```

In this version there was no need for the **let**, since the lemma `(y = zero)` was derived (from `hyp`) at the step where it was needed, on line 5. A backward step `<==` was used for the derivation, for readability, though a forward step would work as well:

```
conclude (forall y . y <= zero ==> zero = (zero - y) + y)
  pick-any y
    assume hyp := (y <= zero)
      (!chain-> [((zero - y) + y)
                 = ((zero - y) + zero)     [hyp ==> (y = zero) [R3]]
                 = (zero + zero)           [R1]
                 = zero                    [R2]
               ==> (zero = (zero - y) + y) [sym]])
```

The justification list `[hyp ==> (y = zero) [R3]]` is essentially an abbreviated application of `chain->` (more precisely, it has the exact form of an argument to `chain->`). Likewise, the list `[(y = zero) <== hyp [R3]]` in the previous proof can be viewed as an abbreviated application of `chain-<`. In general, the direction of the arrows determines whether Athena understands a nested chain as a tacit application of `chain->` or `chain-<`.[16]

6.9 Exercises

Exercise 6.1: Consider the following problems:

(a)

```
assert* premise-1 := (B | (A ==> B))
assert* premise-2 := A

conclude B
  D1
```

(b)

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

conclude B
  D2
```

16 These are the only two options for nested implication or equivalence chains. Such chains are always treated as applications of `chain-last` or `chain-first`, never as applications of `chain`.

(c)

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

conclude (~ exists x . P x)
  D3
```

(d)

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

conclude ((forall x . Q x) ==> ~ exists z . z R z)
  D4
```

(e)

```
assert* prem-1 := (exists x . P x)
assert* prem-2 := (exists x . Q x)
assert* prem-3 := (forall x . P x ==> forall y . Q y ==> x R y)

conclude (exists x y . x R y)
  D5
```

Find appropriate deductions $D_1, \ldots, D_5$, in chaining style, for each of the above. □

Exercise 6.2 (ite: If-Then-Else operator): Athena has `ite` predeclared as a polymorphic ternary function symbol with the following signature:

```
declare ite: (S) [Boolean S S] -> S
```

The meaning of `ite` is captured by the following two axioms, which are also predefined and contained in the initial assumption base:

```
assert* ite-axioms := [((ite true x _) = x)
                       ((ite false _ x) = x)]
```

Thus, this operator can be viewed as implementing an "if-then-else" mechanism for conditional branching: If b is `true` then `(ite` b t_1 t_2`)` "returns" t_1, while if b is `false` then `(ite` b t_1 t_2`)` returns t_2. We refer to b as the *guard* of the `ite` term.

In this book we do not make extensive use of `ite` because all of the functions we study are definable just as well with more generic constructs. However, `ite` is a very useful operator and quite pervasive in software and hardware modeling. In this exercise we provide

some brief information on how `ite` works in Athena and illustrate its use in a simple problem.

The `ite` operator is natively supported both by `eval` and by `chain`.[17] To illustrate, introduce a `min` operator on natural numbers as follows:[18]

```
load "nat-less"

declare min: [N N] -> N [[int->nat int->nat]]

overload < N.<

define [x y] := [?x:N ?y:N]

assert* min-def := [(x min y = (ite (x < y) x y))]

> (eval 0 min 1)

Term: zero

> (eval 2 min 1)

Term: S zero
```

We can nest `ite` occurrences to an arbitrary depth. Constructions of the form

$$\texttt{(ite } b_1 \ s_1 \ \texttt{(ite } b_2 \ t_2 \ \texttt{(}\cdots\texttt{)))}$$

are common. However, such explicit nesting can be cumbersome to read and write. Athena provides an `ite*` procedure that takes a list of guard-term pairs, each such pair separated by `-->` and with the final guard possibly being the wildcard, and automatically produces the correctly nested `ite` term. For instance, suppose we want to define binary search on search trees of natural numbers:

```
datatype BT := null | (node N BT BT)

declare search: [N BT] -> BT
```

Then the interesting recursive clause of the definition could be written as follows:

```
((search x (node y L R)) = ite* [(x = y) --> (node y L R)
                                 | (x < y) --> (search x L)
                                 | _       --> (search x R)])
```

(In Chapter 11 and later chapters we define similar functions using another construct, `fun`.)

17 It is also specially handled in translating Athena theories to external theorem provers such as first-order ATPs and SMT solvers.

18 We load the file `lib/main/nat-less.ath` (we omit the `lib/main/` part because Athena looks there by default), as that is where a less-than relation `N.<` on the natural numbers is introduced and defined, inside a module named `N`. We will have more to say about the contents of that module in subsequent chapters.

In equational chaining, `ite` is handled essentially by desugaring into conditional equations. As long as the corresponding guard holds (is in the assumption base), an `ite` term can be rewritten into the corresponding term. As an example, we ask you to prove the following: `(forall x y . x min y = x | x min y = y)`. □