COMP30026 Models of Computation

Review Lecture

Bach Le / Anna Kalenkova

Lecture Week 12. Part 2

Semester 2, 2021

Propositional Logic

Propositional formulas: Syntax and semantics.

Semantics is simple, in principle—just a matter of constructing truth tables.

However, it is useful to develop an understanding of the algebraic rules, how to rewrite formulas to equivalent formulas in normal form, and so on.

Important concepts: Satisfiability and validity of formulas, logical consequence and equivalence.

Normal forms: CNF and DNF.

Mechanical proof: Propositional resolution.

Relevance of Propositional Logic

Historically: Use in hardware design, fault finding and verification (model checking).

Boolean modelling is increasingly important because of the availability of powerful SAT solvers.

First-Order Predicate Logic

Syntax and semantics.

Important semantic tools: the concepts of interpretation, and of an interpretation satisfying a formula (making it true).

Components of an interpretation: Domain and mappings giving meaning to relation symbols, function symbols, constants.

To define the meaning of quantifiers we also need to consider valuations.

Important concepts: Models and counter-models, satisfiability and validity, logical consequence and equivalence.

First-Order Predicate Logic

Useful to develop an understanding of how formulas can be rewritten, rules of passage for the quantifiers and so on.

Normal forms: Clausal form.

Obtaining equi-satisfiable formulas in clausal form: Skolemization.

Mechanical proof: Resolution, including unification.

Relevance of First-Order Predicate Logic

Historically: Use in artificial intelligence, proof assistants, automated theorem proving.

Logic programming sprang from this.

First-order predicate logic is a computer scientists' lingua franca.

Constraint solvers for various theories play central roles in tools for software verification, vulnerability detection, test data generation, planning and scheduling, ...

Proof

There is an expectation that you can provide readable and valid proofs for simple assertions (about the material covered in the subject).

The proofs will not call for induction, even though we have discussed important induction techniques in the subject: Mathematical and structural induction, including more general forms of mathematical induction.

Relevance of Proof

A formal notation is the basis for precise and unambiguous expression.

Proof is at the core of clear and rigorous thinking.

Proof is how you conduct the ultimate persuasive argument.

Discrete Mathematics: Sets and Relations

Set operations, algebra of sets.

Binary relations, domains, ranges.

Properties of relations, including reflexivity, symmetry, anti-symmetry, transitivity.

Total and partial orders.

Equivalence relations.

Well-founded relations.

Termination.

Discrete Mathematics: Functions

Domain, co-domain, and range of a function.

Image of a set under a function.

Properties of functions, including
injectivity, surjectivity, bijectivity.

Inverse functions.

Relevance of Discrete Maths

Discrete maths gives us simple but powerful modelling tools.

A major focus for us has been to understand infinite objects such as functions and languages.

Recursion allows us to define infinite objects with just a few rules.

Induction principles give us tools for reasoning about countably infinite sets.

Wellfoundedness gives us a handle on termination.

And so on ...

Regular Languages

Finite-state automata: DFAs and NFAs.

Finite-state automata as recognisers.

The regular operations.

Regular expressions.

Closure properties of regular languages.

Important techniques: Translating NFAs to DFAs, regular expressions to NFAs, and vice versa.

Using the pumping lemma for regular languages to prove non-regularity.

Relevance of Automata Theory

Compilers and other meta-programming tools.

Fast string search.

Regular expression features in JavaScript, Ruby, Python, C#, Java, ...

Vulnerability detection in string-processing programs.

Context-Free Languages

Context-free grammars, derivations of sentences.

Parse trees, ambiguity.

Push-down automata.

(Lack of) closure properties of context-free languages.

Important techniques: Translating a CFG to an equivalent PDA.

Using the pumping lemma for context-free languages to prove languages non-context-free.

Relevance of Formal Language Theory

Much of our technology would be hard to design and impossible to understand without it:

- Parsing algorithms
- Parser generators like bison (for C) and happy (for Haskell)
- Compilers' semantic analysis components
- Natural-language processing and machine translation

Computability, Turing Machines

We based our concept of "computable" on the Turing machine model.

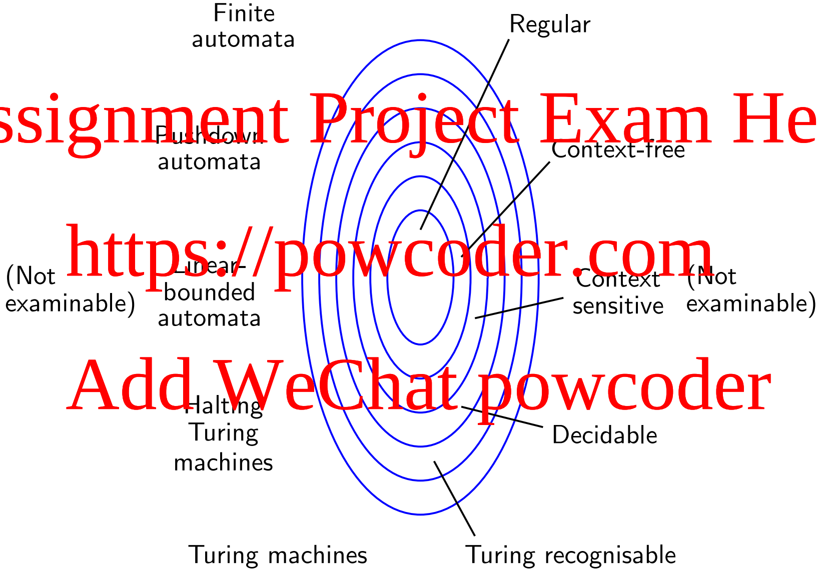
We could have used any other of a large number of equivalent models (partial recursive functions, Markov algorithms, register machines, ...)

The Church-Turing thesis:
Computable is what a Turing machine can compute.

Decidable languages: Those that are recognised by some Turing machine which halts for all input.

Turing recognisable languages: Those that have a Turing machine that acts as a recogniser (and does not necessarily halt).

Models of Computation



Closure Properties

	$\cup$	$\circ$	$*$	$R\cap$	$\cap$	compl
Reg	Y	Y	Y	Y	Y	Y
DCFL	N	N	N	Y	N	Y
CFL	Y	Y	Y	Y	N	N
Decidable	Y	Y	Y	Y	Y	Y
Recognisable	Y	Y	Y	Y	Y	N

Here '$\circ$' means concatenation, '$*$' means Kleene star, and '$R\cap$' means "intersection with a regular language".

DCFL is the class of languages that can be recognised by deterministic PDAs (DPDAs).

Decidability of Language Properties

Question	Reg	DCFL	CFL	Decidable	Recognisable
$w \in L$	D	D	D	D	U
$L = \emptyset$	D	D	D	U	U
$L = \Sigma^*$	D	D	U	U	U
$L_1 = L_2$	D	D	U	U	U
$L =$ given R	D	D	U	U	U
L regular	D	D	U	U	U
$L_1 \subseteq L_2$	D	U	U	U	U

Here ‘D’ = decidable; ‘U’ = undecidable.

Proof Techniques for (Un-)Decidability

Diagonalisation.

Reduction.

Simulation.

Exploitation of closure properties.

Relevance of Computability Theory

Knowing the limits of what can be done allows us to focus on decidable problems and functions that can be captured as algorithms.

It tells us to settle for the less-than-perfect when we are up against undecidable properties.

For example, tools for software and protocol verification, optimizing compilers, and program repair are all based on reasoning with safe approximations of programs' runtime states.

The Last COMP30026 Slide!

Phew... Thank you!