CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Chapter 6

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

Implication Chaining

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

- implication chains
- sentences as justifiers
- structural implication chaining
- chain-last
- backward chains and chain-first
- equivalence chains
- mixing equational, implication, and equivalence steps
- chain nesting example

Equality chains

Recall the general form of a proof by equational chaining:

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

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

$$t_1 = t_{n+1}.$$

But identity is not the only important logical relation that is transitive. Implication and equivalence are also transitive.

Implication chains

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

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

Implication chains

- 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)
```

Implication chains

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.

Implication chains with anonymous methods

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.

Implication chains with binary justifiers

- 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.

Implication chains with binary justifiers

For example, to 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, e.g.:

```
assert A, B

> (!chain [C ==> (A & B & C) [augment]])
Theorem: (if C
             (and A
                  (and B C)))

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

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

Implication chains with binary justifiers

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`:

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

`conj-intro` can be regarded as a generalization of `both`. It 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.

Implication chains with binary justifiers

`existence` allows us to go from p to an existential generalization of p:

```
> (!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))))
```

`existence` can be implemented as follows:

```
define existence :=
  method (premise eg-goal)
    match (match-sentences premise (quant-body eg-goal)) {
     (some-sub sub) => (!egen* eg-goal (sub (qvars-of eg-goal)))
    }
```

where `match-sentences` returns a substitution `sub`, and `quant-body` and `qvars-of` return the body and the quantified variables of a quantified sentence.

Binary procedure `with` as justification

Consider the following implication chain:

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

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

Theorem: (if (not B)
             (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, namely `(~ B)`, will successfully derive the right-hand side, `(~ A)`, as required by the specification of `chain`.

Binary procedure `with` as justification

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)
```

Binary procedure `with` as justification

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`.

Binary procedure `with` as justification

Using `with` in infix notation, 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)`."

Binary procedure `with` as justification

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`.

Non-local arguments can be listed in any order.

Binary procedure `with` as justification

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 write the element by itself.

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:

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

where $k, n \geq 0$, $m > 0$.

Using sentences as justifiers

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)
```

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 exceedingly common. 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))
```

Using sentences as justifiers

The chain implementation

- 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`,

- 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.

Using sentences as justifiers

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))
```

Conjuncts can be listed in any order on the left-hand side of the step.

Using sentences as justifiers

Biconditionals can also be used as rules.

```
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))
```

Using sentences as justifiers

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

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

where $k, n \geq 0$ and $m > 0$, e.g.:

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

> (!chain [(x = null) ==> (x intersection y = null) [R]])
Theorem: (if (= ?x:Set null)
             (= (intersection ?x:Set ?y:Set)
                null))
```

Using sentences as justifiers

The right hand side of an implication admits a single conjunct:

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

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

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

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

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

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

Using sentences as justifiers

For rules of the form:

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

`chain` can also be used in the contrapositive direction:

- If p matches the *complement* of *some* consequent conjunct, and
- q matches
  - either the complement of the antecedent
  - or else

    $$(p'_1 \texttt{ | } \cdots \texttt{ | } p'_n)$$

    where p'_i is the complement of p_i,
- then the step goes through.

Using sentences as justifiers

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

> (!chain [(~ (D & E & F)) ==> (~ (A & B & C)) [R]])
Theorem: (if (not (and D
                      (and E F)))
             (not (and A
                       (and B C))))

> (!chain [(~ F) ==> (~ (A & B & C)) [R]])
Theorem: (if (not F)
             (not (and A
                       (and B C))))

> (!chain [(~ E) ==> (~A | ~B | ~C) [R]])
Theorem: (if (not E)
             (or (not A)
                 (or (not B)
                     (not C))))
```

Using sentences as justifiers

Similarly for the form:

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

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)
```

Using sentences as justifiers

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)))
```

Using sentences as justifiers

It is possible to use "rules" without an explicit antecedent. 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)))
```

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.

Nested rules

Consider:

```
declare A,B: Set
assert* subset-definition :=
   (s1 subset s2 <==> forall x . x in s1 ==> x in s2)
assert (A subset B)

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]])
```

Athena allows us to use the outer rule directly:

```
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)))
```

Structural implication chaining

There is another noteworthy way of enabling an implication-chain step from p to q: by using the structure of sentences 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 $(R\ s_1 \cdots s_n)$ and $(R\ t_1 \cdots t_n)$
  - if J is a justifier (e.g., a list of identities and/or conditional identities) licensing the conclusions $(s_i = t_i)$ for $i = 1, \ldots, n$,
  - then the step

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

will succeed (in some appropriate assumption base β).

Structural implication chaining

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

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.

Structural implication chaining

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.

Structural implication chaining

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)))
```

Structural implication chaining

Ultimately, steps of this form succeed owing to the validity of the following inference rule, where $\odot \in \{\wedge, \vee\}$:

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

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

Structural implication chaining

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)))
```

Using chains with chain-last

- Sometimes when we put together an implication chain of the form

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

  we are operating in an assumption base that contains p_1.
- In such cases we are usually not content with merely showing that p_1 implies p_{n+1}. Instead, we want to derive 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->`.

Using chains with chain-last

For instance, the `chain-last` call below will produce the conclusion `B`:

```
> 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

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

is equivalent to:

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

Using chains with chain-last

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))
```

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)
```

Backward chains and chain-first

Suppose now we want to prove that for any 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]])
```

Starting with a goal and going back to necessary conditions is known as *backward chaining*.

Backward chains and chain-first

- In symmetry 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.

Backward chains and chain-first

For example:

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

assert* p-def := (x parent y <==> x = father y | x = mother y)
assert* gp-def := (x grandparent z <== x parent y & y parent z)

assert fact1 := (Mary = mother Bob)
assert fact2 := (Peter = father Mary)

> (!chain-first
     [(Peter grandparent Bob)
   <== (Peter parent Mary & Mary parent Bob)        [gp-def]
   <== (Peter = father Mary & Mary = mother Bob)    [p-def]
   <== (true & true)                               [fact1 fact2]
   <== true                                        [augment]])

Theorem: (grandparent Peter Bob)
```

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

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

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 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.

Equivalence chains

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`.

Equivalence chains

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)))
```

Equivalence chains

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

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

The rule 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:

```
> assume  R := (A & B <==> A)
  (!chain [(~ (A & B)) <==> (~ A) [R]])

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

Equivalence chains

Whereas implication chaining ([*SC*]) could not make such inference step:

```
>assume  R := (A & B ==> A)
  (!chain [(~ (A & B)) ==> (~ A) [R]]);;

Error: Implicational chaining error
on the 1st step of the chain, in going from:
(not (and A B))
to:
(not A)
.
```

since from $p \implies q$, we cannot conclude $\neg p \implies \neg q$.

Equivalence chains

The quantified analogues of the rule hold as well:

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

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

Equivalence chains

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))))))
```

Mixing 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:

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

Mixing implication and equivalence steps

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)))
```

Even with only one implication step, the conclusion is a conditional.

Mixing implication and equivalence steps

- 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)))
```

Mixing equational steps

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:

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

This can be useful when we first establish an identity $t_1 = t_{n+1}$ by rewriting and then use implication chaining to conclude the last element of the chain, p_{m+1}.

Mixing equational steps

For example, suppose we have defined the `divides` relation and 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]])
```

Mixing equational steps

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.

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

If `chain` is used instead of `chain->`, then the conditional $p_1 \Rightarrow s = t_m$ is returned.

Mixing equational steps

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

Mixing equational steps

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

```
> 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))))))
```

Mixing equational steps

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

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

Chain nesting example

To prove by induction:

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

The base case from premises `R1`, . . . , `R3` follows:

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

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

`[(y = zero) <== hyp [R3]]` can be viewed as an arg. to `chain-<`.