CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Chapter 1

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

Overview of fundamental proof methods

Goal: to develop proof skills applicable to

- input-output correctness, termination, memory safety
- inductive properties of recursive data structures
- abstract algorithms and data structures

Some of the most basic and useful proof methods:

- Equation chaining
- Induction
- Case analysis
- Proof by contradiction
- Abstraction and specialization

Equation chaining

Consider proving the algebraic identity

$$(a^{-1})^{-1} = a \tag{1}$$

where we are given the identities:

$$\begin{array}{lrcl} \text{Right-Identity:} & x \cdot I & = & x \\ \text{Left-Identity:} & I \cdot x & = & x \\ \text{Right-Inverse:} & x \cdot x^{-1} & = & I \end{array}$$

A proof

$$
\begin{aligned}
(a^{-1})^{-1} &= I \cdot (a^{-1})^{-1} \\
&= (a \cdot a^{-1}) \cdot (a^{-1})^{-1} \\
&= a \cdot (a^{-1} \cdot (a^{-1})^{-1}) \\
&= a \cdot I \\
&= a
\end{aligned}
$$

A proof with justifications

$$
\begin{aligned}
(a^{-1})^{-1} &= I \cdot (a^{-1})^{-1} && \text{[Left-Identity]} \\
&= (a \cdot a^{-1}) \cdot (a^{-1})^{-1} && \text{[Right-Inverse]} \\
&= a \cdot (a^{-1} \cdot (a^{-1})^{-1}) && \text{[?]} \\
&= a \cdot I && \text{[Right-Inverse]} \\
&= a && \text{[Right-Identity]}
\end{aligned}
$$

The missing identity

$$\text{Associativity:} \quad (x \cdot y) \cdot z \quad = \quad x \cdot (y \cdot z)$$

So we have

$$\begin{array}{rcll}
(a^{-1})^{-1} & = & I \cdot (a^{-1})^{-1} & \text{[Left-Identity]} \\
& = & (a \cdot a^{-1}) \cdot (a^{-1})^{-1} & \text{[Right-Inverse]} \\
& = & a \cdot (a^{-1} \cdot (a^{-1})^{-1}) & \text{[Associativity]} \\
& = & a \cdot I & \text{[Right-Inverse]} \\
& = & a & \text{[Right-Identity]}
\end{array}$$

Specializing an identity

Look more closely at the first link of the chain:

$$(a^{-1})^{-1} \quad = \quad I \cdot (a^{-1})^{-1} \quad \text{[Left-Identity]}$$

Recall

$$\text{Left-Identity:} \quad I \cdot x \quad = \quad x$$

The variable x of Left-Identity is specialized to the particular term $(a^{-1})^{-1}$.

Universal quantification

To clarify the role of the variables in the given identities, restate them showing explicitly that x, y and z are *universally quantified* (over D):

Right-Identity: $(\forall x{:}D.\ x \cdot I = x)$

Left-Identity: $(\forall x{:}D.\ I \cdot x = x)$

Right-Inverse: $(\forall x{:}D.\ x \cdot x^{-1} = I)$

Associativity: $(\forall x{:}D\ y{:}D\ z{:}D.\ x \cdot (y \cdot z) = (x \cdot y) \cdot z)$

An arbitrarily chosen element

We can clarify the role of a by beginning the proof with "Let a be an arbitrarily chosen element (of the domain)."

The full proof

To prove $(\forall a{:}D.\ (a^{-1})^{-1} = a)$, let a be an arbitrarily chosen element of domain D. Then

$$
\begin{aligned}
(a^{-1})^{-1} &= I \cdot (a^{-1})^{-1} && \text{[Left-Identity]} \\
&= (a \cdot a^{-1}) \cdot (a^{-1})^{-1} && \text{[Right-Inverse]} \\
&= a \cdot (a^{-1} \cdot (a^{-1})^{-1}) && \text{[Associativity]} \\
&= a \cdot I && \text{[Right-Inverse]} \\
&= a && \text{[Right-Identity].}
\end{aligned}
$$

Subterm replacement

- In the second link of the chain,

$$\begin{aligned} (a^{-1})^{-1} &= I \cdot (a^{-1})^{-1} && \text{[Left-Identity]} \\ &= (a \cdot a^{-1}) \cdot (a^{-1})^{-1} && \text{[Right-Inverse]} \end{aligned}$$

  there is another aspect of the use of the identity Right-Inverse: the instance $a \cdot a^{-1} = I$ of $(\forall x{:}D.\ x \cdot x^{-1} = I)$ is used to replace only a *subterm* of $(a \cdot a^{-1}) \cdot (a^{-1})^{-1}$, not the whole term.

- What justification is there for such a subterm replacement?

Athena's chain method

- Identification of the subterm being replaced and the variable substitution involved: Athena provides efficient subterm search and term matching capabilities.
- Athena uses these capabilities in its `chain` method.

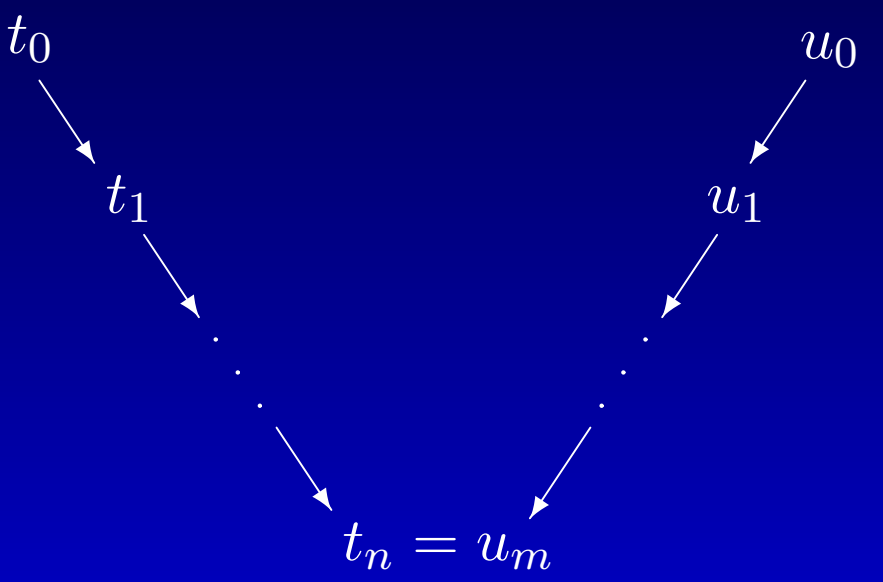
```
conclude (forall a:D . (inv (inv x)) = x)
  pick-any a:D
    (!chain
       [(inv (inv a))
         = (I * (inv (inv a)))              [Left-Identity]
         = ((a * (inv a)) * (inv (inv a))) [Right-Inverse]
         = (a * ((inv a) * (inv (inv a)))) [Associativity]
         = (a * I)                          [Right-Inverse]
         = a                                [Right-Identity]
       ])
```

Creating equation chains

- Starting with the equation $t = u$ to be proved, and letting $t_0 = t$, how do we find the right sequence of terms $t_1, t_2, \ldots, t_n = u$ to link together?
- Not always easy to see how to get started.

Reduction to a common term

- One stategy: working from both sides, t and u, try to *reduce them to a common term*.
- Diagrammatically:



Induction

- Not every valid equation can be proved just by chaining together instances of other valid equations.
- Proving an equation may require some form of *mathematical induction*.
  - Ordinary induction
  - Strong induction

Ordinary induction

With natural numbers, to prove $(\forall n \,.\, P(n))$, it suffices to prove

Basis Case: $P(0)$

Induction Step: $(\forall n \,.\, P(n) \Longrightarrow P(n+1))$

In the Induction Step, $P(n)$ is called the *Induction Hypothesis*.

Strong induction

With natural numbers, to prove $(\forall n \,.\, P(n))$, it suffices to prove

$$(\forall n \,.\, (\forall k \,.\, k < n \Longrightarrow P(k)) \Longrightarrow P(n)).$$

Here, $(\forall k \,.\, k < n \Longrightarrow P(k))$ is called the *Strong Induction Hypothesis*.

Ordinary induction for lists

To prove $(\forall\, L\ .\ P(L))$, where L ranges over lists, it suffices to prove:

Basis Case: $P(\text{nil})$

Induction Step: $(\forall\, L\ .\ P(L) \Longrightarrow (\forall\, x\ .\ P(x :: L)))$

In the Induction Step, the assumption $P(L)$ is called the *Induction Hypothesis*.

A little puzzle

Consider the following problem involving three persons, Jack, Anne, and George.

> Jack is looking at Anne, and Anne is looking at George. Jack is married, George is not. Is some married person looking at an unmarried person?

Is your answer:

A) yes

B) no

C) cannot be determined from the given information

Case analysis

Jack is looking at Anne, and Anne is looking at George. Jack is married, George is not. Is some married person looking at an unmarried person?

The correct answer is "A), yes, some married person is looking at an unmarried person".

Proof: There are two cases:

- If Anne is married, then since she is looking at George, and George is not married, a married person is looking at an unmarried person.
- If Anne is unmarried, then since Jack, a married person, is looking at Anne, we know in this case too that a married person is looking at an unmarried person.

This form of reasoning is called *case analysis*.

Case analysis in Athena (I)

Setup:

```
domain Person
declare married: [Person] -> Boolean
declare is-looking-at: [Person Person] -> Boolean
declare Jack, Anne, George: Person

define [p1 p2] := [?p1:Person ?p2:Person]

assert [(Jack looking-at Anne)
        (Anne looking-at George)
        (married Jack)
        (~ married George)]

conclude goal := (exists p1 p2 . married p1 &
                                 p1 looking-at p2 &
                                 ~ married p2)
```

Case analysis in Athena (II)

```
conclude goal := (exists p1 p2 . married p1 &
                                 p1 looking-at p2 &
                                 ~ married p2)

conclude goal {
    (!two-cases
      assume case1 := (married Anne)
        (!chain-> [case1
                ==> (married Anne &
                     Anne looking-at George
                     & ~ married George)      [augment]
                ==> goal                      [existence]])
      assume case2 := (~ married Anne)
        (!chain-> [case2
                ==> (married Jack &
                    Jack looking-at Anne &
                    ~ married Anne)           [augment]
                ==> goal                      [existence]]))
```

Proof by contradiction

- To prove $(\sim p)$, assume p and show that leads to a contradiction.
- Example:
  Consider a relation $<$ on some domain D, with the properties

$$\textit{Irreflexivity}: \quad \forall\, x{:}D \;.\; \sim x < x.$$

$$\textit{Transitivity}: \quad \forall\, x{:}D \;\; y{:}D \;\; z{:}D \;.\; x < y \wedge y < z \Rightarrow x < z.$$

  Then the $<$ relation also has the property

$$\textit{Asymmetry}: \; \forall\, x{:}D \;\; y{:}D \;.\; x < y \Rightarrow \sim y < x.$$

  PROOF: By contradiction. Choose any a and b in D such that $a < b$ and assume, contrary to the stated conclusion, that $b < a$. Then from $a < b$, $b < a$, and transitivity, we infer $a < a$. But, by irreflexivity, we have $\sim a < a$, and hence we have a contradiction.

Proof by contradiction in Athena

```
domain D
declare <: [D D] -> Boolean
define [x y z] := [?x:D ?y:D ?z:D]

assert* irreflexivity := (~ x < x)
assert* transitivity := (x < y & y < z ==> x < z)

conclude asymmetry := (forall x y . x < y ==> ~ y < x)
  pick-any a:D b:D
    assume (a < b)
      (!by-contradiction (~ b < a)
        assume (b < a)
          let {less := (!chain-> [(a < b & b < a)
                                  ==> (a < a)          [transitivity]]);
               not-less := (!chain-> [true
                                      ==> (~ a < a)    [irreflexivity]])}
            (!absurd less not-less))
```

Abstraction/specialization

Another strategy to limit the number of difficult proofs one has to write: *work at an abstract level*:

- Prove theorems in the most general setting
- Then specialize them to concrete instances as needed

Example of abstraction

Recall the identities,

$$\begin{array}{ll} \text{Right-Identity:} & (\forall x{:}D.\ x \cdot I = x) \\ \text{Left-Identity:} & (\forall x{:}D.\ I \cdot x = x) \\ \text{Right-Inverse:} & (\forall x{:}D.\ x \cdot x^{-1} = I) \\ \text{Associativity:} & (\forall x{:}D\ y{:}D\ z{:}D.\ x \cdot (y \cdot z) = (x \cdot y) \cdot z) \end{array}$$

- Regard these as *axioms of an abstract theory* T. Using them we are able to prove the identity $(a^{-1})^{-1} = a$ as a theorem of T.
- Then for *any* concrete domain D in which the identities hold, we can specialize that proof to prove the corresponding concrete specialization of $(a^{-1})^{-1} = a$.

Example of specialization (I)

- The binary operator $\cdot$ might even be specialized to an addition operator rather than a multiplication operator in a concrete domain.
- For example, let D be the integer domain Z, and specialize $\cdot$ to integer addition: $+$; I to its neutral element: 0; and the inverse operator to integer negation: (unary) $-$.
- Then the given identities become

Right-Identity: $(\forall x{:}Z \;.\; x + 0 = x)$

Left-Identity: $(\forall x{:}Z \;.\; 0 + x = x)$

Right-Inverse: $(\forall x{:}Z \;.\; x + (-x) = 0)$

Associativity: $(\forall x{:}Z\ y{:}Z\ z{:}Z \;.\; x + (y + z) = (x + y) + z)$

Example of specialization (II)

The proof of $(-(-a)) = a$ becomes:

Let a be an arbitrarily chosen element of Z. Then

$$\begin{aligned}
(-(-a)) &= 0 + (-(-a)) && \text{[Left-Identity]} \\
&= (a + (-a)) + (-(-a)) && \text{[Right-Inverse]} \\
&= a + ((-a) + (-(-a))) && \text{[Associativity]} \\
&= a + 0 && \text{[Right-Inverse]} \\
&= a && \text{[Right-Identity].}
\end{aligned}$$

But we don't have to construct this proof from scratch!

Advantages of abstraction

- In mathematics, the advantages have long been known.
- More recently, in computer science, some textbooks and journal articles develop theories at an abstract level and then specialize them in different ways.
- Similar principles have also been applied in *computer programming*, with *generic* (or *abstract*) algorithms and data structures
- For constructing proofs in computer science, with an approach like Athena's—in which proofs are programs, we can realize the same advantages.

Proof methods in combination

- We've looked at
  - Equation chaining
  - Induction
  - Case analysis
  - Proof by contradiction
  - Abstraction and specialization
- In simple cases, proofs can be done with only one of these methods, but most proofs involve combinations of them.
- Developing skills in creating such proofs is the goal of *Fundamental Proof Methods in Computer Science*.