Formal Methods Summary

Simon Schneider

January 2020

1 Chapter 1: Computability and Complexity

1.1 Decidability

A decision problem $\mathcal{P}$ is called decidable if there exists an algorithm for $\mathcal{P}$. Otherwise, if there doesn't exist an algorithm for $\mathcal{P}$ then $\mathcal{P}$ is called undecidable. Undecidable problems include

- Halting problem: (Π, I), does Π terminate on I
- Correctness: (Π, I_1, I_2), does Π return I_2 when run on I_1
- Reachable-Code: (Π, n), is there an input I for Π such that Π with input I reaches code line n in Π.

1.1.1 Semi-decidable problems

A decision problem $\mathcal{P}$ is called semi-decidable if we can build a program Π such that

- Π takes as input instances I of $\mathcal{P}$
- if I is a "yes" instance, then Π return $true$
- if I is a "no" instance, then Π returns $false$ or runs forever

To prove semi-decidability it's often necessary to apply Cantor's enumeration principle.

Provide a semi-decision procedure for ALOH

> **AT-LEAST-ONE-HALTS (ALOH)**
>
> INSTANCE: A tuple (Π_1, Π_2, I), where Π_1, Π_2 are programs that take a string as input, and I is a string.
>
> QUESTION: Does either $\Pi_1(I)$ halt, $\Pi_2(I)$ halt, or do both halt?

In the procedure we can't simply run $\Pi_1(I)$ or $\Pi_2(I)$ since one of them could run forever leading whereas the second one could terminate. Thus we have to run them interleaved (similar to Cantor's principle):
Assume the existence of an interpreter function Π_{INT} that runs the first n steps of a program on a given input and returns true if the program terminates within the given n steps. We can now write a procedure like this:

```
bool ALOH(Π1,Π2,I) {
    int n = 1;
    while (true) {
        bool terminate1 = ΠINT(Π1,I);
        bool terminate2 = ΠINT(Π2,I);
        if (terminate1 || terminate2) return true;
        n++;
    }
}
```

1.2 Complement of a Decision Problem

- The complement of a decision problem $\mathcal{P}$ is obtained by "inverting" the question of $\mathcal{P}$.
- If $\mathcal{P}$ is decidable an co-$\mathcal{P}$ is the complement of $\mathcal{P}$, then co-$\mathcal{P}$ is also decidable.
- If both $\mathcal{P}$ and co-$\mathcal{P}$ are semi-decidable, then $\mathcal{P}$ is decidable.

1.3 Undecidable problems

- Same-output: (Π_1, Π_2, I), do both Π_1 and Π_2 behave the same on input I - e.g. return the same value or both not terminate.
- All-halting: (Π), does Π halt on all input strings I (we would need to check all possible strings which are infinitely many)
- Program equivalence: (Π_1, Π_2), is it true for all inputs I that both return the same value or both do not terminate?

1.4 N vs NP

1.4.1 P

- P is the collection of all problems that can be solved in polynomial time in the size of the instance (e.g. the run time is $O(|I|^k)$ where k is constant)

- Model-checking of propositional formulas, 2-SAT,... are in P

1.4.2 NP

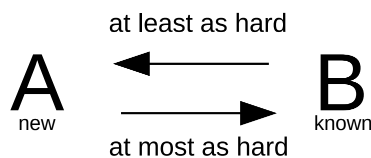
- Problems that require guess and check, e.g. it's easy to verify a potential solution for correctness, but there is no polynomial algorithm to create a solution.
- Consists of all problems $\mathcal{P}$ where there is a `choice` program Π that decides $\mathcal{P}$ and Π is such that for all instances I of $\mathcal{P}$, the run time of Π on I is polynomial in $|I|$ (e.g. $O(|I|^k)$ where k is constant).

SAT with extended SIMPLE

```
Boolean test(String s, Integer n) /* s a forumla over atoms a_1, ..., a_n */
    for all 1 <= i <= n do {
        choice(t_i = true, t_i = false);
    }
    return modelcheck (s,[t_1, ..., t_n]);    /* checks if s is true under
              the assignment mapping a_i to t_i (1 <= i <= n) */
```

1.5 Reductions

- Suppose you want to prove hardness of a new problem A - the idea is to construct an algorithm for a known hard problem B that makes use of a (hypothetical) algorithm for problem A. If such a problem reduction from B to A exists, then problem A must be at least as hard as problem B.
- For proving complexity (e.g. also in P) the reduction must be feasible in polynomial time, for just showing computability (e.g. undecidability) there are no such requirements.
- Turing reductions use the algorithm for problem B as subroutine.
- Many-one reductions define a function to map instances of problem A to instances of of B.



Provide a reduction from the Halting problem to prove ALOH is undecidable

We provide a (many-one) reduction from the Halting problem. Assume an arbitrary instance of Halting (Π, I). We construct an instance of ALOH (Π_1, Π_2, I') by setting $\Pi_1 = \Pi$, $I = I'$ and Π_2 to a program that never terminates. We have to show that positive instances of Halting map to positive instances of ALOH and negative instances of Halting map to negative instances of ALOH:

- $\Rightarrow$ Suppose a positive instance of the Halting problem, that is Π holds on I, by the above definition that means that also Π_1 holds on I' and thus regardless of Pi_2 we have a positive instance of ALOH (as at least one of the programs terminated).
- $\Leftarrow$ Suppose a negative instance of the Halting problem, that is Π does not hold on I, by the above definition that means that neither does Π_1 hold on I' and also by definition, Π_2 does not hold. Thus we have a negative instance of ALOH (since neither Π_1 nor Π_2 hold on I'.

Provide a reduction from the Halting problem to prove Different runtime is undecidable

DIFFERENT RUNTIME

INSTANCE: A tuple (Π, I_1, I_2), where Π is a program that takes a string as input, and I_1, I_2 are strings.

QUESTION: Does Π halt on I_1 within a different number of computation steps than on I_2?

Remark: If a program Π neither terminates on I_1 nor on I_2, we say that Π requires the same number of computation steps (i.e., infinitly many) for I_1 and I_2.

We provide a (polynomial time) reduction from Halting to DiffRuntime: Assume an arbitrary instance of Halting (Π, I). We construct an instance (Π', I_1, I_2) of DiffRuntime by setting $I_1 = I$ and $I_2 \neq I$ and Π' to:

```
bool Π'(String S) {
    if (S == I2) { //I2 is static
        while (true) do {}
    } else {
        return Π(S)
    }
}
```

We have to show the correctness of the reduction:

- $\Rightarrow$ Suppose a positive instance of Halting, that is Π terminates on I. Then by definition of Π' the program terminates after a different amount of steps (since for I_2 it will run forever) and thus we have a positive instance of DiffRuntime.
- $\Leftarrow$ Suppose a negative instance of Halting, then Π doesn't terminate on I. By definition of Π' it also does not terminate on neither input I_1 nor on I_2 thus we have the same runtime and further a negative instance of DiffRuntime

1.6 NP-hardness

- $\mathcal{P}$ is called NP-hard if any problem $\mathcal{P}' \in NP$ is reducible to $\mathcal{P}$
- $\mathcal{P}$ is called NP-complete if $\mathcal{P}$lies in NP and $\mathcal{P}$is NP-hard
- completeness = membership and hardness
- If a problem $\mathcal{P}$ is NP-hard and there is a polynomial-time many-one reduction from $\mathcal{P}$ to $\mathcal{P}'$, then also $\mathcal{P}'$ is NP-hard

2 Chapter 2: Satisfiability

2.1 Ackermann reduction

Given the following: $\varphi^{EUF} : F(F(x_1)) \doteq F(G(F(x_1))) \rightarrow p(x_1, y)$

1. Replace predicates e.g. p with $H_p(x_1, y) \doteq x_p$
2. Number the function instances from inside out and associate them to a new term variable (lowercase). Different occurrences of the same instance are associated with the same name. Above we would get $F_2(F_1(x_1)) \doteq F_3(G_1(F_1(x_1))) \rightarrow H_p(x_1, y) \doteq x_p$.
3. Compute $flat^E(\varphi^{EUF})$ by reading off the top level instances of the lower case term variables, e.g. $f_2 \doteq f_3 \rightarrow h_p \doteq x_p$
4. Compute $FC^E(\varphi^{EUF})$ according to:

$$\bigwedge_{i=1}^{m_F-1} \bigwedge_{j=1+1}^{m_F} (\mathcal{T}(arg(F_i)) \doteq \mathcal{T}(arg(F_j)) \rightarrow f_i \doteq f_j)$$

   (a) For each letter (f, g,...) run a double loop, the outer loop starts with e.g. $f_1 ... f_{n-1}$ and the inner loop runs from $f_2 ... f_n$. For f it would mean:

$$\rightarrow f_1 \doteq f_2 \wedge$$
$$\rightarrow f_1 \doteq f_3$$

   (b) Now for each line, look at the left side of the and check from your annotated φ^{EUF} what it's argument is. E.g. for $F(F(x_1))$ the argument of f_1 is x_1 and for f_2 the argument is f_1. Note that in case there are multiple arguments (as in e.g. $G(x_1, x_2) \wedge G(x_3, x_4)$) you need to map it as $x_1 \doteq x_3 \wedge x_2 \doteq x_4$. Complete $FC^E(\varphi^{EUF})$:

$$x_1 \doteq f_1 \rightarrow f_1 \doteq f_2 \wedge$$
$$x_1 \doteq g_1 \rightarrow f_1 \doteq f_3$$

5. Write $\varphi^E : FC^E(\varphi^{EUF}) \rightarrow flat^E(\varphi^{EUF})$

Ackermann reduction

Consider $\varphi^{EUF} : x_1 \doteq x_2 \to F(F(G(x_1))) \doteq F(F(G(x_2)))$:

$$x_1 \doteq x_2 \to \underbrace{F_2(\underbrace{F_1(\overbrace{G_1(x_1)}^{g_1})}_{f_1})}_{f_2} \doteq \underbrace{F_4(\underbrace{F_3(\overbrace{G_2(x_2)}^{g_2})}_{f_3})}_{f_4}$$

Compute $flat^E(\varphi^{EUF})$: $x_1 \doteq x_2 \to f_2 \doteq f_4$.
Add functionality constraints:

$$\begin{aligned}
g_1 \doteq f_1 &\to f_1 \doteq f_2 \quad \wedge \\
g_1 \doteq g_2 &\to f_1 \doteq f_3 \quad \wedge \\
g_1 \doteq f_3 &\to f_1 \doteq f_4 \quad \wedge \\
f_1 \doteq g_2 &\to f_2 \doteq f_3 \quad \wedge \\
f_1 \doteq f_3 &\to f_2 \doteq f_4 \quad \wedge \\
g_2 \doteq f_3 &\to f_3 \doteq f_4 \quad \wedge \\
x_1 \doteq x_2 &\to g_1 \doteq g_2
\end{aligned}$$

Then we have $\varphi^E : FC_F^E(\varphi^{EUF}) \wedge FC_G^E(\varphi^{EUF}) \to flat^E(\varphi^{EUF})$

3 Chapter 3: Deductive Verification

3.1 Operational semantics

- $\sigma(x)$ is the value of x in state σ
- $\langle exp, \sigma \rangle \to v$ means the expression exp in the state σ evaluates to the value v
- Expression have no side-effects, thus there is no state on the right side of $\langle exp, \sigma \rangle \to$
- $\langle \mathtt{p}, \sigma \rangle \to \sigma'$ means the program $\mathtt{p}$ in the state σ terminates in the final state σ'
- Evaluate of programs may terminate in a final state or may run forever

3.2 Hoare rules

3.2.1 Partial correctnes

$$\{A\}\ \mathtt{p}\ \{B\}$$

For all states σ that satisfy A, if $\langle \mathtt{p}, \sigma \rangle \to \sigma'$, then σ' satisfies B.

$$\frac{}{\{B[x/a]\}\ \texttt{x:=a}\ \{B\}}$$

$$\frac{}{\{A\}\ \texttt{skip}\ \{A\}}$$

$$\frac{\{A\}\ p_1\ \{C\} \qquad \{C\}\ p_2\ \{B\}}{\{A\}\ p_1\, ;p_2\ \{B\}}$$

$$\frac{}{\{\text{true}\}\ \texttt{abort}\ \{B\}}$$

$$\frac{\{I \land b\}\ p\ \{I\}}{\{I\}\ \texttt{while}\ b\ \texttt{do}\ p\ \texttt{od}\ \{I \land \neg b\}}$$

$$\frac{\{A \land b\}\ p_1\ \{B\} \qquad \{A \land \neg b\}\ p_2\ \{B\}}{\{A\}\ \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \{B\}}$$

$$\frac{A \Rightarrow A' \qquad \{A'\}\ p\ \{B'\} \qquad B' \Rightarrow B}{\{A\}\ p\ \{B\}}$$

Where I in the loop rule is an inductive loop invariant (meaning it needs to hold before and after each invocation of the loop).

3.2.2 Total correctness

$$[A]\ \texttt{p}\ [B]$$

For all states σ that satisfy A, then $\langle \texttt{p}, \sigma\rangle \to \sigma'$, for some σ' and σ' satisfies B. p is required to terminate!

For total correctness use the above rules (with [x] instead of {x}) and replace the abort rule and the loop rule with the following two rules:

$$\frac{}{[\text{false}]\ \texttt{abort}\ [\text{B}]}$$

$$\frac{[A \land b \land t = t_0]\ p\ [A \land t < t_0] \qquad A \land b \Rightarrow t \geq 0}{[A]\ \texttt{while}\ b\ \texttt{do}\ p\ \texttt{od}\ [A \land \neg b]}$$

Using Hoare Logic to prove a loop

$$\{i = 0 \land i \geq 0\}\ \texttt{while}\ i < n\ \texttt{do}\ \texttt{i:=i+1}\ \texttt{od}\ \{\text{i=n}\}$$

First we need to find the inductive loop invariant. Let's try simply using *true* and check if we can verify the following implication (which can be found again in the proof tree - see the *):

$$true \land \neg(i < n) \Rightarrow i = n$$
$$true \land i \geq n \Rightarrow i = n$$

true doesn't work, what would it need for this implication to be valid? Use condition of the loop for inspiration: $i < n$. Close, but it doesn't work for two reasons, a) it's not an inductive loop invariant (it doesn't hold after the last iteration) and b) it still doesn't make the implication valid. However we can use something similar: $i \leq n$.

$$\frac{(i = 0 \land n \geq 0) \Rightarrow i \leq n \qquad \dfrac{\dfrac{i \leq n \land i < n \Rightarrow i + 1 \leq n \qquad \dfrac{}{\{i + 1 \leq n\}\ \texttt{i:=i+1}\ \{i \leq n\}}}{\{i \leq n \land i < n\}\ \texttt{i:=i+1}\ \{i \leq n\}}}{\{i \leq n\}\ \texttt{while}\ i < n\ \texttt{do}\ \texttt{i:=i+1}\ \texttt{od}\ \{i \leq n \land \neg(i < n)\}} \qquad *}{\{i = 0 \land n \geq 0\}\ \texttt{while}\ i < n\ \texttt{do}\ \texttt{i:=i+1}\ \texttt{od}\ \{i = n\}}$$

With * being equal to $i \leq n \land \neg(i < n) \Rightarrow i = n$

3.3 Verification Conditions

- $VC(x := a, B) = \text{true}$
- $VC(\textbf{skip}, B) = \text{true}$
- $VC(\textbf{abort}, B) = \text{true}$
- $VC(p_1; p_2, B) = VC(p_2, B) \wedge VC(p_1, wlp(p_2, B))$
- $VC(\textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, B) = VC(p_1, B) \wedge VC(p_2, B)$
- $VC(\textbf{while } b \textbf{ do } p \textbf{ od}, B) =$
  $(I \wedge \neg b) \Rightarrow B \bigwedge (I \wedge b) \Rightarrow wlp(p, I) \bigwedge VC(p, I)$,
  where I is an inductive loop invariant.

For total correctness, replace the while rule with:

- $VC(\textbf{while } b \textbf{ do } p \textbf{ od}, B) =$
  $(I \wedge \neg b) \Rightarrow B \bigwedge (I \wedge b) \Rightarrow t \geq 0 \bigwedge (I \wedge b \wedge t = t_0) \Rightarrow wp(p, I \wedge t < t_0) \bigwedge VC(p, I \wedge t < t_0)$,
  where I is an inductive loop invariant and t is a loop variant.

3.4 WLP / WP

- $wlp(x := a, B) = B[x/a]$
- $wlp(\textbf{skip}, B) = B$
- $wlp(\textbf{abort}, B) = \text{true}$
- $wlp(p_1; p_2, B) = wlp(p_1, wlp(p_2, B))$
- $wlp(\textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, B) = (b \Rightarrow wlp(p_1, B) \wedge \neg b \Rightarrow wlp(p_2, B))$
- $wlp(\textbf{while } b \textbf{ do } p \textbf{ od}, B) = I$, where I is an inductive loop invariant.

WP is the same except for

- $wp(\textbf{abort}, B) = \text{false}$

3.5 Proving loops using VC/WLP

Given the following

$$\{A\}\textbf{while } b \textbf{ do } p \textbf{ od}\{C\}$$

we have to prove

$$VC(\textbf{while } b \textbf{ do } p \textbf{ od}, C) \wedge (A \Rightarrow wlp(\textbf{while } b \textbf{ do } p \textbf{ od}, C))$$

By using the definition of VC and wlp we can further expand it to

$$(I \wedge \neg b) \Rightarrow C \bigwedge (I \wedge b) \Rightarrow wlp(p, I) \bigwedge VC(p, I) \bigwedge A \Rightarrow I$$

Where I is an inductive loop invariant.

Example how to prove loop using VC and WLP

$$\{i = 0 \land n \geq 0\}\textbf{while } i < n \textbf{ do } i := i + 1 \textbf{ od}\{i = n\}$$

We need to prove the conjunction of the following terms:

$$(I \land \neg(i < n)) \Rightarrow i = n \quad (1)$$
$$(I \land i < n) \Rightarrow wlp(i := i + 1, I) \quad (2)$$
$$VC(i := i + 1, I) \quad (3)$$
$$(i = 0 \land n \geq 0) \Rightarrow I \quad (4)$$

For the inductive loop invariant we use $i \leq n$:

$$(i \leq n \land \neg(i < n)) \Rightarrow i = n \quad (1)$$
$$(i \leq n \land i < n) \Rightarrow wlp(i := i + 1, i \leq n) \quad (2)$$
$$VC(i := i + 1, i \leq n) \quad (3)$$
$$(i = 0 \land n \geq 0) \Rightarrow i \leq n \quad (4)$$

We can further expand this by using the defintion of wlp and VC again:

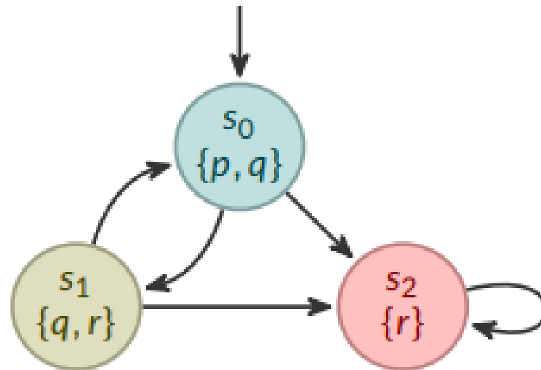
$$(i \leq n \land \neg(i < n)) \Rightarrow i = n \quad (1)$$
$$(i \leq n \land i < n) \Rightarrow i + 1 \leq n \quad (2)$$
$$true \quad (3)$$
$$(i = 0 \land n \geq 0) \Rightarrow i \leq n \quad (4)$$

1. $i \leq n \land \neg(i < n)$ can be refactored into $i \leq n \land i \geq n$ which is the same as $i = n$
2. Can be summed up as $i < n \Rightarrow i + 1 \leq n$ which is true for all integers
3. *true*
4. Also *true* because assuming the left side true, $i \leq n$ holds.

Since all four conjuncts are true the initial program is valid.

4 Chapter 4: Model Checking

4.1 Kripke Structures



- Can be represented as a directed graph with an initial state but must not have any end state (endless)
- S - finite set of states
- $S_0 \subseteq S$ - set of initial states
- $R \subseteq SxS$ - transition relation
- AP - finite set of atomic propositions
- $L : S- > 2^A P$ - function that labels each state with the set of propositions that are true in that state

4.2 Temporal Logic

4.2.1 Path quantifiers

- $A\varphi$ - all paths from given state have property ϕ
- $E\varphi$ - at least one path from given state has property φ

4.2.2 Temporal operators

- $X\varphi$ - the property φ holds in the next state of the path
- $F\varphi$ - the property ϕ holds eventually at some state of the path
- $G\varphi$ - the property φ always holds
- $\varphi U \psi$ - path has the property if there is a state on the path where ψ holds, and at every preciding state on the path, the proposition φ holds

4.2.3 CTL* vs CTL vs ACTL vs LTL

- **CTL*** contains all propositions, combined with A, E, X, F, G, U
- **CTL** is a sublogic of CTL* where path quantifiers (A, E) and temporal operators (X, F, G, U) always occur in pairs
- **ACTL*** is a sublogic of CTL* where only universal path quantification (A) is allowed
- **ACTL** is a sublogic of CTL where only universal path quantification (A) is allowed.
- **LTL** is a sublogic of CTL* without path quantifiers (no A, E). It can be thought of having an invisible A quantifier at the begining of the formula.

4.2.4 Equivalences

- $AX\varphi \equiv \neg EX\neg\varphi$
- $EF\varphi \equiv E[\text{True}U\varphi]$
- $AG\varphi \equiv \neg EF\neg\varphi$
- $AF\varphi \equiv \neg EG\neg\varphi$
- $A[\varphi U\psi] \equiv \neg E[\neg\psi U(\neg\varphi \wedge \neg\psi)] \wedge \neg EG\psi$

4.3 Marking algorithms