Formal Methods in Software Engineering

Theory of Computation — Spring 2025

Konstantin Chukharev

§1 Computability

Computable Functions

Definition 1 (Church–Turing thesis): *Computable functions* are exactly the functions that can be calculated using a mechanical (that is, automatic) calculation device given unlimited amounts of time and storage space.

“*Every model of computation that has ever been imagined can compute* only *computable functions, and* all *computable functions can be computed by any of several* models of computation *that are apparently very different, such as Turing machines, register machines, lambda calculus and general recursive functions.*”

Definition 2 (Computable function): A partial function $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ is *computable* (“can be calculated”) if there exists a computer program with the following properties:

- If $f(x)$ is defined, then the program terminates on the input x with the value $f(x)$ stored in memory.
- If $f(x)$ is undefined, then the program never terminates on the input x.

Effective Procedures

Definition 3 (Effective procedure): An *effective procedure* is a finite, deterministic, mechanical algorithm that guarantees to terminate and produce the correct answer in a finite number of steps.

An algorithm (set of instructions) is called an *effective procedure* if it:

- Consists of *exact*, finite steps.
- Always *terminates* in finite time.
- Produces the *correct* answer for given inputs.
- Requires no external assistance to execute.
- Can be performed *manually*, with pencil and paper.

Definition 4: A function is *computable* if there exists an effective procedure that computes it.

Examples of Computable Functions

Examples:

- The function $f(x) = x^2$ is computable.
- The function $f(x) = x!$ is computable.
- The function $f(n) =$ “n-th prime number” is computable.
- The function $f(n) =$ “the n-th digit of π” is computable.
- The Ackermann function is computable.
- The function that answers the question “Does God exist?” is computable.
- If the Collatz conjecture is true, the stopping time (number of steps to reach 1) of any n is computable.

§2 Decidability

Decidable Sets

Definition 5 (Decidable set): Given a universal set $\mathcal{U}$, a set $S \subseteq \mathcal{U}$ is *decidable* (or *computable*, or *recursive*) if there exists a computable function $f : \mathcal{U} \longrightarrow \{0, 1\}$ such that $f(x) = 1$ iff $x \in S$.

Examples:

- The set of all WFFs is decidable.
  - *We can check if a given string is well-formed by recursively verifying the syntax rules.*
- For a given finite set Γ of WFFs, the set $\{\alpha \mid \Gamma \models \alpha\}$ of all tautological consequences of Γ is decidable.
  - *We can decide $\Gamma \models \alpha$ using a truth table algorithm by enumerating all possible interpretations (at most $2^{|\Gamma|}$) and checking if each satisfies all formulas in Γ.*
- The set of all tautologies is decidable.
  - *It is the set of all tautological consequences of the empty set.*

Undecidable Sets

Definition 6 (Undecidable set): A set S is *undecidable* if it is not decidable.

Example: The existence of *undecidable* sets of expressions can be shown as follows.

An algorithm is completely determined by its *finite* description. Thus, there are only *countably many* effective procedures. But there are uncountably many sets of expressions. (Why? The set of expressions is countably infinite. Therefore, its power set is uncountable.) Hence, there are *more* sets of expressions than there are possible effective procedures.

§3 Undecidability

Halting Problem

Definition 7 (Halting problem ⊠): The *halting problem* is the problem of determining, given a program and an input, whether the program will eventually halt when run with that input.

Theorem 1 (Turing): The halting problem is undecidable.

Proof sketch: Suppose there exists a procedure H that decides the halting problem. We can construct a program P that takes itself as input and runs H on it. If H says that P halts, then P enters an infinite loop. If H says that P does not halt, then P halts. This leads to a contradiction, proving that H cannot exist. □

Halting Problem [2]

```
def halts(P, x) -> bool:
  """
  Returns True if program P halts on input x.
  Returns False if P loops forever.
  """

def self_halts(P):
  if halts(P, P):
    while True: # loop forever
  else:
    return # halt
```

Observe that `halts(self_halts, self_halts)` cannot return neither `True` nor `False`. **Contradition!**

Thus, the `halts` *does not exist* (cannot be implemented), and thus the halting problem is *undecidable*.

§4 Semi-decidability

Semi-decidability

Suppose we want to determine $\Sigma \vDash \alpha$ where Σ is infinite. In general, it is undecidable.

However, it is possible to obtain a weaker result.

Definition 8 (Semi-decidable set): A set S is *computably enumerable* if there is an *enumeration procedure* which lists, in some order, every member of S: $s_1, s_2, s_3 \ldots$

Equivalently (see Theorem 2), a set S is *semi-decidable* if there is an algorithm such that the set of inputs for which the algorithm halts is exactly S.

Note that if S is infinite, the enumeration procedure will *never* finish, but every member of S will be listed *eventually*, after some finite amount of time.

Some properties:

- Decidable sets are closed under union, intersection, Cartesian product, and complement.
- Semi-decidable sets are closed under union, intersection, and Cartesian product.

Enumerability and Semi-decidability

Theorem 2: A set S is computably enumerable iff it is semi-decidable.

Proof (⇒): If S is computably enumerable, then it is semi-decidable.

Since S is computably enumerable, we can check if $\alpha \in S$ by enumerating all members of S and checking if α is among them. If it is, we answer "yes"; otherwise, we continue enumerating. Thus, if $\alpha \in S$, the procedure produces "yes". If $\alpha \notin S$, the procedure runs forever. □

Enumerability and Semi-decidability [2]

Proof (⇐): *If S is semi-decidable, then it is computably enumerable.*

Suppose we have a procedure P which, given α, terminates and produces "yes" iff $\alpha \in S$. To show that S is computably enumerable, we can proceed as follows.

1. Construct a systematic enumeration of *all* expressions (for example, by listing all strings over the alphabet in length-lexicographical order): $\beta_1, \beta_2, \beta_3, \ldots$
2. Break the procedure P into a finite number of "steps" (for example, by program instructions).
3. Run the procedure on each expression in turn, for an increasing number of steps (see dovetailing):
   - Run P on β_1 for 1 step.
   - Run P on β_1 for 2 steps, then on β_2 for 2 steps.
   - …
   - Run P on each of $\beta_1, \ldots, \beta_n$ for n steps each.
   - …
4. If P produces "yes" for some β_i, output (yield) β_i and continue enumerating.

This procedure will eventually list all members of S, thus S is computably enumerable. □

Dual Enumerability and Decidability

Theorem 3: A set is decidable iff both it and its complement are semi-decidable.

Proof (⇒): *If A is decidable, then both A and its complement $\overline{A}$ are effectively enumerable.*

Since A is decidable, there exists an effective procedure P that halts on all inputs and returs “yes” if $\alpha \in A$ and “no” otherwise.

To enumerate A:

- Systematically generate all expressions $\alpha_1, \alpha_2, \alpha_3, \ldots$
- For each α_i, run P. If P outputs “yes”, yield α_i. Otherwise, continue.

Similarly, enumerate $\overline{A}$ by yielding α_i when P outputs “no”.

Both enumerations are effective, since P always halts, so A and its complement are semi-decidable. □

Dual Enumerability and Decidability [2]

Proof ($\Leftarrow$): *If both A and its complement $\overline{A}$ are effectively enumerable, then A is decidable.*

Let E be an enumerator for A and $\overline{E}$ an enumerator for $\overline{A}$.

To decide if $\alpha \in A$, *interleave* the execution of E and $\overline{E}$, that is, for $n = 1, 2, 3, \ldots$

- Run E for n steps and if it produces α, *halt* and output "yes".
- Run $\overline{E}$ for n steps and if it produces α, *halt* and output "no".

Since α is either in A or in $\overline{A}$, one of the enumerators will eventually produce α. The interleaving with increasing number of steps ensures fair scheduling without starvation.

Remark: The "dovetailing" technique (alternating between enumerators with increasing step) avoids infinite waiting while maintaining finite memory requirements. The alternative is to run both enumerators *simultaneosly*, in parallel, using, for example, two computers. □

Enumerability of Tautological Consequences

Theorem 4: If Σ is an effectively enumerable set of WFFs, then the set $\{\alpha \mid \Sigma \vDash \alpha\}$ of tautological consequences of Σ is effectively enumerable.

Proof: Consider an enumeration of the elements of Σ: $\sigma_1, \sigma_2, \sigma_3, \ldots$

By the compactness theorem, $\Sigma \vDash \alpha$ iff $\{\sigma_1, \ldots, \sigma_n\} \vDash \alpha$ for some n.

Hence, it is sufficient to successively test (using truth tables)

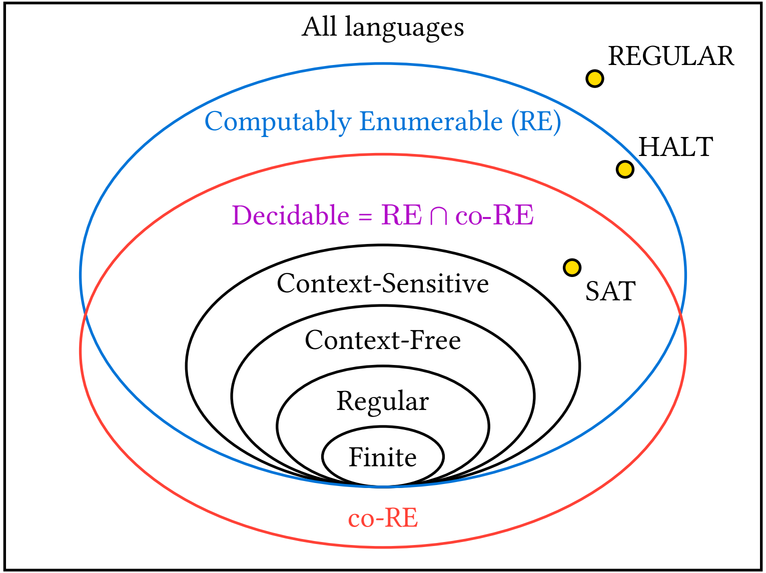
$$\begin{aligned} \emptyset &\vDash \alpha, \\ \{\sigma_1\} &\vDash \alpha, \\ \{\sigma_1, \sigma_2\} &\vDash \alpha, \end{aligned}$$

and so on. If any of these tests succeeds (each is decidable), then $\Sigma \vDash \alpha$.

This demonstrates that there is an effective procedure that, given any WFF α, will output "yes" iff α is a tautological consequence of Σ. Thus, the set of tautological consequences of Σ is effectively enumerable. □

§5 Languages

Language Classes



Language Classes [2]

$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{R} \subseteq \mathrm{RE}$

- RE
  contains languages which are *accepted* by any TM
- R = RE ∩ co-RE
  contains all languages which are *decided* by any TM
- NP
  contains all languages which are *accepted* by any *polynomial-time* TM
- P
  contains all languages which are *decided* by any *polynomial-time* and *deterministic* TM

§6 Complexity Zoo

Complexity Classes

TODO

See also: https://complexityzoo.net/Petting_Zoo

TODO

- ☑ Computability
- ☑ Decidability
- ☑ Undecidable sets
- ☑ Semi-decidability
- □ Decidable language outside of NP
- ◧ Diagram of language classes (Finite, Regular, Context-Free, Recursive (Decidable), CE(RE), co-CE)
- □ Complexity classes
- □ NP-completeness
- □ Polytime reductions