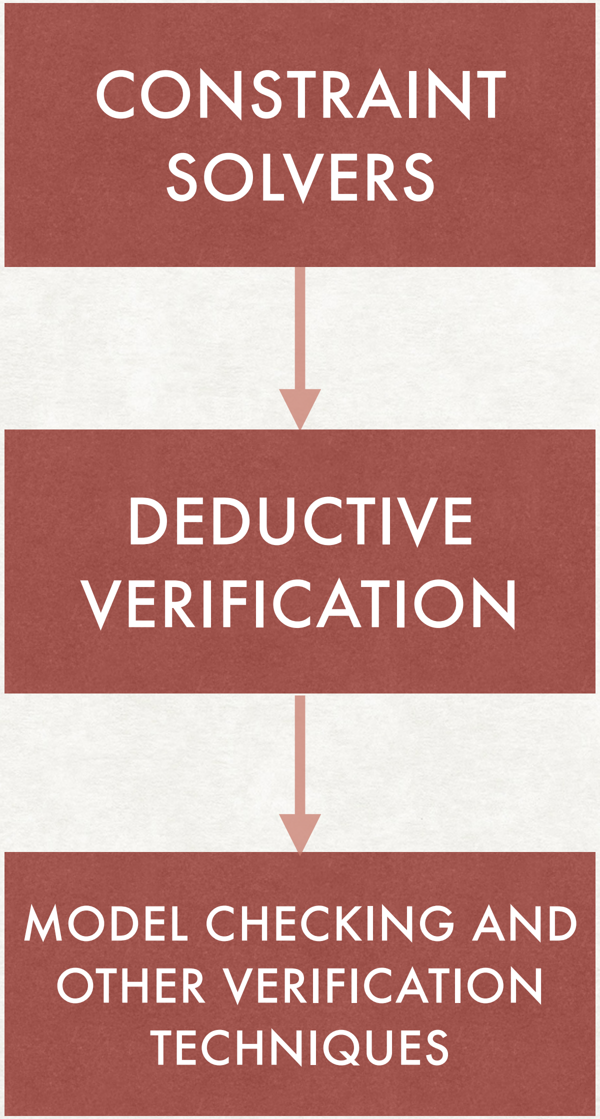
COURSE STRUCTURE



CONSTRAINT SOLVERS

- Propositional Logic, SAT solving, DPLL
- First-Order Logic, SMT
- First-Order Theories

DEDUCTIVE VERIFICATION

- Operational Semantics
- Strongest Post-condition, Weakest Pre-condition
- Hoare Logic

MODEL CHECKING AND OTHER VERIFICATION TECHNIQUES

- Predicate Abstraction, CEGAR
- Abstract Interpretation
- Property-directed Reachability

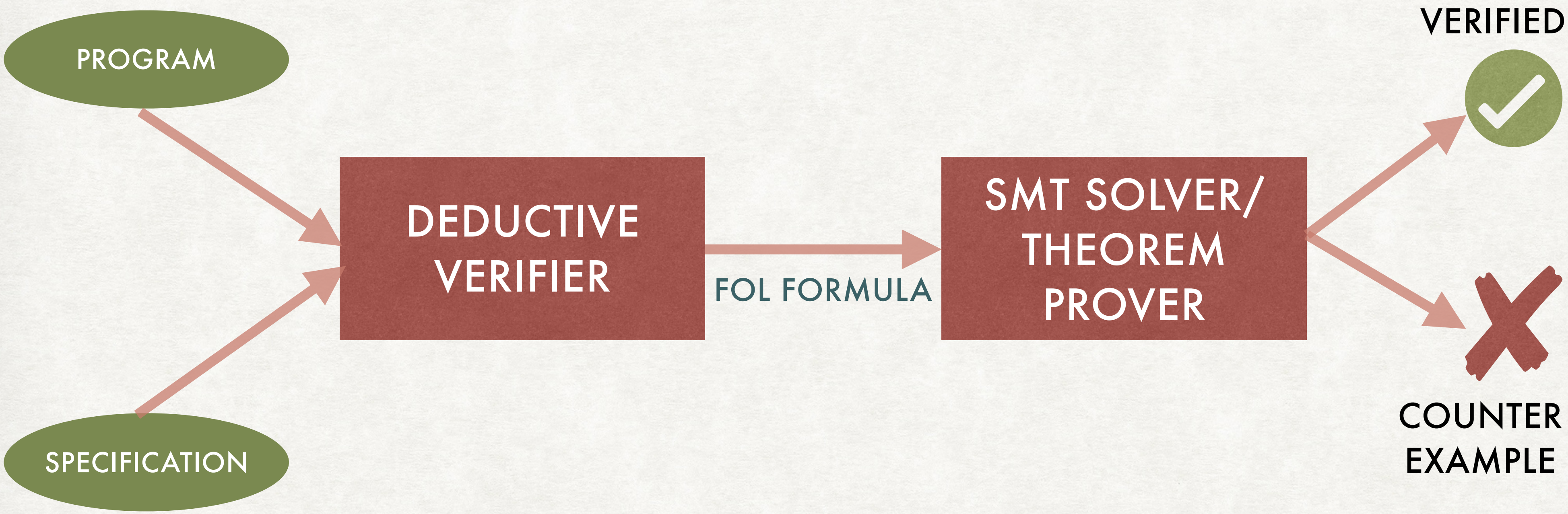
FORMAL SPECIFICATION AND VERIFICATION OF PROGRAMS

INTRODUCTION

- So far we have seen…
    - Syntax, Semantics for Propositional Logic and First-Order Logic and (some examples of) Decision Procedures for Validity/Satisfiability
    - Underlying engine for Deductive Verification of programs
- Now we will how to reduce the program verification problem to the satisfiability problem in first-order logic.

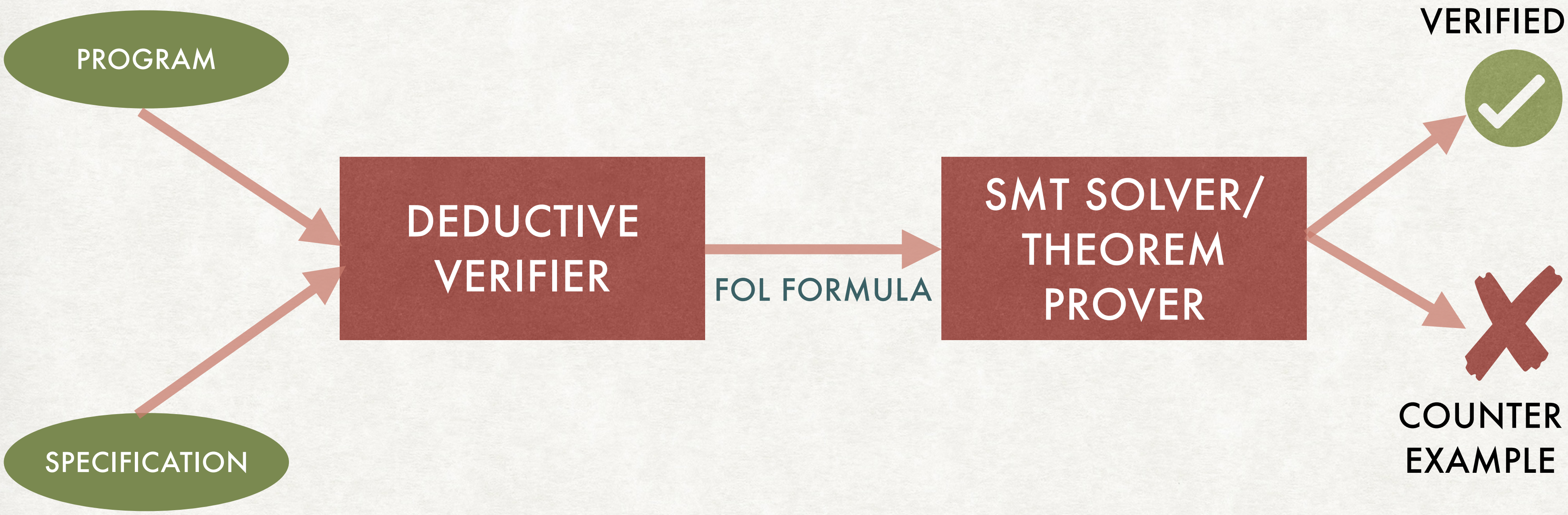
AUTOMATED VERIFICATION

OVERVIEW



AUTOMATED VERIFICATION

OVERVIEW



PROGRAM

SPECIFICATION

DEDUCTIVE VERIFIER

FOL FORMULA

SMT SOLVER/ THEOREM PROVER

VERIFIED

COUNTER EXAMPLE

- Assertions
- Pre-conditions/Post-conditions
- Invariants
- …

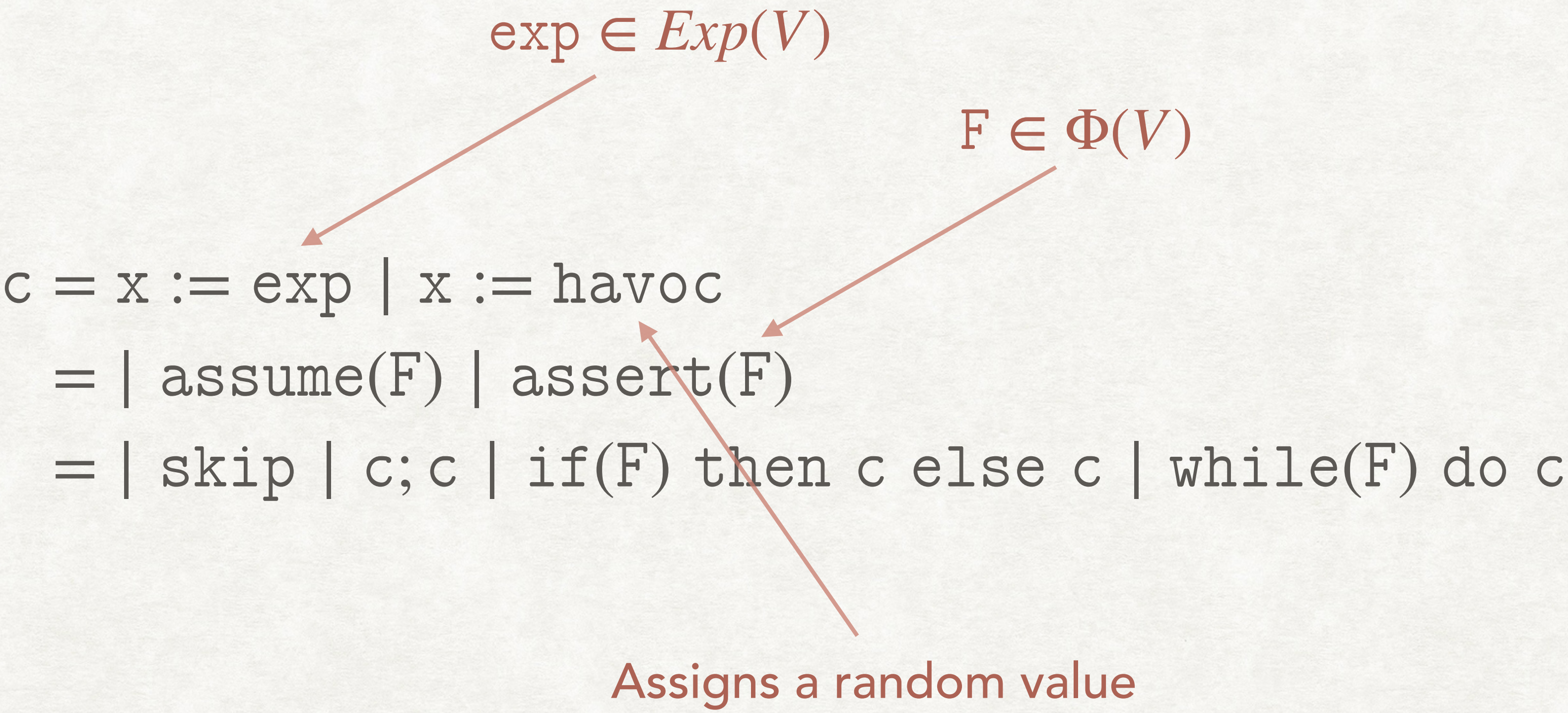
IMP

A SMALL IMPERATIVE PROGRAMMING LANGUAGE

- Let V be a set of program variables
- Let $Exp(V)$ be the set of linear expressions, and $\Phi(V)$ be the set of linear formulae over V
  - $Exp(V)$ are terms in LRA or LIA
  - $\Phi(V)$ are formulae in LRA or LIA
- Examples
  - $3x + 2y \in Exp(\{x, y\})$
  - $x \leq y + z \wedge z = w \in \Sigma(\{x, y, z, w\})$

IMP

A SMALL IMPERATIVE PROGRAMMING LANGUAGE



exp $\in Exp(V)$

F $\in \Phi(V)$

c = x := exp | x := havoc

= | assume(F) | assert(F)

= | skip | c; c | if(F) then c else c | while(F) do c

Assigns a random value

EXAMPLES

```
assume(i = 0 ∧ n ≥ 0);
while(i < n) do
  i := i + 1;
assert(i = n);
```

PRE-CONDITION

POST-CONDITION

EXAMPLES

```
assume(i = 0 ∧ n ≥ 0);

while(i < n) do

  i := i + 1;

assert(i = n);
```

FOL formula in LIA whose
free variables are program variables

EXAMPLES

```
{i = 0 ∧ n ≥ 0}

while(i < n) do

  i := i + 1;

{i = n}
```

EXAMPLES

```
{i = 0 ∧ n ≥ 0}

while(i < n) do

  i := i + 1;

{i = n}
```

```
{Pre-condition}

Program

{Post-condition}
```

EXAMPLES

Linear Search

Input: Array `a`, Lower limit `l`, Upper limit `u`, Element to be searched `e`

Output: true if element is present, false otherwise

```
i := l;
present := false;
while(i <= u && !present)
{
  if (a[i] == e) then
    present := true;
  else
    i := i + 1;
}
```

EXAMPLES

Linear Search

Input: Array `a`, Lower limit `l`, Upper limit `u`, Element to be searched `e`

Output: true if element is present, false otherwise

```
assume(?);
i := l;
present := false;
while(i <= u && !present)
{
  if (a[i] == e) then
    present := true;
  else
    i := i + 1;
}
assert(?);
```

EXAMPLES

Linear Search

Input: Array `a`, Lower limit `l`, Upper limit `u`, Element to be searched `e`

Output: true if element is present, false otherwise

```
assume(l ≥ 0 ∧ u ≤ |a|);
i := l;
present := false;
while(i <= u && !present)
{
  if (a[i] == e) then
    present := true;
  else
    i := i + 1;
}
assert(?);
```

EXAMPLES

Linear Search

Input: Array `a`, Lower limit `l`, Upper limit `u`, Element to be searched `e`

Output: true if element is present, false otherwise

```
assume(l ≥ 0 ∧ u ≤ |a|);
i := l;
present := false;
while(i <= u && !present)
{
  if (a[i] == e) then
    present := true;
  else
    i := i + 1;
}
assert(present ↔ l ≤ i ≤ u ∧ a[i] = e);
```

EXAMPLES

Linear Search

Input: Array `a`, Lower limit `l`, Upper limit `u`, Element to be searched `e`

Output: true if element is present, false otherwise

```
assume(l ≥ 0 ∧ u ≤ |a|);
i := l;
present := false;
while(i <= u && !present)
{
  if (a[i] == e) then
    present := true;
  else
    i := i + 1;
}
assert(present ↔ ∃x. l ≤ x ≤ u ∧ a[x] = e);
```

OPERATIONAL SEMANTICS OF IMP

- In order to formally define the verification problem, i.e. ‘the program satisfies its specification’, we will first define Operational Semantics of Imp.
- The operational semantics formally define how the program state evolves during execution.
- A program state (σ, c) consists of two components:
    - $\sigma : V \to \mathbb{R}$ is a valuation of program variables
    - c is the rest of the program to be executed
- Let $\Sigma = (\mathbb{R}^{|V|} \cup \{Error\}) \times \mathscr{P}$ be the set of all states
    - $\mathscr{P}$ is the set of all Imp programs.
- A transition $(\sigma_1, \mathrm{c}_1) \hookrightarrow (\sigma_2, \mathrm{c}_2)$ denotes a step taken by the program

TRANSITIONS OF IMP

$$\frac{\sigma_2 = \sigma_1[x \mapsto \sigma_1(e)]}{(\sigma_1, \texttt{x := e}) \hookrightarrow (\sigma_2, \texttt{skip})}$$

TRANSITIONS OF IMP

$$\frac{\sigma_2 = \sigma_1[x \mapsto \sigma_1(e)]}{(\sigma_1, \texttt{x := e}) \hookrightarrow (\sigma_2, \texttt{skip})}$$

NOTATION ALERT:

$f = g[a \mapsto b]$ means:

- $f(a) = b$
- $\forall x \in dom(g)\,.\, x \neq a \rightarrow f(x) = g(x)$

TRANSITIONS OF IMP

$$\frac{\sigma_2 = \sigma_1[x \mapsto \sigma_1(e)]}{(\sigma_1, \texttt{x := e}) \hookrightarrow (\sigma_2, \texttt{skip})}$$

NOTATION ALERT:

For $e \in Exp(V)$ and $\sigma \in \mathbb{R}^{|V|}$, $\sigma(e)$ denotes the evaluation of e at σ using the standard interpretations of Arithmetic operators.

TRANSITIONS OF IMP

$$\frac{\sigma_2 = \sigma_1[x \mapsto \sigma_1(e)]}{(\sigma_1, \texttt{x := e}) \hookrightarrow (\sigma_2, \texttt{skip})}\ \text{[T-ASSIGN]}$$

TRANSITIONS OF IMP

$$\frac{\sigma_2 = \sigma_1[x \mapsto \sigma_1(e)]}{(\sigma_1, \texttt{x := e}) \hookrightarrow (\sigma_2, \texttt{skip})}$$

[T-ASSIGN]

$$\frac{\sigma_2 = \sigma_1[x \mapsto n] \quad n \in \mathbb{R}}{(\sigma_1, \texttt{x := havoc}) \hookrightarrow (\sigma_2, \texttt{skip})}$$

[T-HAVOC]

TRANSITIONS OF IMP

$$\frac{\sigma_2 = \sigma_1[x \mapsto \sigma_1(e)]}{(\sigma_1, \texttt{x := e}) \hookrightarrow (\sigma_2, \texttt{skip})}\ \text{[T-ASSIGN]}$$

$$\frac{\sigma_2 = \sigma_1[x \mapsto n] \quad n \in \mathbb{R}}{(\sigma_1, \texttt{x := havoc}) \hookrightarrow (\sigma_2, \texttt{skip})}\ \text{[T-HAVOC]}$$

$$\frac{???}{(\sigma_1, \texttt{assume(F)}) \hookrightarrow (\sigma_1, \texttt{skip})}\ \text{[T-ASSUME]}$$

TRANSITIONS OF IMP

$$\frac{\sigma_2 = \sigma_1[x \mapsto \sigma_1(e)]}{(\sigma_1, \texttt{x := e}) \hookrightarrow (\sigma_2, \texttt{skip})}$$

[T-ASSIGN]

$$\frac{\sigma_2 = \sigma_1[x \mapsto n] \quad n \in \mathbb{R}}{(\sigma_1, \texttt{x := havoc}) \hookrightarrow (\sigma_2, \texttt{skip})}$$

[T-HAVOC]

$$\frac{\sigma_1 \vDash F}{(\sigma_1, \texttt{assume(F)}) \hookrightarrow (\sigma_1, \texttt{skip})}$$

[T-ASSUME]

TRANSITIONS OF IMP

$$\frac{\sigma_2 = \sigma_1[x \mapsto \sigma_1(e)]}{(\sigma_1, \texttt{x := e}) \hookrightarrow (\sigma_2, \texttt{skip})}$$

[T-ASSIGN]

$$\frac{\sigma_2 = \sigma_1[x \mapsto n] \quad n \in \mathbb{R}}{(\sigma_1, \texttt{x := havoc}) \hookrightarrow (\sigma_2, \texttt{skip})}$$

[T-HAVOC]

$$\frac{\sigma_1 \vDash F}{(\sigma_1, \texttt{assume(F)}) \hookrightarrow (\sigma_1, \texttt{skip})}$$

[T-ASSUME]

$$\frac{\sigma_1 \vDash F}{(\sigma_1, \texttt{assert(F)}) \hookrightarrow (\sigma_1, \texttt{skip})}$$

[T-ASSERT-TRUE]

$$\frac{\sigma_1 \nvDash F}{(\sigma_1, \texttt{assert(F)}) \hookrightarrow (Error, \texttt{skip})}$$

[T-ASSERT-FALSE]

TRANSITIONS OF IMP

[T-SEQ-1]

$$\frac{(\sigma_1, \mathtt{c}_1) \hookrightarrow (\sigma_2, \mathtt{c}_1')}{(\sigma_1, \mathtt{c}_1; \mathtt{c}_2) \hookrightarrow (\sigma_2, \mathtt{c}_1'; \mathtt{c}_2)}$$

[T-SEQ-2]

$$\frac{}{(\sigma_1, \mathtt{skip}; \mathtt{c}_2) \hookrightarrow (\sigma_1, \mathtt{c}_2)}$$

TRANSITIONS OF IMP

[T-SEQ-1]

$$\frac{(\sigma_1, \texttt{c}_1) \hookrightarrow (\sigma_2, \texttt{c}_1')}{(\sigma_1, \texttt{c}_1; \texttt{c}_2) \hookrightarrow (\sigma_2, \texttt{c}_1'; \texttt{c}_2)}$$

[T-SEQ-2]

$$\frac{}{(\sigma_1, \texttt{skip}; \texttt{c}_2) \hookrightarrow (\sigma_1, \texttt{c}_2)}$$

[T-IF-TRUE]

$$\frac{\sigma_1 \models F}{(\sigma_1, \texttt{if(F) then c}_1 \texttt{ else c}_2) \hookrightarrow (\sigma_1, \texttt{c}_1)}$$

[T-IF-FALSE]

$$\frac{\sigma_1 \not\models F}{(\sigma_1, \texttt{if(F) then c}_1 \texttt{ else c}_2) \hookrightarrow (\sigma_1, \texttt{c}_2)}$$

TRANSITIONS OF IMP

[T-SEQ-1]

$$\frac{(\sigma_1, \mathtt{c}_1) \hookrightarrow (\sigma_2, \mathtt{c}_1')}{(\sigma_1, \mathtt{c}_1; \mathtt{c}_2) \hookrightarrow (\sigma_2, \mathtt{c}_1'; \mathtt{c}_2)}$$

[T-SEQ-2]

$$\frac{}{(\sigma_1, \mathtt{skip}; \mathtt{c}_2) \hookrightarrow (\sigma_1, \mathtt{c}_2)}$$

[T-IF-TRUE]

$$\frac{\sigma_1 \models F}{(\sigma_1, \mathtt{if(F)\ then}\ \mathtt{c}_1\ \mathtt{else}\ \mathtt{c}_2) \hookrightarrow (\sigma_1, \mathtt{c}_1)}$$

[T-IF-FALSE]

$$\frac{\sigma_1 \not\models F}{(\sigma_1, \mathtt{if(F)\ then}\ \mathtt{c}_1\ \mathtt{else}\ \mathtt{c}_2) \hookrightarrow (\sigma_1, \mathtt{c}_2)}$$

[T-WHILE-TRUE]

$$\frac{\sigma_1 \models F}{(\sigma_1, \mathtt{while(F)\ do\ c}) \hookrightarrow (\sigma_1, \mathtt{c}; \mathtt{while(F)\ do\ c})}$$

TRANSITIONS OF IMP

[T-SEQ-1]

$$\frac{(\sigma_1, \texttt{c}_1) \hookrightarrow (\sigma_2, \texttt{c}_1')}{(\sigma_1, \texttt{c}_1; \texttt{c}_2) \hookrightarrow (\sigma_2, \texttt{c}_1'; \texttt{c}_2)}$$

[T-SEQ-2]

$$\frac{}{(\sigma_1, \texttt{skip}; \texttt{c}_2) \hookrightarrow (\sigma_1, \texttt{c}_2)}$$

[T-IF-TRUE]

$$\frac{\sigma_1 \models F}{(\sigma_1, \texttt{if(F) then c}_1 \texttt{ else c}_2) \hookrightarrow (\sigma_1, \texttt{c}_1)}$$

[T-IF-FALSE]

$$\frac{\sigma_1 \not\models F}{(\sigma_1, \texttt{if(F) then c}_1 \texttt{ else c}_2) \hookrightarrow (\sigma_1, \texttt{c}_2)}$$

$$\frac{\sigma_1 \models F}{(\sigma_1, \texttt{while(F) do c}) \hookrightarrow (\sigma_1, \texttt{c}; \texttt{while(F) do c})}$$

[T-WHILE-TRUE]

$$\frac{\sigma_1 \not\models F}{(\sigma_1, \texttt{while(F) do c}) \hookrightarrow (\sigma_1, \texttt{skip})}$$

[T-WHILE-FALSE]

EXAMPLE

```
assume(i = 0 ∧ n ≥ 0);
while(i < n) do
  i := i + 1;
assert(i = n);
```

$(\{\mathtt{i} \mapsto 0, \mathtt{n} \mapsto 2\}, \texttt{assume(i=0} \wedge \texttt{n} \geq \texttt{0);}\ldots)$

$\hookrightarrow (\{\mathtt{i} \mapsto 0, \mathtt{n} \mapsto 2\}, \texttt{skip;}\ldots)$ [T-SEQ-1, T-ASSUME]

$\hookrightarrow (\{\mathtt{i} \mapsto 0, \mathtt{n} \mapsto 2\}, \texttt{while(i < n) do i:=i+1;}\ldots)$ [T-SEQ-2]

$\hookrightarrow (\{\mathtt{i} \mapsto 0, \mathtt{n} \mapsto 2\}, \texttt{i:=i+1;while(i < n) do i:=i+1;}\ldots)$ [T-WHILE-TRUE]

$\hookrightarrow (\{\mathtt{i} \mapsto 1, \mathtt{n} \mapsto 2\}, \texttt{while(i < n) do i:=i+1;}\ldots)$ [T-SEQ-1, T-ASSIGN, T-SEQ-2]

$\hookrightarrow (\{\mathtt{i} \mapsto 1, \mathtt{n} \mapsto 2\}, \texttt{i:=i+1;while(i < n) do i:=i+1;}\ldots)$ [T-WHILE-TRUE]

$\hookrightarrow (\{\mathtt{i} \mapsto 2, \mathtt{n} \mapsto 2\}, \texttt{while(i < n) do i:=i+1;}\ldots)$ [T-SEQ-1, T-ASSIGN, T-SEQ-2]

$\hookrightarrow (\{\mathtt{i} \mapsto 2, \mathtt{n} \mapsto 2\}, \texttt{assert(i=n);})$ [T-WHILE-FALSE,T-SEQ-2]

$\hookrightarrow (\{\mathtt{i} \mapsto 2, \mathtt{n} \mapsto 2\}, \texttt{skip;})$ [T-ASSERT-TRUE]

REACHABILITY AND VERIFICATION

- Let $\Delta \subseteq \Sigma \times \Sigma$ be the set of transitions ($\hookrightarrow$) defined in the previous slides.
  - Is Δ finite?
  - Is Δ defined for a specific program c or for any program?
  - Is Δ finite if restricted to a specific program c?
- Given a program c, a sequence of transitions $(\sigma_0, \mathrm{c}) \hookrightarrow (\sigma_1, \mathrm{c}_1)\ldots \hookrightarrow (\sigma_n, \mathrm{c}_n)$ is called an execution of c.
  - A program state σ is called reachable if there exists an execution $(\sigma_0, \mathrm{c}) \hookrightarrow \ldots \hookrightarrow (\sigma, \mathrm{c}_n)$ which ends in the state σ.
- Verification Problem: Is $(Error, \mathrm{c}')$ reachable for some c'?
  - Program c is called safe if the error state is not reachable.
  - What about the initial state?

EXAMPLE

```
assume(i = 0 ∧ n ≥ 0);
while(i < n) do
  i := i + 1;
assert(i = n);
```

- Is $(Error, c')$ reachable?

PRE/POST-CONDITIONS AND VERIFICATION

- Alternatively, we can express the Verification problem in terms of pre-conditions and post-conditions.
- A program c satisfies the specification $\{P\}\mathsf{c}\{Q\}$ if:
  - $\forall \sigma, \sigma' .\ \sigma \vDash P \land (\sigma, \mathsf{c}) \hookrightarrow^{*} (\sigma', \mathsf{skip}) \rightarrow \sigma' \vDash Q$
- $\{P\}\mathsf{c}\{Q\}$ is also called a 'Hoare Triple'.
  - If c satisfies the specification $\{P\}\mathsf{c}\{Q\}$, then we also say that the Hoare Triple $\{P\}\mathsf{c}\{Q\}$ is valid.

TOTAL CORRECTNESS

- Both ways of specifying the verification problem deal with Partial Correctness.
  - They only consider terminating executions. Non-terminating executions trivially satisfy both definitions.
- Total Correctness also requires all program executions to be of finite length.
- A program c satisfies the specification $[P]$c$[Q]$ if every execution which begins in a state satisfying P should terminate in a state satisfying Q.

TOTAL CORRECTNESS

- A program c satisfies the specification $[P]\mathsf{c}[Q]$ if every execution which begins in a state satisfying P should terminate in a state satisfying Q.
  - $\forall \sigma \,.\, \sigma \vDash P \to \exists n, \sigma' \,.\, (\sigma, \mathsf{c}) \hookrightarrow^{n} (\sigma', \mathsf{skip}) \land \sigma' \vDash Q$
- Is this correct?
  - This only says that for every state satisfying P, there is some execution which terminates in a state obeying Q.
  - However, IMP is non-deterministic (due to havoc).
  - Hence, we need to say that every execution beginning from a state satisfying P should terminate.

$$\boxed{\begin{aligned} &\forall \sigma \,.\, \exists n \,.\, \sigma \vDash P \to \neg(\exists m, \sigma' \,.\, m > n \land (\sigma, \mathsf{c}) \hookrightarrow^{m} (\sigma', \mathsf{c}')) \\ &\land \forall \sigma, \sigma' \,.\, \sigma \vDash P \land (\sigma, \mathsf{c}) \hookrightarrow^{*} (\sigma', \mathsf{skip}) \to \sigma' \vDash Q \end{aligned}}$$

TOTAL CORRECTNESS

```
[⊤]
i := havoc;
if (i = 5)
  while(i > 0) do
    i := i+1;
else
  skip;
[i ≠ 5]
```

- The above program satisfies the definition of total correctness as per $\forall \sigma \,.\, \sigma \vDash P \rightarrow \exists n, \sigma' \,.\, (\sigma, \mathrm{c}) \hookrightarrow^{n} (\sigma', \mathrm{skip}) \wedge \sigma' \vDash Q$.
- However, there are clearly non-terminating executions.
- It does not satisfy the following formula
  - $\forall \sigma \,.\, \exists n \,.\, \sigma \vDash P \rightarrow \neg(\exists m, \sigma' \,.\, m > n \wedge (\sigma, \mathrm{c}) \hookrightarrow^{m} (\sigma', \mathrm{c}'))$ $\wedge\ \forall \sigma, \sigma' \,.\, \sigma \vDash P \wedge (\sigma, \mathrm{c}) \hookrightarrow^{*} (\sigma', \mathrm{skip}) \rightarrow \sigma' \vDash Q$

EXAMPLES OF HOARE TRIPLES

- What can be said about the following triples?
  - $\{\ \top\ \}$ c $\{Q\}$
  - $\{\ \bot\ \}$ c $\{Q\}$
  - $\{P\}$ c $\{\ \top\ \}$
  - $\{\ \top\ \}$ c $\{\ \bot\ \}$
- Partial and total correctness
  - Is $\{x = 0\}$ while(x $\geq$ 0) do x:=x+1 $\{x = 1\}$ valid?
  - What about $[x = 0]$ while(x $\geq$ 0) do x:=x+1 $[x = 1]$?

AUTOMATED VERIFICATION

- We will reduce the verification problem to the satisfiability problem (modulo theories) in FOL.
- As a first step, we encode the operational semantics in FOL.
- If V is the set of variables used in a program c, then an FOL formula $F[V]$ encodes a set of states of the program.
  - E.g. If $V = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$, then the formula $\mathsf{x} + \mathsf{y} > 0$ encodes the set of states $\{(\mathsf{x} \mapsto m, \mathsf{y} \mapsto n, \mathsf{z} \mapsto o) \mid m + n > 0\}$

AUTOMATED VERIFICATION

If $(\sigma, \text{c}) \hookrightarrow (\sigma', \text{skip})$, then we will use the FOL formula $\rho(\mathbf{c})[\mathbf{V}, \mathbf{V}']$ to encode the states σ and σ'.

- If $(\sigma, \text{c}) \hookrightarrow (\sigma', \text{skip})$ then σ, σ' are models of $\rho(c)$.
- Example: $\rho(\text{x:=y+1}) \triangleq \text{x}' = \text{y} + 1 \wedge \text{y}' = \text{y}$
- We will use a special variable error $\in V$ to indicate the $Error$ state (obtained after assertion failure). error $= 0$ indicates a non-error state.

SEMANTICS IN FOL

For $U \subseteq V$, we define $\mathit{frame}(\mathrm{U})$ to be the formula $\bigwedge_{\mathrm{v} \in V \setminus U} \mathrm{v}' = \mathrm{v}$

- E.g. $V = \{x, y, z\}$, $\mathit{frame}(\mathrm{x}) \triangleq (\mathrm{y}' = \mathrm{y}) \wedge (\mathrm{z}' = \mathrm{z})$

Now, the semantics of commands in FOL can be defined as follows:

- ρ(x:=e) $\triangleq$
- ρ(x:=havoc) $\triangleq$
- ρ(assume(F)) $\triangleq$
- ρ(assert(F)) $\triangleq$

SEMANTICS IN FOL

For $U \subseteq V$, we define $frame$(U) to be the formula $\bigwedge_{v \in V \setminus U} v' = v$

- E.g. $V = \{x, y, z\}$, $frame(x) \triangleq (y' = y) \land (z' = z)$

Now, the semantics of commands in FOL can be defined as follows:

- $\rho(x{:=}e) \triangleq x' = e \land frame(x)$
- $\rho(x{:=}\text{havoc}) \triangleq frame(x)$
- $\rho(\text{assume}(F)) \triangleq F \land frame(\emptyset)$
- $\rho(\text{assert}(F)) \triangleq F \rightarrow frame(\emptyset)$

SEMANTICS IN FOL

- What is $\rho(c; c')$? How can we express it in terms of $\rho(c)$ and $\rho(c')$?
  - $\rho(c; c') = \rho(c)[V''/V'] \wedge \rho(c')[V''/V]$
- What is $\rho(\mathit{if}\ F\ \mathit{then}\ c\ \mathit{else}\ c')$?
  - $(F \wedge \rho(c)) \vee (\neg F \wedge \rho(c'))$