Deductive Methods in Computer Science

Excerpts from a lecture by
Björn Lisper,
Mälardalens Högskola

Overview

1. Scientific methods, general
2. Theoretical vs. empirical methods in Computer Science
3. Theoretical models in Computer Science
4. Theoretical problems in Computer Science
5. Deductive methods in Computer Science
6. Recursive definitions and proofs by induction
7. Inference systems and their applications

Scientific Methods, General

Theoretical methods:

Create formal models (mathematics, logic)

Define concepts within these

Prove properties of the concepts

Abstraction, hide details to make the whole more understandable (and to make it possible to prove properties of it)

Proofs of properties by deductive methods

Empirical methods:

Perform experiments

See how it turned out

Draw conclusions

Simulation:

Start with a formal model at some "easy-to-understand" level

Make "artificial experiments" in your computer

Collect statistics and draw conclusions

In physics:

Make hypotheses about the surrounding world (theory), observe it (experiment)

Relate the result of experiment to theory

Adjust the theory if it doesn’t predict the reality well enough

Theory is used to predict the future (e.g., if a bridge will hold for a certain load, or an asteroid fall down on our heads)

Common pattern in Computer Science:

The system is constructed to behave according to some theoretical model

Deviations are seen as construction errors rather than deficiencies in the theory (hardware error, bug in OS, . . .)

In both cases: the theory helps us understand and predict, but in different ways!

Theoretical vs. Empirical Methods in Computer Science

Computer Science really has a “spectrum”, from “extreme constructivism” to a use of theory close the one in physics:

“Extreme constructivism”: (ideal) programming language design:

- Formal semantics for the language, pure construction of model defining the mathematical meaning of each program
- Abstraction of details to make the meaning of the language simpler (for instance, assume that data structures can grow arbitrarily big)
- Implement the language according to the semantics

One can prove formally within the model that a program is correct – valuable!

But the model does not cover all kinds of errors. E.g., hardware errors, or stack overflow (or an asteroid falling down on the computer)

Extreme “physics” approach: performance modelling of complex computer- and communication systems

- Extremely hard to make analytical calculations
- Simplified performance models, tested against experiments (e.g., long suites of benchmarks)
- Discrepancy leads to a modified theory, as in physics
- Often simulation (desire to evaluate systems before building them)

In-between: algorithm analysis

- Build on some form of formal model for how the algorithm executed (metalanguage with formal semantics), and some performance model (how long does a step in the algorithm take, how much memory is needed to store an entity)
- Performance model often of type “one arithmetic operation $=$ one time unit”
- Given that the performance model is correct, one proves mathematically that the algorithm needs certain resources (time, memory) to be carried out
- But the performance model is often very approximate
- Sometimes possible to refine the performance model, but this can make it impossible to calculate the resource needs of the algorithm

Theoretical Models in Computer Science

Discrete mathematics: basic set theory, relations, functions, graphs, algebra, combinatorics, category theory, etc.

The *science* logic: different logical systems, how to make “proofs about proofs”

Theory for complete partial orders (formal semantics)

Topology (mathematics with notions of distance and convergence)

Probability theory, statistics

(Traditional analysis)

Theoretical *Problems* in Computer Science

What do we want to prove theoretically within Computer Science?

For instance properties of programs, systems, algorithms, and problems

Some examples:

"FFT uses $O(n \log n)$ operations"

"With 99% confidence the program p runs faster than 1.3 ms on machine m"

"The program p terminates for all indata"

"If the method M says that a program terminates then this is true"

"There is no method that can decide, for any program, whether it terminates or not"

"For each CREW PRAM-algorithm there is an EREW PRAM-algorithm that can simulate it with a certain slowdown"

$P = NP$ (or $P \neq NP$)

"The two semantics S_1 och S_2 agree for each program in the programming language P"

Deductive Methods in Computer Science

1. “Ordinary” mathematical proofs:

- Often *finite entities*: defined *recursively*, properties proved with *induction*
- But also reasoning about limits (“go to the limit”), when infinite behaviours are modelled
- Encodings and translations – common in Complexity Theory
- Sometimes also more conventional mathematical techniques

2. Direct modelling with logical inference systems:

- Common in semantics of programming languages (operational semantics)
- Proof methods from logic (proofs about proofs), again induction!

Let's see some examples. . .

Example 2: Algorithm Analysis

Purpose: to find the *cost* of executing an algorithm (that solves a given problem)

(Archetypal problem: to sort a sequence of numbers)

Cost is typically *running time*, but can also be memory requirements, power consumption, etc.

To calculate the cost requires:

- a *machine model*
- a *notation*, i.e., “programming language” for the machine

Typically only interested in the *asymptotic complexity* of the algorithm

“How fast does the execution time grow with the size of the input?”

An example: insertion sort

```
1 for j = 2 to length(A) do
2   key = A[j]
3   i = j - 1
4   while i > 0 and A[i] > key do
5     A[i+1] = a[i]
6     i = i - 1
7   A[i+1] = key
```

We want to find the execution time as a function of input size (`length(A)`)

Let us informally analyze insertion sort!

Assume execution time of a program is sum of the time of all executions of individual statements,

and that the execution time of an individual statement is constant

(How reasonable is this assumption, really?)

Thus, we can, for each statement take its execution time times the number of times it is executed, and then sum over all statements

Say statements 1 - 7 have execution times $c_1, \ldots, c_7$

Each statement s is executed t_s times

Then total execution time is

$$\sum_{s=1}^{7} t_s \cdot c_s$$

Let's calculate the different t_s on wyteboard and see what we get. . .

Results of analysis:

Best-case execution time (with $n =$ length of A):

$c_1 n + (c_2 + c_3 + c_4 + c_7)(n - 1)$

order $\Theta(n)$ (what do we mean by this?)

Worst-case execution time:

$(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2})n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - (c_2 + c_3 + c_4 + c_7)$

order $\Theta(n^2)$

Average-case execution time:

order $\Theta(n^2)$

What is $\Theta(n)$ (and $\Theta(n^2)$)?

$\Theta(f)$ means *the set of functions that grow as fast as f when the argument becomes large enough*

Mathematically,

$$\Theta(f) = \{\, g \mid \exists c_1, c_2, n_0 > 0. \forall n \geq n_0. 0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n) \,\}$$

"h is order $\Theta(f)$" means $h \in \Theta(f)$

$\theta(f)$ is similar to the more commonly used $O(f)$ (ordo)

What kind of mathematics did we use?

Proving $h \in \Theta(f)$ is done by ordinary mathematical methods (reasoning about inequalities, deciding the existence of certain entities, . . .)

Facts about sums, algebraic manipulations

Probability theory to get the average-case execution time

In short, traditional mathematics

Note, though, that certain details are swept under the carpet!

In particular, implicit assumptions about semantics of loops etc. (how do we know the body of `for j = 2 to n` is executed exactly $\mathtt{n} - 1$ times?)

Example 3: Complexity Theory

Deals with *problems*, or *classes* of problems, rather than single algorithms

Tries to find limits for how costly a certain problem (or class of problems) is on a certain machine model

An example of a problem is sorting:

- $O(n \log\ n)$ algorithms are known (for sequential machine model)
- Not proved whether this is the ultimate lower limit!

A famous class of problems (a *complexity class*):

NP, the set of all problems that can be solved in polynomial time ($O(n^k)$ for some k) by a non-deterministic Turing machine ($\approx$ set of problems solvable by "brute force parallel search" in polynomial time)

A “hardest” problem in NP is known, 3-SAT: if 3-SAT always can be solved in polynomial time then *each* problem in NP can be solved in polynomial time

Proof by encoding: that each problem in NP can be translated into 3-SAT such that a solution of the translated problem solves the original problem (in polynomial time relative to the time to solve the translated problem in 3-SAT)

(Or the reverse: if there is *any* problem in NP that *cannot* be solved in polynomial time, then 3-SAT cannot either!)

3-SAT is NP*-complete*

Proof that *another* problem Q is NP-complete:

1. Show $Q \in NP$

2. Show that if one can solve Q in (sequential) polynomial time then 3-SAT can be solved in polynomial time (via translation of 3-SAT into Q)

Complexity theory uses *encodings* a lot

Another famous complexity class:

P, the set of all problems that can be solved in polynomial time by a *deterministic* Turing machine (cf. NP)

The class of problems that can be solved *sequentially* in polynomial time (like, for instance, sorting)

Open question: is $P = NP$?

Generally assumed that $P \neq NP$, but has not been proved!

If indeed $P = NP$, then the concept of NP-completeness becomes quite meaningless

Example 4: Complete Partial Orders

A mathematical model that is good for describing the meaning of recursive definitions

It can be thought of as describing *information contents*

The idea is that more and more information about the result of a computation becomes available as the computation proceeds

We may want to describe infinite computations (e.g., a server computing an unbounded number of results)

Therefore we need to “go to the limit” in the model

A *complete partial order* (cpo) is a structure $\langle D, \sqsubseteq \rangle$, where D is a set and $\sqsubseteq$ is a binary relation on D such that:

- it is a partial order; and
- there is a *bottom element* $\bot \in D$ such that $\bot \sqsubseteq d$ for all $d \in D$; and
- for each infinitely non-decreasing chain $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_i \sqsubseteq \cdots$, there is a *least upper bound* $\bigsqcup_{i=0}^{\infty} d_i$.

Least upper bound means:

- $d_j \sqsubseteq \bigsqcup_{i=0}^{\infty} d_i$ for all j; and
- if $e \sqsubset \bigsqcup_{i=0}^{\infty} d_i$ for some $e \in D$ then there is a k such that, for all $j > k$, holds that $e \sqsubset d_j$.

Definition: a function $f\colon D \to D$ is *continuous* if $f(\bigsqcup_{i=0}^{\infty} d_i) = \bigsqcup_{i=0}^{\infty} f(d_i)$ for all least upper bounds $\bigsqcup_{i=0}^{\infty} d_i$

An interesting result in the theory of cpo's is *Kleene's Fixed-Point Theorem*:

Let f be continuous. Define $\mathit{fix}(f) = \bigsqcup_{i=0}^{\infty} f^i(\bot)$. Then $\mathit{fix}(f)$ is the least solution w.r.t. $\sqsubseteq$ of the equation $d = f(d)$.

We say it is the *least fixed point* of f

Many recursive definitions are of the form $d = f(d)$

For instance, a recursively defined function in a simple functional language:

```
fac(n) = if n == 0 then 0 else n * fac(n-1)
```

This can be seen as an equation $\texttt{fac} = f(\texttt{fac})$

With suitably chosen cpo, Kleene's fixed-point theorem gives a well-defined mathematical meaning to `fac`

Cpo's are central in *denotational semantics* of programming languages

Recursive Definitions and Proofs by Induction

Induction over natural numbers

Show that the property P is true for all natural numbers (whole numbers ≥ 0)

1. Show that P holds for 0
2. Show, for all natural numbers n, that if P holds for n then P holds also for $n+1$
3. Conclude that P holds for all n

Formulated in formal logic:

$$[P(0) \wedge \forall n.P(n) \implies P(n+1)] \implies \forall n.P(n)$$

Example: show that for all natural numbers n holds that

$$\sum_{i=0}^{n}(2i-1) = n^2 - 1$$

Why does induction over the natural numbers work?

The set of natural numbers $\mathbf{N}$ is an *inductively defined set*

(A variation of) Peano's axiom:

- $0 \in \mathbf{N}$
- $\forall x.x \in \mathbf{N} \implies s(x) \in \mathbf{N}$
- $\forall x.0 \neq s(x)$
- $\forall x, y.x \neq y \implies s(x) \neq s(y)$

$s(x)$ "successor" to x, or $x + 1$

$$\begin{array}{ccccccccc} 0 & \rightarrow & s(0) & \rightarrow & s(s(0)) & \rightarrow & s(s(s(0))) & \rightarrow & \cdots \\ 0 & & 1 & & 2 & & 3 & & \cdots \end{array}$$

Note how proofs by induction over the natural numbers follow the structure of their definition

$$\begin{array}{ccccccccc} 0 & \rightarrow & s(0) & \rightarrow & s(s(0)) & \rightarrow & s(s(s(0))) & \rightarrow & \cdots \\ 0 & & 1 & & 2 & & 3 & & \cdots \\ P(0) & \Longrightarrow & P(1) & \Longrightarrow & P(2) & \Longrightarrow & P(3) & & \cdots \end{array}$$

Also note that the definition of $\mathbf{N}$ is given a well-defined meaning by Kleene's fixed-point theorem:

$$\emptyset \subseteq \{0\} \subseteq \{0,1\} \subseteq \{0,1,2\} \subseteq \cdots$$

Inductively defined sets are typically sets of *infinitely* many *finite* objects

Entities in Computer Science are often finite (data structures, programs, . . .)

Example: mathematical definition of the set of (finite) lists of integers, $List$

- $NIL \in List$
- $z \in \mathbf{Z} \wedge l \in List \implies z : l \in List$
- $\forall z, l.(z : l \neq NIL)$
- $\forall z, z', l, l'.(z : l = z' : l' \implies z = z' \wedge l = l')$

$List$ is a kind of *abstract data type* – the internal representation is hidden

Need *not* be represented as linked structures in memory (but could be)

Typical elements in $List$: $NIL,\ 3:(4:NIL)$

Note similarity with the set of natural numbers

$List$ is the set of finite (but arbitrarily long) lists of numbers

We can define mathematical functions over lists. An example:

$$\begin{aligned} length(NIL) &= 0 \\ length(z : l) &= 1 + length(l), \quad \text{for all } z \in \mathbf{Z} \text{ och } l \in List \end{aligned}$$

Defines $length$ as a function $List \rightarrow \mathbf{N}$

Recursive definition: $length$ itself is used in the definition! (Seemingly circular definition, but note that $length$ is not applied on the same argument in the right-hand side)

Exercise: show that $length$ really is a well-defined partial function! (That is, that each function value is uniquely determined by the definition.)

Note the similarity with function definitions in some functional languages

Exercise: show $\forall l.length(l) \geq 0$

How to do this?

Each inductively defined set has an *induction principle* that follows the inductive definition of the set. Induction is performed on the "pieces" of an entity built up from smaller entitites (e.g., a list built of elements put in front of shorter lists).

Induction principle for $List$. Show that the property P is true for all lists of integers:

1. Show that P holds for NIL
2. Show, for all lists l and integers z, that if P holds for l, then P holds also for $z : l$
3. Conclude that P holds for all lists of integers

"Mathematical" lists, and functions like $length$, can be seen as *abstract specifications* of what lists are and how functions on them should work

Consider the following piece of C code:

```
#define NIL 0

struct list
{ int contents;
  list *succ;
}

int len(list *l)
{ int length;
  length = 0;
  while(l != NIL)
    {length++; l = l -> l.succ;}
  return(length);
}
```

Interesting things to verify:

- That lists of `list`-structs represent "mathematical" lists in $List$ correctly
- That `len(l)` $= length(l)$ always, when `l` is the representation in C of l

The verification requires that a *formal semantics* is defined for C programs, and that we define exactly what it means that a C entity represents a "mathematical" entity

Logic deals with *formal systems for derivations*, that is, “how to prove things”, and *properties of derivations* (proofs).

Thus, logic is a *metatheory*, which deals with properties of other theories!

Example of a result in logic: “in all logical systems that can express arithmetics on whole numbers, it is possible to formulate statements that can neither be proved nor disproved” (Gödel’s incompleteness theorem)

Logical systems consist of:

Axioms, which are assumed to hold without proof

Inference rules, of the form “given these premises, this conclusion can be inferred”

Inference rules are often written on the form

$$\frac{\text{premise 1} \ \cdots \ \text{premise } n}{\text{conclusion}}$$

Example (modus ponens in propositional logic):

$$\frac{P \quad P \implies Q}{Q}$$

A proof of a statement in a logic is a *finite derivation of the statement as a conclusion, starting from axioms*

The set of provable statements is thus an inductively defined set, starting with the axioms as “base cases”. Hmmm. . .

Even the set of *proofs* is an inductively defined set! Hmmm, hmmm. . .

Induction principle for proofs (in some given logic). Show that the property P is true for all proofs:

1. Show that P holds for all axioms (“least possible proofs”)

2. Show, for each derivation rule and its possible premises, that if P holds for each of the proofs of the premises, then P holds also for the proof of the conclusion

3. Conclude that P holds for all proofs in the proof system

In Computer Science, systems are sometimes modelled directly with logical inference systems!

Example:

- Operational semantics for programming languages
- Type systems

Properties of these can be proved with induction over derivations

We will consider a type system and an operational semantics for a small, typed language

Our language:

Three types: `int` for integers, `bool` for boolean values, `void` for programs

Constants `17`, `0`, `true`, `false`, `...`

Identifiers (program variables) `X`, `Y`, `Z`, `...`

Arithmetical expressions `17`, `X+99*Y`, `...`

Boolean expressions `false`, `Y and Z`, `X > 17`, `...`

Program statements:

- Assignments `X = e`, where `e` is an arithmetical or boolean expression
- Sequencing of statements `c1;c2`, where `c1` and `c2` are statements
- Conditional statements `if b then c1 else c2`, where `b` is a boolean expression, and `c1` and `c2` are statements
- Looping statements `while b do c`, where `b` is a boolean expression, and `c` is a statement

Example: `X = 5; while X > 0 do (X = X - 1)`

A formal, operational semantics for the language:

We give the semantics as a logical inference system (just as the type system)

Derivable facts are statements about programs and program parts, that tell what result(s) their executions can yield

The facts use *states*

A state σ is a mapping from program variables to values (i.e., a description of the "current contents" in memory)

Executing a program, starting in a state σ, will transform the state into some new state σ'

Derivable facts are relations $\langle c, \sigma \rangle \rightarrow \sigma'$

"Starting in state σ, executing program c can yield the state σ'"

Inference rules to derive facts of this form

For arithmetic and boolean expressions we have similar relations as derivable facts:

$\langle \mathtt{a}, \sigma \rangle \to n$ and $\langle \mathtt{b}, \sigma \rangle \to b$

Here, n is a natural number and b a boolean value

This is since evaluating pure expressions in our language does not change the state, only a value is returned

Read $\langle \mathtt{a}, \sigma \rangle \to n$ as “if $\mathtt{a}$ is evaluated when in state σ, then the number n can be returned”

Similarly for boolean expressions

Axioms and inference rules for arithmetic expressions:

Evaluation of numbers:

$$\langle \mathtt{n}, \sigma \rangle \to n$$

Evaluation of numeric program variable:

$$\langle \mathtt{X}, \sigma \rangle \to \sigma(\mathtt{X})$$

Evaluation of sums:

$$\frac{\langle \mathtt{a1}, \sigma \rangle \to n_1 \quad \langle \mathtt{a2}, \sigma \rangle \to n_2}{\langle \mathtt{a1+a2}, \sigma \rangle \to n_1 + n_2}$$

Etc.

Axioms and inference rules for boolean expressions:

Evaluation of boolean constants:

$$\langle \texttt{true}, \sigma \rangle \rightarrow \mathit{true} \quad \langle \texttt{false}, \sigma \rangle \rightarrow \mathit{false}$$

Evaluation of boolean program variable:

$$\langle \texttt{X}, \sigma \rangle \rightarrow \sigma(\texttt{X})$$

Evaluation of inequality:

$$\frac{\langle \texttt{a1}, \sigma \rangle \rightarrow n_1 \quad \langle \texttt{a2}, \sigma \rangle \rightarrow n_2 \quad n_1 > n_2}{\langle \texttt{a1 > a2}, \sigma \rangle \rightarrow \mathit{true}}$$

$$\frac{\langle \texttt{a1}, \sigma \rangle \rightarrow n_1 \quad \langle \texttt{a2}, \sigma \rangle \rightarrow n_2 \quad n_1 \leq n_2}{\langle \texttt{a1 > a2}, \sigma \rangle \rightarrow \mathit{false}}$$

And similarly for the other relational operators . . .

Evaluation of negation:

$$\frac{\langle \texttt{b}, \sigma \rangle \rightarrow \mathit{false}}{\langle \texttt{not b}, \sigma \rangle \rightarrow \mathit{true}} \qquad \frac{\langle \texttt{b}, \sigma \rangle \rightarrow \mathit{true}}{\langle \texttt{not b}, \sigma \rangle \rightarrow \mathit{false}}$$

Exercise: figure out suitable inference rules that give the semantics of boolean connectives `and`, `or`

Axioms and inference rules for execution of programs:

Execution of assignment of numeric variable:

$$\frac{\langle \mathtt{a}, \sigma \rangle \rightarrow n}{\langle \mathtt{X} \ = \ \mathtt{a}, \sigma \rangle \rightarrow \sigma[n/\mathtt{X}]}$$

$\sigma[n/\mathtt{X}]$ is state for which:

- $\sigma[n/\mathtt{X}](\mathtt{X}) = n$
- $\sigma[n/\mathtt{X}](\mathtt{Y}) = \sigma(\mathtt{Y})$, $\mathtt{X} \neq \mathtt{Y}$

Execution of sequenced programs:

$$\frac{\langle \texttt{c1}, \sigma \rangle \rightarrow \sigma'' \quad \langle \texttt{c2}, \sigma'' \rangle \rightarrow \sigma'}{\langle \texttt{c1;c2}, \sigma \rangle \rightarrow \sigma'}$$

Execution of conditionals:

$$\frac{\langle \texttt{b}, \sigma \rangle \rightarrow \mathit{true} \quad \langle \texttt{c1}, \sigma \rangle \rightarrow \sigma'}{\langle \texttt{if b then c1 else c2}, \sigma \rangle \rightarrow \sigma'}$$

$$\frac{\langle \texttt{b}, \sigma \rangle \rightarrow \mathit{false} \quad \langle \texttt{c2}, \sigma \rangle \rightarrow \sigma'}{\langle \texttt{if b then c1 else c2}, \sigma \rangle \rightarrow \sigma'}$$

Execution of while-statement:

$$\frac{\langle \mathtt{b}, \sigma \rangle \rightarrow \mathit{false}}{\langle \mathtt{while\ b\ do\ c}, \sigma \rangle \rightarrow \sigma}$$

$$\frac{\langle \mathtt{b}, \sigma \rangle \rightarrow \mathit{true} \quad \langle \mathtt{c}, \sigma \rangle \rightarrow \sigma'' \quad \langle \mathtt{while\ b\ do\ c}, \sigma'' \rangle \rightarrow \sigma'}{\langle \mathtt{while\ b\ do\ c}, \sigma \rangle \rightarrow \sigma'}$$

This one tends to require some thinking!

Exercise: find state σ' such that

$\langle \mathtt{while\ X > 0\ do\ X = X - 1}, \sigma \rangle \rightarrow \sigma'$

if $\sigma(\mathtt{X}) = 1$!

What can we do with this theory?

We can define a sensible *equivalence* between programs:

$$\texttt{c1} \sim \texttt{c2} \quad \text{iff} \quad \forall \sigma, \sigma'. \langle \texttt{c1}, \sigma \rangle \to \sigma' \iff \langle \texttt{c2}, \sigma \rangle \to \sigma'$$

"`c1`, when started in state σ, can yield state σ' precisely when `c2` can"

We can now use the equivalence to prove *correctness of program transformations* (as used by, e.g., an optimizing compiler)

Exercise: let $\texttt{w} = \texttt{while b do c}$. Show that

$\texttt{w} \sim \texttt{if b then c;w else skip}$

(Correctness of "loop unrolling")

(Extended language with `skip` statement, you figure out how to give its semantics)