Program Verification

Bow-Yaw Wang

Institute of Information Science
Academia Sinica, Taiwan

December 19, 2017

Motivation I

- Computers are commonly used in modern society.
  - aircrafts, high-speed trains, cars, nuclear plants, banks, hospitals, governments, etc.
- What if computer programs go wrong?
  - Therac-25, Ariane 5, Pentium FDIV, high-speed rail, etc.
- A prominent application of logic in computer science is to verify critical computer systems.
- With logic, we are able to state and prove properties about computer systems formally.

Motivation II

- Engineering techniques are also used to build critical systems.
  - testing, metrics, documentation, good programming practices, etc.
- Such techniques cannot guarantee correctness.
- Since mid-1990's, formal logic has been deployed in computer industry.
- Many organizations are asking manufacturers to apply formal methods in development cycles.

Classification of Verification Techniques I

- Proof- versus model-based. Is the technique syntactic or semantic?
- Degree of automation. Does the technique need human guidance? How much?
- Full- versus property-verification. Does the technique verify all or some requirements?
- Intended domain. What types of systems (hardware/software, interactive/reactive etc) the technique is designed for?
- Pre- versus post-development. Is the technique applied before or after system development?

Classification of Verification Techniques II

- In this chapter, we will discuss a proof-based, semi-automatic, property-oriented verification technique for sequential programs.
- We are given a sequential program P and an intended property ϕ.
- We will construct a proof for P satisfying ϕ based on a proof calculus.
- The proof calculus is similar to the ones for propositional or predicate logic.

Why not Model Checking?

- Generally speaking, model checking works better for finite-state concurrent systems.
  - That is, systems with complex control flows but simple data types.
- For sequential programs, integer variables are often used.
- Since integer variables can have arbitrary values, sequential programs can have infinitely many states in theory.
- Verifying infinite-state systems is often undecidable.
- It is impossible to have an automatic tool for such systems.

Outline

Outline

1. A framework for software verification
   - A core programming language
   - Hoare triples
   - Partial and total correctness
   - Program variables and logical variables
2. Proof calculus for partial correctness
3. Proof calculus for total correctness
4. Introduction to Z3

A Framework for Software Verification I

- Consider the following framework:
  - Convert the informal description R of requirements into a formula ϕ_R in some symbolic logic;
  - Write a program P to realize ϕ_R;
  - Show that P indeed satisfies ϕ_R.

A Framework for Software Verification II

- Our framework is overly simplified.
- Translating informal descriptions to formal logic is always difficult.
- Even if formal specifications are available, writing programs is not easy.
- These two steps are to be done manually.
- What we will do is to show that a program satisfies its specification formally.

Outline

1. A framework for software verification
   - A core programming language
   - Hoare triples
   - Partial and total correctness
   - Program variables and logical variables
2. Proof calculus for partial correctness
3. Proof calculus for total correctness
4. Introduction to Z3

A Core Programming Language I

- We begin with the description of our programming language.
- Popular programming languages such as C, C++, Java are of our interests.
- However, we would like to focus on fundamentals of program verification.
- Thus we consider a (very) simple subset of these programming languages.
- The core programming language will help us to grasp key concepts more easily.

A Core Programming Language II

- Let us assume our core programming language has three syntactic classes: integer expressions, boolean expressions, and commands.
- Integer expressions have the following syntax:

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$

  where $n \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and x is an integer variable.
- For instance, $((-x) * 4)$ and $((x * x) - (y * y))$ are integer expressions.
- To reduce parentheses, we assume the following binding convention:

strongest		weakest
negation $(-)$	multiplication $(*)$	addition $(+)$ substraction $(-)$

- With the convention, we write $-x * 4$ for $((-x) * 4)$ and $x * x - y * y$ for $((x * x) - (y * y))$.

A Core Programming Language III

- Boolean expressions have the following syntax:

$$B ::= \texttt{false} \mid \texttt{true} \mid (!B) \mid (B \&\& B) \mid (B || B) \mid (E < E).$$

- ! stands for negation, && for conjunction, || for disjunction.
- We will use $E_1 == E_2$ for $!(E_1 < E_2) \&\& !(E_2 < E_1)$ and $E_1 != E_2$ for $!(E_1 == E_2)$.
- To reduce parentheses, we assume the following binding convention:

strongest		weakest
less (<)	not (!)	and (&&) or (||)

A Core Programming Language IV

- Commands have the following syntax:

$$C ::= x{=}E \mid C;C \mid \texttt{if } B \; \{ \; C \; \} \; \texttt{else} \; \{ \; C \; \} \mid \texttt{while } B \; \{ \; C \; \}.$$

- $x{=}E$ is an assignment; it evaluates E and then assigns the result to x.
- C_0; C_1 is a compound statement; it first executes C_0 and then C_1.
- $\texttt{if } B \; \{ \; C_0 \; \} \; \texttt{else} \; \{ \; C_1 \; \}$ is an if-statement. It evaluates B first and then executes C_0 if the result is true; otherwise, C_1 is executed.
- $\texttt{while } B \; \{ \; C \; \}$ is a while-statement.
  1. It evaluates B;
  2. If the result is false, it terminates;
  3. If the result is true, it executes C and go to Step 1.

Example

- Recall the factorial $n!$:

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) \times n! \end{aligned}$$

- Consider the following program `Fac1`:

```
y = 1;
z = 0;
while (z != x) {
  z = z + 1;
  y = y * z;
}
```

- When it terminates, `Fac1` is intended to compute $\mathtt{y} = \mathtt{x}!$.
- How do we prove it?

Outline

1. A framework for software verification
   - A core programming language
   - Hoare triples
   - Partial and total correctness
   - Program variables and logical variables

2. Proof calculus for partial correctness

3. Proof calculus for total correctness

4. Introduction to Z3

Hoare Triples I

- Let P be a program that computes a number y with $y^2 < x$ on an input number x.
- How do we specify P?

Hoare Triples II

- Let P be a program that computes a number y with $y^2 < x$ on an input number x.
- Here is a first attempt:

  On an input number x, we have $y^2 < x$ after executing P.
- This is fine. But what if $x \leq 0$?

Hoare Triples III

- Let P be a program that computes a number y with $y^2 \leq x$ on an input number x.
- To be more precise, we can say:

  On an input number $x > 0$, we have $y^2 \leq x$ after executing P.

Hoare Triples IV

Definition

Let P be a program, ϕ and ψ are logic formulae. A Hoare triple is of the form $(\!|\phi|\!)P(\!|\psi|\!)$ where ϕ is the precondition and ψ the postcondition of P. Moreover, we require that quantifiers in ϕ and ψ only bind variables not in P.

- Let P be a program that computes a number y with $y^2 < x$ on an input number x.
- We may write a Hoare triple to specify P:

$$(\!|x > 0|\!)P(\!|y^2 < x|\!)$$

where $x > 0$ is the precondition and $y^2 < x$ the postcondition.

States

- In a Hoare triple $(\!|\phi|\!) P (\!|\psi|\!)$, we have two logic formulae ϕ and ψ with free variables.
- In order to interpret ϕ and ψ, consider the standard model $\mathcal{M}$ for integers with function symbols $-$ (unary), $+$, $-$, $*$ (binary), and predicate symbols $<$ and $=$ (binary).
- A state (or store) of programs is a function I from variables to integers.
- Let I be a state and ϕ a logic formula. We say I satisfies ϕ or I is a ϕ-state (written $I \models \phi$) if $\mathcal{M} \models_I \phi$.
- Consider a state I with $I(x) = -2$, $I(y) = 5$, and $I(z) = -1$.
  - $I \models \neg(x + y < z)$ holds.
  - $I \models y - x * z < z$ does not hold.
  - $I \models \forall u(y < u \implies y * z < u * z)$ does not hold.

Example (Revisited)

- Let us consider again the Hoare triple:

$$(\!|x > 0|\!) P (\!|y^2 < x|\!)$$

- Consdier the following program P_0:

```
y = 0
```

Does $(\!|x > 0|\!) P_0 (\!|y^2 < x|\!)$ hold?

- Consider the following program P_1:

```
y = 0;
while (y * y < x) {
  y = y + 1;
}
y = y - 1;
```

Does $(\!|x > 0|\!) P_1 (\!|y^2 < x|\!)$ hold?

- How do we prove it?

Outline

1. A framework for software verification
   - A core programming language
   - Hoare triples
   - Partial and total correctness
   - Program variables and logical variables
2. Proof calculus for partial correctness
3. Proof calculus for total correctness
4. Introduction to Z3

Partial and Total Correctness I

- There are two interpretations for a Hoare triple $(\!|\phi|\!) P (\!|\psi|\!)$.

Definition

$(\!|\phi|\!) P (\!|\psi|\!)$ holds under partial correctness (written $\models_{\text{par}} (\!|\phi|\!) P (\!|\psi|\!)$) if for all states satisfying ϕ, the state resulting from executing P's execution satisfies ψ provided P terminates. We say $\models_{\text{par}}$ the satisfaction relation for partial correctness.

- That is, $\models_{\text{par}} (\!|\phi|\!) P (\!|\psi|\!)$ requires ψ holds after executing P from ϕ only when P terminates.
- Particularly, $\models_{\text{par}} (\!|\top|\!)$ `while true x = x` $(\!|\bot|\!)$ holds.

Partial and Total Correctness II

- Here is the other interpretation.

Definition

$(|\phi|)P(|\psi|)$ holds under total correctness (written $\models_{\text{tot}} (|\phi|)P(|\psi|)$) if for all states satisfying ϕ, the state resulting from executing P's execution satisfies ψ and P terminates. We say $\models_{\text{tot}}$ the satisfaction relation for total correctness.

- That is, $\models_{\text{tot}} (|\phi|)P(|\psi|)$ requires ψ holds after executing P from ϕ and P must terminate.
- Particularly, $\models_{\text{tot}} (|\top|)\ \texttt{while true x} = \texttt{x}\ (|\bot|)$ does not hold.

Partial and Total Correctness III

- Recall the program `Fac1`:

```
y = 1;
z = 0;
while (z != x) {
  z = z + 1;
  y = y * z;
}
```

- Clearly, we have $\models_{\text{tot}} (\!|x \geq 0|\!)\texttt{Fac1}(\!|y = x!|\!)$ but not $\models_{\text{tot}} (\!|\top|\!)\texttt{Fac1}(\!|y = x!|\!)$.
- On the other hand, both $\models_{\text{par}} (\!|x \geq 0|\!)\texttt{Fac1}(\!|y = x!|\!)$ and $\models_{\text{par}} (\!|\top|\!)\texttt{Fac1}(\!|y = x!|\!)$ hold.
- Naturally, total correctness is more difficult to establish.
- However, we may find insights to prove total correctness when we establish partial correctness.
- Hence we will focus on proving partial correctness.

Proof Systems for Hoare Triples

- Similar to formal logics, we will discuss a proof system for $(\!|\phi|\!)P(\!|\psi|\!)$ under partial correctness.
- We say $\vdash_{\text{par}} (\!|\phi|\!)P(\!|\psi|\!)$ is <u>valid</u> if there is a proof for $(\!|\phi|\!)P(\!|\psi|\!)$ in our proof system for partial correctness.
- Similarly, $\vdash_{\text{tot}} (\!|\phi|\!)P(\!|\psi|\!)$ is <u>valid</u> if there is a proof $(\!|\phi|\!)P(\!|\psi|\!)$ in a proof system for total correctness.
- We say a proof calculus for partial correctness is <u>sound</u> if for every ϕ, ψ and P

  $$\models_{\text{par}} (\!|\phi|\!)P(\!|\psi|\!) \text{ holds whenever } \vdash_{\text{par}} (\!|\phi|\!)P(\!|\psi|\!) \text{ is valid.}$$
- We say a proof calculus for partial correctness is <u>complete</u> if for every ϕ, ψ and P

  $$\vdash_{\text{par}} (\!|\phi|\!)P(\!|\psi|\!) \text{ is valid whenever } \models_{\text{par}} (\!|\phi|\!)P(\!|\psi|\!) \text{ holds.}$$

Outline

1. A framework for software verification
   - A core programming language
   - Hoare triples
   - Partial and total correctness
   - Program variables and logical variables
2. Proof calculus for partial correctness
3. Proof calculus for total correctness
4. Introduction to $\mathrm{Z3}$

Program Variables and Logical Variables I

- So far, free variables in the pre- and post-conditions of a Hoare triple $(\!|\phi|\!)P(\!|\psi|\!)$ are program variables.
- Consider the following program `Fac2`:

```
y = 1;
while (x != 0) {
  y = y * x;
  x = x - 1;
}
```

- Can you specify `Fac2` in a Hoare triple?
  - $(\!|x \geq 0|\!)$`Fac2`$(\!|y = x!|\!)$ is incorrect. Why?

- Consider the program Sum:

```
z = 0;
while (x > 0) {
  z = z + x;
  x = x - 1;
}
```

- Can you specify Sum?
  - $(\!| x \geq 0 |\!) \mathtt{Sum} (\!| z = \frac{x(x+1)}{2} |\!)$ is incorrect. Why?

Program Variables and Logical Variables III

- One way to specify `Fac2` and `Sum` is to introduce logical variables.
  - A logical variable occurs only in logic formulae but not programs.
- Recall the program `Fac2`:

```
y = 1;
while (x != 0) {
  y = y * x;
  x = x - 1;
}
```

- We have $(\!|x = x_0 \wedge x \geq 0|\!)\texttt{Fac2}(\!|y = x_0!|\!)$.
- Recall the program `Sum`:

```
z = 0;
while (x > 0) {
  z = z + x;
  x = x - 1;
}
```

- We have $(\!|x = x_0 \wedge x \geq 0|\!)\texttt{Sum}(\!|z = \frac{x_0(x_0+1)}{2}|\!)$.

Outline

Proof Calculus for Partial Correctness

- We will discuss a proof calculus for partial correctness of core programs.
- It was developed by R. Floyd and C. A. R. Hoare.
- Similar to proof calculus for propositional or predicate logics, we will give proof rules.
- We could write formal proofs in proof trees. But we will use a linear presentation called <u>proof tableaux</u>

Outline

1. A framework for software verification
2. Proof calculus for partial correctness
   - Proof rules
   - Proof tableaux
   - A case study: minimal-sum section
3. Proof calculus for total correctness
4. Introduction to $\mathrm{Z3}$

Proof Rules – Composition

- For each statement P, we present a proof rule for $(\!|\phi|\!)P(\!|\psi|\!)$.
- The proof rule for $(\!|\phi|\!)C_0;C_1(\!|\psi|\!)$ is

$$\frac{(\!|\phi|\!)C_0(\!|\eta|\!) \quad (\!|\eta|\!)C_1(\!|\psi|\!)}{(\!|\phi|\!)C_0;C_1(\!|\psi|\!)}\ \textit{Composition}$$

- That is, if we want to show $(\!|\phi|\!)C_0;C_1(\!|\psi|\!)$, it suffices to find η and show $(\!|\phi|\!)C_0(\!|\eta|\!)$ and $(\!|\eta|\!)C_1(\!|\psi|\!)$.
- η is called a <u>midcondition</u>.

Proof Rules – Assignment I

- The proof rule for assignment statements is

$$\frac{}{(\!|\psi[E/x]|\!)x = E(\!|\psi|\!)}\ \textit{Assignment}$$

- That is, if we start from a state satisfying $\psi[E/x]$, we end with a state satisfying ψ after executing $x = E$.

Proof Rules – Assignment II

- The proof rule may look strange at first.
- Let us see how it works.
- Consider the assignment statement $\mathtt{x} = \mathtt{x} + 1$ with the postcondition $x \geq 1$.
- By the proof rule for assignments, we have

$$\frac{}{(\!| x + 1 \geq 1 |\!) \mathtt{x} = \mathtt{x} + 1 (\!| x \geq 1 |\!)} \ \textit{Assignment}$$

- That is, if we start from a state with $x \geq 0$, we will arrive at a state with $x \geq 1$ after executing $\mathtt{x} = \mathtt{x} + 1$.

Proof Rules – Assignment III

- One might think the proof rule would be

$$\frac{}{(\!|\phi|\!)x = E(\!|\phi[E/x]|\!)}\ \mathit{WrongAssignment}$$

  - From a state satisfying ϕ, we will get a state satisfying $\phi[E/x]$ after executing $x = E$.
- But this is wrong!
- Consider the assignment $\mathtt{x} = 5$ with the precondition $x = 0$.
- We would have

$$(\!|x = 0|\!)\mathtt{x} = 5(\!|x[5/x] = 0|\!). \qquad \text{WRONG}$$

Proof Rules – Assignment IV

- Let us rethink the proof rule for assignments:

$$\frac{}{(\!|\psi[E/x]|\!)\,\mathtt{x} = E\,(\!|\psi|\!)}\ \textit{Assignment}$$

- The right way to understand this proof rule is to think backward.
- Suppose we have a postcondition ψ after executing $\mathtt{x} = E$.
- What can make the postcondition hold before $\mathtt{x} = E$?
- We want all occurrences of x in ψ equal to E <u>after</u> executing $x = E$.
- Hence $\psi[E/x]$ should hold <u>before</u> executing $x = E$.

- Let P be $\mathtt{x} = 2$. We have
  1. $(\!| 2 = 2 |\!) P (\!| x = 2 |\!)$.
  2. $(\!| 2 = 4 |\!) P (\!| x = 4 |\!)$.
  3. $(\!| 2 = y |\!) P (\!| x = y |\!)$.
  4. $(\!| 2 > 0 |\!) P (\!| x > 0 |\!)$.
- Let Q be $\mathtt{x} = \mathtt{x} + 1$. We have
  1. $(\!| x + 1 = 2 |\!) Q (\!| x = 2 |\!)$.
  2. $(\!| x + 1 = y |\!) Q (\!| x = y |\!)$.
  3. $(\!| x + 1 + 5 = y |\!) Q (\!| x + 5 = y |\!)$.
  4. $(\!| x + 1 > 0 \wedge y > 0 |\!) Q (\!| x > 0 \wedge y > 0 |\!)$.

Proof Rules – If-Statement

- The proof rule for if-statements is:

$$\frac{(\!|\phi \wedge B|\!)\, C_0\, (\!|\psi|\!) \quad (\!|\phi \wedge \neg B|\!)\, C_1\, (\!|\psi|\!)}{(\!|\phi|\!)\, \texttt{if}\ B\ \{\ C_0\ \}\ \texttt{else}\ \{\ C_1\ \}\, (\!|\psi|\!)}\ \mathit{If}-\mathit{statement}$$

- That is, if we want to show $(\!|\phi|\!)\texttt{if}\ B\ \{\ C_0\ \}\ \texttt{else}\ \{\ C_1\ \}(\!|\psi|\!)$, it suffices to show $(\!|\phi \wedge B|\!)C_0(\!|\psi|\!)$ and $(\!|\phi \wedge \neg B|\!)C_1(\!|\psi|\!)$.
- Note that $\phi \wedge B$ and $\phi \wedge \neg B$ mix a logic formula ϕ with a Boolean expression B or $\neg B$.
  - Strictly speaking, such mixed notations need be defined formally.

Proof Rules – Partial-While

- The proof rule for while-statements is:

$$\frac{(\!|\psi \wedge B|\!) C (\!|\psi|\!)}{(\!|\psi|\!) \texttt{while } B \ \{ \ C \ \} (\!|\psi \wedge \neg B|\!)} \ \textit{Partial} - \textit{while}$$

- That is, if we want to show $(\!|\psi|\!) \texttt{while } B \ \{ \ C \ \} (\!|\psi \wedge \neg B|\!)$, it suffices to show $(\!|\psi \wedge B|\!) C (\!|\psi|\!)$.
  - Note that $\neg B$ must hold after executing the while-statement.
- $(\!|\psi \wedge B|\!) C (\!|\psi|\!)$ in fact asserts that ψ remains unchanged after executing the loop body C.
  - No matter how many times the body C is executed, ψ always holds after each execution.
- To prove properties about while-statements, the key is to find such an "invariant" ψ.

Proof Rules – Implied

- We have an additional proof rule to strengthen (weaken) the precondition (postcondition):

$$\frac{\vdash_{AR} \phi' \implies \phi \quad (\!|\phi|\!)\, C\, (\!|\psi|\!) \quad \vdash_{AR} \psi \implies \psi'}{(\!|\phi'|\!)\, C\, (\!|\psi'|\!)}\ \textit{Implied}$$

  where $\vdash_{AR} \eta$ is valid if there is a natural deduction proof for η in standard laws of arithmetic.
- This proof rule is not really about program statements.
- Rather, it links program logic (Hoare triples) with predicate logic ($\vdash_{AR}$).
- Particularly, the implied proof rule helps us simplify pre- and post-conditions.

Proof Rules – Summary

$$\frac{(\!|\phi|\!)\, C_0\, (\!|\eta|\!) \quad (\!|\eta|\!)\, C_1\, (\!|\psi|\!)}{(\!|\phi|\!)\, C_0; C_1\, (\!|\psi|\!)}\ \mathit{Composition}$$

$$\frac{}{(\!|\psi[E/x]|\!)\, x = E\, (\!|\psi|\!)}\ \mathit{Assignment}$$

$$\frac{(\!|\phi \wedge B|\!)\, C_0\, (\!|\psi|\!) \quad (\!|\phi \wedge \neg B|\!)\, C_1\, (\!|\psi|\!)}{(\!|\phi|\!)\, \texttt{if}\ B\ \{\ C_0\ \}\ \texttt{else}\ \{\ C_1\ \}\, (\!|\psi|\!)}\ \mathit{If-statement}$$

$$\frac{(\!|\psi \wedge B|\!)\, C\, (\!|\psi|\!)}{(\!|\psi|\!)\, \texttt{while}\ B\ \{\ C\ \}\, (\!|\psi \wedge \neg B|\!)}\ \mathit{Partial-while}$$

$$\frac{\vdash_{AR} \phi' \implies \phi \quad (\!|\phi|\!)\, C\, (\!|\psi|\!) \quad \vdash_{AR} \psi \implies \psi'}{(\!|\phi'|\!)\, C\, (\!|\psi'|\!)}\ \mathit{Implied}$$

Outline

- Using proof rules, we can formally prove that $\vdash_{\text{par}} \llparenthesis \top \rrparenthesis \texttt{Fac1} \llparenthesis y = x! \rrparenthesis$ is valid.
- Here is the proof tree:

$$\llparenthesis y \cdot (z+1) = (z+1)! \rrparenthesis \texttt{z} = \texttt{z} + \texttt{1} \llparenthesis y \cdot z = z! \rrparenthesis$$

$$\llparenthesis y = z! \wedge z \neq x \rrparenthesis \texttt{z} = \texttt{z} + \texttt{1} \llparenthesis y \cdot z = z! \rrparenthesis \qquad \llparenthesis y \cdot z = z! \rrparenthesis \texttt{y}$$

$$\frac{\llparenthesis 1 = 1 \rrparenthesis \texttt{y} = \texttt{1} \llparenthesis y = 1 \rrparenthesis}{\llparenthesis \top \rrparenthesis \texttt{y} = \texttt{1} \llparenthesis y = 1 \rrparenthesis} \qquad \frac{\llparenthesis y = 1 \wedge 0 = 0 \rrparenthesis \texttt{z} = \texttt{0} \llparenthesis y = 1 \wedge z = 0 \rrparenthesis}{\llparenthesis y = 1 \rrparenthesis \texttt{z} = \texttt{0} \llparenthesis y = 1 \wedge z = 0 \rrparenthesis} \qquad \llparenthesis y = z! \wedge z \neq x \rrparenthesis \texttt{z} = \texttt{z} + \texttt{1}; \texttt{y} = \texttt{y} * \texttt{z} \llparenthesis y =$$

$$\llparenthesis y = z! \rrparenthesis \texttt{while(z!} = \texttt{x)\{z} = \texttt{z} + \texttt{1}; \texttt{y} = \texttt{y} * \texttt{z\}} \llparenthesis y =$$

$$\llparenthesis \top \rrparenthesis \texttt{y} = \texttt{1}; \texttt{z} = \texttt{0} \llparenthesis y = 1 \wedge z = 0 \rrparenthesis \qquad \llparenthesis y = 1 \wedge z = 0 \rrparenthesis \texttt{while(z!} = \texttt{x)\{z} = \texttt{z} + \texttt{1}; \texttt{y} = \texttt{y} *$$

$$\llparenthesis \top \rrparenthesis \texttt{y} = \texttt{1}; \texttt{z} = \texttt{0}; \texttt{while(z!} = \texttt{x)\{z} = \texttt{z} + \texttt{1}; \texttt{y} = \texttt{y} * \texttt{z\}} \llparenthesis y = x! \rrparenthesis$$

Proof Tableaux I

- Similar to proof systems for propositional and predicate logic, we will use a linear presentation of proof trees called proof tableaux.
- Let $P = C_1; C_2; \cdots ; C_n$ be a program.
- Suppose we would like to show $\vdash_{\text{par}} (\!|\phi_0|\!)P(\!|\phi_n|\!)$ is valid.
- We will write its proof as follows.

$$
\begin{array}{lll}
 & (\!|\phi_0|\!) & \\
C_1 & & \\
 & (\!|\phi_1|\!) & \text{justification} \\
\vdots & & \\
 & (\!|\phi_{n-1}|\!) & \text{justification} \\
C_n & & \\
 & (\!|\phi_n|\!) &
\end{array}
$$

where $\phi_1, \phi_2, \ldots, \phi_{n-1}$ are midconditions.

Proof Tableaux II

- Recall the proof rule for assignments:

$$\frac{}{(\!|\psi[E/x]|\!)\,x = E\,(\!|\psi|\!)}\ \textit{Assignment}$$

- Since it is easier to compute preconditions from postconditions for assignments, we often start with ϕ_n and work backward until ϕ_0.
- Ideally, we should compute the weakest precondition ϕ_{i-1} from ϕ_i for each i.
  - A logical formula ψ is weaker than ϕ if $\models \phi \implies \psi$ holds; similarly, we say ϕ is stronger than ψ.
- That is, we would like to compute the weakest condition ϕ_{i-1} to guarantee ϕ_i after execution C_i.
- When we get to ϕ_0, we can then apply the Implied proof rule to strengthen ϕ_0 if needed.

Proof Tableaux – Assignment

- Let us begin with a proof tableau for assignments.

$$\begin{array}{ll} \quad (\!|\psi[E/x]|\!) & \\ x = E & \\ \quad (\!|\psi|\!) & \qquad\qquad \text{Assignment} \end{array}$$

- Proof tableaux are read from top.
- We obtain the postcondition ψ by the Assignment proof rule.
- Hence the justification for ψ is Assignment.

Proof Tableaux – Implied

- A proof tableau for the Implied proof rule links predicate logic with arithmetic with program logic.

$$\begin{array}{lll} & (\!|\phi'|\!) & \\ & (\!|\phi|\!) & \text{Implied} \\ C & & \\ & (\!|\psi|\!) & \\ & (\!|\psi'|\!) & \text{Implied} \end{array}$$

  provided $\vdash_{AR} \phi' \implies \phi$ and $\vdash_{AR} \psi \implies \psi'$.
- We do not give a formal proof of $\vdash_{AR} \phi' \implies \phi$ nor of $\vdash_{AR} \psi \implies \psi'$ in proof tableaux.
  - Oftentimes such proofs are simple.
  - You should know how to give their formal proofs by now.

Examples I

Example

Show $\vdash_{\text{par}} (\!|y = 5|\!) \mathtt{x = y + 1} (\!|x = 6|\!)$ is valid.

Proof.

$$
\begin{array}{lll}
 & (\!|y = 5|\!) & \\
 & (\!|y + 1 = 6|\!) & \text{Implied} \\
\mathtt{x = y + 1} & & \\
 & (\!|x = 6|\!) & \text{Assignment}
\end{array}
$$

$\square$

Examples II

Example

Show $\vdash_{\text{par}} (\!|y < 3|\!) \mathtt{y = y + 1} (\!|y < 4|\!)$.

Proof.

$$
\begin{array}{lll}
 & (\!|y < 3|\!) & \\
 & (\!|y + 1 < 4|\!) & \text{Implied} \\
\mathtt{y = y + 1} & & \\
 & (\!|y < 4|\!) & \text{Assignment}
\end{array}
$$

$\square$

Examples III

Example

Let P be `z = x; z = z + y; u = z`. Show $\vdash_{\text{par}} (\!|\top|\!) P (\!|u = x + y|\!)$.

Proof.

$$
\begin{array}{lll}
 & (\!|\top|\!) & \\
 & (\!|x + y = x + y|\!) & \text{Implied} \\
\mathtt{z} = \mathtt{x}; & & \\
 & (\!|z + y = x + y|\!) & \text{Assignment} \\
\mathtt{z} = \mathtt{z} + \mathtt{y}; & & \\
 & (\!|z = x + y|\!) & \text{Assignment} \\
\mathtt{u} = \mathtt{z} & & \\
 & (\!|u = x + y|\!) & \text{Assignment}
\end{array}
$$

□

Non-Examples

- What is wrong in the following "proof?"

$$
\begin{array}{lll}
 & (\!|\top|\!) & \\
 & (\!|x+1=x+1|\!) & \text{Implied} \\
\mathtt{x = x + 1} & & \\
 & (\!|x=x+1|\!) & \text{Assignment}
\end{array}
$$

  - The Assignment proof rule replaces all occurrences of x by $x+1$.
  - We have $(\!|x+1=x+1+1|\!)\mathtt{x = x + 1}(\!|x=x+1|\!)$.

- What is wrong in the following "proof?"

$$
\begin{array}{lll}
 & (\!|x+2=y+1|\!) & \\
\mathtt{y = y + 10001;} & & \\
\mathtt{x = x + 2} & & \\
 & (\!|x=y+1|\!) & \text{Assignment}
\end{array}
$$

  - The proof rule Assignment must apply to every assignment statement.

Proof Tableaux – If

- Given ψ, consider a Hoare triple for if-statement:

$$(\!|\phi|\!)\texttt{if } B \ \{ \ C_0 \ \} \texttt{ else } \{ \ C_1 \ \}(\!|\psi|\!)$$

  How do we compute the weakest ϕ to guarantee ψ?
- ϕ is computed as follows.
  - Compute ϕ_0 such that $(\!|\phi_0|\!) C_0 (\!|\psi|\!)$;
  - Compute ϕ_1 such that $(\!|\phi_1|\!) C_1 (\!|\psi|\!)$;
  - Define ϕ as $(B \implies \phi_0) \wedge (\neg B \implies \phi_1)$.
- Observe that $\vdash_{AR} (\phi \wedge B) \implies \phi_0$ and $\vdash_{AR} (\phi \wedge \neg B) \implies \phi_1$. Hence

$$\dfrac{\dfrac{(\!|\phi_0|\!) C_0 (\!|\psi|\!)}{(\!|\phi \wedge B|\!) C_0 (\!|\psi|\!)}\ \mathit{Implied} \qquad \dfrac{(\!|\phi_1|\!) C_1 (\!|\psi|\!)}{(\!|\phi \wedge \neg B|\!) C_1 (\!|\psi|\!)}\ \mathit{Implied}}{(\!|\phi|\!)\texttt{if } B \ \{ \ C_0 \ \} \texttt{ else } \{ \ C_1 \ \}(\!|\psi|\!)}\ \mathit{If-statement}$$

Example I

Example

Let `Succ` be

```
a = x + 1;
if (a - 1 == 0) {
  y = 1
} else {
  y = a
}
```

Show $\vdash_{\text{par}} (\!|\top|\!)\texttt{Succ}(\!|y = x + 1|\!)$.

Example II

Proof.

$(\mid \top \mid)$	
$(\mid (x+1-1=0 \implies 1=x+1) \wedge (\neg(x+1-1=0) \implies x+1=x+1) \mid)$	Implied
`a = x + 1;`	
$(\mid (a-1=0 \implies 1=x+1) \wedge (\neg(a-1=0) \implies a=x+1) \mid)$	Assignment
`if (a - 1 == 0) {`	
$(\mid 1=x+1 \mid)$	If-statement
`y = 1`	
$(\mid y=x+1 \mid)$	Assignment
`} else {`	
$(\mid a=x+1 \mid)$	If-statement
`y = a`	
$(\mid y=x+1 \mid)$	Assignment
`}`	
$(\mid y=x+1 \mid)$	If-statement

□

Proof Tableaux – While I

- Recall the following proof rule:

$$\frac{(\!|\eta \wedge B|\!)\, C\, (\!|\eta|\!)}{(\!|\eta|\!)\,\texttt{while}\ B\ \{\ C\ \}\,(\!|\eta \wedge \neg B|\!)}\ \textit{Partial} - \textit{while}$$

- Suppose the while statement terminates from a state satisfying η and $(\!|\eta \wedge B|\!)\, C\, (\!|\eta|\!)$.
  - If B is false at the start of the while statement, the statement is never executed. We end up a state satisfying $\eta \wedge \neg B$.
  - If B is true at the start of the while statement, we execute C from a state satisfying $\eta \wedge B$. Since $(\!|\eta \wedge B|\!)\, C\, (\!|\eta|\!)$, η is true after executing C.
    - ★ If B is now false, we stop with $\eta \wedge \neg B$;
    - ★ If B is still true, we execute C from a state satisfying $\eta \wedge B$ again and have η after executing C.
  - The while statement terminates iff B becomes false after executing C finitely many times. Hence we have $\eta \wedge \neg B$ when the statement terminates.

Proof Tableaux – While II

- More generally, suppose we are asked to show

$$(\!|\phi|\!)\texttt{while}\ B\ \{\ C\ \}(\!|\psi|\!).$$

- How do we proceed?
  - We apply the Partial-while and Implied proof rules.

$$\frac{\vdash_{AR} \phi \implies \eta \quad \dfrac{(\!|\eta \land B|\!)C(\!|\eta|\!)}{(\!|\eta|\!)\texttt{while}\ B\ \{\ C\ \}(\!|\eta \land \neg B|\!)} \quad \vdash_{AR} \eta \land \neg B \implies \psi}{(\!|\phi|\!)\texttt{while}\ B\ \{\ C\ \}(\!|\psi|\!)}$$

- The key is to find a proper invariant η!

Proof Tableaux – While III

Definition

An <u>invariant</u> of `while` B { C } is a formula η that $\models_{\text{par}} (\!|\eta \wedge B|\!) C (\!|\eta|\!)$ holds.

- A while statement have many invariants.
  - For instance, $\top$ and $\bot$ are trivial invariants for any while statement.
- We are looking for an invariant η that establishes the precondition ϕ and postcondition ψ.
  - That is, we must have $\vdash_{AR} \phi \implies \eta$ and $\vdash_{AR} \eta \wedge \neg B \implies \psi$.
- But how do we find such an invariant?
  - we can examine program traces carefully.
  - is it possible to find invariants automatically?

Example I

Example

Recall the program `Fac1`:

```
y = 1;
z = 0;
while (z != x) {
  z = z + 1;
  y = y * z;
}
```

Show $\vdash_{\text{par}} (\!|\top|\!)\texttt{Fac1}(\!|y = x!|\!)$ is valid.

What is an invariant η such that

1. $\vdash_{AR} y = 1 \land z = 0 \implies \eta$;
2. $\vdash_{AR} \eta \land z = x \implies y = x!$; and
3. $(\!|\eta \land \neg(z = x)|\!)\mathtt{z} = \mathtt{z} + 1; \mathtt{y} = \mathtt{y} * \mathtt{z}(\!|\eta|\!)$?

Take η to be $y = z!$.

Example II

Proof.

$$
\begin{array}{ll}
\quad (\!|\top|\!) & \\
\quad (\!|1 = 0!|\!) & \text{Implied} \\
\mathtt{y} = 1; & \\
\quad (\!|y = 0!|\!) & \text{Assignment} \\
\mathtt{z} = 0; & \\
\quad (\!|y = z!|\!) & \text{Assignment} \\
\mathtt{while}\ (\mathtt{z} \ != \ \mathtt{x})\ \{ & \\
\qquad (\!|y = z! \wedge \neg(z = x)|\!) & \\
\qquad (\!|y \cdot (z+1) = (z+1)!|\!) & \text{Implied} \\
\ \ \mathtt{z} = \mathtt{z} + 1; & \\
\qquad (\!|y \cdot z = z!|\!) & \text{Assignment} \\
\ \ \mathtt{y} = \mathtt{y} * \mathtt{z}; & \\
\qquad (\!|y = z!|\!) & \text{Assignment} \\
\} & \\
\quad (\!|y = z! \wedge z = x|\!) & \text{Partial-while} \\
\quad (\!|y = x!|\!) & \text{Implied}
\end{array}
$$

□

Outline

1. A framework for software verification
2. Proof calculus for partial correctness
   - Proof rules
   - Proof tableaux
   - A case study: minimal-sum section
3. Proof calculus for total correctness
4. Introduction to $\mathrm{Z3}$

Minimal-Sum Sections of Arrays

- Let `int a[n]` declare an integer array with elements $\mathtt{a}[0], \mathtt{a}[1], \ldots, \mathtt{a}[\mathtt{n}-1]$.

Definition

Let a be an array with elements $a[0], \ldots, a[n-1]$. A section of a consists of elements $a[i], \ldots, a[j]$ for some $0 \leq i \leq j < n$. We write $S_{i,j}$ for $a[i] + a[i+1] + \cdots + a[j]$ (the sum of the section). A minimal-sum section is a section $a[i], \ldots, a[j]$ of a such that $S_{i,j}$ is less than or equal to $S_{i',j'}$ for every section $a[i'], \ldots, a[j']$.

- Example: consider the array $[-1, 3, 15, -6, 4, -5]$.
    - $[3, 15, -6]$ and $[-6]$ are sections but $[-1, 15, -6]$ isn't.
    - A minimal-sum section of the array is $[-6, 4, -5]$.
- Minimal-sum sections are in general not unique.
- We will write a program that computes a minimal-sum section for any array and verify the program with our proof calculus.

Solutions to Minimal-Sum Sections I

- A simple solution to minimal-sum sections is by enumeration.
- We enumerate all possible sections and evaluate their sums.
- There are $O(n^2)$ sections. It takes $O(n)$ to evaluate the sum of a section.
- We need $O(n^3)$ to solve the problem.

Solutions to Minimal-Sum Sections II

- Here is a better program MinSum:

```
k = 1;
t = a[0];
s = a[0];
while (k != n) {
  t = min(t + a[k], a[k]);
  s = min(s, t);
  k = k + 1;
}
```

- The variable s stores the minimal-sum of all sections in the subarray $a[0 \cdots k]$.
- The variable t stores the minimal-sum of all sections in the subarray $a[0 \cdots k]$ ending at $a[k]$.
- We will prove its correctness and get some insights from our proof.

Specifications of MinSum

- In order to prove its correctness, let us specify properties about `MinSum` by Hoare triples.
- We want to show that the variable s will be the minimal sum after execution.
- We first specify s is less than or equal to the sum of any section.

$$\textbf{S1}. \quad (\!|\top|\!)\texttt{MinSum}(\!|\forall i \forall j (0 \leq i \leq j < n \implies s \leq S_{i,j})|\!)$$

  - i and j are logical variables.
- Next, we specify s must be the sum of some section.

$$\textbf{S2}. \quad (\!|\top|\!)\texttt{MinSum}(\!|\exists i \exists j (0 \leq i \leq j < n \wedge s = S_{i,j})|\!)$$

- We will show **S1** must hold after executing `MinSum`.

How to Find Invariants?

- We need to come up with an invariant for the while statement.
- This often requires creativity.
  - Some guidelines would not hurt.
- Here are some characteristics about invariants that may help us find invariants from the textbook:
  - Invariants express the fact that the computation performed so far by the while-statement is correct.
  - Invariants typically have the same form as the desired postcondition of the while-statement.
  - Invariants express relationships between the variables manipulated by the while-statement which are re-established each time the body of the while-statement is executed.

$\vdash_{\text{par}} (\!|\top|\!)\texttt{MinSum}(\!|\mathbf{S1}|\!)$ I

- Let us show $\vdash_{\text{par}} (\!|\top|\!)\texttt{MinSum}(\!|\mathbf{S1}|\!)$.
- Consider the invariant:

$$Inv1(s,k) \triangleq \forall i \forall j (0 \leq i \leq j < k \implies s \leq S_{i,j}).$$

- If we tried to prove `MinSum` with the invariant, we would find the invariant is not strong enough.
  - We simply ignore the variable t.
  - Intuitively, this cannot be right.
- Consider another invariant:

$$Inv2(t,k) \triangleq \forall i (0 \leq i < k \implies t \leq S_{i,k-1}).$$

- We use $Inv1(s,k) \wedge Inv2(t,k)$ as an invariant to prove `MinSum`.

$\vdash_{\text{par}} (\!|\top|\!)\texttt{MinSum}(\!|\mathbf{S1}|\!)$ II

Here is the proof:

$$
\begin{array}{ll}
\quad (\!|\top|\!) & \\
\quad (\!|\mathit{Inv1}(a[0],1) \wedge \mathit{Inv2}(a[0],1)|\!) & \text{Implied} \\
\texttt{k} = 1; & \\
\quad (\!|\mathit{Inv1}(a[0],k) \wedge \mathit{Inv2}(a[0],k)|\!) & \text{Assignment} \\
\texttt{t} = \texttt{a}[0]; & \\
\quad (\!|\mathit{Inv1}(a[0],k) \wedge \mathit{Inv2}(t,k)|\!) & \text{Assignment} \\
\texttt{s} = \texttt{a}[0]; & \\
\quad (\!|\mathit{Inv1}(s,k) \wedge \mathit{Inv2}(t,k)|\!) & \text{Assignment} \\
\texttt{while (k != n) \{} & \\
\qquad (\!|\mathit{Inv1}(s,k) \wedge \mathit{Inv2}(t,k) \wedge k \neq n|\!) & \\
\qquad (\!|\mathit{Inv1}(\min(s,\min(t+a[k],a[k])),k+1) \wedge \mathit{Inv2}(\min(t+a[k],a[k]),k+1)|\!) & \text{Implied} \\
\quad \texttt{t} = \texttt{min}(\texttt{t}+\texttt{a}[\texttt{k}],\texttt{a}[\texttt{k}]); & \\
\qquad (\!|\mathit{Inv1}(\min(s,t),k+1) \wedge \mathit{Inv2}(t,k+1)|\!) & \text{Assignment} \\
\quad \texttt{s} = \texttt{min}(\texttt{s},\texttt{t}); & \\
\qquad (\!|\mathit{Inv1}(s,k+1) \wedge \mathit{Inv2}(t,k+1)|\!) & \text{Assignment} \\
\quad \texttt{k} = \texttt{k}+1 & \\
\qquad (\!|\mathit{Inv1}(s,k) \wedge \mathit{Inv2}(t,k)|\!) & \text{Assignment} \\
\texttt{\}} & \\
\quad (\!|\mathit{Inv1}(s,k) \wedge \mathit{Inv2}(t,k) \wedge k = n|\!) & \text{Partial-while} \\
\quad (\!|\mathit{Inv1}(s,n)|\!) & \text{Implied}
\end{array}
$$

Lemma

Let $s, t \in \mathbb{Z}$, a an array of size $n \geq 0$, and $0 < k < n$. Then $Inv1(s,k) \wedge Inv2(t,k) \wedge k \neq n$ implies

1. *$Inv2(\min(t + a[k], a[k]), k + 1)$; and*
2. *$Inv1(\min(s, \min(t + a[k], a[k])), k + 1)$.*

Proof.

1. Let $0 \leq i < k + 1$, we want to show $\min(t + a[k], a[k]) \leq S_{i,k}$.
   - If $i < k$, $S_{i,k} = S_{i,k-1} + a[k]$. We have $\min(t + a[k], a[k]) \leq S_{i,k-1} + a[k]$ for $t \leq S_{i,k-1}$ from $Inv2(t,k)$.
   - If $i = k$, $S_{i,k} = a[k]$. Clearly, $\min(t + a[k], a[k]) \leq S_{i,k} = a[k]$.
2. Let $0 \leq i \leq j < k + 1$. We show $\min(s, \min(t + a[k], a[k])) \leq S_{i,j}$.
   - If $i \leq j < k$, we have $s \leq S_{i,j}$ for $Inv1(s,k)$. Clearly, $\min(s, \min(t + a[k], a[k])) \leq S_{i,j}$.
   - If $i \leq j = k$, we have $\min(t + a[k], a[k]) \leq S_{i,k}$ from the previous case. Then $\min(s, \min(t + a[k], a[k])) \leq S_{i,j}$.

$\square$

Outline

Partial and Total Correctness

- We have introduced a proof calculus for proving $\vdash_{\text{par}} (\!|\phi|\!) P (\!|\psi|\!)$.
- There is always an implicit disclaimer for such proofs: $(\!|\phi|\!) P (\!|\psi|\!)$ "when P terminates."
- If P does not terminate, $\vdash_{\text{par}} (\!|\phi|\!) P (\!|\psi|\!)$ does not tell us anything.
  - $(\!|\top|\!) P (\!|\bot|\!)$ for every non-terminating P as well.
- How do we extend our calculus to prove $\vdash_{\text{tot}} (\!|\phi|\!) P (\!|\psi|\!)$?
- Observe that the while-statement is the only non-terminating statement.
- Hence the proof rules for total correctness are the same for partial correctness except the Partial-while rule.

Proof Rules – Total-while I

- Consider a while statement `while` B { C }.
- To prove total correctness of the while statement, we simply show that the statement is partially correct and terminating.
- To show the statement is terminating, we find an integer expression E such that
  - E is non-negative; and
  - E decreases its value after executing C.
- Such an expression E is called a variant.

Proof Rules – Total-while II

- The proof rule for while statements is

$$\frac{(\!|\eta \wedge B \wedge 0 \leq E = E_0|\!)\, C\, (\!|\eta \wedge 0 \leq E < E_0|\!)}{(\!|\eta \wedge 0 \leq E|\!)\, \texttt{while}\ B\ \{\ C\ \}(\!|\eta \wedge \neg B|\!)}\ \mathit{Total-while}$$

- Note that the logical variable E_0 shows E decreases strictly.
- That is, we start from a state satisfying η and $0 \leq E$. If η is an invariant and E decreases its value after executing C, we have shown the while statement is totally correct.

Proof Tableaux – While

- Proof tableaux for while statements are similar to Partial-while.
- The only difference is that we must show

$$(\!|\eta \wedge B \wedge 0 \leq E = E_0|\!)\, C\, (\!|\eta \wedge 0 \leq E < E_0|\!).$$

- We still work up proof tableaux from the end of programs.
- We just need to find a variant E for each while statement.

Example I

Example

Recall the program `Fac1`:

```
y = 1;
z = 0;
while (z != x) {
  z = z + 1;
  y = y * z;
}
```

Show $\vdash_{\text{tot}} (\!| x \geq 0 |\!) \texttt{Fac1} (\!| y = x! |\!)$.

- We know $y = z!$ is an invariant.
- What is an expression that decreases strictly?
- $x - z$ is our variant!

Example II

Proof.

$$
\begin{array}{lll}
& \quad (\!|x \geq 0|\!) & \\
& \quad (\!|1 = 0! \wedge 0 \leq x - 0|\!) & \text{Implied} \\
\texttt{y = 1;} & & \\
& \quad (\!|y = 0! \wedge 0 \leq x - 0|\!) & \text{Assignment} \\
\texttt{z = 0;} & & \\
& \quad (\!|y = z! \wedge 0 \leq x - z|\!) & \text{Assignment} \\
\texttt{while (x != z) \{} & & \\
& \qquad (\!|y = z! \wedge \neg(x = z) \wedge 0 \leq x - z = E_0|\!) & \\
& \qquad (\!|y \cdot (z+1) = (z+1)! \wedge 0 \leq x - (z+1) < E_0|\!) & \text{Assignment} \\
\quad \texttt{z = z + 1;} & & \\
& \qquad (\!|y \cdot z = z! \wedge 0 \leq x - z < E_0|\!) & \text{Assignment} \\
\quad \texttt{y = y * z;} & & \\
& \qquad (\!|y = z! \wedge 0 \leq x - z < E_0|\!) & \text{Assignment} \\
\texttt{\}} & & \\
& \quad (\!|y = z! \wedge x = z|\!) & \text{Total-while} \\
& \quad (\!|y = x!|\!) & \text{Implied}
\end{array}
$$

- Note that $x \geq 0$ is necessary for termination.
- Observe also that the Boolean guard $x \neq z$ is needed to show $\vdash_{AR} (y = z! \wedge \neg(x = z) \wedge 0 \leq x - z = E_0) \implies (y \cdot (z+1) = (z+1)! \wedge 0 \leq x - (z+1)$ <

□

Finding Variants

- Since the halting problem is undecidable, it is impossible to compute variants automatically.
  - Similar to invariants, techniques are available to find simple variants.
  - Microsoft Research develops a tool for proving termination on device drivers.

Collatz $3n + 1$ Conjecture

- To illustrate difficulties in proving total correctness, consider the program `Collatz`:

```
c = n;
while (c != 1) {
  if (c % 2 == 0) { c = c / 2; }
  else { c = 3 * c + 1; }
}
```

- Consider the input $n = 7$, the values of c are

$$7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$$

- We would like to show whether $\models_{\text{tot}} (\!|0 < n|\!)\texttt{Collatz}(\!|\top|\!)$.
  - Sine the postcondition $\top$ always holds, we just need to show `Collatz` terminates on $n > 0$.
- However, noone knows any variant to show $\vdash_{\text{tot}} (\!|0 < n|\!)\texttt{Collatz}(\!|\top|\!)$.
- In fact, noone knows if $\models_{\text{tot}} (\!|0 < n|\!)\texttt{Collatz}(\!|\top|\!)$ holds or not.

Outline

SMT Solvers I

- SAT solvers have been used in hardware verification.
  - Propositional logic suffices to model digital circuits.
- Can we use SAT solvers to verify programs?
  - Not really.
  - In general, we need predicate logic with mathematical vocabulary.
  - The problem is undecidable.
- However, there are tools that can help us solve simple cases.

SMT Solvers II

- Satisfiability Modulo Theories (SMT) solvers are SAT solvers extended with various theories.
  - For instance, theories of linear arithmetic, uninterpreted functions, etc.
- Such theories allow us to verify properties about programs.
- The basic idea is not complicated.
  - In addition to propositional atoms, we introduce predicate symbols as new propositional atoms.
  - Efficient SAT algorithms can still be used on top of these propositional atoms.
- In fact, many SMT solvers are based on SAT solvers.

SMT Solvers III

- Similar to SAT solvers, there is a competition for SMT solvers.
- Recent SMT solvers thus adopt the SMT-LIB input format.
- In the following, we will introduce the SMT solver Z3.
- Z3 is developed at Microsoft Research.
- Source codes are available.
- We will use its PYTHON interface in class.

Using Z3 in PYTHON

```
from z3 import *    # import Z3 library

s = Solver ()       # create an SMT solver s

print s.check ()    # check satisfiability
print s.model ()    # obtain a model
```

- We first import Z3 PYTHON library.
  - Remember to add your Z3 PYTHON path to `PYTHONPATH`.
- The Z3 solver checks whether the conjunction of formulae is satisfiable.
- When there is no formula, the degenerated conjunction is true.
- The empty model suffices to satisfy the degenerated conjunction.

Equational Theory I

```python
from z3 import *

s = Solver ()

m = Int ('M')  """ create the integer constant 'M' """
n = Int ('N')  """ create the integer constant 'N' """

s.add (m == n)   """ add the formula 'M = N' """
print s.check ()
if s.check () == sat: print s.model ()
```

- The PYTHON variable `m` contains a Z3 Boolean constant `M`.
- The PYTHON variable `n` contains a Z3 Boolean constant `N`.
- `m == n` is the Z3 formula for $M = N$.
- Clearly, $M = N$ is satisfiable.
- The model [`N = 0`, `M = 0`] is returned.

Boolean Theory I

```
from z3 import *

s = Solver ()

x = Bool ('X')        # create the Boolean constant 'X'

s.add (Not (x))       # add the formula ~X
print s.check ()
if s.check () == sat: print s.model ()
```

- `Not(x)` is the Z3 formula for $\neg$`X`.
- The formula $\neg$`X` is satisfiable.
- The model [`X = False`] is returned.

Boolean Theory II

```
from z3 import *

s = Solver ()

x = Bool ('X')

s.add (Not (x))
s.add (x)              # add the formula X
print s.check ()
if s.check () == sat: print s.model ()
```

- The formulae ¬`X` and `X` is not satisfiable.
  - What if we ask $\mathrm{Z3}$ to give a model?

Boolean Theory III

- Boolean sort: `BoolSort()`
- Boolean values: `BoolVal(False)`, `BoolVal(True)`
  - `False` and `True` are PYTHON Boolean values
- Constant declaration: `Bool`(*name*) or `Bools`(*names*)
- Unary operator: `Not` (negation)
- Binary operators: `Or` (disjunction), `And` (conjunction), `Xor` (exclusive or), `Implies` (implication)

Boolean Theory IV

```
""" 3 Pigeons to live in 2 holes """
from z3 import *
s = Solver ()

pigeons = [ BoolVector ('P', 2),
            BoolVector ('Q', 2),
            BoolVector ('R', 2) ]

""" each pigeon must live in one hole """
s.add (Or (pigeons[0][0], pigeons[0][1]))
s.add (Or (pigeons[1][0], pigeons[1][1]))
s.add (Or (pigeons[2][0], pigeons[2][1]))

""" a hole receives at most one pigeon """
s.add (And (Or (Not (pigeons[0][0]), Not (pigeons[1][0])),
            Or (Not (pigeons[0][0]), Not (pigeons[2][0])),
            Or (Not (pigeons[1][0]), Not (pigeons[2][0]))))
s.add (And (Or (Not (pigeons[0][1]), Not (pigeons[1][1])),
            Or (Not (pigeons[0][1]), Not (pigeons[2][1])),
            Or (Not (pigeons[1][1]), Not (pigeons[2][1]))))

print s.check ()
```

Arithmetic Theory I

```
from z3 import *
s = Solver ()

i = Int ('I')
x = Real ('X')

s.add (i < x)
s.add (x < i + 1)
print s.check ()
if s.check () == sat: print s.model ()
```

- $\mathrm{Z3}$ supports integer and real numbers.
  - `Int('I')` declares a $\mathrm{Z3}$ integer constant named `I`.
  - `Real('X')` declares a $\mathrm{Z3}$ real constant named `X`.
- We can use PYTHON arithmetic expressions as $\mathrm{Z3}$.
  - The Z3 PYTHON module overloads arithmetic functions.

Arithmetic Theory II

- Integer sort: $\texttt{IntSort}(), \texttt{RealSort}()$
- Integer values: $\texttt{IntVal}(\mathit{value}), \texttt{RealVal}(\mathit{value})$
- Constant declaration: $\texttt{Int}(\mathit{name})$ or $\texttt{Real}(\mathit{name})$
- Binary operators: $+$, $-$, $*$, $/$, and $\%$.
- Binary relations: $<$, $<=$, $>$, and $>=$.

Bitvector Theory I

```
from z3 import *

s = Solver ()
# create a 32-bit bit-vector constant 'X'
x = BitVec ('x', 16)
s.add (x > 0)
s.add (x & (x - 1) == 0)

# a trick to find all solutions
while s.check () == sat:
    print s.model()[x]
    s.add(x != s.model()[x])
```

- Z3 supports bit-vectors.
  - `BitVec('x', 16)` declares a 16-bit bit-vector constant named `x`.
- Again, PYTHON bit-vector expressions are overloaded to construct Z3 bit-vector expressions.

Bitvector Theory II

- Sort declaration: `BitVecSort`(*width*)
- Constant declaration: `BitVec`(*name*,*width*)
- Binary operators: & (bitwise-and), | (bitwise-or), ˜ (bitwise-invert), ˆ (exclusive-or), $+$, $-$, $*$, $/$, %, $>>$ (right-shift), and $<<$ (left-shift).
- Binary relations: $<$, $<=$, $>$, and $>=$.
- Additional functions:
  - `Concat`(*bitvecs*) represents the concatenation of a list of bit-vectors.
  - `Extract`(*high*, *low*, *bitvec*) represents a sub bit-vector of *bitvec*.
  - `RotateLeft`(*bitvec*, *r*) represents the left rotation of *bitvec*.
  - `RotateRight`(*bitvec*, *r*) represents the right rotation of *bitvec*.

Theory of Uninterpreted Functions I

```
from z3 import *
# declare an unknown sort of universe
U = DeclareSort('U')
# a and b are constants of sort U
a, b = Const('a', U), Const('b', U)
# f is an unterpreted function from U * U to U
f = Function('f', U, U, U)

s = Solver ()
s.add(f(a, b) == a)
print s.check()
s.add(f(f(a, b), b) != a)
print s.check()
```

- $\mathrm{Z3}$ allows uninterpreted functions.
- An uninterpreted function need not be fully specified.
  - If $a \neq b$, $f(a, a)$ can take any value in U.
- However, $\mathrm{Z3}$ deduces $f(f(a, b), b) = a$ from $f(a, b) = a$.

Theory of Uninterpreted Functions II

- Sort declaration: `DeclareSort`(*name*)
- Constant declaration: `Const`(*name*, *sort*)
- Uninterpreted function declaration:
  `Function`(*name*, *domainsorts*, *rangesort*)

Save and Restore Context

```
from z3 import *

s = Solver ()

x = Bool ('X')

s.add (Not (x))
print s.check ()
if s.check () == sat: print s.model ()

s.push ()          # save the current context
s.add (x == BoolVal (True))
print s.check ()
if s.check () == sat: print s.model ()

s.pop ()           # restore the saved context
print s.check ()
if s.check () == sat: print s.model ()
```

- How do you simulate "push" and "pop" in MiniSAT?

McCarthy91

```
int mccarthy91 (int n) {
  int c;
  int ret;
  ret = n;
  c = 1;
  while (c > 0) {
    if (ret > 100) {
      ret = ret - 10;
      c--;
    } else {
      ret = ret + 11;
      c++;
    }
  }
  return ret;
}
```

- For $n \leq 100$, mccarthy91(n) is 91. For $n > 100$, mccarthy91(n) is $n - 10$.
- Let us try to find an invariant to prove it!

Invariant for McCarthy 91 I

- First, we will set up pre- and post-conditions.
- Immediately before entering the loop, we have $ret = n \wedge c = 1$.

```
from z3 import *

n = Int ('n')
ret = Int ('ret')
c = Int ('c')

solver = Solver ()
```

- This is represented by $\mathrm{And}(\mathrm{ret} == \mathrm{n}, \mathrm{c} == 1)$.
- Immediately after leaving the loop, we want to show

$$(\mathrm{n} \leq 100 \implies \mathrm{ret} = 91) \quad \wedge \quad (\mathrm{n} > 100 \implies \mathrm{ret} = \mathrm{n} - 10).$$

- This is represented by $\mathrm{And}(\mathrm{Implies}(n <= 100, \mathrm{ret} == 91), \mathrm{Implies}(\mathrm{n} > 100, \mathrm{ret} == \mathrm{n} - 10))$.

Invariant for McCarthy 91 II

- Any invariant η must have
  - $\vdash_{AR} \mathsf{ret} = \mathsf{n} \land \mathsf{c} = 1 \implies \eta$;
  - $\vdash_{AR} \eta \land \neg(\mathsf{c} > 0) \implies [(\mathsf{n} \leq 100 \implies \mathsf{ret} = 91) \land (\mathsf{n} > 100 \implies \mathsf{ret} = \mathsf{n} - 10)]$; and
  - finally,

$$\begin{array}{l} (\!|\eta \land \mathsf{c} > 0|\!) \\ \mathsf{if}\ (\mathsf{ret}\ >\ 100)\ \{\ \mathsf{ret} = \mathsf{ret} - 10;\ \mathsf{c} - -;\ \}\ \mathsf{else}\ \{\ \mathsf{ret} = \mathsf{ret} + 11;\ \mathsf{c} + +;\ \} \\ (\!|\eta|\!) \end{array}$$

- Suppose we come up with an η and express it in Z3 PYTHON.
- How do we use Z3 to check them?

Invariant for McCarthy 91 III

- The first two requirements are similar.
- They are of the form $\vdash_{AR} \phi \implies \psi$.
- It is equivalent to $\phi \wedge \neg\psi$ is not satisfiable.
- We use the following PYTHON code:

```
def check_implies (phi, psi):
  solver.push ()
  f = And (phi, Not (psi))
  solver.add (f)
  result = solver.check ()
  solver.pop ()
  return result != sat
```

Invariant for McCarthy 91 IV

- For the last requirement, note that

$$
\begin{array}{l}
\left(\!\left| \begin{array}{lll} \mathsf{ret} > 100 \implies \eta[\mathsf{c} \mapsto \mathsf{c} - 1][\mathsf{ret} \mapsto \mathsf{ret} - 10] & & \wedge \\ \neg(\mathsf{ret} > 100) \implies \eta[\mathsf{c} \mapsto \mathsf{c} + 1][\mathsf{ret} \mapsto \mathsf{ret} + 11] & & \end{array} \right|\!\right) \\
\mathsf{if}\ (\mathsf{ret}\ >\ 100)\ \{ \\
\qquad (\!|\eta[\mathsf{c} \mapsto \mathsf{c} - 1][\mathsf{ret} \mapsto \mathsf{ret} - 10]|\!) \\
\quad \mathsf{ret} = \mathsf{ret} - 10; \\
\qquad (\!|\eta[\mathsf{c} \mapsto \mathsf{c} - 1]|\!) \\
\quad \mathsf{c} - -; \\
\qquad (\!|\eta|\!) \\
\}\ \mathsf{else}\ \{ \\
\qquad (\!|\eta[\mathsf{c} \mapsto \mathsf{c} + 1][\mathsf{ret} \mapsto \mathsf{ret} + 11]|\!) \\
\quad \mathsf{ret} = \mathsf{ret} + 11; \\
\qquad (\!|\eta[\mathsf{c} \mapsto \mathsf{c} + 1]|\!) \\
\quad \mathsf{c} + +; \\
\qquad (\!|\eta|\!) \\
\} \\
\quad (\!|\eta|\!)
\end{array}
$$

Invariant for McCarthy 91 V

- Hence it suffices to check

$$\vdash_{AR} \eta \wedge \mathsf{c} > 0 \implies \left(\begin{array}{ll} \mathsf{ret} > 100 \implies \eta[\mathsf{c} \mapsto \mathsf{c} - 1][\mathsf{ret} \mapsto \mathsf{ret} - 10] & \wedge \\ \neg(\mathsf{ret} > 100) \implies \eta[\mathsf{c} \mapsto \mathsf{c} + 1][\mathsf{ret} \mapsto \mathsf{ret} + 11] & \end{array} \right)$$

- When we guess an η, we can check the last requirement after performing 4 substitutions.
- Luckily, Z3 Python can do substitutions for us.

```
def requirement3 (eta):
  b_true  = Implies (ret > 100,
                     substitute (substitute (eta, (c, c - 1)),
                                             (ret, ret - 10)))
  b_false = Implies (Not (ret > 100),
                     substitute (substitute (eta, (c, c + 1)),
                                             (ret, ret + 11)))
  return check_implies (And (Not (c > 0), eta),
                        And (b_true, b_false))
```

Invariant for McCarthy 91 VI

- We have almost everything except η.
- Here is what we will do:
  - Guess η and express it in Z3 PYTHON.
  - Use Z3 PYTHON to check the three requirements on η.
  - If all three requirements pass, we are done.
  - Otherwise, guess another η and repeat.

Invariant for McCarthy 91 VII

- It may be too hard to guess η for all input n.
- We hence consider two sub-problems:
  - $(\!|\mathsf{n} > 100|\!)\mathsf{ret} \ = \ \mathsf{mccarthy91(n)}(\!|\mathsf{ret} = \mathsf{n} - 10|\!)$; and
  - $(\!|\mathsf{n} \leq 100|\!)\mathsf{ret} \ = \ \mathsf{mccarthy91(n)}(\!|\mathsf{ret} = 91|\!)$
- Try to find an invariant for each sub-problem.
- Then combine two sub-invariants into one for the main problem.
- Have fun!