CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Chapter 4.10, 4.14

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

Sentential logic

Goal: to become familiar with *propositional logic* proofs.

- *Boolean constants*
- *conjunctions*
- *conditionals*
- *disjunctions*
- *negations*
- *biconditionals*
- recursive proof methods
- proof heuristics

Recursive proof methods

Consider an iterative version of double negation: a method dn* that takes a premise with an arbitrary number of pairs of negation signs in front of it, and eliminates all such pairs by repeated applications of dn.

- Applying dn* to (~ ~ A) gives A, (same as calling dn.)
- Applying dn* to (~ ~ ~ A) gives (~ A) (same as calling dn);
- Applying dn* to (~ ~ ~ ~ A) gives A;
- Applying dn* to (~ ~ ~ ~ ~ A) gives (~ A); and so on.

The method should have no effect on an input that does not have at least two negation signs in front of it. For example, applying dn* to A should result in A.

Recursive proof methods

The following implementation of dn* fits the bill:

```
define (dn* premise) :=
  match premise {
    (~ (~ p)) => let {_ := conclude p
                                (!dn premise)}
                      (!dn* p)
  | _ => (!claim premise)
  }
```

- When the given premise has at least two negation signs in front of it, that is, when it is of the form (~ ~ p), dn* does two things:
  - it derives the conclusion p by applying dn to the premise;
  - it recursively calls itself with p as the input.
- When all double negation signs have been eliminated dn* will simply claim the given premise.

Recursive proof methods

A more succinct way of defining dn* is the following:

```
define (dn* premise) :=
  match premise {
    (~ (~ p)) => (!dn* (!dn premise))
  | _ => (!claim premise)
  }
```

In general, a deduction of the form

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

is equivalent to `(!M D)`.

- Recall the assumption base when evaluating `M` will include the conclusion of `D` (by *dynamic assumption scoping*.)
- Successive recursive calls of dn* are made in increasingly larger assumption bases.

Recursive proof methods

Consider a method that takes a conditional premise of the form

$$(p \texttt{ ==> } q \texttt{ ==> } r)$$

and derives $(p \texttt{ \& } q \texttt{ ==> } r)$.

This is traditionally known in logic as *importation*.

```
define (import premise) :=
  match premise {
    (p ==> (q ==> r)) => assume (p & q)
                           let {q=>r := (!mp premise p)}
                             (!mp q=>r q)
  }
```

Recursive proof methods

Suppose now that we want a more general version of this method, one that can take an arbitrary conditional of the form
$(p_1$ ==> p_2 ==> p_3 ==> $\cdots)$ and repeatedly apply importation.
For example, obtaining $(((p_1$ & $p_2)$ & $p_3)$ ==> $p_4)$ from

$$(p_1 \texttt{ ==> } p_2 \texttt{ ==> } p_3 \texttt{ ==> } p_4).$$

A recursive method is again the solution:

```
define (import* premise) :=
  match premise {
    (p1 ==> (p2 ==> p3)) => (!import* (!import premise))
  | _ => (!claim premise)
  }
```

A "bidirectional" double-negation method

- dn works only in one direction: It goes from a doubly negated premise (~ ~ p) to p.
- It is often useful to be able to go in the other direction whereby we insert two negation signs in front of an arbitrary premise.
- The method idn below implements that direction:

```
define (idn premise) :=
  (!by-contradiction (~ ~ premise)
     assume -premise := (~ premise)
       (!absurd premise -premise))
```

A "bidirectional" double-negation method

It is also useful to have a "bidirectional" double-negation method, bdn, which

- inserts two negation signs in front of the given premise p *only* when p is not already doubly negated;
- if it is, then it applies dn to it to remove the two leading negation signs:

```
define (bdn premise) :=
  match premise {
    (~ (~ _)) => (!dn premise)
  | _ => (!idn premise)
  }
```

A “bidirectional” double-negation method

Sample use:

```
> assume h := (~ ~ A)
    (!bdn h)

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

> assume A
    (!bdn A)

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

The replacement rule

The *replacement rule* lets us replace an arbitrary part q of a premise p by any equivalent sentence q'.

For example, suppose we have

$$p = \texttt{(A \& \~{} \~{} B)} \tag{1}$$

in the assumption base. Since `(~ ~ B)` is equivalent to `B`, and specifically since we know how to derive each from the other, via the `bdn` method, we should be able to conclude

$$\texttt{(A \& B).} \tag{2}$$

The conclusion is obtained from p by replacing the double negation `(~ ~ B)` with the equivalent `B`.

The replacement rule

- Our reasoning can be roughly expressed as follows: (2) follows from (1) by virtue of the bdn method.
- More precisely, this inference will be captured by the method call

```
(!transform (A & ~ ~ B) (A & B) [bdn]).
```

- This should be read as: "Transform the first argument (the premise) into the second one by replacement, using the method bdn."

The replacement rule

As another example, suppose the following conditional is in the assumption base:

```
(~ ~ A ==> ~ (B & C)).
```
(3)

From De Morgan's laws, we know that the consequent is equivalent to `(~ B | ~ C)`, while from double negation we know that the antecedent is equivalent to `A`. The `transform` method will allow us to derive

```
(A ==> ~ B | ~ C)
```

from (3) simply by citing these two methods, `dm` (De Morgan) and `bdn` (two-way double negation):

```
(!transform (~ ~ A ==> ~ (B & C)) (A ==> ~ B | ~ C) [bdn dm]).
```

The replacement rule

In general, then, calls to `transform` will be of the form

$$\texttt{(!transform } p \ q \texttt{ [} M_1 \cdots M_n \texttt{])}$$

and should be read as: "Transform p into q by replacement, using the methods $M_1 \cdots M_n$."

- The desired conclusion q should be obtainable from p by replacing $k \geq 0$ subsentences of it, $p_1, \ldots, p_k$, by sentences $q_1, \ldots, q_k$, where each equivalence $(p_i \texttt{ <==> } q_i)$, $i = 1, \ldots, k$, is provable by some method M_j, $j \in \{1, \ldots, n\}$.
- The starting sentence p must be in the assumption base, and each M_i will (usually) be a bidirectional unary method, such that if $(!M_i \ p) = q$ then $(!M_i \ q) = p$, in all appropriate assumption bases.

Implementing the replacement rule

We will use a ternary method `prove-equiv`, which

- takes two sentences p and q as inputs and a list of two-way methods `[`$M_1 \cdots M_n$`]` and
- derives the biconditional `(`p `<==>` q`)`, provided that q is obtainable from p by replacing $k \geq 0$ subsentences of it, $p_1, \ldots, p_k$, by sentences $q_1, \ldots, q_k$, where each equivalence `(`p_i `<==>` q_i`)` is provable by some M_j, $j \in \{1, \ldots, n\}$.

With `prove-equiv`, we can readily implement `transform`:

```
define (transform p q methods) :=
  let {E := conclude (p <==> q)
              (!prove-equiv p q methods)}
    conclude q
      (!mp (!left-iff E) p)
```

Implementing `prove-equiv`

- If there is some method M_j that can derive q from p and vice versa, we prove the equivalence of p and q by applying that method.
- If not, then we check if p and q are syntactically identical, and if so, we can readily prove $(p \iff p)$ for any p by:

```
define (ref-equiv p) :=
  (!equiv assume p (!claim p)
          assume p (!claim p))
```

- Finally, if p and q are neither M_j-interconvertible nor syntactically identical, they must be nonatomic sentences obtained by applying the same sentential constructor, i.e., they must both be negations, or both conjunctions, etc.

Implementing `prove-equiv`

Say p and q are both negations of the form `(~` p_1`)` and `(~` q_1`)`, respectively.

- We then recursively apply `prove-equiv` to p_1 and q_1 (and the same list of methods `[`$M_1 \cdots M_n$`]`), which will presumably result in the biconditional `(`p_1 `<==>` q_1`)`.
- But then it is a simple matter to derive `(`p `<==>` q`)`, since for any p_1 and q_1, `(~` p_1 `<==> ~` q_1`)` is derivable from `(`p_1 `<==>` q_1`)`.

Implementing `prove-equiv`

The method `not-cong` to derive `(~` p_1 `<==> ~` p_2`)` from `(`p_1 `<==>` p_2`)`:

```
define (not-cong premise) :=
  match premise {
    (p1 <==> p2) =>
      let {-p1=>-p2 := assume -p1 := (~ p1)
                         (!by-contradiction (~ p2)
                           assume p2
                             (!absurd (!mp (!right-iff premise) p2)
                                      -p1));
           -p2=>-p1 := assume -p2 := (~ p2)
                         (!by-contradiction (~ p1)
                           assume p1
                             (!absurd (!mp (!left-iff premise) p1)
                                      -p2))}
        (!equiv -p1=>-p2 -p2=>-p1)
  }
```

Implementing `prove-equiv`

The reasoning for conjunctions is similar. Say p and q are respectively of the form `(`p_1 `&` p_2`)` and `(`q_1 `&` q_2`)`.

- We place two recursive calls to `prove-equiv`,
  - one applied to p_1 and q_1 (and the given list of methods);
  - and the other applied to p_2 and q_2 (and said list).
- These will presumably result in the biconditionals `(`p_1 `<==>` q_1`)` and `(`p_2 `<==>` q_2`)`, and
- From these two, the biconditional `(`p `<==>` q`)` is deduced by the `and-cong` method.

Implementing `prove-equiv`

and-cong method to derive $(p_1$ `&` p_2 `<==>` q_1 `&` $q_2)$ from $(p_1$ `<==>` $q_1)$ and $(p_2$ `<==>` $q_2)$

```
define (and-cong premise1 premise2) :=
  match [premise1 premise2] {
    [(p1 <==> q1) (p2 <==> q2)] =>
        let {p1&p2=>q1&q2 := assume (p1 & p2)
                                (!both (!mp (!left-iff premise1) p1)
                                       (!mp (!left-iff premise2) p2));
             q1&q2=>p1&p2 := assume (q1 & q2)
                                (!both (!mp (!right-iff premise1) q1)
                                       (!mp (!right-iff premise2) q2))}
          (!equiv p1&p2=>q1&q2 q1&q2=>p1&p2)
  }
```

The treatment of the remaining cases is analogous, and requires corresponding congruence methods for disjunctions, conditionals, and biconditionals.

Implementing `prove-equiv`

We introduce an auxiliary method `find-first-element` that

- takes a unary method M and an arbitrary list $L = \texttt{[}V_1 \cdots V_m\texttt{]}$ (of values of any type), and
- repeatedly applies M to each V_i in turn, $i = 1, \ldots, m$, until one such application succeeds, in which case the conclusion returned will be whatever result is produced by $(\texttt{!}M\ \ V_i)$.
- If $(\texttt{!}M\ \ V_i)$ fails for every $i = 1, \ldots, m$, then the entire call to `find-first-element` fails.

```
define (find-first-element M list) :=
  match list {
    (list-of first rest) =>
      try { (!M first) | (!find-first-element M rest) }
  }
```

Implementing `prove-equiv`

Finally, we introduce an auxiliary selection procedure `choose-cong-method` that takes an arbitrary sentential constructor and chooses the appropriate congruence method:

```
define (choose-cong-method pc) :=
  match pc {
    &    => and-cong
  | |    => or-cong
  | ==>  => if-cong
  | <==> => iff-cong
  }
```

Implementing prove-equiv

We can now define prove-equiv as follows:

```
define (prove-equiv p q methods) :=
  try {
    (!find-first-element method (M)
                          try { (!equiv assume p (!M p)
                                        assume q (!M q))
                              | (!equiv assume p (!M p q)
                                        assume q (!M q p)) }
                         methods)
  | check {
      (p equals? q) => (!ref-equiv p)
    | else => match [p q] {
                [(~ p1) (~ q1)] =>
                    (!not-cong (!prove-equiv p1 q1 methods))
              | [((some-sent-con pc) p1 p2) (pc q1 q2)] =>
                    (!(choose-cong-method pc)
                        (!prove-equiv p1 q1 methods)
                        (!prove-equiv p2 q2 methods)) } } }
```

Proof heuristics for sentential logic

Recall the proof of the conditional

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

from two premises. What process led us to this proof?

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

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

Proof heuristics for sentential logic—Goal tree

Derive `(~ B ==> A ==> ~ E)` from $\beta = \{$`(A & B | A ==> C), (C <==> ~ E)`$\}$

```
assume (~ B)
  (!force (A ==> ~ E))
```

Derive `(A ==> ~ E)` from $\{$`(~ B), (A & B | A ==> C), (C <==> ~ E)`$\}$

```
assume A
  (!force (~ E))
```

Derive `(~ E)` from $\{$`A, (~ B), (A & B | A ==> C), (C <==> ~ E)`$\}$

```
(!cases (A & B | A ==> C))
        assume (A & B)
          (!force (~ E))          # Sub-goal 1
        assume (A ==> C)
          (!force (~ E)))         # Sub-goal 2
```

Each force creates a new sub-goal. Both sub-goals must be proven.

Sub-goal 1 tree

Derive `(~ E)` from {`(A & B)`, `A`, `(~ B)`, `(A & B | A ==> C)`, `(C <==> ~ E)`}

```
let {_ := (!force B)}
  (!from-complements (~ E) B (~ B))
```

Derive `B` from {`(A & B)`, `A`, `(~ B)`, `(A & B | A ==> C)`, `(C <==> ~ E)`}

```
let {conj := (!force (A & B))}
  (!right-and conj)
```

Derive `(A & B)` from {`(A & B)`, `A`, `(~ B)`, `(A & B | A ==> C)`, `(C <==> ~ E)`}

```
(!claim (A & B))
```
$\surd$

No **force** means that we are finished with the sub-goal proof.

Sub-goal 2 tree

Derive `(~ E)` from {`(A ==> C)`, `A`, `(~ B)`, `(A & B | A ==> C)`, `(C <==> ~ E)`}

```
let {imp := (!left-iff (C <==> ~ E));
     _   := (!force C)}
  (!mp imp C)
```

Derive `C` from {`(A ==> C)`, `A`, `(~ B)`, `(A & B | A ==> C)`, `(C <==> ~ E)`}

```
let {imp := (!force (A ==> C));
     _   := (!force A)}
  (!mp imp A)
```

Derive `(A ==> C)` from {`(A & B)`, `A`, `(~ B)`, `(A ==> C)`, `(C <==> ~ E)`}

`(!claim (A ==> C))` $\surd$

Derive `A` from {`(A & B)`, `A`, `(~ B)`, `(A ==> C)`, `(C <==> ~ E)`}

`(!claim A)` $\surd$

Goal trees

Goal trees abstract over the process to construct a proof.

- Every node in such a tree contains a goal, which in the present context is a specification for a proof, or a *proof spec* for short.
- A proof spec is of the form "Derive p from β" where
  - β is some assumption base, that is, a finite set of available assumptions, and
  - p is the desired conclusion that must be derived from β.
- We use the letter τ as a variable ranging over proof specs (goals), which by convention, are enclosed inside a box.
- Occasionally we also refer to the target conclusion p in a proof spec as a "goal."

Tactics

Applying a *tactic* to a proof spec τ produces two things:

1. A partial deduction D—partial in that it may contain $n \geq 0$ applications of **force**.
   - This partial deduction D is capable of deriving the target sentence p from the corresponding assumption base β, provided that the n applications of **force** have been successfully replaced by proper deductions.
2. A list of proof specs $\tau_1, \ldots, \tau_n$ that capture the proof obligations corresponding to the n applications of **force** inside D.
   - We refer to these new proof specs as the *subgoals* generated by the tactic.

Tactics

We tackle sub-goals $\tau_1, \ldots, \tau_n$ by applying new tactics to each τ_i, thereby expanding the goal tree to deeper levels.

- The process ends when the deductions at the leaves of the tree contain no occurrences of **force**.
  - This typically happens when the deductions are single applications of `claim`.
- We visually mark such leaves by placing the sign $\surd$ right next to the corresponding deductions.

When the process is complete, we can scan the tree bottom up completing partial deductions at every internal node, by incrementally eliminating applications of **force**, culminating with the top-level deduction at the root of the tree.

Tactics

In general, every time we are faced with a proof spec, there are two ways to proceed:

- *goal-driven*, or *backward*, or *analytic* approach
  - it tries to move from the target conclusion toward the premises—it "analyzes" or decomposes the target sentence, usually breaking it up into smaller components.
- *information-driven* or *forward* or *synthetic* approach
  - it moves from the premises toward the target conclusion—it tries to "synthesize" the conclusion by utilizing the premises.

Backward tactics

- In the goal-driven approach, we ignore the set of available assumptions β (i.e., we ignore the available *information*), focusing instead on the syntactic form of the target sentence.
- In particular, we look at the main logical connective of that sentence, and we let that dictate the overall form that the desired deduction will take, by using the corresponding introduction mechanism or some closely related method.
- Since there are five logical connectives in sentential logic, there are five corresponding backward tactics, depending on the logical structure of the target conclusion.

Backward tactics—conjunctive goals

Conjunctive goals:

Derive $(p$ `&` $q)$ from β

```
let {left := (!force p);
     right := (!force q)}                    [and<-]
  (!both left right)
```

The choice of deriving the left conjunct (p) first is arbitrary. We could just as well choose to derive q first and p afterward.

`[and<-]` is a mnemonic name for the tactic (*backward tactic* applicable to *conjunction*.)

Backward tactics—disjunctive goals

Disjunctive goals:

Derive $(p \mid q)$ from β

```
let {_ := (!force p)}                    [lor<-]
  (!left-either p q)
```

`[ror<-]` is a similar pattern that derives q first and then introduces $(p \mid q)$ via `right-either`.

An alternative—and often more effective—tactic for deriving a disjunction $(p \mid q)$ is to treat it as the conditional $(\sim p \Longrightarrow q)$:

Derive $(p \mid q)$ from β

```
let {cond := (!force (~ p ==> q))}       [or-if<-]
  (!cond-def cond)
```

Backward tactics—conditional goals

Derive (*p* ==> *q*) from β

```
assume p
  (!force q)                    [if<-]
```

Derive q from $\beta \cup \{p\}$

Note the augmented assumption base in the child proof spec.

Backward tactics—negated goals

Derive `(~ p)` from β

```
(!by-contradiction (~ p)
  assume p                          [not<-]
    (!force false))
```

`false` is usually derived by `absurd`, so the task of the new proof spec will usually consist in the derivation of two contradictory sentences q and `(~ q)`, followed by an application of `absurd`.

Later we will give a more specific tactic for deriving `false`.

Backward tactics—biconditional goals

Derive $(p$ <==> $q)$ from β

```
let {left := (!force (p ==> q));
     right := (!force (q ==> p))}                    [iff<-]
  (!equiv left right)
```

As with conjunctions, the ordering of the two subgoals is immaterial.

Forward tactics

In the information-driven approach,

- we focus on the set of available assumptions β and
- try to find one or more sentences in β that we can use to derive the goal.

Typically, the resulting proof will consist of a sequence of inference steps assembled together in a single proof block (or a `let` if we choose to use naming).

Forward tactics—claim

The ideal situation in forward proof search occurs when the assumption base contains the target sentence, in which case we can simply `claim` it:

Derive p from $\{\ldots, \underline{p}, \ldots\}$

`(!claim` p`)` $\surd$ `[claim->]`

- Note that this tactic does not generate any subgoals, as there are no applications of **force** in the given deduction.
- Goals of this form (that generate no subgoals) will be called *primitive*.
- Thus, primitive goals become leaf nodes of goal trees.

Forward tactics—conjunctions

The next best-case scenario occurs when the assumption base contains a sentence from which the goal can be detached in one step, by a single application of an elimination method.

For instance, if our goal is q and the assumption base contains a conjunction `(`p `&` q`)`, then clearly we can derive q in one step by applying `right-and` to `(`p `&` q`)`.

We can depict this forward proof tactic as follows:

Derive q from $\{\ldots, \underline{\texttt{(}p\ \texttt{\&}\ q\texttt{)}}, \ldots\}$

`(!right-and (`p `&` q`))` $\surd$

Forward tactics—conjunctions

We generalize the preceding proof tactic for conjunctions as follows:

Derive q from $\beta = \{\ldots, (\cdots \underline{(p \texttt{ \& } q)}^{+} \cdots), \ldots\}$

```
let {conj := (!force (p & q))}                    [rand->]
  (!right-and conj)
```

- We indicate that an occurrence of a sentence q inside a sentence p is positive by attaching a $^{+}$ superscript to it. For example,

  $$p = (\cdots (\texttt{A ==> B})^{+} \cdots)$$

  signifies that the superscripted occurrence of the conditional `(A ==> B)` in the surrounding sentence p is positive.
- It suffices to consider only sentences with positive subsentences.

There is an obvious analogue for left conjuncts: `[land->]`

Forward tactics—conditionals

Suppose that the goal is to find a derivation of q from some assumption base β.

- If the assumption base contains a positive occurrence of a conditional of the form $(p$ `==>` $q)$,
- then we can try to derive that conditional from β,
- then derive the antecedent p,
- and finally obtain q via modus ponens:

Derive q from $\beta = \{\ldots, (\cdots \underline{(p \texttt{ ==> } q)}^{+} \cdots), \ldots\}$

```
let {cond := (!force (p ==> q));
     ant  := (!force p)}                    [if->]
  (!mp cond ant)
```

Forward tactics—disjunctions

Suppose that the parent of the goal q is a disjunction, say $(q \mid p)$. Then once we derive that disjunction from the assumption base, we can perform a case analysis. Specifically, we have to show that q follows no matter which disjunct obtains, so we have to show that

- (a) q follows from the first disjunct, q; and
- (b) q follows from the second disjunct, p.

Part (a) is trivial, so (b) will usually be more challenging:

Derive q from $\{\ldots, (\cdots \underline{(q \mid p)}^{+} \cdots), \ldots\}$

```
let {disj := (!force (q | p));
     cnd1 := assume q (!claim q);                    [lor->]
     cnd2 := (!force (p ==> q))}
  (!cases disj cnd1 cnd2)
```

Forward tactics—biconditionals

Finally, we have similar forward tactics involving biconditionals:

Derive q from $\beta = \{\ldots, (\cdots \underline{(p \texttt{ <==> } q)}^{+} \cdots), \ldots\}$

```
let {bcond := (!force (p <==> q));
     cond  := (!left-iff bcond);                    [liff->]
     _     := (!force p)}
  (!mp cond p)
```

As should be expected, there are corresponding `[ror->]` and `[riff->]` tactics.

Forward extraction tactics

The seven forward tactics are collectively called the *extraction tactics*. Given the goal of deriving some q from some β, these tactics::

1. Find some element of β with a positive occurrence of a parent of the goal q, specifically, a positive occurrence of a sentence of the following forms:
   - $(p$ `&` $q)$ or $(q$ `&` $p)$;
   - $(p$ `==>` $q)$;
   - $(p$ `|` $q)$ or $(q$ `|` $p)$;
   - $(p$ `<==>` $q)$ or $(q$ `<==>` $p)$.
2. Derive that parent, and possibly other additional subgoals.
3. Detach the goal q from the parent via the proper elimination method.

Forward tactics example goal tree

Derive `C` from $\beta = \{$`(A ==> (B & C)`$^{+}$`, (A & E)`$\}$

```
let {conj := (!force (B & C))}
  (!right-and conj)
```

Derive `(B & C)` from $\beta = \{$`(A ==> (B & C))`$^{+}$`, (A & E)`$\}$

```
let {cond := (!force (A ==> B & C));
     ant  := (!force A)}
  (!mp cond ant)
```

Derive `(A ==> B & C)` from $\{$`(A ==> (B & C)), (A & E)`$\}$

```
(!claim (A ==> B & C))
```
$\surd$

Derive `A` from $\{$`(A ==> (B & C)), (A & E)`$^{+}\}$

```
let {conj := (!force (A & E))
  (!left-and conj)
```

Derive `(A & E)` from $\{$`(A ==> (B & C)), (A & E)`$\}$

```
(!claim (A & E))
```
$\surd$

Forward tactics example proof

The following is the final assembled proof:

```
let {conj := let {cond := (!claim (A ==> B & C));
                 ant :=  let {conj := (!claim (A & E))}
                           (!left-and conj)}
             (!mp cond ant)}
 (!right-and conj)
```

or, if we remove the extraneous claims:

```
let {conj := (!mp (A ==> B & C)
                  (!left-and (A & E)))}
  (!right-and conj)
```

Forward tactics—general disjunction

A more general forward tactic involving disjunctions is the following:

Derive q from $\{\ldots, (\cdots \underline{(p_1 \texttt{ | } p_2)}^{+} \cdots), \ldots\}$

```
let {disj := (!force (p1 | p2));
     cnd1 := (!force (p1 ==> q));                    [or->]
     cnd2 := (!force (p2 ==> q))}
  (!cases disj cnd1 cnd2)
```

This tactic subsumes `[lor->]` and `[ror->]`, both of which can be viewed as instances of `[or->]`.

It can be used with arbitrary disjunctions in (positive positions of) the assumption base.

Forward tactics—contradictions

Indirect proof, or proof by contradiction (we write $\overline{p}$ for the complement of p):

Derive p from β

```
(!by-contradiction p
  assume p̄                    [ift]
    (!force false))
```

When the goal p is a negation, this tactic reduces to the backward tactic `[not<-]`.

But, unlike `[not<-]`, this tactic can be used even when the goal is not a negation.

Forward tactics—contradictions

Both `[ift]` and `[not<-]` seek to derive `false` from an assumption base.

Are there any heuristics offering guidance on how to derive `false`?

- Clearly, if the assumption base contains two complementary sentences then a single application of `absurd` will suffice, but what should be done in other cases?
- A simple heuristic is this: look for a negation `(~` p`)` in a positive position in the assumption base, derive it, and then try to derive p:

Derive `false` from $\beta = \{\ldots, (\cdots \underline{(\sim p)}^{+} \cdots), \ldots\}$

```
let {neg := (!force (~ p));
     pos := (!force p);                    [false->]
  (!absurd pos neg)
```

Forward tactics—contradictions

If we already have $\overline{p}$ in the assumption base and we can also derive the positively embedded p, then clearly the assumption base is inconsistent and any goal whatsoever can be derived from it:

Derive q from $\beta = \{\ldots, \overline{p}, \ldots, (\cdots p^{+} \cdots), \ldots\}$

```
let {_ := (!force p)}                                    [cft]
  (!from-complements q p p̄)
```

This tactic is particularly apt when the assumption base directly contains p as well as its complement $\overline{p}$, in which case the tactic takes the following form:

Derive q from $\beta = \{\ldots, p, \ldots, (\sim p), \ldots\}$

```
(!from-complements q p (~ p))
```

Replacement tactics

Recall that we are allowed to replace any part of a sentence by an equivalent part.

- We can take advantage of such replacements both in a forward and in a backward direction.
- First, in a backward direction, we can transform the goal q to an equivalent goal q' that may be easier to tackle.
- If and when we succeed in deriving q', and assuming that q' and q are indeed equivalent in that each can be derived from the other by certain bidirectional methods $M_1, \ldots, M_n$, we can then `transform` q' to the desired q.

Replacement tactics

Derive q from β

```
let {_ := (!force q')}                    [replace<-]
  (!transform q' q [M_1 ··· M_n])
```

- Likewise, in a forward direction, we can convert any element of the assumption base into an equivalent form and then proceed to tackle the goal anew:

Derive q from $\beta = \{\ldots, \underline{p}, \ldots\}$

```
let {_ := (!transform p p' [M_1 ··· M_n])}    [replace->]
  (!force q)
```

Heuristic for ranking tactics

Tactics should be considered for application in the following order:

- reiteration (`[claim->]`) and constant tactics (`[true<-]` and `[false<-]`);
- the complement tactic (`[cft]`);
- extraction tactics;
- replacement tactics;
- backward tactics, with the exception of `[not<-]`;
- generalized disjunction tactic;
- indirect tactic and `[not<-]`.

Heuristic for ranking tactics

- Replacement tactics should be used so as to make other tactics applicable.
  - A particularly useful backward replacement is the transformation of a disjunctive goal into a conditional, via `cond-def`.
- Multiple extraction tactics might be applicable, so a selection algorithm is necessary for such cases.
- If a cycle is encountered, backtrack to the first occurrence of the duplicate subgoal and try another tactic.

Heuristic for ranking tactics

To select among multiple extraction tactics:

- When the assumption base has two or more parents of the goal in positive positions, choose the parent that is embedded in the least complex (smallest) sentence, unless there is a parent embedded in a pure conjunction, in which case that should be the selection.
- If two or more parents are positively embedded in sentences of the same complexity, break ties by choosing conjunctions over conditionals, conditionals over biconditionals, and any of these over disjunctions.

Applying the heuristic for ranking tactics

Consider the problem of finding a proof D that derives the following goal from the given premises:

```
assert premise-1 := (A | (B ==> C))
assert premise-2 := (C <==> D & E)
assert premise-3 := (B & ~ A)

define goal := (B & D)
```

Applying the heuristic for ranking tactics

```
let {left := (!left-and (B & ~ A));
     right :=
       let {conj :=
              let {bcond := (!claim (C <==> D & E));
                   _ := let {disj := (!claim (A | (B ==> C)));
                             cnd1 := assume A
                                       (!from-complements
                                         (B ==> C)
                                         A
                                         (!right-and premise-3));
                             cnd2 := assume (B ==> C)
                                       (!claim (B ==> C));
                             cond := (!cases disj cnd1 cnd2)}
                        (!mp cond B)}
               (!mp (!left-iff bcond) C)}
         (!left-and conj)}
  (!both B D)
```

Applying the heuristic for ranking tactics

Or, after a few simplifications and some general cleaning up:

```
let {left := (!left-and (B & ~ A));
     right := let {B=>C := (!cases (A | (B ==> C))
                                 assume A
                                   (!from-complements
                                     (B ==> C)
                                     A
                                     (!right-and premise-3))
                                 assume h := (B ==> C)
                                   (!claim h));
                    _ := (!mp B=>C B);
                    D&E := (!mp conclude (C ==> D & E)
                                   (!left-iff premise-2)
                                  C)}
                (!left-and D&E)}
  (!both B D)
```

Proof heuristics summary

- Given that we typically have a choice of several tactics that we can apply to a given goal, we need some guidance on how to select a tactic judiciously.
- An inauspicious choice might result in a kludgy proof, or it might lead us to a dead end or get us stuck in a cycle. Then we have to backtrack and try another tactic.
- Choosing cleverly which tactics to apply can avoid such pitfalls and lead to success quickly and elegantly.
- Heuristics provide some guidance on this selection task. They are not guaranteed to always work (which is why they are only heuristics after all), but they are generally helpful.