E0 205 Mathematical logic and theorem proving

Kamal Lodaya and Deepak D'Souza

February 25, 2021

Logics

- Tools are used to verify programs or systems, and they use logics. Why?
- How did logic get to be used?
- What do logics have to do with programs?

Formal methods

A logic is a formal language equipped with a method for making inferences in this language.

P1: Side AB = Side DE (a “formula” in the formal language)

P2: Side AC = Side DF (another formula)

P3: $\angle BAC = \angle EDF$ (another formula)

C: $\therefore \triangle ABC \equiv \triangle DEF$ (a formula prefixed by “therefore”)

Domain of discourse is geometry.

Formal methods

A logic is a formal language equipped with a method for making inferences in this language.

- P1: Side AB = Side DE (a “formula” in the formal language)
- P2: Side AC = Side DF (another formula)
- P3: $\angle BAC = \angle EDF$ (another formula)
- C: $\therefore \triangle ABC \equiv \triangle DEF$ (a formula prefixed by “therefore”)

Domain of discourse is geometry.

We say “from the finite set of premisses $P1$, $P2$, $P3$, infer that the conclusion C holds”.
In short, “from $P1$, $P2$, $P3$, infer C”.
In short, $P1, P2, P3 \vdash C$.

Formal methods (continued)

- Why formal? Weird symbols like $\angle$ and $\triangle$ are used, we have to be told what they mean.
- Words like “Side” in English (or any other natural language) are used in a technical sense—in the example above, the word has nothing to do with its use in “left side” and “right side” which also happens in English—and we have to be told what this technical meaning is.
- We are also told when from a given set of formulas (premisses), we can infer a formula (the conclusion). That is, the inference method (abbreviated above by $\vdash$) is also told to us.

Exercises (Ramanujam 2021)

Question
Evaluate $x^2 - 1/x - 1$ *at* $x = 1$.

Exercises (Ramanujam 2021)

Question

Evaluate $x^2 - 1/x - 1$ *at* $x = 1$.

Question

Evaluate $\lim_{x \to 1} x^2 - 1/x - 1$.

Exercises (Ramanujam 2021)

Question

Evaluate $x^2 - 1/x - 1$ *at* $x = 1$.

Question

Evaluate $\lim_{x \to 1} x^2 - 1/x - 1$.

Question

Show that sum of two odd numbers is even.

Domain of discourse is integers.

Exercises (Ramanujam 2021)

Question

Evaluate $x^2 - 1/x - 1$ *at* $x = 1$.

Question

Evaluate $\lim_{x \to 1} x^2 - 1/x - 1$.

Question

Show that sum of two odd numbers is even.

Domain of discourse is integers.

Question

Solve $2\sin x - 2\sqrt{3}\cos x - \sqrt{3}\tan x + 3 = 0$.

Domain of discourse is angles.

Exercises (Ramanujam 2021)

Question

Evaluate $x^2 - 1/x - 1$ *at* $x = 1$.

Question

Evaluate $\lim_{x \to 1} x^2 - 1/x - 1$.

Question

Show that sum of two odd numbers is even.

Domain of discourse is integers.

Question

Solve $2\sin x - 2\sqrt{3}\cos x - \sqrt{3}\tan x + 3 = 0$.

Domain of discourse is angles.

Question (by school student)

If substitution had to be done for verification, why not try it first?

School mathematics (Ramanujam 2021)

- In $x + 3 = 5$, solve.
- In $x + y = 5$, x can now take many values.
- In $x + y = y + x$, x can be any number whatsoever.
- In $x = k$, k a constant. A variable is a constant?

School mathematics (Ramanujam 2021)

- In $x + 3 = 5$, solve.
- In $x + y = 5$, x can now take many values.
- In $x + y = y + x$, x can be any number whatsoever.
- In $x = k$, k a constant. A variable is a constant?

- Is $x^2 - 2 = 0$ true? Depends on what x ranges over.
- Is the angle sum property true? For angles in the plane.
- Is multiplication repeated addition? For integer multipliers.
- In general $Th \models C$.

School mathematics (Ramanujam 2021)

- In $x + 3 = 5$, solve.
- In $x + y = 5$, x can now take many values.
- In $x + y = y + x$, x can be any number whatsoever.
- In $x = k$, k a constant. A variable is a constant?

- Is $x^2 - 2 = 0$ true? Depends on what x ranges over.
- Is the angle sum property true? For angles in the plane.
- Is multiplication repeated addition? For integer multipliers.
- In general $Th \models C$.

- In factoring, polynomials are expressions.
- In graphing, polynomials are functions.
- In finding roots, polynomials are equations.
- In college math, polynomials are elements of a ring.

Programs

A programming language is a formal language equipped with rules which specify how a program in this programming language is to be executed.

```
(S1) void mergesort(int *A, unsigned i, unsigned j) {
(S2) int m = (i+j)/2;
(S3) if (i < j)
(S4) {mergesort(A,i,m); || (S5) mergesort(A,m+1,j);}
(S6) merge(A,i,m+1,j);
(S7) }
```

Programs

A programming language is a formal language equipped with rules which specify how a program in this programming language is to be executed.

(S1) void mergesort(int *A, unsigned i, unsigned j) {
(S2) int m = (i+j)/2;
(S3) if (i < j)
(S4) {mergesort(A,i,m); || (S5) mergesort(A,m+1,j);}
(S6) merge(A,i,m+1,j);
(S7) }

We might say mergesort$([987654321], 1, 9) = [123456789]$, where mergesort is an abbreviation for the finite sequence $S1\,S2\ldots S7$.

A program computes a partial function.

Logic programming (Kowalski 1974)

- So logic is not that different from a programming language.
- This can be taken further, and in languages like Prolog, logic is itself used as a programming language.
- We will not adopt this approach. For us, logic and programs are different.

Our view of verification

- Logic is a specification language, and programs are written in an implementation language.
- Verification means checking that a program (or more generally a system) which is supposed to implement a specification actually does so or not.
- Verification may lead to finding a counterexample, which shows some case in which the system does not implement the specification.
- This may be a bug in the system, and hence the implementation has to be changed.
- During an interactive process of verification (or debugging), one may also find that the specification has to be changed and then verification (debugging) has to be redone.

Tool jargon

- A theorem prover is a procedure for checking whether an inference of the form $P1, P2, \ldots, Pn \vdash C$ is valid or not. $P1, P2, \ldots, Pn$ and C are all formulas.

Tool jargon

- A theorem prover is a procedure for checking whether an inference of the form $P1, P2, \ldots, Pn \vdash C$ is valid or not. $P1, P2, \ldots, Pn$ and C are all formulas.
- A model checker is a procedure for checking whether an inference of the form $Th \models C$ is valid or not. Here C is a formula, but Th is a theory, for example, it could be integer arithmetic, or it could be a program with statements $S1 S2 \ldots Sn$.

Tool jargon

- A theorem prover is a procedure for checking whether an inference of the form $P1, P2, \ldots, Pn \vdash C$ is valid or not. $P1, P2, \ldots, Pn$ and C are all formulas.
- A model checker is a procedure for checking whether an inference of the form $Th \models C$ is valid or not.
  Here C is a formula, but Th is a theory, for example, it could be integer arithmetic, or it could be a program with statements $S1\, S2 \ldots Sn$.
- A sat solver is a procedure for checking whether a formula is satisfiable, $\not\models \neg C$. That is, $\neg C$ is not a theorem, so it finds a way of assigning to the variables of formula C so that it evaluates to true.
- These procedures are algorithmic, but they may fail to terminate, and the user of the tool may have to manually terminate it.

Direct or indirect

- Why do this in such a convoluted way?
- Why not just take a program and check what is the (partial) function that it computes?
- Or why not take a program and check whether giving it the inputs $I1, I2, I3$ will produce the output O?

Direct or indirect

- Why do this in such a convoluted way?
- Why not just take a program and check what is the (partial) function that it computes?
- Or why not take a program and check whether giving it the inputs $I1$, $I2$, $I3$ will produce the output O?

Theorem (Turing 1936)

There is no algorithm which can check the two questions stated above.

We can, of course, start running the program to check this (or simulate it in some way) but that defeats the very idea of verification. Also, how do we simulate the program on every possible input from an infinite set of values?

Note: Alan Turing's parents lived in Chhatrapur, Odisha, in 1911. He was born in London in 1912.

The verification process

- So we need to look at the system (which has a finite implementation) and reason about whether some possibilities can happen or not, and from this infer whether the specification is met. In short, we need to use logic.
- From Turing's theorem, we know that it is not possible to always get an answer to the verification question.
- Another more common reason is that the system, although finite, may be very large and analyzing it might take a very long time or may require the tool to use a very large amount of space. So often the tool may crash, or may have to be crashed!
- Then we build a coarser model of the system and run the tool on the model. Hopefully that will not crash.

The verification person

- Becoming a verification expert means that you have to learn different tools, their specification languages, what kind of things they can do, what kind of things they cannot.
- Becoming a verification expert also means that you have to learn how to change parts of the implementation, or the specification, so that you get the tool to give you an answer.
- Then from this answer try to lift the verification from the "model" to the "real" implementation and from the formula to the "real" specification.

Can’t we use machine learning?

- As we develop AI more and take decisions based on them, we need to come up with explanations as to why the AI program came up with a certain answer.
- Imagine an AI program grading your Board exam and deciding that you got 85 marks, whereas someone else whose exam performance was something like yours, got 95 marks.
- That might have made the other person eligible for some admission whereas you weren’t, or had to undergo another test. So you would like an explanation of your 85 marks.

Can’t we use machine learning?

- As we develop AI more and take decisions based on them, we need to come up with explanations as to why the AI program came up with a certain answer.
- Imagine an AI program grading your Board exam and deciding that you got 85 marks, whereas someone else whose exam performance was something like yours, got 95 marks.
- That might have made the other person eligible for some admission whereas you weren’t, or had to undergo another test. So you would like an explanation of your 85 marks.
- More seriously, if your self-driving car has an accident, the insurance company will want an explanation that it was not some flawed reasoning within the AI program before deciding to pay for your hospital bills.

Can’t we use machine learning?

- As we develop AI more and take decisions based on them, we need to come up with explanations as to why the AI program came up with a certain answer.
- Imagine an AI program grading your Board exam and deciding that you got 85 marks, whereas someone else whose exam performance was something like yours, got 95 marks.
- That might have made the other person eligible for some admission whereas you weren’t, or had to undergo another test. So you would like an explanation of your 85 marks.
- More seriously, if your self-driving car has an accident, the insurance company will want an explanation that it was not some flawed reasoning within the AI program before deciding to pay for your hospital bills.
- It turns out that what AI calls an explanation has strong parallels with what logic calls a proof.

Logic of programs (Floyd 1968, Hoare 1969)

To verify a structured program, we associate a correctness triple $\{p\}$ C $\{q\}$ with every construct C in the program. The precondition p and the postcondition q are formulas in a logic.
eg, $\{x > 0\}$ x = x-1; $\{x \geq 0\}$
Here are some inference rules for Hoare logic.

STRUCTURED PROGRAMS

$\vdash \{q[e/x]\}\text{x = e;}\{q\}$ assignment

$$\frac{\vdash \{p\}C_1\{q\}, \ \vdash \{q\}C_2\{r\}}{\vdash \{p\}C_1; C_2\{r\}}$$ sequencing

$$\frac{\vdash \{p \land b\}C_1\{q\}, \ \vdash \{p \land \neg b\}C_2\{q\}}{\vdash \{p\}\text{if } (b) \text{ /*then*/} C_1 \text{ else } C_2\{q\}}$$ if

$$\frac{\vdash \{q \land b\}C\{q\}}{\vdash \{q\}\text{while } (b)\ C\{q \land \neg b\}}$$ while

Propositional logic (de Morgan 1847, Boole 1854)

- Atomic formulas are propositions from a set *Prop*, which we do not analyze any further.
- Formulas are built up from atomic formulas using the boolean operations $\wedge, \vee, \neg$.

SYNTAX (BACKUS-NAUR/CHOMSKY/CONTEXT-FREE GRAMMAR)
$p, q ::= a \in Prop \mid (\neg p) \mid (p \vee q) \mid (p \wedge q)$
We can construct a parse tree for any well-formed formula (wff).
We can also define derived formulas like $(p \supset q) \overset{\text{def}}{=} ((\neg p) \vee q)$.
$(p \leftrightarrow q) \overset{\text{def}}{=} ((p \supset q) \wedge (q \supset p))$.

Propositional logic (de Morgan 1847, Boole 1854)

- Atomic formulas are propositions from a set *Prop*, which we do not analyze any further.
- Formulas are built up from atomic formulas using the boolean operations $\wedge, \vee, \neg$.

SYNTAX (BACKUS-NAUR/CHOMSKY/CONTEXT-FREE GRAMMAR)

$p, q ::= a \in Prop \mid (\neg p) \mid (p \vee q) \mid (p \wedge q)$

We can construct a parse tree for any well-formed formula (wff).

We can also define derived formulas like $(p \supset q) \overset{\text{def}}{=} ((\neg p) \vee q)$.

$(p \leftrightarrow q) \overset{\text{def}}{=} ((p \supset q) \wedge (q \supset p))$.

Example: Let a be the proposition “Augustus de Morgan was born in Madura, Madras presidency”.

Let b be the proposition “Augustus de Morgan was born in 1806”.

Examine the formulas $(a \wedge b)$, $(\neg a)$, $(\neg b)$, $((\neg a) \vee (\neg b))$.

Mathematics in logic (Dedekind 1888, Frege 1893)

- The success of de Morgan and Boole led mathematicians to wonder if *all* of mathematics could be described using a finite set of inference rules.
- Richard Dedekind and Giuseppe Peano began modelling numbers and calculations on them in logic. It was soon evident that boolean logic was not sufficient. Peano's "mathematical induction" required dealing with "terms" whose values were numbers and not just *true* and *false*.
- Ever since the Greeks, mathematicians had used statements like "every integer is odd or even".
- Gottlob Frege invented the formal quantifier such as $(\forall x)$, using which one could say this statement as $(\forall x)(odd(x) \vee even(x))$, where the variable x in the right hand parentheses is bound in the left hand parentheses.

Predicate logic with quantifiers

- The names *odd* and *even* that we used are called predicates (in one variable) and serve the same role as propositions did in our earlier logic. That is, $odd(x)$ has a truth value if the variable x is given an integer value.
- So now we have two kinds of things.
- Terms are built up from variables and constant symbols using function symbols (eg, numbers with $+$, $\times$). The value of a term is in a domain of discourse such as the integers.

Predicate logic with quantifiers

- The names *odd* and *even* that we used are called predicates (in one variable) and serve the same role as propositions did in our earlier logic. That is, $odd(x)$ has a truth value if the variable x is given an integer value.
- So now we have two kinds of things.
- Terms are built up from variables and constant symbols using function symbols (eg, numbers with $+, \times$). The value of a term is in a domain of discourse such as the integers.
- Atomic formulas are built up from terms by applying an n-ary predicate symbol to n terms. For example, the binary relational operators $<, \leq, >, \geq, =, \neq$ return boolean value.
- Formulas are built up from atomic formulas using the boolean operations $\wedge, \vee, \neg$ and quantifiers $(\forall x), (\exists x)$. A formula can only take a boolean value *true* or *false*.

Verification of programs with data structures

- Rod Burstall 1969 pointed out that you need to do induction on the data type (eg, lists)
- Zohar Manna and Richard Waldinger 1985, 1990 present theories for inducting over commonly used data types, and techniques which theorem provers can employ to use them. For example, in proving programs with lists, we might use the inductive predicate *list* taking a sequence of values (the “data” in the list) and a natural number (an “index” or “position” in the list):
  $\mathit{list}\ \varepsilon\ i \overset{\text{def}}{=} i = \mathit{nil}$, and
  $\mathit{list}\ aV\ i \overset{\text{def}}{=} \mathit{head}(i) = a \wedge \mathit{list}\ V\ \mathit{tail}(i)$

List program

Let *list V i* say that *i* points to a list of items *V*.
What does this program do?

```
pre {list V i}
j = null;
while (i != null) {
    k = i->next;
    i->next = j;
    j = i;
    i = k;
}
```

List reversal: program

Let $\mathit{list}\ \varepsilon\ i \overset{\text{def}}{=} i = \mathit{nil}$,
and $\mathit{list}\ aV\ i \overset{\text{def}}{=} ((i \to \mathit{val}) = a) \wedge (\mathit{list}\ V\ (i \to \mathit{next}))$.
How do we prove this program?

```
pre {$list\ V\ i$}
j = null;
while (i != null) {
    k = i->next;
    i->next = j;
    j = i;
    i = k;
}
post {$list\ V^R\ j$}
```

List reversal: top level proof

$\star$ has separation built-in, and we can define precise assertions: $\mathit{list}\ \varepsilon\ i \overset{\text{def}}{=} i = \mathit{nil}$ and $\mathit{list}\ aV\ i \overset{\text{def}}{=} \exists j (i \mapsto (a, j) \star \mathit{list}\ V\ j)$.

```
pre $\{\mathit{list}\ V\ i\}$
j = null;
inv $\{\exists W, X((\mathit{list}\ W\ i \star \mathit{list}\ X\ j) \wedge V^R = W^R X)\}$
while (i != null) {
    $\{\exists a, W, X((\mathit{list}\ aW\ i \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$
    k = i->next;
    i->next = j;
    j = i;
    i = k;
inv $\{\exists W, X((\mathit{list}\ W\ i \star \mathit{list}\ X\ j) \wedge V^R = W^R X)\}$
}
post $\{\mathit{list}\ V^R\ j\}$
```

List reversal: proof of loop body

pre $\{\exists a, W, X((\mathit{list}\ aW\ i \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

$\Longrightarrow\{\exists a, W, X, k((i \mapsto a, k \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

```
    k = i->next;
```

List reversal: proof of loop body

pre $\{\exists a, W, X((\mathit{list}\ aW\ i \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

$\Longrightarrow \{\exists a, W, X, k((i \mapsto a, k \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

```
    k = i->next;
```

$\{\exists a, W, X((i \mapsto a, k \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

```
    i->next = j;
```

List reversal: proof of loop body

pre $\{\exists a, W, X((list\ aW\ i \star list\ X\ j) \wedge V^R = (aW)^R X)\}$
$\Longrightarrow \{\exists a, W, X, k((i \mapsto a, k \star list\ W\ k \star list\ X\ j) \wedge V^R = (aW)^R X)\}$

```
    k = i->next;
    $\{\exists a, W, X((i \mapsto a, k \star list\ W\ k \star list\ X\ j) \wedge V^R = (aW)^R X)\}$
    i->next = j;
    $\{\exists a, W, X((i \mapsto a, j \star list\ W\ k \star list\ X\ j) \wedge V^R = (aW)^R X)\}$
```

List reversal: proof of loop body

pre $\{\exists a, W, X((\mathit{list}\ aW\ i \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

$\Longrightarrow \{\exists a, W, X, k((i \mapsto a, k \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

```
    k = i->next;
```

$\quad \{\exists a, W, X((i \mapsto a, k \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

```
    i->next = j;
```

$\quad \{\exists a, W, X((i \mapsto a, j \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

$\Longrightarrow \{\exists a, W, X((\mathit{list}\ W\ k \star \mathit{list}\ aX\ i) \wedge V^R = W^R aX)\}$

List reversal: proof of loop body

pre $\{\exists a, W, X((\mathit{list}\ aW\ i \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

$\Longrightarrow \{\exists a, W, X, k((i \mapsto a, k \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

    k = i->next;

  $\{\exists a, W, X((i \mapsto a, k \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

    i->next = j;

  $\{\exists a, W, X((i \mapsto a, j \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

$\Longrightarrow \{\exists a, W, X((\mathit{list}\ W\ k \star \mathit{list}\ aX\ i) \wedge V^R = W^R aX)\}$

$\Longrightarrow \{\exists W, X((\mathit{list}\ W\ k \star \mathit{list}\ X\ i) \wedge V^R = W^R X)\}$

    j = i; i = k;

List reversal: proof of loop body

pre $\{\exists a, W, X((\mathit{list}\ aW\ i \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

$\Longrightarrow\{\exists a, W, X, k((i \mapsto a, k \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

```
    k = i->next;
```

$\{\exists a, W, X((i \mapsto a, k \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

```
    i->next = j;
```

$\{\exists a, W, X((i \mapsto a, j \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$

$\Longrightarrow\{\exists a, W, X((\mathit{list}\ W\ k \star \mathit{list}\ aX\ i) \wedge V^R = W^R aX)\}$

$\Longrightarrow\{\exists W, X((\mathit{list}\ W\ k \star \mathit{list}\ X\ i) \wedge V^R = W^R X)\}$

```
    j = i; i = k;
```

post $\{\exists W, X((\mathit{list}\ W\ i \star \mathit{list}\ X\ j) \wedge V^R = W^R X)\}$

List reversal: proof of loop body

pre $\{\exists a, W, X((\mathit{list}\ aW\ i \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$
$\Longrightarrow \{\exists a, W, X, k((i \mapsto a, k \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$
    k = i->next;
    $\{\exists a, W, X((i \mapsto a, k \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$
    i->next = j;
    $\{\exists a, W, X((i \mapsto a, j \star \mathit{list}\ W\ k \star \mathit{list}\ X\ j) \wedge V^R = (aW)^R X)\}$
$\Longrightarrow \{\exists a, W, X((\mathit{list}\ W\ k \star \mathit{list}\ aX\ i) \wedge V^R = W^R aX)\}$
$\Longrightarrow \{\exists W, X((\mathit{list}\ W\ k \star \mathit{list}\ X\ i) \wedge V^R = W^R X)\}$
    j = i; i = k;
post $\{\exists W, X((\mathit{list}\ W\ i \star \mathit{list}\ X\ j) \wedge V^R = W^R X)\}$

Theorem (Cook 1978)

Given expressive loop invariants and intermediate assertions, theorem proving in Floyd-Hoare logic reduces to theorem proving in first-order logic (in polynomial time).