CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Chapter 4.1-4.8

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*

Sentential logic

Sentential logic is concerned with zero-order sentences:

- either a Boolean term, say, `(zero < S zero)`,
- or the result of applying one of the five sentential connectives (`not`, `and`, `or`, `if`, `iff`) to other zero-order sentences.

Working with the Boolean constants

We can derive `true` any time by applying the nullary method `true-intro`:

```
> (!true-intro)

Theorem: true
```

The constant `false` can only be derived if the assumption base is inconsistent. Applying the binary method `absurd` to p and `(~` p`)` will derive `false`:

```
> assume A
    assume (~ A)
      (!absurd A (~ A))

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

Working with the Boolean constants

Finally, we can derive `(~ false)` at any time through the nullary method `false-elim`:

```
> (!false-elim)

Theorem: (not false)
```

Using conjunctions

The unary method `left-and` takes a conjunction `(`p `&` q`)` that is present in the assumption base, and produces the conclusion p:

```
assert p := (A & B)

> (!left-and p)

Theorem: A
```

There is a similar unary method, `right-and`, that does the same thing for the right component of a conjunction: If `(`p `&` q`)` is in the assumption base,

`(!right-and (`p `&` q`))`

will produce the conclusion q.

Deriving conjunctions

Given any two sentences p and q in the assumption base, the binary method call

$$\texttt{(!both } p \; q\texttt{)}$$

will produce the conclusion (p & q).

As an example that uses all three methods dealing with conjunctions, consider the derivation of `(D & A)` from the premises `(A & B)` and `(C & D)`:

```
assert A-and-B := (A & B)
assert C-and-D := (C & D)

> (!both (!right-and C-and-D)
         (!left-and  A-and-B))

Theorem: (and D A)
```

Using conditionals: modus ponens

Starting from two premises of the form (p ==> q) and p, modus ponens yields the conclusion q, thereby detaching ("eliminating") the conditional connective.

- In Athena, modus ponens is performed by the primitive binary method `mp`.
- When the first argument to `mp` is of the form (p ==> q), the second argument is p, and both arguments are in the assumption base, `mp` will derive q.
- For instance, assuming that `(A ==> B)` and `A` are both in the assumption base, we have:

```
> (!mp (A ==> B) A)

Theorem: B
```

Using conditionals: modus tollens

Given two premises of the form (p `==>` q) and (`~` q), modus tollens generates the conclusion (`~` p).

- In Athena, modus tollens is performed by the binary method `mt`.
- When the first argument to `mt` is a conditional (p `==>` q), the second is (`~` q), and both are in the assumption base, `mt` will derive (`~` p).
- For instance, assuming that `(A ==> B)` and `(~ B)` are both in the assumption base, we have:

```
> (!mt (A ==> B) (~ B))

Theorem: (not A)
```

Deriving conditionals with `assume`

The standard way of proving a conditional (p `==>` q) is to assume the antecedent p (i.e., to add p to the current assumption base) and proceed to derive the consequent q.

- In Athena, conditional deductions are written: `assume` p D, where D is a proof that derives q from the augmented assumption base.
- A proof of `(A ==> A)`:

```
> assume A
    (!claim A)

Theorem: (if A A)
```

- We refer to p and D as the *hypothesis* (or *assumption*) and the *body* of the conditional deduction, respectively.

Deriving conditionals with `assume`

To evaluate a deduction of the form

$$\texttt{assume} \;\; p \;\; D \tag{1}$$

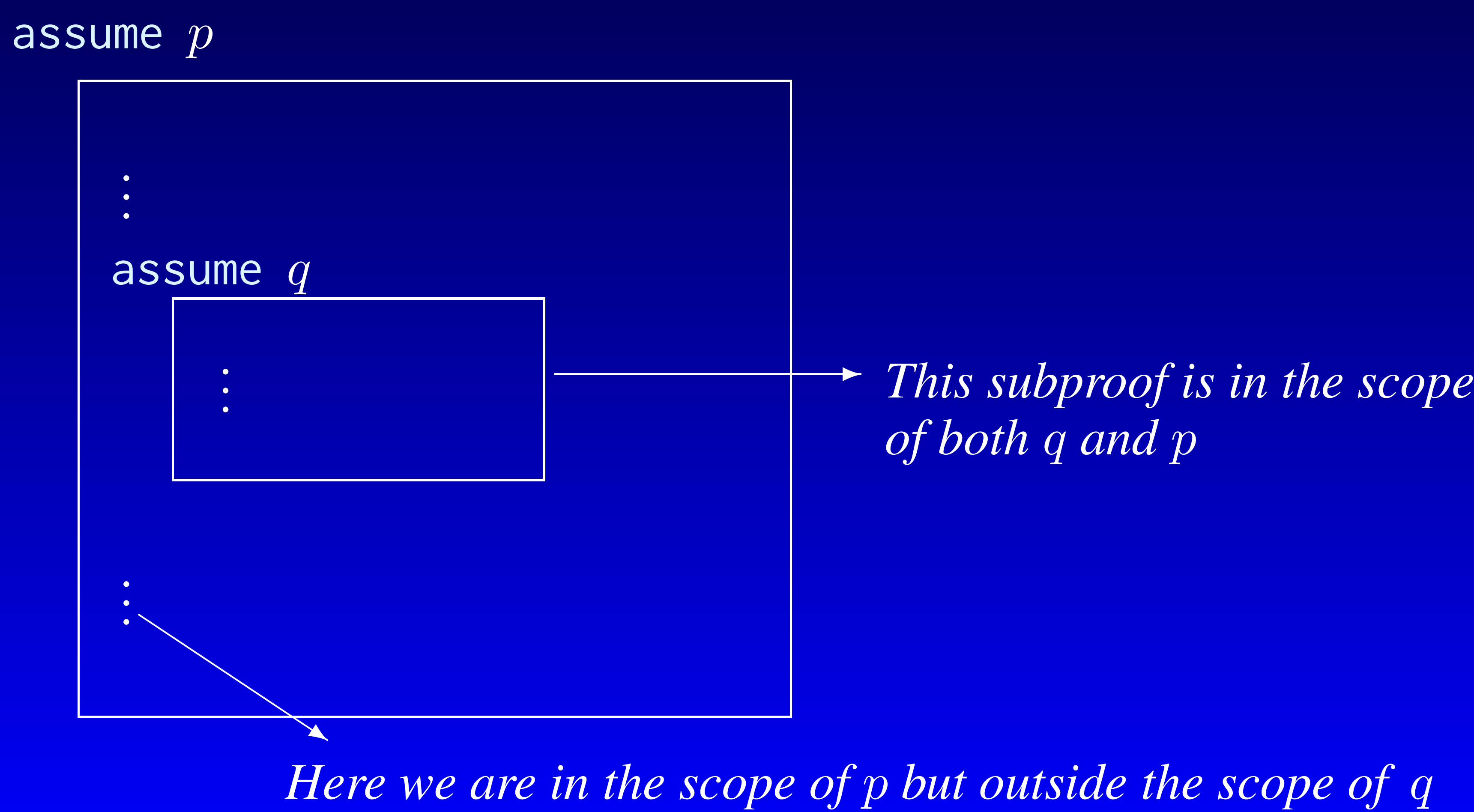
in an assumption base β:

- We add p to β and go on to evaluate the body D in the augmented assumption base $\beta \cup \{p\}$.
- The fact that D is evaluated in $\beta \cup \{p\}$ means that *the assumption p can be freely used anywhere within its scope*, that is, anywhere inside D.
- If and when the evaluation of D in $\beta \cup \{p\}$ produces a conclusion q, we return the conditional $(p \;\texttt{==>}\; q)$ as the result of (1).

Deriving conditionals with `assume`

The body D is said to be a *subproof* (or *subdeduction*) of the conditional proof (1).

Subproofs can be nested inside one another



Deriving conditionals with `assume`

For instance, in the following proof the body `(!claim A)` is in the scope of both the inner assumption `B` and the outer assumption `A`:

```
> assume A
    assume B
      (!claim A)

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

Evaluating this proof in any assumption base β whatsoever will successfully produce the result

`(A ==> B ==> A).`

That is the case for all and only those sentences that are *tautologies*. A *tautology* is precisely a sentence that can be derived from every assumption base. Or alternatively, from the empty assumption base.

Deriving conditionals with `assume`

Consider next this implication:

```
((A ==> B ==> C) ==> (B ==> A ==> C)).
```

The following proof derives it:

```
> assume hyp := (A ==> B ==> C)
    assume B
      assume A
        let {B=>C := (!mp hyp A)}
          conclude C
            (!mp B=>C B)

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

Using disjunctions: Reasoning by cases

Suppose we are trying to derive some goal p.

- If the assumption base contains a disjunction $(p_1 \ | \ p_2)$, we can often put that disjunction to use as follows:
  - We know that p_1 holds or p_2 holds.
  - If we can show that p_1 implies the goal p *and* that p_2 also implies p, then we can conclude p.
  - For, if p_1 holds, then p follows from the implication $(p_1$ ==> $p)$, p_1, and modus ponens; while, if p_2 holds, then p follows from $(p_2$ ==> $p)$, p_2, and modus ponens.
- This type of reasoning (called “case analysis,” or “reasoning by cases”) is pervasive, both in mathematics and in real life.

Using disjunctions: Reasoning by cases

In Athena, reasoning by cases is carried out by the ternary method `cases`.

- The first argument of this method must be a disjunction, say $(p_1 \mid p_2)$; while the second and third arguments must be conditionals of the form $(p_1$ `==>` $p)$ and $(p_2$ `==>` $p)$
- If all three sentences are in the assumption base, then the conclusion p is produced as the result, e.g.:

```
assert (C1 | C2), (C1 ==> B), (C2 ==> B)

> conclude B
    (!cases (C1 | C2)
            (C1 ==> B)
            (C2 ==> B))

Theorem: B
```

Using disjunctions: Reasoning by cases

- We know that for any given p, either p or `(~` p`)` holds; this is the law of the *excluded middle*.
- Therefore, if we can show that a goal q follows both from p and from `(~` p`)`, we should be able to conclude q.
- This is done with the binary method `two-cases`, which takes two premises of the form `(`p `==>` q`)` and `(~` p `==>` q`)` and derives q.
- For example:

```
assert (A ==> B), (~ A ==> B)

> (!two-cases
    (A ==> B)
    (~ A ==> B))

Theorem: B
```

Deriving disjunctions

To derive a disjunction `(`p `|` q`)` we can derive the left component, p, or the right component q.

- If we have p in the assumption base, then `(`p `|` q`)` can be derived by applying the binary method `left-either` to p and q:

  `(!left-either` p q`)`.

- Or if q is in the assumption base, then

  `(!right-either` p q`)`

```
> (!left-either (A ==> A) B)
Theorem: (or (if A A)
             B)

> (!right-either B (A ==> A))
Theorem: (or B
             (if A A))
```

Deriving disjunctions

- Athena offers a third, more versatile mechanism for disjunction introduction, the binary method `either`.
- If either *p or q* is in the assumption base, then `(!either` p q`)` derives the disjunction `(`p `|` q`)`.
- Otherwise, if neither argument is in the assumption base, `either` fails.

We can also use the logical equivalence between a disjunction `(`p `|` q`)` and the conditional

`(~` p `==>` q`)`

to derive a disjunction using the prior techniques to derive conditional sentences.

Using negations

The only primitive method for negation elimination is dn, which stands for "double negation."

It is a unary method whose argument must be of the form (~ ~ p).

If that sentence is in the assumption base, then the call

(!dn (~ ~ p))

will produce the conclusion p.

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

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

There are two other primitive methods that require some of its arguments to be negations: `mt` and `absurd`.

Deriving negations: Proof by contradiction

Proof by contradiction is one of the most useful and common forms of deductive reasoning.

The basic idea is to establish a negation `(~` p`)` by

- assuming p
- showing that this assumption (perhaps in tandem with other working assumptions) leads to an absurdity, namely, to `false`.
- which entitles us to reject the hypothesis p and conclude the desired `(~` p`)`.

Deriving negations: Proof by contradiction

The binary method `by-contradiction` is one way to perform this type of reasoning in Athena.

- The first argument to `by-contradiction` is simply the sentence we are trying to establish, typically a negation `(~` p`)`.
- The second argument must be the conditional `(`p `==> false)`, essentially stating that the hypothesis p leads to an absurdity.
- If that conditional is in the assumption base, then the desired conclusion `(~` p`)` will be produced.

Deriving negations: Proof by contradiction

Suppose the assumption base contains the premises `(A ==> B & C)` and `(~ B)`, and we want to derive `(~ A)`.

We can reason by contradiction as follows:

- Suppose `A` holds.
- Then, by the first premise and modus ponens, we would have `(B & C)`, and hence, by conjunction elimination, `B`.
- But this contradicts the second premise, `(~ B)`, which allows us to reject the hypothesis `A`, inferring `(~ A)`.

Deriving negations: Proof by contradiction

In Athena, this proof can be written as follows:

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

> (!by-contradiction (~ A)
    assume A
      let {p1 := conclude (B & C)
                   (!mp premise-1 A);
           _ := conclude B
                  (!left-and p1)}
        (!absurd B premise-2))

Theorem: (not A)
```

Deriving negations: Proof by contradiction

As another example, here is a proof that derives `(~ B)` from `(~ (A ==> B))`:

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

> (!by-contradiction (~ B)
    assume B
      let {A==>B := assume A
                        (!claim B)}
        (!absurd A==>B premise))

Theorem: (not B)
```

Deriving negations: Proof by contradiction

The most direct way to derive `false` is to apply the binary method `absurd` to two contradictory sentences of the form q and `(~` q`)` in the assumption base.

```
> (!absurd A (~ A))

Theorem: false
```

Therefore, a proof of `(~` p`)` by contradiction often has the following logical structure:

```
(!by-contradiction (~ p)
  assume p
    let {p1 := conclude q
                  D1;
         p2 := conclude (~ q)
                  D2}
      (!absurd p1 p2))
```

Proof by contradiction

If the sentence p we want to establish by contradiction is not a negation, recall every sentence p is equivalent to the double negation (~ ~ p). We can simply infer (~ ~ p) by assuming (~ p) and deriving a contradiction. After that, we can eliminate the double negation sign with dn.

For example, suppose from A and (~ (A & ~ B)), we want to derive B:

```
assert premise-1 := (~ (A & ~ B))
assert premise-2 := A

> let {--B := (!by-contradiction (~ ~ B)
                 assume (~ B)
                  (!absurd (!both A (~ B)) premise-1))}
    (!dn --B)

Theorem: B
```

Proof by contradiction

`by-contradiction` offers a shortcut:

- When the conclusion p to be established is not in the explicit form of a negation, it suffices to establish the conditional `(~` p `==> false)`.
- We can then apply `by-contradiction` directly to p and this conditional:

```
(!by-contradiction p
   (~ p ==> false))
```

and the desired p will be obtained.

```
> (!by-contradiction B
     assume (~ B)
       (!absurd (!both A (~ B)) premise-1))

Theorem: B
```

Proof by contradiction

There are two other auxiliary methods for reasoning by contradiction:

1. The unary method `from-false` derives any given sentence, provided that the assumption base contains `false`. That is, `(!from-false` p`)` will produce the theorem p whenever the assumption base contains `false`. This captures the principle that "everything follows from `false`."
2. The ternary method `from-complements` derives any given sentence p provided that the assumption base contains two complementary sentences q and $\overline{q}$. Specifically,

   `(!from-complements` p q $\overline{q}$`)`

   will derive p provided that both q and $\overline{q}$ are in the assumption base. Such an application can be read as: "Infer p from the complements q and $\overline{q}$."

Using biconditionals

There are two elimination methods for biconditionals, `left-iff` and `right-iff`.

For any given biconditional (p `<==>` q) in the assumption base, the method call

`(!left-iff (p <==> q))`

will produce the conclusion (p `==>` q), while

`(!right-iff (p <==> q))`

will yield (q `==>` p), e.g.:

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

> (!left-iff bc)
Theorem: (if A B)

> (!right-iff bc)
Theorem: (if B A)
```

Deriving biconditionals

The introduction method for biconditionals is `equiv`.

Given two conditionals (p `==>` q) and (q `==>` p) in the assumption base, the call

`(!equiv (`p `==>` q`) (`q `==>` p`))`

will derive the biconditional (p `<==>` q):

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

> (!equiv (A ==> B) (B ==> A))

Theorem: (iff A B)
```

Putting it all together

Suppose we are given the following two premises:

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

assert premise-2 := (C <==> ~ E)
```

and our task is to write a proof D that derives the conditional

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

from the two premises.

Putting it all together

```
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))
```