Program verification 2016

Lecture 2: Symbolic methods

Instructor: Ashutosh Gupta

TIFR, India

Compile date: 2016-02-08

Where are we and where are we going?

We have

- defined a simple language
- defined small step operation semantics of the language
- defined logical view of program statements
- defined strongest post and weakest pre
- defined logical strongest post and weakest pre

Where are we and where are we going?

We have

- defined a simple language
- defined small step operation semantics of the language
- defined logical view of program statements
- defined strongest post and weakest pre
- defined logical strongest post and weakest pre

We will

- Hoare logic
- labelled transition system
- we cover some methods that try/avoid to compute lfp

Topic 3.1

Hoare logic

Hoare logic - our first method of verification

Hoare logic - our first method of verification

- Computing a super set of the reachable states(lfp) that does not intersect with error states should be suffice for our goal

Hoare logic - our first method of verification

- Computing a super set of the reachable states(lfp) that does not intersect with error states should be suffice for our goal
- Since we do not know how to compute lfp, we will first see a method of writing pen-paper proofs of program safety

Hoare logic - our first method of verification

- Computing a super set of the reachable states(lfp) that does not intersect with error states should be suffice for our goal
- Since we do not know how to compute lfp, we will first see a method of writing pen-paper proofs of program safety
- Such a proof method has following steps
  - write (guess) a super set of reachable states
  - show it is actually a super set
  - show it does not intersect with error states

Hoare logic - our first method of verification

- Computing a super set of the reachable states(lfp) that does not intersect with error states should be suffice for our goal
- Since we do not know how to compute lfp, we will first see a method of writing pen-paper proofs of program safety
- Such a proof method has following steps
  - write (guess) a super set of reachable states
  - show it is actually a super set
  - show it does not intersect with error states
- First such method was proposed by Tony Hoare
  - it is sometimes called axiomatic semantics

Hoare Triple

Definition 3.1

$$\{P\}\mathsf{c}\{Q\}$$

- $P : \Sigma(V)$, *usually called precondition*
- $\mathsf{c} : \mathcal{P}$
- $Q : \Sigma(V)$, *usually called postcondition*

Hoare Triple

Definition 3.1

$$\{P\}\mathtt{c}\{Q\}$$

- $P : \Sigma(V)$, *usually called precondition*
- $\mathtt{c} : \mathcal{P}$
- $Q : \Sigma(V)$, *usually called postcondition*

Definition 3.2

$\{P\}\mathtt{c}\{Q\}$ *is valid if all the executions of* $\mathtt{c}$ *that start from* P *end in* Q*, i.e.,*

$$\forall v, v'.\ v \models P \wedge ((v, \mathtt{c}), (v', \mathtt{skip})) \in T^* \Rightarrow v' \models Q.$$

Hoare proof obligation/goal

The safety verification problem is slightly differently stated in Hoare logic.

Hoare proof obligation/goal

The safety verification problem is slightly differently stated in Hoare logic.

We remove assert statement from the language and no *err* variable.

Hoare proof obligation/goal

The safety verification problem is slightly differently stated in Hoare logic.

We remove assert statement from the language and no *err* variable.

Here, a verification problem is proving validity of a Hoare triple.

Hoare proof obligation/goal

The safety verification problem is slightly differently stated in Hoare logic.

We remove assert statement from the language and no *err* variable.

Here, a verification problem is proving validity of a Hoare triple.

Example 3.1

Program

$$
\begin{array}{l}
\texttt{assume}(\top) \\
\texttt{r} := 1; \\
\texttt{i} := 1; \\
\texttt{while}(\texttt{i} < 3) \\
\{ \\
\quad \texttt{r} := \texttt{r} + \texttt{z}; \\
\quad \texttt{i} := \texttt{i} + 1 \\
\} \\
\texttt{assert}(\texttt{r} = 2\texttt{z} + 1)
\end{array}
$$

Hoare proof obligation/goal

The safety verification problem is slightly differently stated in Hoare logic.

We remove assert statement from the language and no *err* variable.

Here, a verification problem is proving validity of a Hoare triple.

Example 3.1

Program

```
assume(⊤)
r := 1;
i := 1;
while(i < 3)
{
  r := r + z;
  i := i + 1
}
assert(r = 2z + 1)
```

→

Hoare triple

$\{\top\}$
r := 1;
i := 1;
while(i < 3)
{
  r := r + z;
  i := i + 1
}
$\{r = 2z + 1\}$

Hoare Proof System

$$\frac{}{\{P\}\texttt{skip}\{P\}}(\text{skip rule})$$

Hoare Proof System

$$\frac{}{\{P\}\texttt{skip}\{P\}}(\text{skip rule})$$

$$\frac{}{\{P[exp/x]\}\texttt{x} := exp\{P\}}(\text{assign rule})$$

Hoare Proof System

$$\frac{}{\{P\}\texttt{skip}\{P\}}\text{(skip rule)}$$

$$\frac{}{\{P[exp/x]\}\texttt{x} := exp\{P\}}\text{(assign rule)}$$

$$\frac{}{\{\forall x.P\}\texttt{x} := havoc()\{P\}}\text{(havoc rule)}$$

Hoare Proof System

$$\frac{}{\{P\}\texttt{skip}\{P\}}\text{(skip rule)}$$

$$\frac{}{\{P[exp/x]\}\texttt{x} := exp\{P\}}\text{(assign rule)}$$

$$\frac{}{\{\forall x.P\}\texttt{x} := havoc()\{P\}}\text{(havoc rule)}$$

$$\frac{}{\{P\}\texttt{assume(F)}\{\texttt{F} \wedge P\}}\text{(assume rule)}$$

Hoare Proof System

$$\frac{}{\{P\}\texttt{skip}\{P\}}\text{(skip rule)}$$

$$\frac{}{\{P[exp/x]\}\texttt{x} := exp\{P\}}\text{(assign rule)}$$

$$\frac{}{\{\forall x.P\}\texttt{x} := havoc()\{P\}}\text{(havoc rule)}$$

$$\frac{}{\{P\}\texttt{assume(F)}\{\texttt{F} \wedge P\}}\text{(assume rule)}$$

We may freely choose any of sp and wp for pre/post pairs for data statements.

Hoare Proof System (contd.)

$$\frac{\{P\}c_1\{Q\} \quad \{Q\}c_2\{R\}}{\{P\}c_1; c_2\{R\}}(\text{composition rule})$$

Hoare Proof System (contd.)

$$\frac{\{P\}c_1\{Q\} \quad \{Q\}c_2\{R\}}{\{P\}c_1; c_2\{R\}}(\text{composition rule})$$

$$\frac{\{P\}c_1\{Q\} \quad \{P\}c_2\{Q\}}{\{P\}c_1[]c_2\{Q\}}(\text{nondet rule})$$

Hoare Proof System (contd.)

$$\frac{\{P\}\mathtt{c}_1\{Q\} \quad \{Q\}\mathtt{c}_2\{R\}}{\{P\}\mathtt{c}_1;\mathtt{c}_2\{R\}}\text{(composition rule)}$$

$$\frac{\{P\}\mathtt{c}_1\{Q\} \quad \{P\}\mathtt{c}_2\{Q\}}{\{P\}\mathtt{c}_1[]\mathtt{c}_2\{Q\}}\text{(nondet rule)}$$

$$\frac{\{\mathtt{F} \wedge P\}\mathtt{c}_1\{Q\} \quad \{\neg\mathtt{F} \wedge P\}\mathtt{c}_2\{Q\}}{\{P\}\mathtt{if(F)\ c}_1\ \mathtt{else\ c}_2\{Q\}}\text{(if rule)}$$

Hoare Proof System (contd.)

$$\frac{\{P\}\mathtt{c}_1\{Q\} \quad \{Q\}\mathtt{c}_2\{R\}}{\{P\}\mathtt{c}_1;\mathtt{c}_2\{R\}}\text{(composition rule)}$$

$$\frac{\{P\}\mathtt{c}_1\{Q\} \quad \{P\}\mathtt{c}_2\{Q\}}{\{P\}\mathtt{c}_1[]\mathtt{c}_2\{Q\}}\text{(nondet rule)}$$

$$\frac{\{\mathtt{F} \wedge P\}\mathtt{c}_1\{Q\} \quad \{\neg\mathtt{F} \wedge P\}\mathtt{c}_2\{Q\}}{\{P\}\mathtt{if(F)\ c}_1\ \mathtt{else\ c}_2\{Q\}}\text{(if rule)}$$

$$\frac{P_1 \Rightarrow P_2 \quad \{P_2\}\mathtt{c}\{Q_2\} \quad Q_2 \Rightarrow Q_1}{\{P_1\}\mathtt{c}\{Q_1\}}\text{(Consequence rule)}$$

Hoare Proof System (contd.)

$$\frac{\{P\}c_1\{Q\} \quad \{Q\}c_2\{R\}}{\{P\}c_1; c_2\{R\}} \text{(composition rule)}$$

$$\frac{\{P\}c_1\{Q\} \quad \{P\}c_2\{Q\}}{\{P\}c_1[]c_2\{Q\}} \text{(nondet rule)}$$

$$\frac{\{\mathtt{F} \wedge P\}c_1\{Q\} \quad \{\neg \mathtt{F} \wedge P\}c_2\{Q\}}{\{P\}\mathtt{if(F)}\ c_1\ \mathtt{else}\ c_2\{Q\}} \text{(if rule)}$$

$$\frac{P_1 \Rightarrow P_2 \quad \{P_2\}c\{Q_2\} \quad Q_2 \Rightarrow Q_1}{\{P_1\}c\{Q_1\}} \text{(Consequence rule)}$$

$$\frac{\{\mathtt{I} \wedge \mathtt{F}\}c\{\mathtt{I}\}}{\{\mathtt{I}\}\mathtt{while(F)}\ c\{\neg F \wedge \mathtt{I}\}} \text{(While rule)}$$

Hoare Proof System (contd.)

$$\frac{\{P\}\mathtt{c}_1\{Q\} \quad \{Q\}\mathtt{c}_2\{R\}}{\{P\}\mathtt{c}_1;\mathtt{c}_2\{R\}}\text{(composition rule)}$$

$$\frac{\{P\}\mathtt{c}_1\{Q\} \quad \{P\}\mathtt{c}_2\{Q\}}{\{P\}\mathtt{c}_1[]\mathtt{c}_2\{Q\}}\text{(nondet rule)}$$

$$\frac{\{\mathtt{F} \wedge P\}\mathtt{c}_1\{Q\} \quad \{\neg\mathtt{F} \wedge P\}\mathtt{c}_2\{Q\}}{\{P\}\mathtt{if(F)\ c}_1\ \mathtt{else\ c}_2\{Q\}}\text{(if rule)}$$

$$\frac{P_1 \Rightarrow P_2 \quad \{P_2\}\mathtt{c}\{Q_2\} \quad Q_2 \Rightarrow Q_1}{\{P_1\}\mathtt{c}\{Q_1\}}\text{(Consequence rule)}$$

$$\frac{\{\mathtt{I} \wedge \mathtt{F}\}\mathtt{c}\{\mathtt{I}\}}{\{\mathtt{I}\}\mathtt{while(F)\ c}\{\neg F \wedge \mathtt{I}\}}\text{(While rule)}$$

Non-mechanical step: invent `I` such that the while rule holds. `I` is called loop-invariant.

Example Hoare proof

Example 3.2

$\{\top\}$
$\mathtt{r} := 1;$

$\mathtt{i} := 1;$

$\mathtt{while}(\mathtt{i} < 3)$
$\{$

$\quad \mathtt{r} := \mathtt{r} + \mathtt{z}$

$\quad \mathtt{i} := \mathtt{i} + 1$
$\}$
$\{r = 2z + 1\}$

Example Hoare proof

Example 3.2

$\{\top\}$
$\texttt{r} := 1;$

$\texttt{i} := 1;$

$\texttt{while}(\texttt{i} < 3)$
$\{$

$\quad \texttt{r} := \texttt{r} + \texttt{z}$

$\quad \texttt{i} := \texttt{i} + 1$
$\}$
$\{r = 2z + 1\}$

$$\{\top\}\texttt{r} := 1; ..; \texttt{while}(..)..\{r = 2z + 1\}$$

Example Hoare proof

Example 3.2

$\{\top\}$
$\texttt{r} := 1;$

$\texttt{i} := 1;$
$\{\texttt{I}\}$
$\texttt{while}(\texttt{i} < 3)$
$\{$

$\quad \texttt{r} := \texttt{r} + \texttt{z}$

$\quad \texttt{i} := \texttt{i} + 1$
$\}$
$\{r = 2z + 1\}$

$$\frac{\{\top\}\texttt{r} := 1; ..; \texttt{while}(..)..\{\texttt{I} \wedge \texttt{i} \geq 3\} \qquad \texttt{I} \wedge \texttt{i} \geq 3 \Rightarrow r = 2z + 1}{\{\top\}\texttt{r} := 1; ..; \texttt{while}(..)..\{r = 2z + 1\}}$$

Example Hoare proof

Example 3.2

$\{\top\}$
$\texttt{r} := 1;$

$\texttt{i} := 1;$
$\{\texttt{I}\}$
$\texttt{while}(\texttt{i} < 3)$
$\{$
$\quad \{\texttt{I} \wedge \texttt{i} < 3\}$
$\quad \texttt{r} := \texttt{r} + \texttt{z}$

$\quad \texttt{i} := \texttt{i} + 1$
$\}$
$\{r = 2z + 1\}$

$$\frac{\{\top\}\texttt{r} := 1; \texttt{i} := 1; \{\texttt{I}\} \quad \{i < 3 \wedge \texttt{I}\}\texttt{r} := \texttt{r} + \texttt{z}; \texttt{i} := \texttt{i} + 1\{\texttt{I}\}}{\{\top\}\texttt{r} := 1; ..; \texttt{while}(..)..\{\texttt{I} \wedge \texttt{i} \geq 3\} \qquad \texttt{I} \wedge \texttt{i} \geq 3 \Rightarrow r = 2z + 1}$$
$$\{\top\}\texttt{r} := 1; ..; \texttt{while}(..)..\{r = 2z + 1\}$$

Example Hoare proof

Example 3.2

Consider loop invariant: $\mathtt{I} = (i \leq 3 \wedge r = (i-1)z + 1)$

$\{\top\}$
$\mathtt{r} := 1;$

$\mathtt{i} := 1;$
$\{\mathtt{I}\}$
$\mathtt{while}(\mathtt{i} < 3)$
$\{$
  $\{\mathtt{I} \wedge \mathtt{i} < 3\}$
  $\mathtt{r} := \mathtt{r} + \mathtt{z}$

  $\mathtt{i} := \mathtt{i} + 1$
$\}$
$\{r = 2z + 1\}$

$$\frac{\{\top\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\} \quad \{i < 3 \wedge \mathtt{I}\}\mathtt{r} := \mathtt{r} + \mathtt{z}; \mathtt{i} := \mathtt{i} + 1\{\mathtt{I}\}}{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{\mathtt{I} \wedge \mathtt{i} \geq 3\} \qquad \mathtt{I} \wedge \mathtt{i} \geq 3 \Rightarrow r = 2z + 1}$$
$$\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{r = 2z + 1\}$$

Example Hoare proof

Example 3.2

Consider loop invariant: $\mathtt{I} = (i \leq 3 \wedge r = (i-1)z + 1)$

$\{\top\}$
$\mathtt{r} := 1;$

$\mathtt{i} := 1;$
$\{\mathtt{I}\}$
$\mathtt{while}(\mathtt{i} < 3)$
$\{$
  $\{\mathtt{I} \wedge \mathtt{i} < 3\}$
  $\mathtt{r} := \mathtt{r} + \mathtt{z}$

  $\mathtt{i} := \mathtt{i} + 1$
$\}$
$\{r = 2z + 1\}$

$$\frac{}{\{\mathtt{r} = 1\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\}}$$

$$\frac{\dfrac{\{\top\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\} \quad \{i < 3 \wedge \mathtt{I}\}\mathtt{r} := \mathtt{r} + \mathtt{z}; \mathtt{i} := \mathtt{i} + 1\{\mathtt{I}\}}{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{\mathtt{I} \wedge \mathtt{i} \geq 3\}} \qquad \mathtt{I} \wedge \mathtt{i} \geq 3 \Rightarrow r = 2z + 1}{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{r = 2z + 1\}}$$

Example Hoare proof

Example 3.2

Consider loop invariant: $\mathtt{I} = (i \leq 3 \wedge r = (i-1)z + 1)$

$\{\top\}$
$\mathtt{r} := 1;$
$\{\mathtt{r} = 1\}$
$\mathtt{i} := 1;$
$\{\mathtt{I}\}$
$\mathtt{while}(\mathtt{i} < 3)$
$\{$
  $\{\mathtt{I} \wedge \mathtt{i} < 3\}$
  $\mathtt{r} := \mathtt{r} + \mathtt{z}$

  $\mathtt{i} := \mathtt{i} + 1$
$\}$
$\{r = 2z + 1\}$

$$\frac{\dfrac{}{\{\mathtt{r}=1\}\mathtt{i}:=1\{\mathtt{I}\}}}{\{\mathtt{r}=1\}\mathtt{r}:=1;\mathtt{i}:=1;\{\mathtt{I}\}}$$

$$\frac{\{\top\}\mathtt{r}:=1;\mathtt{i}:=1;\{\mathtt{I}\} \quad \{i<3\wedge\mathtt{I}\}\mathtt{r}:=\mathtt{r}+\mathtt{z};\mathtt{i}:=\mathtt{i}+1\{\mathtt{I}\}}{\{\top\}\mathtt{r}:=1;..;\mathtt{while}(..)..\{\mathtt{I}\wedge\mathtt{i}\geq 3\} \qquad \mathtt{I}\wedge\mathtt{i}\geq 3 \Rightarrow r=2z+1}$$
$$\{\top\}\mathtt{r}:=1;..;\mathtt{while}(..)..\{r=2z+1\}$$

Example Hoare proof

Example 3.2

Consider loop invariant: $\mathtt{I} = (i \leq 3 \wedge r = (i-1)z + 1)$

$\{\top\}$
$\mathtt{r} := 1;$
$\{\mathtt{r} = 1\}$
$\mathtt{i} := 1;$
$\{\mathtt{I}\}$
$\mathtt{while}(\mathtt{i} < 3)$
$\{$
  $\{\mathtt{I} \wedge \mathtt{i} < 3\}$
  $\mathtt{r} := \mathtt{r} + \mathtt{z}$

  $\mathtt{i} := \mathtt{i} + 1$
$\}$
$\{r = 2z + 1\}$

$$\frac{\overline{\{\top\}\mathtt{r} := 1\{\mathtt{r} = 1\}} \quad \overline{\{\mathtt{r} = 1\}\mathtt{i} := 1\{\mathtt{I}\}}}{\{\mathtt{r} = 1\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\}}$$

$$\frac{\dfrac{\{\top\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\} \quad \{i < 3 \wedge \mathtt{I}\}\mathtt{r} := \mathtt{r} + \mathtt{z}; \mathtt{i} := \mathtt{i} + 1\{\mathtt{I}\}}{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{\mathtt{I} \wedge \mathtt{i} \geq 3\}} \qquad \mathtt{I} \wedge \mathtt{i} \geq 3 \Rightarrow r = 2z + 1}{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{r = 2z + 1\}}$$

Example Hoare proof

Example 3.2

Consider loop invariant: $\mathtt{I} = (i \leq 3 \wedge r = (i-1)z + 1)$

$\{\top\}$
$\mathtt{r} := 1;$
$\{\mathtt{r} = 1\}$
$\mathtt{i} := 1;$
$\{\mathtt{I}\}$
$\mathtt{while}(\mathtt{i} < 3)$
$\{$
  $\{\mathtt{I} \wedge \mathtt{i} < 3\}$
  $\mathtt{r} := \mathtt{r} + \mathtt{z}$

  $\mathtt{i} := \mathtt{i} + 1$
$\}$
$\{r = 2z + 1\}$

$$\frac{\overline{\{\top\}\mathtt{r} := 1\{\mathtt{r} = 1\}} \quad \overline{\{\mathtt{r} = 1\}\mathtt{i} := 1\{\mathtt{I}\}}}{\{\mathtt{r} = 1\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\}}$$

$$\frac{}{\{i < 3 \wedge \mathtt{I}\}\mathtt{r} := \mathtt{r} + \mathtt{z}; \mathtt{i} := \mathtt{i} + 1\{\mathtt{I}\}}$$

$$\frac{\{\top\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\} \quad \{i < 3 \wedge \mathtt{I}\}\mathtt{r} := \mathtt{r} + \mathtt{z}; \mathtt{i} := \mathtt{i} + 1\{\mathtt{I}\}}{\dfrac{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{\mathtt{I} \wedge \mathtt{i} \geq 3\} \qquad \mathtt{I} \wedge \mathtt{i} \geq 3 \Rightarrow r = 2z + 1}{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{r = 2z + 1\}}}$$

Example Hoare proof

Example 3.2

Consider loop invariant: $\mathtt{I} = (i \leq 3 \wedge r = (i-1)z + 1)$

$\{\top\}$
$\mathtt{r} := 1;$
$\{\mathtt{r} = 1\}$
$\mathtt{i} := 1;$
$\{\mathtt{I}\}$
$\mathtt{while}(\mathtt{i} < 3)$
$\{$
  $\{\mathtt{I} \wedge \mathtt{i} < 3\}$
  $\mathtt{r} := \mathtt{r} + \mathtt{z}$
  $\{P_5\}$
  $\mathtt{i} := \mathtt{i} + 1$
$\}$
$\{r = 2z + 1\}$

$$\frac{\dfrac{}{\{\top\}\mathtt{r} := 1\{\mathtt{r} = 1\}} \quad \dfrac{}{\{\mathtt{r} = 1\}\mathtt{i} := 1\{\mathtt{I}\}}}{\{\mathtt{r} = 1\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\}}$$

$$\frac{\dfrac{}{\{i < 3 \wedge r = iz + 1\}\ \mathtt{i} := \mathtt{i} + 1\ \{\mathtt{I}\}}}{\{i < 3 \wedge \mathtt{I}\}\mathtt{r} := \mathtt{r} + \mathtt{z}; \mathtt{i} := \mathtt{i} + 1\{\mathtt{I}\}}$$

$$\frac{\dfrac{\{\top\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\} \quad \{i < 3 \wedge \mathtt{I}\}\mathtt{r} := \mathtt{r} + \mathtt{z}; \mathtt{i} := \mathtt{i} + 1\{\mathtt{I}\}}{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{\mathtt{I} \wedge \mathtt{i} \geq 3\}} \quad \mathtt{I} \wedge \mathtt{i} \geq 3 \Rightarrow r = 2z + 1}{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{r = 2z + 1\}}$$

Example Hoare proof

Example 3.2

Consider loop invariant: $\mathtt{I} = (i \leq 3 \wedge r = (i-1)z + 1)$

$\{\top\}$
$\mathtt{r} := 1;$
$\{\mathtt{r} = 1\}$
$\mathtt{i} := 1;$
$\{\mathtt{I}\}$
$\mathtt{while}(\mathtt{i} < 3)$
$\{$
  $\{\mathtt{I} \wedge \mathtt{i} < 3\}$
  $\mathtt{r} := \mathtt{r} + \mathtt{z}$
  $\{P_5\}$
  $\mathtt{i} := \mathtt{i} + 1$
$\}$
$\{r = 2z + 1\}$

$$\frac{\dfrac{}{\{\top\}\mathtt{r} := 1\{\mathtt{r} = 1\}} \quad \dfrac{}{\{\mathtt{r} = 1\}\mathtt{i} := 1\{\mathtt{I}\}}}{\{\mathtt{r} = 1\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\}}$$

$$\frac{\dfrac{}{\{i < 3 \wedge \mathtt{I}\}\ \mathtt{r} := \mathtt{r} + \mathtt{z}\ \{i < 3 \wedge r = iz + 1\}} \quad \dfrac{}{\{i < 3 \wedge r = iz + 1\}\ \mathtt{i} := \mathtt{i} + 1\ \{\mathtt{I}\}}}{\{i < 3 \wedge \mathtt{I}\}\mathtt{r} := \mathtt{r} + \mathtt{z}; \mathtt{i} := \mathtt{i} + 1\{\mathtt{I}\}}$$

$$\frac{\dfrac{\{\top\}\mathtt{r} := 1; \mathtt{i} := 1; \{\mathtt{I}\} \quad \{i < 3 \wedge \mathtt{I}\}\mathtt{r} := \mathtt{r} + \mathtt{z}; \mathtt{i} := \mathtt{i} + 1\{\mathtt{I}\}}{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{\mathtt{I} \wedge \mathtt{i} \geq 3\}} \quad \mathtt{I} \wedge \mathtt{i} \geq 3 \Rightarrow r = 2z + 1}{\{\top\}\mathtt{r} := 1; ..; \mathtt{while}(..)..\{r = 2z + 1\}}$$

Topic 3.2

Program as labeled transition system

A more convenient program model

- Simple language has many cases to write an algorithm

A more convenient program model

- Simple language has many cases to write an algorithm
- automata like program models allow more succinct description of verification methods

Program as labeled transition system (LTS)

Definition 3.3

A program P is a tuple $(V, L, \ell_0, \ell_e, E)$, where

- *V is a vector of variables,*
- *L be set of program locations,*
- *ℓ_0 is initial location,*
- *ℓ_e is error location, and*
- *$E \subseteq L \times \Sigma(V, V') \times L$ is a set of labeled transitions between locations.*

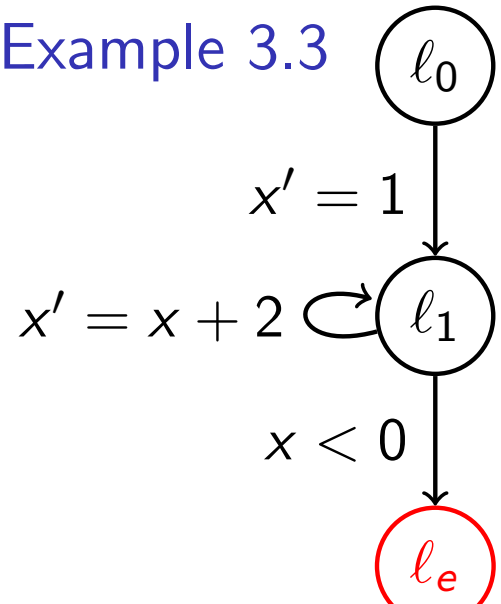
Program as labeled transition system (LTS)

Definition 3.3

A program P is a tuple $(V, L, \ell_0, \ell_e, E)$, where

- *V is a vector of variables,*
- *L be set of program locations,*
- *ℓ_0 is initial location,*
- *ℓ_e is error location, and*
- *$E \subseteq L \times \Sigma(V, V') \times L$ is a set of labeled transitions between locations.*

Example 3.3



$$V = [\mathtt{x}]$$
$$L = \{\ell_0, \ell_1, \ell_e\}$$
$$E = \{(\ell_0, \mathtt{x}' = 1, \ell_1),$$
$$(\ell_1, \mathtt{x}' = \mathtt{x} + 2, \ell_1),$$
$$(\ell_1, \mathtt{x} < 0, \ell_e)\}$$

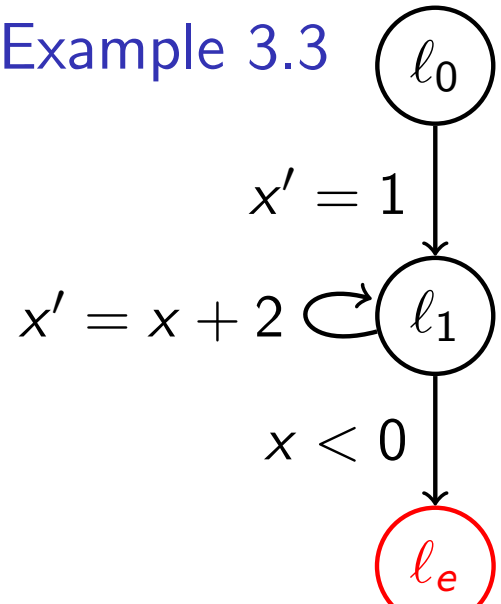
Program as labeled transition system (LTS)

Definition 3.3

A program P is a tuple $(V, L, \ell_0, \ell_e, E)$, where

- *V is a vector of variables,*
- *L be set of program locations,*
- *ℓ_0 is initial location,*
- *ℓ_e is error location, and*
- *$E \subseteq L \times \Sigma(V, V') \times L$ is a set of labeled transitions between locations.*

Example 3.3



$V = [\mathtt{x}]$

$L = \{\ell_0, \ell_1, \ell_e\}$

$E = \{(\ell_0, \mathtt{x}' = 1, \ell_1),$
$(\ell_1, \mathtt{x}' = \mathtt{x} + 2, \ell_1),$
$(\ell_1, \mathtt{x} < 0, \ell_e)\}$

Notation:

If $e = (\ell, \rho(V, V'), \ell') \in E$, then

$e(V, V') \triangleq \rho(V, V')$,

$e(loc) \triangleq \ell$ and

$e(loc') \triangleq \ell'$

Guarded command

Definition 3.4 (Guarded command)

A guarded command is a pair of a formula in $\Sigma(V)$ and a sequence of update constraints (including havoc) of variables in V.

Guarded command

Definition 3.4 (Guarded command)

A guarded command is a pair of a formula in $\Sigma(V)$ and a sequence of update constraints (including havoc) of variables in V.

Note: we may write transition formulas as guarded commands. Havoc encodes inputs.

Guarded command

Definition 3.4 (Guarded command)

A guarded command is a pair of a formula in $\Sigma(V)$ and a sequence of update constraints (including havoc) of variables in V.

Note: we may write transition formulas as guarded commands. Havoc encodes inputs.

Example 3.4

Consider $V = [\mathtt{x}, \mathtt{y}]$. The formula represented by the guarded command $(\mathtt{x} > \mathtt{y}, [\mathtt{x} := \mathtt{x} + 1])$ is $\mathtt{x} > \mathtt{y} \wedge \mathtt{x}' = \mathtt{x} + 1 \wedge \mathtt{y}' = \mathtt{y}$.

Guarded command

Definition 3.4 (Guarded command)

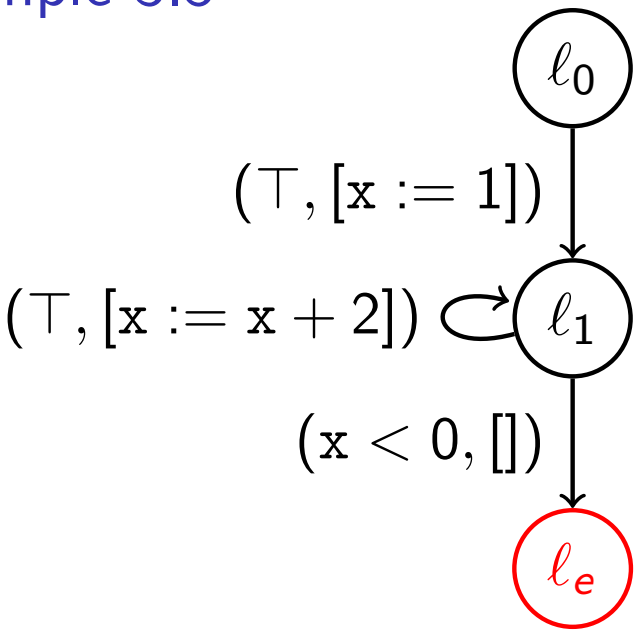
A guarded command is a pair of a formula in $\Sigma(V)$ and a sequence of update constraints (including havoc) of variables in V.

Note: we may write transition formulas as guarded commands. Havoc encodes inputs.

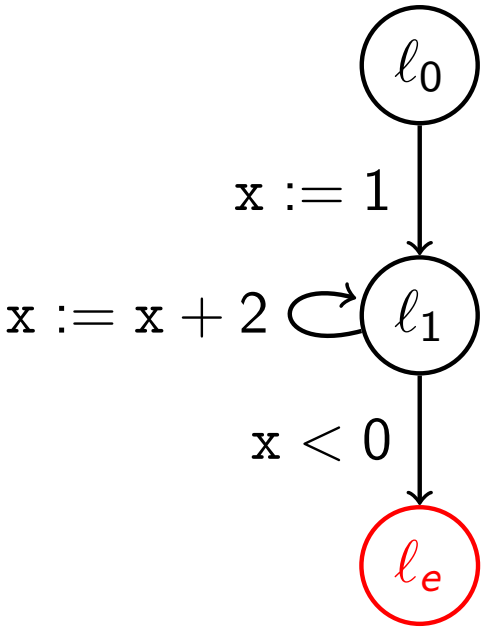
Example 3.4

Consider $V = [\mathrm{x}, \mathrm{y}]$. The formula represented by the guarded command $(\mathrm{x} > \mathrm{y}, [\mathrm{x} := \mathrm{x} + 1])$ is $\mathrm{x} > \mathrm{y} \wedge \mathrm{x}' = \mathrm{x} + 1 \wedge \mathrm{y}' = \mathrm{y}$.

Example 3.5



simplified view $\rightarrow$



Semantics

Consider program $P = (V, L, \ell_0, \ell_e, E)$.

Definition 3.5

A state $s = (\ell, v)$ of a program is program location ℓ and a valuation v of V.

Semantics

Consider program $P = (V, L, \ell_0, \ell_e, E)$.

Definition 3.5

A state $s = (\ell, v)$ of a program is program location ℓ and a valuation v of V.

Let $v(x) \triangleq$ value of variable x in v
For state $s = (\ell, v)$, let $s(x) \triangleq v(x)$ and $s(loc) \triangleq \ell$

Semantics

Consider program $P = (V, L, \ell_0, \ell_e, E)$.

Definition 3.5

A state $s = (\ell, v)$ of a program is program location ℓ and a valuation v of V.

Let $v(x) \triangleq$ value of variable x in v
For state $s = (\ell, v)$, let $s(x) \triangleq v(x)$ and $s(loc) \triangleq \ell$

Definition 3.6

A path $\pi = e_1, \ldots, e_n$ in P is a sequence of transitions such that, for each $0 < i < n$, $e_i = (\ell_{i-1}, \_, \ell_i)$ and $e_{i+1} = (\ell_i, \_, \ell_{i+1})$.

Semantics

Consider program $P = (V, L, \ell_0, \ell_e, E)$.

Definition 3.5

A state $s = (\ell, v)$ of a program is program location ℓ and a valuation v of V.

Let $v(x) \triangleq$ value of variable x in v
For state $s = (\ell, v)$, let $s(x) \triangleq v(x)$ and $s(loc) \triangleq \ell$

Definition 3.6

A path $\pi = e_1, \ldots, e_n$ in P is a sequence of transitions such that, for each $0 < i < n$, $e_i = (\ell_{i-1}, \_, \ell_i)$ and $e_{i+1} = (\ell_i, \_, \ell_{i+1})$.

Definition 3.7

An execution corresponding to path $e_1, \ldots, e_n$ is a sequence of states $(\ell_0, v_0), \ldots, (\ell_n, v_n)$ such that $\forall i \in 1..n$, $e_i(v_{i-1}, v_i)$ holds true.
An execution belongs to P if there is a corresponding path in P.

Semantics

Consider program $P = (V, L, \ell_0, \ell_e, E)$.

Definition 3.5

A state $s = (\ell, v)$ of a program is program location ℓ and a valuation v of V.

Let $v(x) \triangleq$ value of variable x in v
For state $s = (\ell, v)$, let $s(x) \triangleq v(x)$ and $s(loc) \triangleq \ell$

Definition 3.6

A path $\pi = e_1, \ldots, e_n$ in P is a sequence of transitions such that, for each $0 < i < n$, $e_i = (\ell_{i-1}, \_, \ell_i)$ and $e_{i+1} = (\ell_i, \_, \ell_{i+1})$.

Definition 3.7

An execution corresponding to path $e_1, \ldots, e_n$ is a sequence of states $(\ell_0, v_0), \ldots, (\ell_n, v_n)$ such that $\forall i \in 1..n$, $e_i(v_{i-1}, v_i)$ holds true.
An execution belongs to P if there is a corresponding path in P.

Definition 3.8

P is safe if there is no execution of P from ℓ_0 to ℓ_e.

Path constraints

$V_i \triangleq$ variable vector obtained by adding subscript i after each variable in V.

Path constraints

$V_i \triangleq$ variable vector obtained by adding subscript i after each variable in V.

Definition 3.9

For a path $\pi e_1, \ldots, e_n$, path constraints is $\bigwedge_{i \in 1..n} e_i(V_{i-1}, V_i)$.
A path is feasible if corresponding path constraints is satisfiable.

Path constraints

$V_i \triangleq$ variable vector obtained by adding subscript i after each variable in V.

Definition 3.9

For a path $\pi e_1, \ldots, e_n$, path constraints is $\bigwedge_{i \in 1..n} e_i(V_{i-1}, V_i)$.
A path is feasible if corresponding path constraints is satisfiable.

Let $\textsc{PathCons}(\pi)$ returns path constraints of π.

Path constraints

$V_i \triangleq$ variable vector obtained by adding subscript i after each variable in V.

Definition 3.9

For a path $\pi e_1, \ldots, e_n$, path constraints is $\bigwedge_{i \in 1..n} e_i(V_{i-1}, V_i)$.
A path is feasible if corresponding path constraints is satisfiable.

Let $\textsc{PathCons}(\pi)$ returns path constraints of π.

Path constraints are also known as "SSA formulas"

Path constraints

$V_i \triangleq$ variable vector obtained by adding subscript i after each variable in V.

Definition 3.9

For a path $\pi e_1, \ldots, e_n$, path constraints is $\bigwedge_{i \in 1..n} e_i(V_{i-1}, V_i)$.
A path is feasible if corresponding path constraints is satisfiable.

Let $\textsc{PathCons}(\pi)$ returns path constraints of π.

Path constraints are also known as "SSA formulas"

Theorem 3.1

A path is feasible then there is an execution that corresponds to the path.

Path constraints

$V_i \triangleq$ variable vector obtained by adding subscript i after each variable in V.

Definition 3.9

For a path $\pi e_1, \ldots, e_n$, path constraints is $\bigwedge_{i \in 1..n} e_i(V_{i-1}, V_i)$.
A path is feasible if corresponding path constraints is satisfiable.

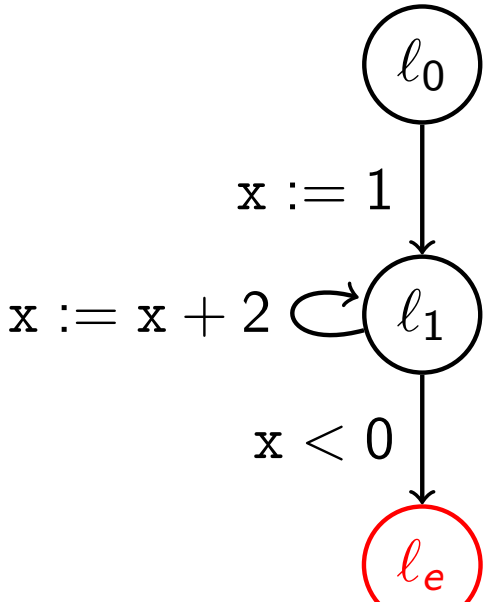
Let $\textsc{PathCons}(\pi)$ returns path constraints of π.

Path constraints are also known as "SSA formulas"

Theorem 3.1

A path is feasible then there is an execution that corresponds to the path.

Example 3.6



Consider path
$(\ell_0, \mathtt{x} := 1, \ell_1)$, $(\ell_1, \mathtt{x} := \mathtt{x} + 2, \ell_1)$, $(\ell_1, \mathtt{x} < 0, \ell_e)$

Path constraints

$V_i \triangleq$ variable vector obtained by adding subscript i after each variable in V.

Definition 3.9

For a path $\pi e_1, \ldots, e_n$, path constraints is $\bigwedge_{i \in 1..n} e_i(V_{i-1}, V_i)$.
A path is feasible if corresponding path constraints is satisfiable.

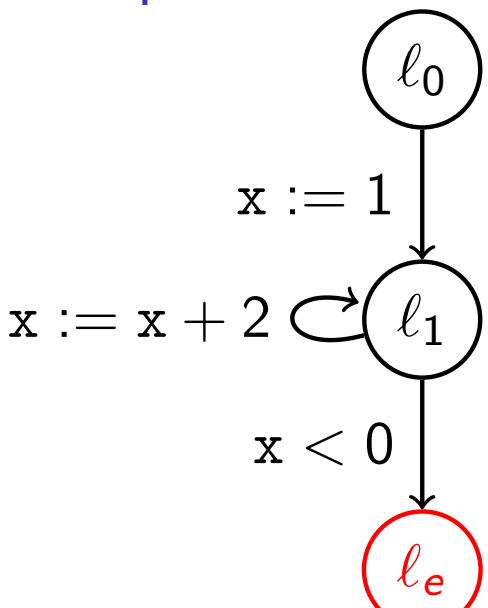
Let $\textsc{PathCons}(\pi)$ returns path constraints of π.

Path constraints are also known as "SSA formulas"

Theorem 3.1

A path is feasible then there is an execution that corresponds to the path.

Example 3.6



Consider path
$(\ell_0, \mathtt{x} := 1, \ell_1)$, $(\ell_1, \mathtt{x} := \mathtt{x} + 2, \ell_1)$, $(\ell_1, \mathtt{x} < 0, \ell_e)$

Path constraint for the path is
$F = (\mathtt{x}_1 = 1 \wedge \mathtt{x}_2 = \mathtt{x}_1 + 2 \wedge \mathtt{x}_2 < 0)$

Path constraints

$V_i \triangleq$ variable vector obtained by adding subscript i after each variable in V.

Definition 3.9

For a path $\pi e_1, \ldots, e_n$, path constraints is $\bigwedge_{i \in 1..n} e_i(V_{i-1}, V_i)$.
A path is feasible if corresponding path constraints is satisfiable.

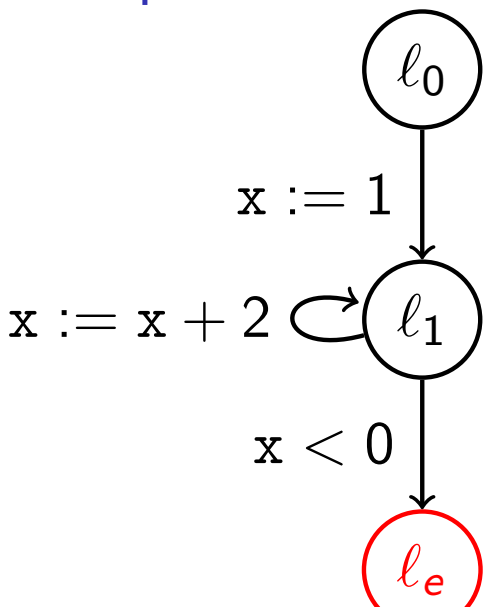
Let $\textsc{PathCons}(\pi)$ returns path constraints of π.

Path constraints are also known as "SSA formulas"

Theorem 3.1

A path is feasible then there is an execution that corresponds to the path.

Example 3.6



Consider path
$(\ell_0, \mathtt{x} := 1, \ell_1)$, $(\ell_1, \mathtt{x} := \mathtt{x} + 2, \ell_1)$, $(\ell_1, \mathtt{x} < 0, \ell_e)$

Path constraint for the path is
$F = (\mathtt{x}_1 = 1 \wedge \mathtt{x}_2 = \mathtt{x}_1 + 2 \wedge \mathtt{x}_2 < 0)$

Since F is unsat, there is no execution along the path

From simple language to labelled transition system

Theorem 3.2

Show simple programming language is isomorphic to the labelled transition system.

From simple language to labelled transition system

Theorem 3.2

Show simple programming language is isomorphic to the labelled transition system.

Example 3.7

```
L0: i = 0;
L1: while( x < 10 ) {
L2:  if x > 0 then
L3:   i := i + 1
     else
L4:   skip
    }
L5: assert( i >= 0 )
```

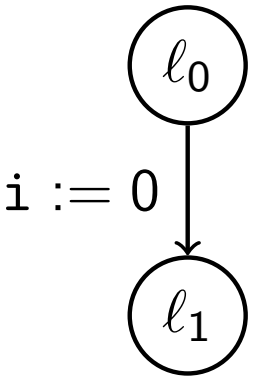
From simple language to labelled transition system

Theorem 3.2

Show simple programming language is isomorphic to the labelled transition system.

Example 3.7

```
L0: i = 0;
L1: while( x < 10 ) {
L2:  if x > 0 then
L3:   i := i + 1
     else
L4:   skip
    }
L5: assert( i >= 0 )
```



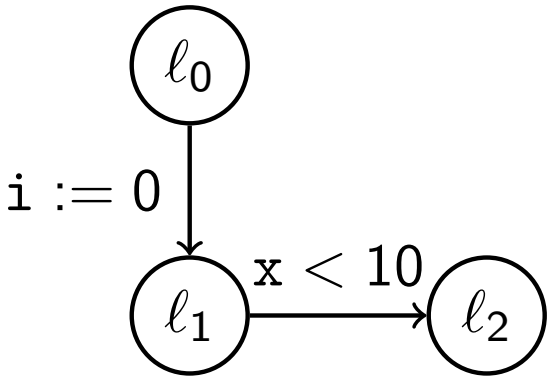
From simple language to labelled transition system

Theorem 3.2

Show simple programming language is isomorphic to the labelled transition system.

Example 3.7

```
L0: i = 0;
L1: while( x < 10 ) {
L2:  if x > 0 then
L3:   i := i + 1
     else
L4:   skip
    }
L5: assert( i >= 0 )
```



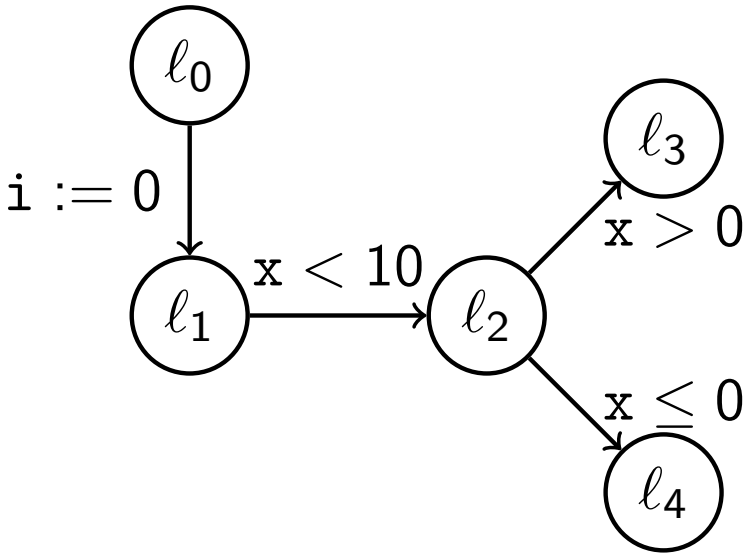
From simple language to labelled transition system

Theorem 3.2

Show simple programming language is isomorphic to the labelled transition system.

Example 3.7

```
L0: i = 0;
L1: while( x < 10 ) {
L2:  if x > 0 then
L3:   i := i + 1
     else
L4:   skip
    }
L5: assert( i >= 0 )
```



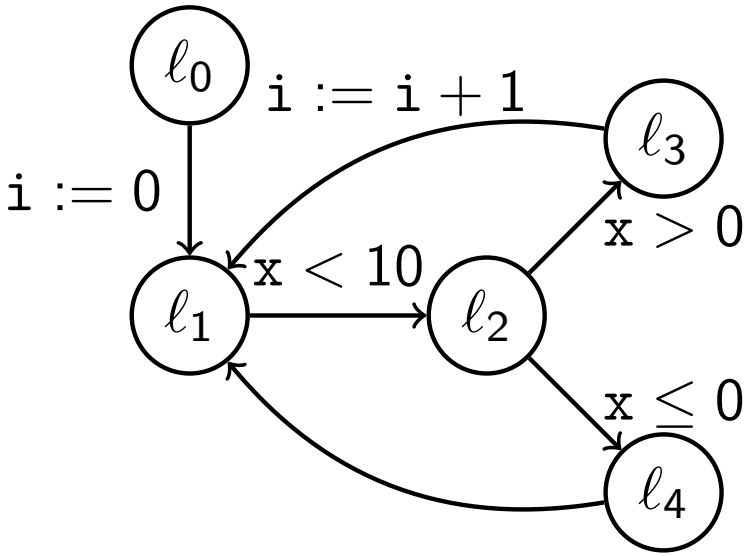
From simple language to labelled transition system

Theorem 3.2

Show simple programming language is isomorphic to the labelled transition system.

Example 3.7

```
L0: i = 0;
L1: while( x < 10 ) {
L2:  if x > 0 then
L3:   i := i + 1
     else
L4:   skip
    }
L5: assert( i >= 0 )
```



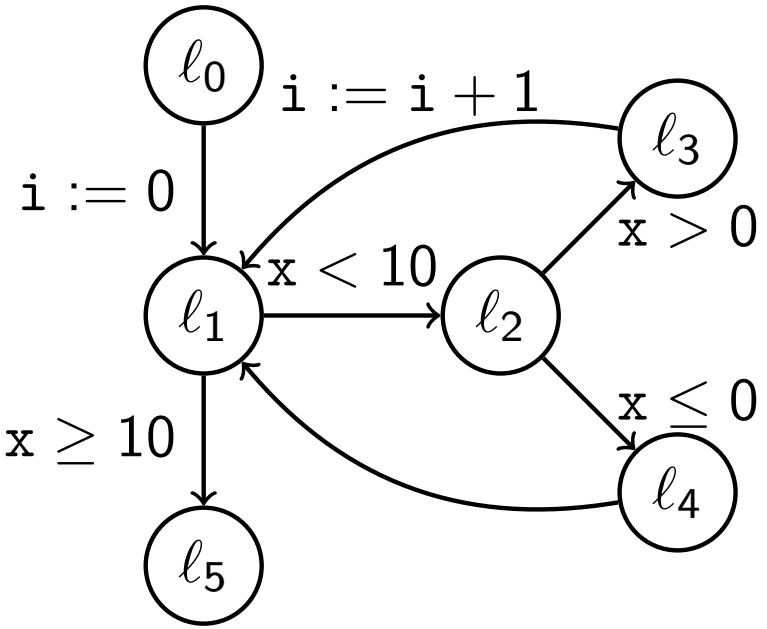
From simple language to labelled transition system

Theorem 3.2

Show simple programming language is isomorphic to the labelled transition system.

Example 3.7

```
L0: i = 0;
L1: while( x < 10 ) {
L2:  if x > 0 then
L3:   i := i + 1
     else
L4:   skip
    }
L5: assert( i >= 0 )
```



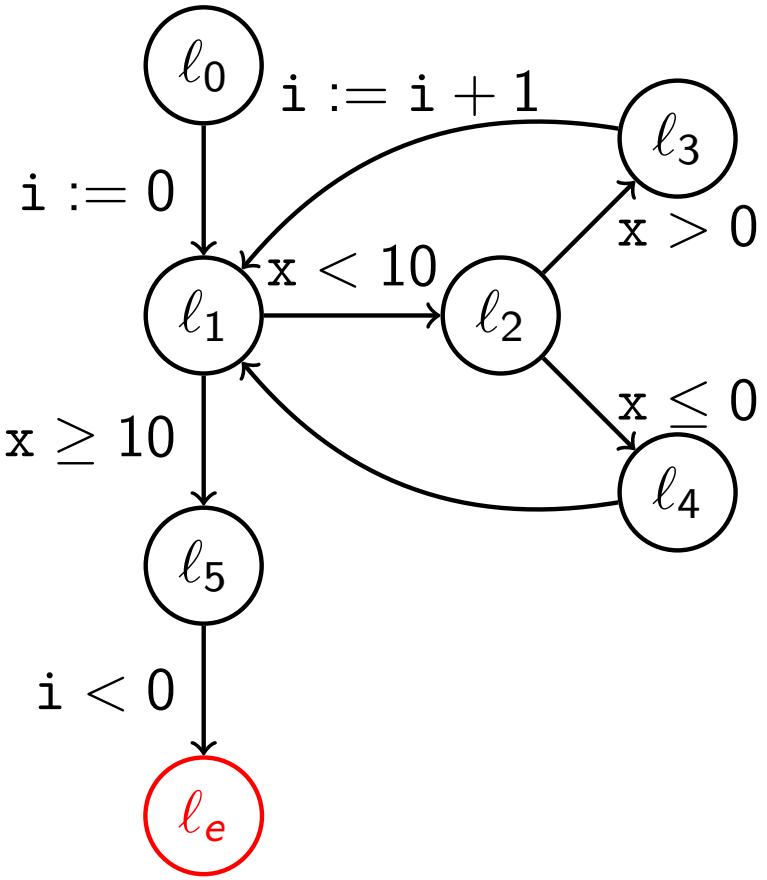
From simple language to labelled transition system

Theorem 3.2

Show simple programming language is isomorphic to the labelled transition system.

Example 3.7

```
L0: i = 0;
L1: while( x < 10 ) {
L2:  if x > 0 then
L3:   i := i + 1
     else
L4:   skip
    }
L5: assert( i >= 0 )
```



Cut-points

Definition 3.10

For a program $P = (V, L, \ell_0, \ell_e, E)$, $\textsc{CutPoints}(P)$ is the a minimal subset of L such that every path of P containing a loop passes through one of the location in $\textsc{CutPoints}(P)$.

Cut-points

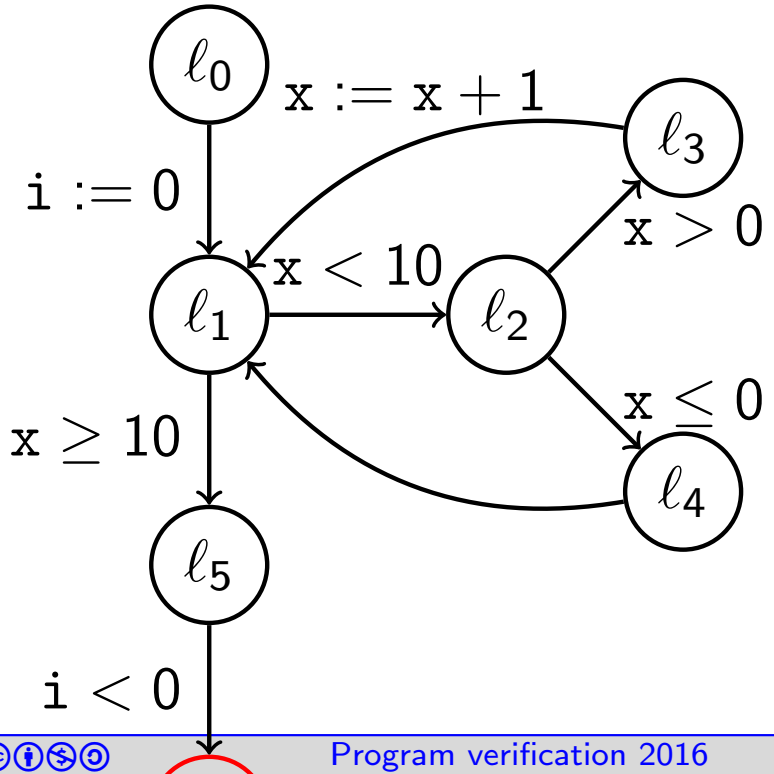
Definition 3.10

For a program $P = (V, L, \ell_0, \ell_e, E)$, $\textsc{CutPoints}(P)$ is the a minimal subset of L such that every path of P containing a loop passes through one of the location in $\textsc{CutPoints}(P)$.

$\textsc{CutPoints}(P)$ in LTS loop heads in simple language

Example 3.8

Consider the following program P.



$\textsc{CutPoints}(P) = \{\ell_1\}$

Reminder: symbolic strongest post

$$sp : \Sigma(V) \times \Sigma(V, V') \to \Sigma(V)$$

We define symbolic post over labels of P as follows.

$$sp(F, \rho) \triangleq (\exists V : F(V) \wedge \rho(V, V'))[V/V']$$

We assume that ρ and F are in a theory that admits quantifier elimination

Reminder: symbolic strongest post

$$sp : \Sigma(V) \times \Sigma(V, V') \to \Sigma(V)$$

We define symbolic post over labels of P as follows.

$$sp(F, \rho) \triangleq (\exists V : F(V) \wedge \rho(V, V'))[V/V']$$

We assume that ρ and F are in a theory that admits quantifier elimination

Using polymorphism, we also define $sp((\ell, F), (\ell, \rho, \ell') \in E) \triangleq (\ell', sp(F, \rho))$.

Reminder: symbolic strongest post

$$sp : \Sigma(V) \times \Sigma(V, V') \to \Sigma(V)$$

We define symbolic post over labels of P as follows.

$$sp(F, \rho) \triangleq (\exists V : F(V) \wedge \rho(V, V'))[V/V']$$

We assume that ρ and F are in a theory that admits quantifier elimination

Using polymorphism, we also define $sp((\ell, F), (\ell, \rho, \ell') \in E) \triangleq (\ell', sp(F, \rho))$.

For path $\pi = e_1, .., e_n$ of P, $sp((\ell, F), \pi) \triangleq sp(sp((\ell, F), e_1), e_2..e_n)$.

Topic 3.3

Loop invariants

Invariants

Definition 3.11

For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called invariant map if, for each $\ell \in L$, all reachable states at ℓ satisfy $\mathrm{I}(\ell)$.

Invariants

Definition 3.11

For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called invariant map if, for each $\ell \in L$, all reachable states at ℓ satisfy $\mathrm{I}(\ell)$.

Definition 3.12

For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called inductive if, for each $(\ell, \rho, \ell') \in E$,

$$sp(\mathrm{I}(\ell), \rho) \Rightarrow \mathrm{I}(\ell').$$

Invariants

Definition 3.11

For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called invariant map if, for each $\ell \in L$, all reachable states at ℓ satisfy $\mathrm{I}(\ell)$.

Definition 3.12

For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called inductive if, for each $(\ell, \rho, \ell') \in E$,

$$sp(\mathrm{I}(\ell), \rho) \Rightarrow \mathrm{I}(\ell').$$

Definition 3.13

For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called safe if $\mathrm{I}(\ell_0) = \top$ and $\mathrm{I}(\ell_e) = \bot$

Theorem 3.3

For P, if I is inductive and safe then I is an invariant and P is safe.

Invariants

Definition 3.11

For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called invariant map if, for each $\ell \in L$, all reachable states at ℓ satisfy $\mathrm{I}(\ell)$.

Definition 3.12

For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called inductive if, for each $(\ell, \rho, \ell') \in E$,

$$sp(\mathrm{I}(\ell), \rho) \Rightarrow \mathrm{I}(\ell').$$

Definition 3.13

For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called safe if $\mathrm{I}(\ell_0) = \top$ and $\mathrm{I}(\ell_e) = \bot$

Theorem 3.3

For P, if I is inductive and safe then I is an invariant and P is safe.

Invariant checking: is I a safe inductive invariant map?

Invariants

Definition 3.11

For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called invariant map if, for each $\ell \in L$, all reachable states at ℓ satisfy $\mathrm{I}(\ell)$.

Definition 3.12

For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called inductive if, for each $(\ell, \rho, \ell') \in E$,

$$sp(\mathrm{I}(\ell), \rho) \Rightarrow \mathrm{I}(\ell').$$

Definition 3.13

For P, a map $\mathrm{I} : L \to \Sigma(V)$ is called safe if $\mathrm{I}(\ell_0) = \top$ and $\mathrm{I}(\ell_e) = \bot$

Theorem 3.3

For P, if I is inductive and safe then I is an invariant and P is safe.

Invariant checking: is I a safe inductive invariant map?

Exercise 3.1

What is the algorithm for invariant checking?

Cut-point invariant maps

Let P be a program and $C = \textsc{CutPoints}(P) \cup \{\ell_0, \ell_e\}$.

Cut-point invariant maps

Let P be a program and $C = \textsc{CutPoints}(P) \cup \{\ell_0, \ell_e\}$.

Definition 3.14

A map $\mathrm{I} : C \to \Sigma(V)$ *is called cut-point invariant map if, for each* $\ell \in C$*, all reachable states at* ℓ *satisfy* $\mathrm{I}(\ell)$*.*

Cut-point invariant maps

Let P be a program and $C = \textsc{CutPoints}(P) \cup \{\ell_0, \ell_e\}$.

Definition 3.14

A map $\mathrm{I} : C \to \Sigma(V)$ *is called cut-point invariant map if, for each* $\ell \in C$*, all reachable states at* ℓ *satisfy* $\mathrm{I}(\ell)$*.*

Definition 3.15

A map $\mathrm{I} : C \to \Sigma(V)$ *is called inductive if, for each* $\ell, \ell' \in C$ *and* $\pi \in \textsc{LoopFreePaths}(P, \ell, \ell')$*,* $sp(\mathrm{I}(\ell), \pi) \Rightarrow \mathrm{I}(\ell')$*.*

Cut-point invariant maps

Let P be a program and $C = \textsc{CutPoints}(P) \cup \{\ell_0, \ell_e\}$.

Definition 3.14

A map $\mathrm{I} : C \to \Sigma(V)$ *is called cut-point invariant map if, for each* $\ell \in C$*, all reachable states at* ℓ *satisfy* $\mathrm{I}(\ell)$*.*

Definition 3.15

A map $\mathrm{I} : C \to \Sigma(V)$ *is called inductive if, for each* $\ell, \ell' \in C$ *and* $\pi \in \textsc{LoopFreePaths}(P, \ell, \ell')$*,* $sp(\mathrm{I}(\ell), \pi) \Rightarrow \mathrm{I}(\ell')$*.*

Definition 3.16

A map $\mathrm{I} : C \to \Sigma(V)$ *is called safe if* $\mathrm{I}(\ell_0) = \top$ *and* $\mathrm{I}(\ell_e) = \bot$

Cut-point invariant maps

Let P be a program and $C = \textsc{CutPoints}(P) \cup \{\ell_0, \ell_e\}$.

Definition 3.14

A map $\mathrm{I} : C \to \Sigma(V)$ *is called cut-point invariant map if, for each* $\ell \in C$*, all reachable states at* ℓ *satisfy* $\mathrm{I}(\ell)$*.*

Definition 3.15

A map $\mathrm{I} : C \to \Sigma(V)$ *is called inductive if, for each* $\ell, \ell' \in C$ *and* $\pi \in \textsc{LoopFreePaths}(P, \ell, \ell')$*,* $sp(\mathrm{I}(\ell), \pi) \Rightarrow \mathrm{I}(\ell')$*.*

Definition 3.16

A map $\mathrm{I} : C \to \Sigma(V)$ *is called safe if* $\mathrm{I}(\ell_0) = \top$ *and* $\mathrm{I}(\ell_e) = \bot$

Theorem 3.4

If I *is inductive and safe then* I *is an cut-point invariant map and* P *is safe.*

Cut-point invariant maps

Let P be a program and $C = \textsc{CutPoints}(P) \cup \{\ell_0, \ell_e\}$.

Definition 3.14

A map $\mathrm{I} : C \to \Sigma(V)$ *is called cut-point invariant map if, for each* $\ell \in C$*, all reachable states at* ℓ *satisfy* $\mathrm{I}(\ell)$*.*

Definition 3.15

A map $\mathrm{I} : C \to \Sigma(V)$ *is called inductive if, for each* $\ell, \ell' \in C$ *and* $\pi \in \textsc{LoopFreePaths}(P, \ell, \ell')$*,* $sp(\mathrm{I}(\ell), \pi) \Rightarrow \mathrm{I}(\ell')$*.*

Definition 3.16

A map $\mathrm{I} : C \to \Sigma(V)$ *is called safe if* $\mathrm{I}(\ell_0) = \top$ *and* $\mathrm{I}(\ell_e) = \bot$

Theorem 3.4

If I *is inductive and safe then* I *is an cut-point invariant map and* P *is safe.*

Proof.

Every path from ℓ_0 to ℓ_e can be segmented into loop free paths between cut-points. Therefore, no such path is feasible. □

Annotated verification: VCC demo

http://rise4fun.com/Vcc

Annotated verification: VCC demo

http://rise4fun.com/Vcc

Exercise 3.2

Complete the following program such that Vcc proves it correct

```
#include <vcc.h>
int main()
{
  int x, y;
  _(assume x > y +3 && x < 3000 )
  while( 0 < y ) _(invariant ....) {
    x = x + 1;
    y = y -1;
  }
  _(assert x >= y)
  return 0;
}
```

Annotated verification

- There are many tools like VCC that require user to write invariants at the loop heads and function boundaries
- Rest of the verification is done as discussed in earlier slides

Annotated verification

- There are many tools like VCC that require user to write invariants at the loop heads and function boundaries
- Rest of the verification is done as discussed in earlier slides
- User needs to do a lot of work, not a very desirable method

Annotated verification

- There are many tools like VCC that require user to write invariants at the loop heads and function boundaries
- Rest of the verification is done as discussed in earlier slides
- User needs to do a lot of work, not a very desirable method

What if we want to compute the invariants automatically?

Topic 3.4

Concrete model checking - enumerate reachable states

Isn't enumeration impossible?

Isn't enumeration impossible?

- Explore the transition graph explicitly, light weight machinery
- If edge labels are guarded commands then finding next values are trivial

Isn't enumeration impossible?

- Explore the transition graph explicitly, light weight machinery
- If edge labels are guarded commands then finding next values are trivial
- After resolving non-determinism, concrete model checking reduces to program execution

Isn't enumeration impossible?

- Explore the transition graph explicitly, light weight machinery
- If edge labels are guarded commands then finding next values are trivial
- After resolving non-determinism, concrete model checking reduces to program execution
- May be only finitely many states are reachable

Isn't enumeration impossible?

- Explore the transition graph explicitly, light weight machinery
- If edge labels are guarded commands then finding next values are trivial
- After resolving non-determinism, concrete model checking reduces to program execution
- May be only finitely many states are reachable
- May be impossible to cover all states explicitly, but it may cover a portion of interest

Isn't enumeration impossible?

- Explore the transition graph explicitly, light weight machinery
- If edge labels are guarded commands then finding next values are trivial
- After resolving non-determinism, concrete model checking reduces to program execution
- May be only finitely many states are reachable
- May be impossible to cover all states explicitly, but it may cover a portion of interest
- Useful for learning design principles of computing reachable states

Concrete model checking

Algorithm 3.1: Concrete model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: SAFE if P is safe, UNSAFE otherwise

Concrete model checking

Algorithm 3.1: Concrete model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: SAFE if P is safe, UNSAFE otherwise

$reach := \emptyset$;

$worklist := \{(\ell_0, v) | v \in \mathbb{Z}^{|V|}\}$;

Concrete model checking

Algorithm 3.1: Concrete model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: SAFE if P is safe, UNSAFE otherwise

$reach := \emptyset$;

$worklist := \{(\ell_0, v) | v \in \mathbb{Z}^{|V|}\}$;

while $worklist \neq \emptyset$ **do**

  choose $(\ell, v) \in worklist$;

  $worklist := worklist \setminus \{(\ell, v)\}$;

Concrete model checking

Algorithm 3.1: Concrete model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: SAFE if P is safe, UNSAFE otherwise

$reach := \emptyset$;

$worklist := \{(\ell_0, v) | v \in \mathbb{Z}^{|V|}\}$;

while $worklist \neq \emptyset$ **do**

  choose $(\ell, v) \in worklist$;

  $worklist := worklist \setminus \{(\ell, v)\}$;

  if $(\ell, v) \notin reach$ **then**

    $reach := reach \cup \{(\ell, v)\}$;

    foreach $(\ell, F(V, V'), \ell') \in E$ **do**

      $worklist := worklist \cup \{(\ell', v') | F(v, v')\}$;

Concrete model checking

Algorithm 3.1: Concrete model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: SAFE if P is safe, UNSAFE otherwise

$reach := \emptyset$;

$worklist := \{(\ell_0, v) | v \in \mathbb{Z}^{|V|}\}$;

while $worklist \neq \emptyset$ **do**

  choose $(\ell, v) \in worklist$;

  $worklist := worklist \setminus \{(\ell, v)\}$;

  if $(\ell, v) \notin reach$ **then**

    $reach := reach \cup \{(\ell, v)\}$;

    foreach $(\ell, F(V, V'), \ell') \in E$ **do**

      $worklist := worklist \cup \{(\ell', v') | F(v, v')\}$;

if $(\ell_e, \_) \in reach$ **then**

  return UNSAFE

else

  return SAFE

Concrete model checking

Algorithm 3.1: Concrete model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$
Output: SAFE if P is safe, UNSAFE otherwise

```
reach := ∅;
worklist := {(ℓ0, v) | v ∈ ℤ^|V|};
while worklist ≠ ∅ do
    choose (ℓ, v) ∈ worklist;
    worklist := worklist \ {(ℓ, v)};
    if (ℓ, v) ∉ reach then
        reach := reach ∪ {(ℓ, v)};
        foreach (ℓ, F(V, V'), ℓ') ∈ E do
            worklist := worklist ∪ {(ℓ', v') | F(v, v')};
if (ℓe, _) ∈ reach then
    return UNSAFE
else
    return SAFE
```

Exercise 3.3

Suggest improvements in the algorithm

Concrete model checking

Algorithm 3.1: Concrete model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$
Output: SAFE if P is safe, UNSAFE otherwise

```
reach := ∅;
worklist := {(ℓ0, v) | v ∈ ℤ^|V|};
while worklist ≠ ∅ do
    choose (ℓ, v) ∈ worklist;
    worklist := worklist \ {(ℓ, v)};
    if (ℓ, v) ∉ reach then
        reach := reach ∪ {(ℓ, v)};
        foreach (ℓ, F(V, V'), ℓ') ∈ E do
            worklist := worklist ∪ {(ℓ', v') | F(v, v')};
if (ℓe, _) ∈ reach then
    return UNSAFE
else
    return SAFE
```

the choice defines the nature of exploration

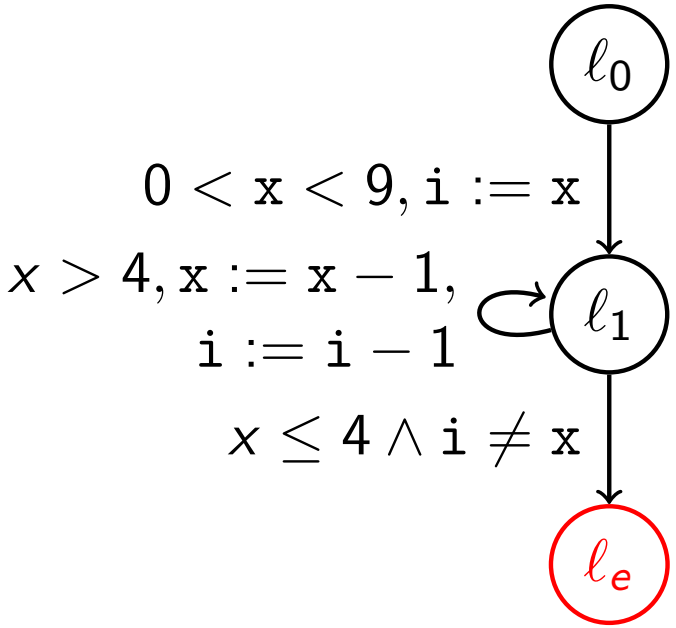
Exercise 3.3

Suggest improvements in the algorithm

Example: concrete model checking

Example 3.9

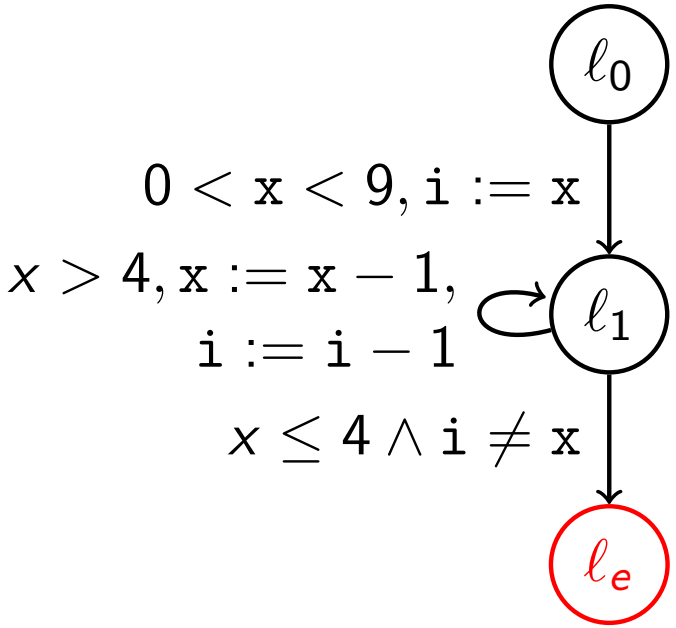
Consider the following example



Example: concrete model checking

Example 3.9

Consider the following example

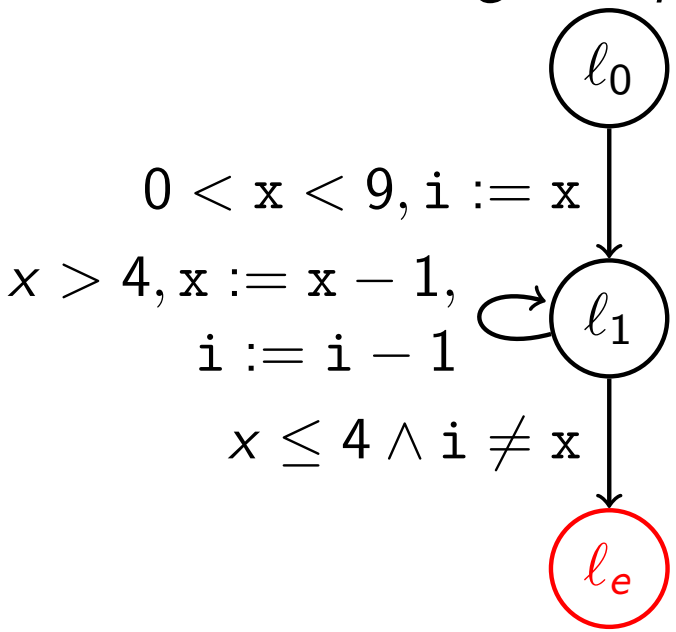


Let $V = [\mathtt{x}, \mathtt{i}]$

Example: concrete model checking

Example 3.9

Consider the following example



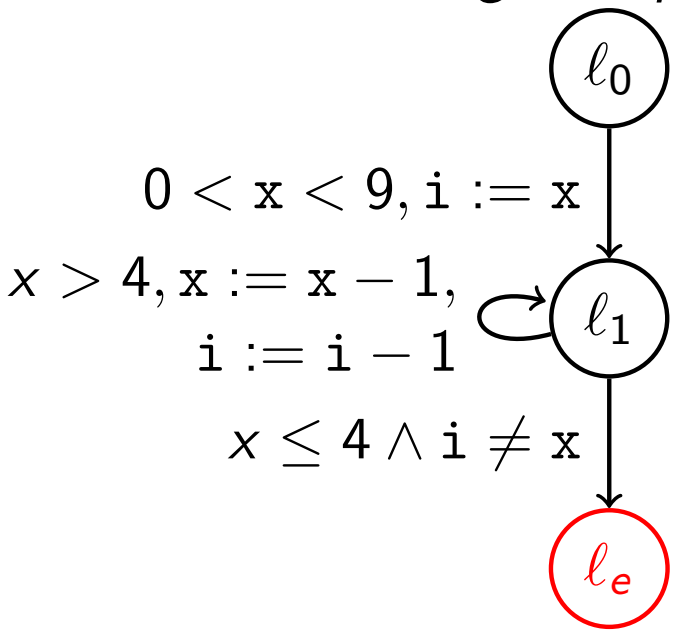
Initialization:
reach $= \emptyset$, *worklist* $= \{(\ell_0, v) | v \in \mathbb{Z}^2\}$

Let $V = [\mathtt{x}, \mathtt{i}]$

Example: concrete model checking

Example 3.9

Consider the following example



Initialization:
reach $= \emptyset$, *worklist* $= \{(\ell_0, v) | v \in \mathbb{Z}^2\}$

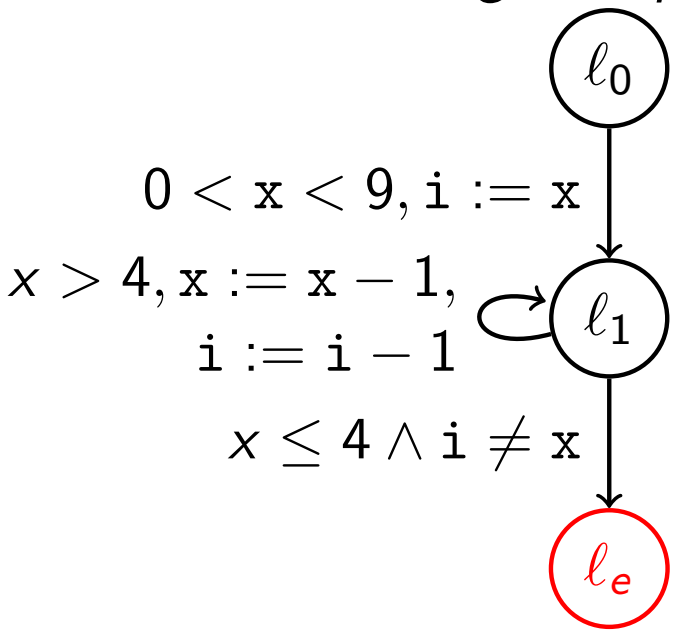
Choose a state:
Lets choose $(\ell_0, [8, 0])$

Let $V = [\mathtt{x}, \mathtt{i}]$

Example: concrete model checking

Example 3.9

Consider the following example



Initialization:
reach $= \emptyset$, *worklist* $= \{(\ell_0, v) | v \in \mathbb{Z}^2\}$

Choose a state:
Lets choose $(\ell_0, [8, 0])$

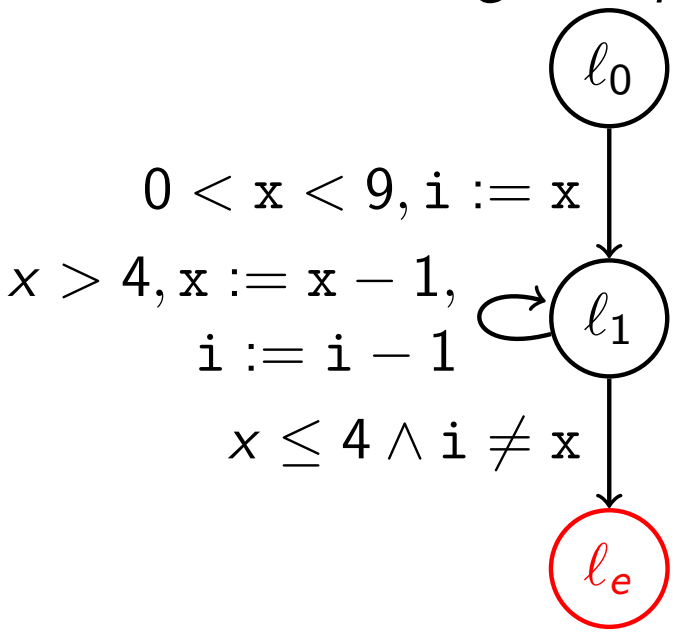
Update worklist:
worklist $:=$ *worklist* $\setminus \{(\ell_0, [8, 8])\}$

Let $V = [\mathtt{x}, \mathtt{i}]$

Example: concrete model checking

Example 3.9

Consider the following example



Let $V = [\mathtt{x}, \mathtt{i}]$

Initialization:
reach $= \emptyset$, *worklist* $= \{(\ell_0, v) | v \in \mathbb{Z}^2\}$

Choose a state:
Lets choose $(\ell_0, [8, 0])$

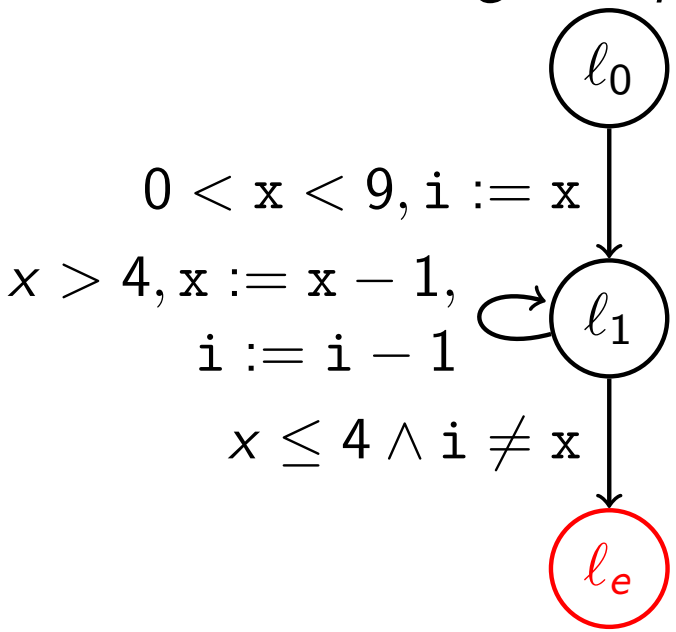
Update worklist:
worklist $:=$ *worklist* $\setminus \{(\ell_0, [8, 8])\}$

Add successors in worklist if state not visited:
worklist $:=$ *worklist* $\cup \{(\ell_1, [8, 8])\}$
reach $:=$ *reach* $\cup \{(\ell_0, [8, 0])\}$

Example: concrete model checking

Example 3.9

Consider the following example



Let $V = [\mathtt{x}, \mathtt{i}]$

Initialization:
reach $= \emptyset$, *worklist* $= \{(\ell_0, v) | v \in \mathbb{Z}^2\}$

Choose a state:
Lets choose $(\ell_0, [8, 0])$

Update worklist:
worklist $:=$ *worklist* $\setminus \{(\ell_0, [8, 8])\}$

Add successors in worklist if state not visited:
worklist $:=$ *worklist* $\cup \{(\ell_1, [8, 8])\}$
reach $:=$ *reach* $\cup \{(\ell_0, [8, 0])\}$

... go back to choosing a new state from worklist

Search strategies

- DFS
- BFS

Search strategies

- DFS
- BFS
- A^*
  - worklist is a priority queue,
  - weights are assigned to states based on estimate on possibility of reaching error

Search strategies

- DFS
- BFS
- A^*
  - worklist is a priority queue,
  - weights are assigned to states based on estimate on possibility of reaching error

Exercise 3.4

Describe A^* search strategy

Optimizations: exploiting structure

- Symmetry reduction
- Assume guarantee
- Partial order reduction (for concurrent systems)

Optimizations: reducing space

- hashed states - reach set contains hash of states (not sound)
- Stateless exploration - no reach set (redundant)

Trade-off among time, space, and soundness

Optimizations: reducing space

- hashed states - reach set contains hash of states (not sound)
- Stateless exploration - no reach set (redundant)

Trade-off among time, space, and soundness

Exercise 3.5

Write concrete model checking using hash tables

Proof and counterexample

Definition 3.17

A proof of a program is an object that allows one to check safety of the program using a low complexity (preferably linear) algorithm in the size of the object.

Proof and counterexample

Definition 3.17

A proof of a program is an object that allows one to check safety of the program using a low complexity (preferably linear) algorithm in the size of the object.

Example 3.10

In our concrete model checking algorithm, reach set is the proof. The checker needs to find that no more states can be reached from reach.

Proof and counterexample

Definition 3.17

A proof of a program is an object that allows one to check safety of the program using a low complexity (preferably linear) algorithm in the size of the object.

Example 3.10

In our concrete model checking algorithm, reach set is the proof. The checker needs to find that no more states can be reached from reach.

Definition 3.18

A counterexample of a program is an execution that ends at ℓ_e.

Proof and counterexample

Definition 3.17

A proof of a program is an object that allows one to check safety of the program using a low complexity (preferably linear) algorithm in the size of the object.

Example 3.10

In our concrete model checking algorithm, reach set is the proof. The checker needs to find that no more states can be reached from reach.

Definition 3.18

A counterexample of a program is an execution that ends at ℓ_e.

A verification method may produce three possible outcomes for a program

- proof
- counterexample
- unknown or non-termination

Enabling counterexample generation

Algorithm 3.2: Concrete model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$
Output: SAFE if P is safe, UNSAFE otherwise

reach := $\emptyset$;
worklist := $\{(\ell_0, v) | v \in \mathbb{Z}^{|V|}\}$;
while *worklist* $\neq \emptyset$ **do**
    choose $(\ell, v) \in$ *worklist*;
    worklist := *worklist* $\setminus \{(\ell, v)\}$;
    if $(\ell, v) \notin$ *reach* **then**
        reach := *reach* $\cup \{(\ell, v)\}$;
        foreach v' *s.t.* $F(v, v')$ *is sat* $\wedge(\ell, F(V, V'), \ell') \in E$ **do**
            worklist := *worklist* $\cup \{(\ell', v')\}$;
if $(\ell_e, v) \in$ *reach* **then**
    return UNSAFE
else
    return SAFE

Exercise 3.6
add data structure to report counterexample

Enabling counterexample generation

Algorithm 3.2: Concrete model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$
Output: SAFE if P is safe, UNSAFE otherwise

reach := $\emptyset$; *parents* := $\lambda x.\text{NAN}$;
worklist := $\{(\ell_0, v) | v \in \mathbb{Z}^{|V|}\}$;
while *worklist* $\neq \emptyset$ **do**
  choose $(\ell, v) \in$ *worklist*;
  worklist := *worklist* $\setminus \{(\ell, v)\}$;
  if $(\ell, v) \notin$ *reach* **then**
    reach := *reach* $\cup \{(\ell, v)\}$;
    foreach v' *s.t.* $F(v, v')$ *is sat* $\wedge(\ell, F(V, V'), \ell') \in E$ **do**
      worklist := *worklist* $\cup \{(\ell', v')\}$; *parents*$((\ell', v'))$:= (ℓ, v);
if $(\ell_e, v) \in$ *reach* **then**
  return UNSAFE*(traverseToInit(parents,* (ℓ_e, v)*))*
else
  return SAFE

Exercise 3.6
add data structure to report counterexample

Topic 3.5

Symbolic methods

Why symbolic?

To avoid, state explosion problem

Symbolic methods

Now, we cover some methods that try/avoid to compute lfp

- Symbolic model checking
- Constraint based invariant generation

Symbolic state

Definition 3.19

A symbolic state s of $P = (V, L, \ell_0, \ell_e, E)$ is a pair (ℓ, F), where

- $\ell \in L$
- *F is a formula over variables V in a given theory*

Symbolic model checking

Algorithm 3.3: Symbolic model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: SAFE if P is safe, UNSAFE otherwise

Symbolic model checking

Algorithm 3.3: Symbolic model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: SAFE if P is safe, UNSAFE otherwise

$reach : L \to \Sigma(V) \ := \ \lambda x.\bot;$

$worklist \ := \ \{(\ell_0, \top)\};$

Symbolic model checking

Algorithm 3.3: Symbolic model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: SAFE if P is safe, UNSAFE otherwise

$reach : L \to \Sigma(V) \ := \ \lambda x.\bot$;

$worklist \ := \ \{(\ell_0, \top)\}$;

while $worklist \neq \emptyset$ **do**

  choose $(\ell, F) \in worklist$;

  $worklist \ := \ worklist \setminus \{(\ell, F)\}$;

Symbolic model checking

Algorithm 3.3: Symbolic model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: SAFE if P is safe, UNSAFE otherwise

$reach : L \to \Sigma(V) \ := \ \lambda x.\bot$;

$worklist \ := \ \{(\ell_0, \top)\}$;

while $worklist \neq \emptyset$ **do**

  choose $(\ell, F) \in worklist$;

  $worklist \ := \ worklist \setminus \{(\ell, F)\}$;

  if $\neg(F \Rightarrow reach(\ell))$ *is sat* **then**

    $reach \ := \ reach[\ell \mapsto reach(\ell) \vee F]$;

Symbolic model checking

Algorithm 3.3: Symbolic model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: SAFE if P is safe, UNSAFE otherwise

$reach : L \to \Sigma(V) \ := \ \lambda x.\bot$;

$worklist \ := \ \{(\ell_0, \top)\}$;

while $worklist \neq \emptyset$ **do**

  choose $(\ell, F) \in worklist$;

  $worklist \ := \ worklist \setminus \{(\ell, F)\}$;

  if $\neg(F \Rightarrow reach(\ell))$ *is sat* **then**

    $reach \ := \ reach[\ell \mapsto reach(\ell) \vee F]$;

    foreach $(\ell, \rho(V, V'), \ell') \in E$ **do**

      $worklist \ := \ worklist \cup \{(\ell', sp(F, \rho))\}$;

Symbolic model checking

Algorithm 3.3: Symbolic model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$

Output: SAFE if P is safe, UNSAFE otherwise

$reach : L \to \Sigma(V) \ := \ \lambda x.\bot$;

$worklist \ := \ \{(\ell_0, \top)\}$;

while $worklist \neq \emptyset$ **do**

  choose $(\ell, F) \in worklist$;

  $worklist \ := \ worklist \setminus \{(\ell, F)\}$;

  if $\neg(F \Rightarrow reach(\ell))$ *is sat* **then**

    $reach \ := \ reach[\ell \mapsto reach(\ell) \vee F]$;

    foreach $(\ell, \rho(V, V'), \ell') \in E$ **do**

      $worklist \ := \ worklist \cup \{(\ell', sp(F, \rho))\}$;

if $reach(\ell_e) \neq \bot$ **then**

  return UNSAFE

else

  return SAFE

Symbolic model checking

Algorithm 3.3: Symbolic model checking

Input: $P = (V, L, \ell_0, \ell_e, E)$
Output: SAFE if P is safe, UNSAFE otherwise

$reach : L \to \Sigma(V) := \lambda x.\bot$;
$worklist := \{(\ell_0, \top)\}$;
while $worklist \neq \emptyset$ **do**
    choose $(\ell, F) \in worklist$;
    $worklist := worklist \setminus \{(\ell, F)\}$;
    if $\neg(F \Rightarrow reach(\ell))$ *is sat* **then**
        $reach := reach[\ell \mapsto reach(\ell) \vee F]$;
        foreach $(\ell, \rho(V, V'), \ell') \in E$ **do**
            $worklist := worklist \cup \{(\ell', sp(F, \rho))\}$;

if $reach(\ell_e) \neq \bot$ **then**
    return UNSAFE
else
    return SAFE

Note: We need efficient implementations of various logical operators!

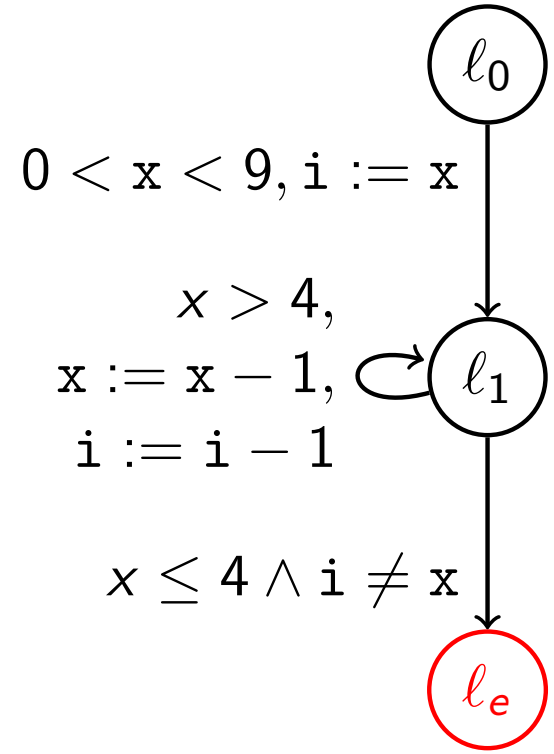
Exercise 3.7

Give a condition for definite termination?

Example: symbolic model checking

Example 3.11

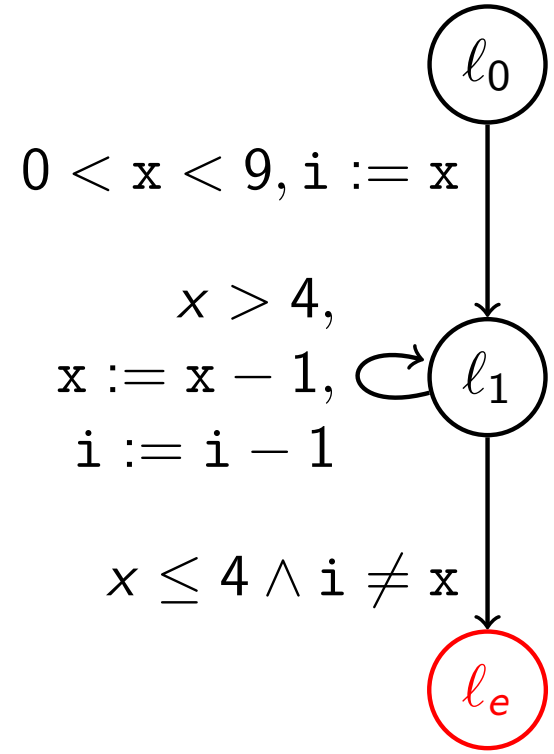
Consider the following example



Example: symbolic model checking

Example 3.11

Consider the following example

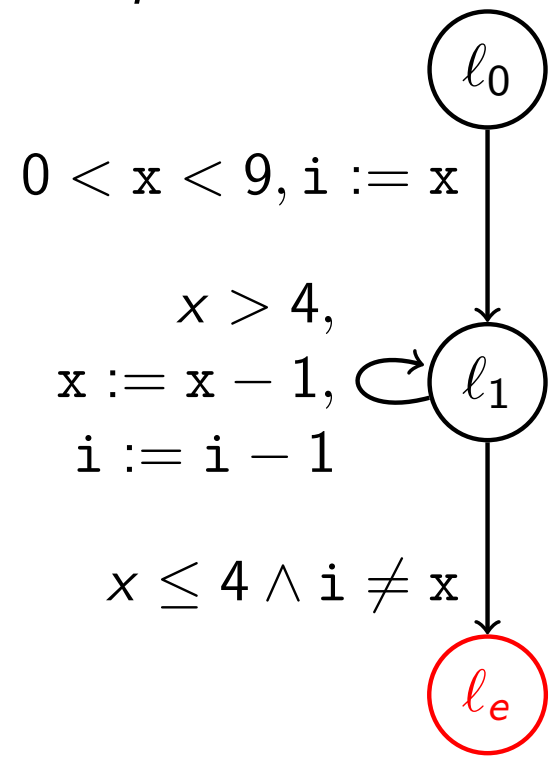


Let $V = [\mathtt{x}, \mathtt{i}]$

Example: symbolic model checking

Example 3.11

Consider the following example



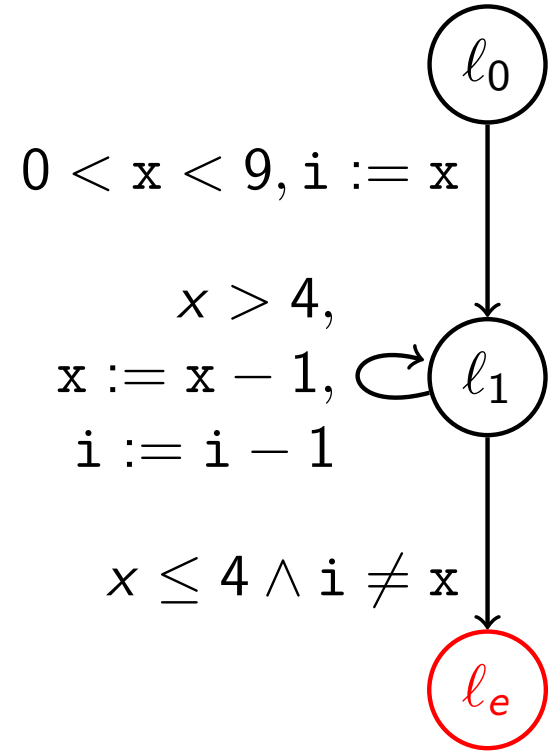
Init: $\mathit{reach} = \lambda x.\bot$, $\mathit{worklist} = \{(\ell_0, \top)\}$

Let $V = [\mathtt{x}, \mathtt{i}]$

Example: symbolic model checking

Example 3.11

Consider the following example



Init: $reach = \lambda x.\bot$, $worklist = \{(\ell_0, \top)\}$

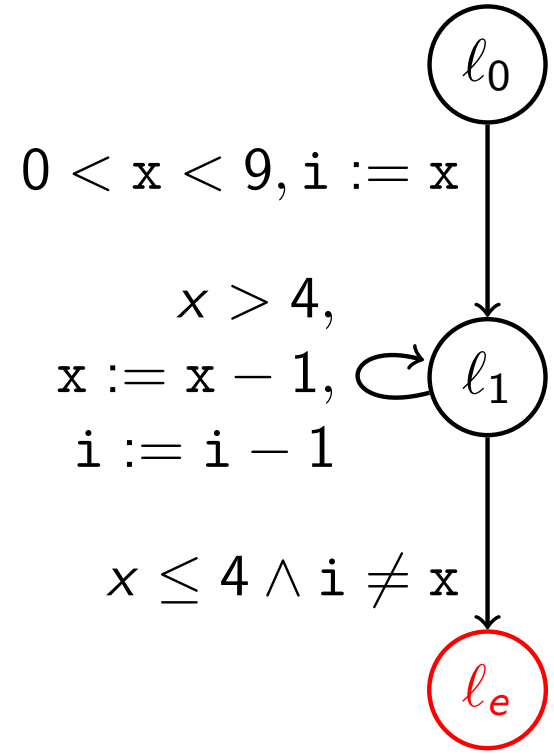
Choose a state: $(\ell_0, \top)$ *(only choice)*

Let $V = [\mathtt{x}, \mathtt{i}]$

Example: symbolic model checking

Example 3.11

Consider the following example



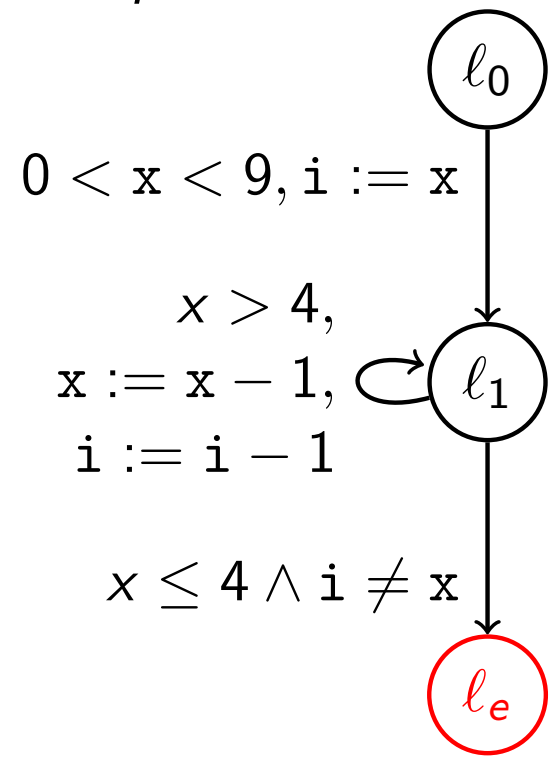
Init: $reach = \lambda x.\bot$, $worklist = \{(\ell_0, \top)\}$
Choose a state: $(\ell_0, \top)$ *(only choice)*
Update worklist: $worklist := \emptyset$

Let $V = [\mathtt{x}, \mathtt{i}]$

Example: symbolic model checking

Example 3.11

Consider the following example



Init: reach $= \lambda x.\bot$, *worklist* $= \{(\ell_0, \top)\}$

Choose a state: $(\ell_0, \top)$ *(only choice)*

Update worklist: worklist $:= \emptyset$

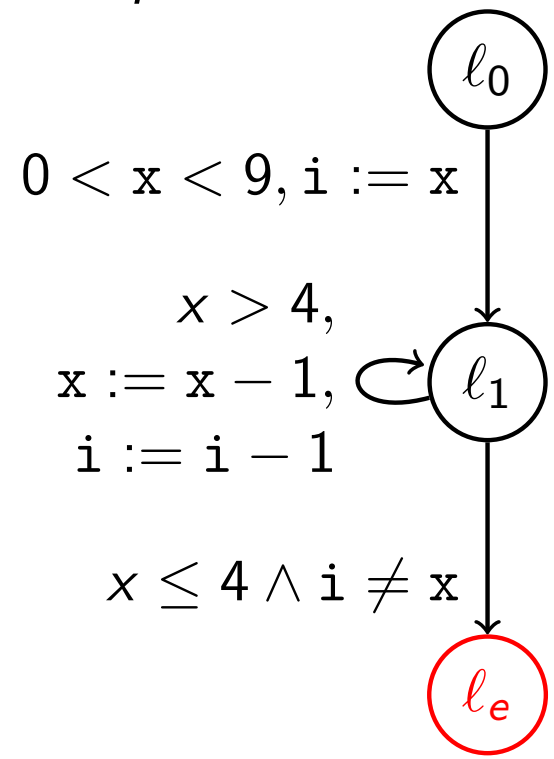
Add successors in worklist:

Let $V = [\mathtt{x}, \mathtt{i}]$

Example: symbolic model checking

Example 3.11

Consider the following example



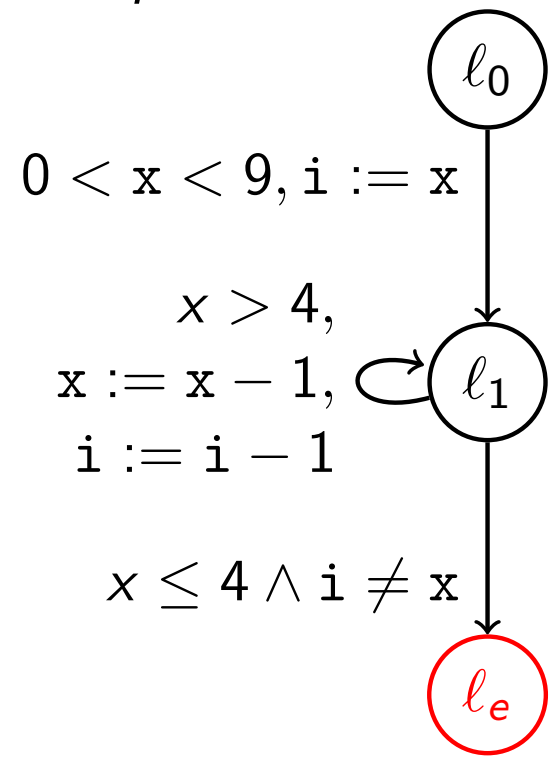
Let $V = [\mathtt{x}, \mathtt{i}]$

Init: $reach = \lambda x.\bot$, $worklist = \{(\ell_0, \top)\}$
Choose a state: $(\ell_0, \top)$ *(only choice)*
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(\top \Rightarrow reach(\ell_0))$ *is sat,*

Example: symbolic model checking

Example 3.11

Consider the following example



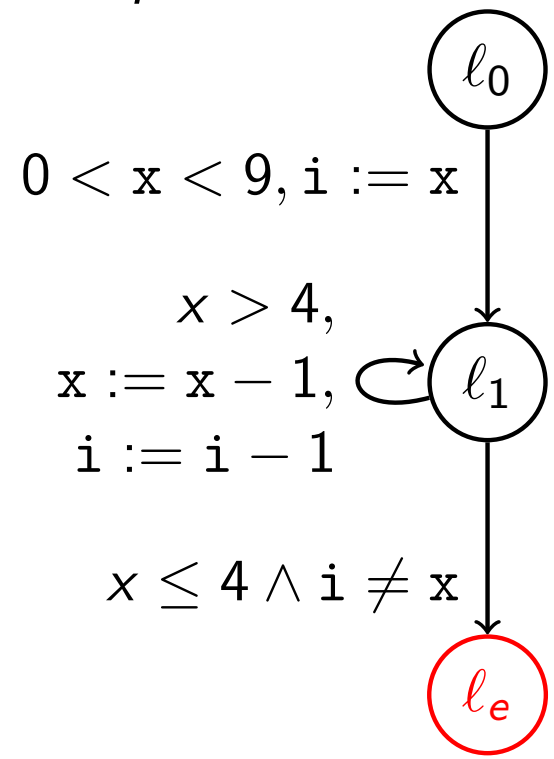
Let $V = [\mathtt{x}, \mathtt{i}]$

Init: $reach = \lambda x.\bot$, $worklist = \{(\ell_0, \top)\}$
Choose a state: $(\ell_0, \top)$ *(only choice)*
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(\top \Rightarrow reach(\ell_0))$ *is sat,*
$worklist := worklist \cup \{(\ell_1, 0 < x = i < 9)\}$

Example: symbolic model checking

Example 3.11

Consider the following example



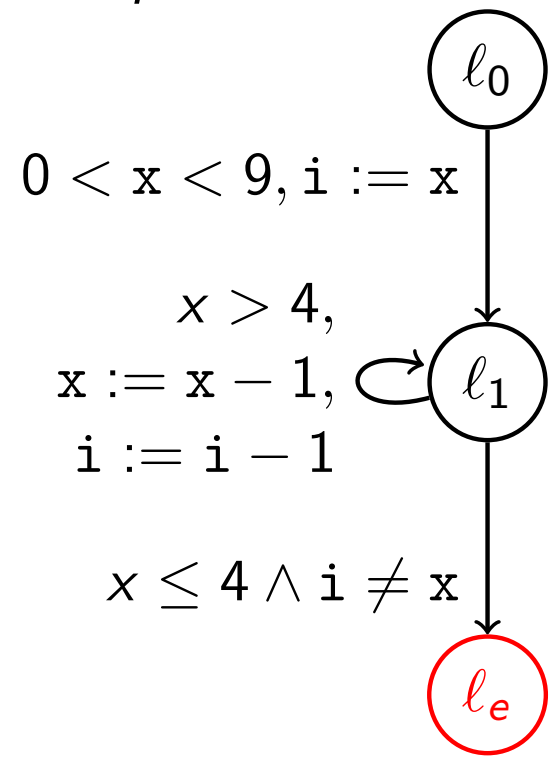
Let $V = [\mathtt{x}, \mathtt{i}]$

Init: $reach = \lambda x.\bot$, $worklist = \{(\ell_0, \top)\}$
Choose a state: $(\ell_0, \top)$ *(only choice)*
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(\top \Rightarrow reach(\ell_0))$ *is sat,*
$worklist := worklist \cup \{(\ell_1, 0 < x = i < 9)\}$
$reach(\ell_0) := reach(\ell_0) \vee \top := \top$

Example: symbolic model checking

Example 3.11

Consider the following example



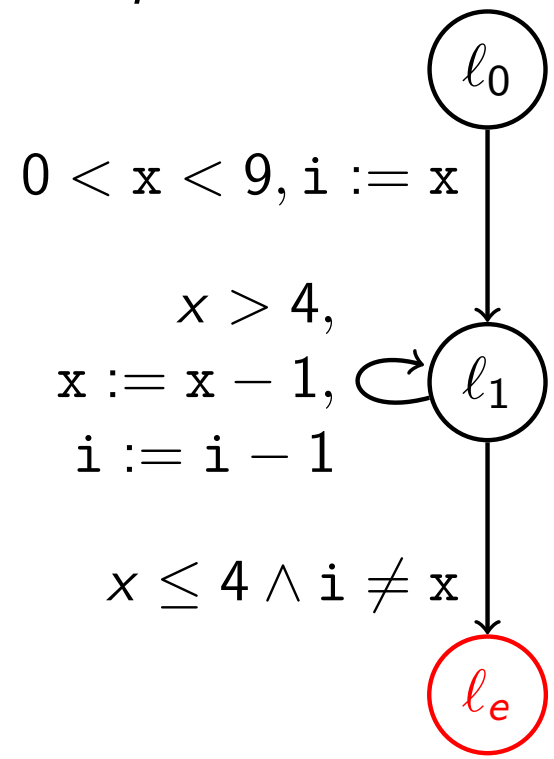
Let $V = [\mathtt{x}, \mathtt{i}]$

Init: $reach = \lambda x.\bot$, $worklist = \{(\ell_0, \top)\}$
Choose a state: $(\ell_0, \top)$ *(only choice)*
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(\top \Rightarrow reach(\ell_0))$ *is sat,*
$worklist := worklist \cup \{(\ell_1, 0 < x = i < 9)\}$
$reach(\ell_0) := reach(\ell_0) \vee \top := \top$
Again choose a state: $\{(\ell_1, 0 < x = i < 9)\}$

Example: symbolic model checking

Example 3.11

Consider the following example



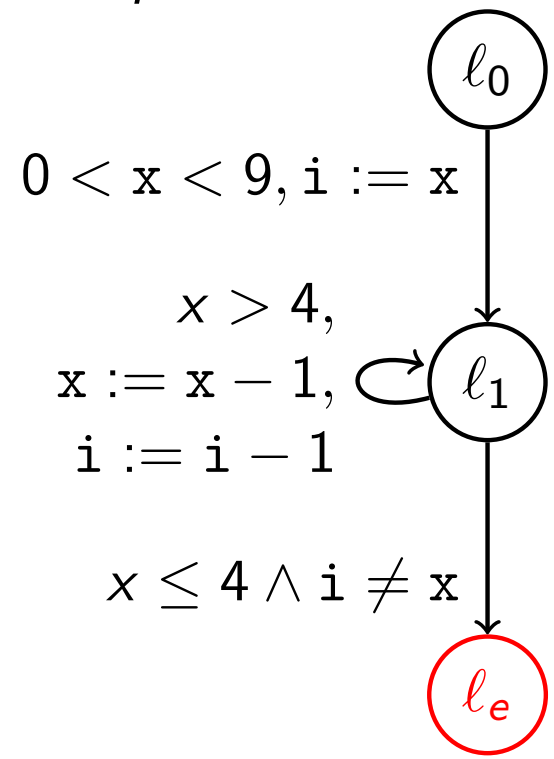
Let $V = [\mathtt{x}, \mathtt{i}]$

Init: $reach = \lambda x.\bot$, $worklist = \{(\ell_0, \top)\}$
Choose a state: $(\ell_0, \top)$ *(only choice)*
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(\top \Rightarrow reach(\ell_0))$ *is sat,*
$worklist := worklist \cup \{(\ell_1, 0 < x = i < 9)\}$
$reach(\ell_0) := reach(\ell_0) \vee \top := \top$
Again choose a state: $\{(\ell_1, 0 < x = i < 9)\}$
Update worklist: $worklist := \emptyset$
Add successors in worklist:

Example: symbolic model checking

Example 3.11

Consider the following example



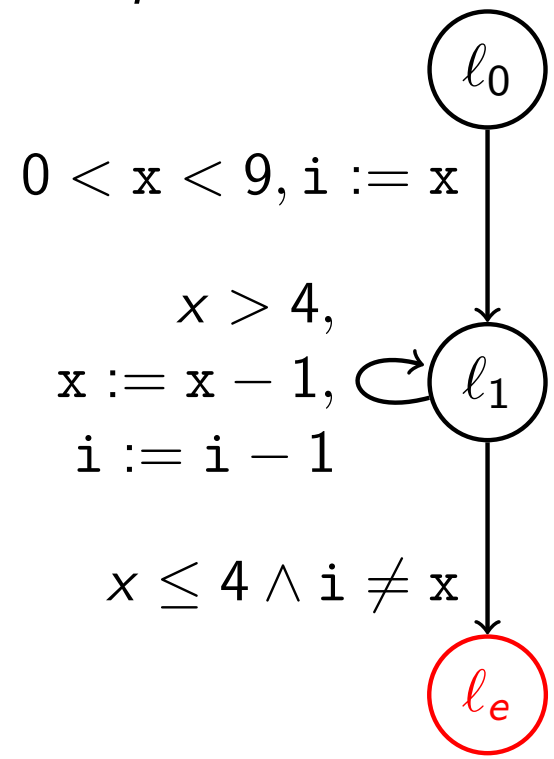
Let $V = [\mathtt{x}, \mathtt{i}]$

Init: $reach = \lambda x.\bot$, $worklist = \{(\ell_0, \top)\}$
Choose a state: $(\ell_0, \top)$ *(only choice)*
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(\top \Rightarrow reach(\ell_0))$ *is sat,*
$worklist := worklist \cup \{(\ell_1, 0 < x = i < 9)\}$
$reach(\ell_0) := reach(\ell_0) \vee \top := \top$
Again choose a state: $\{(\ell_1, 0 < x = i < 9)\}$
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(0 < x = i < 9 \Rightarrow reach(\ell_1))$ *is sat,*

Example: symbolic model checking

Example 3.11

Consider the following example



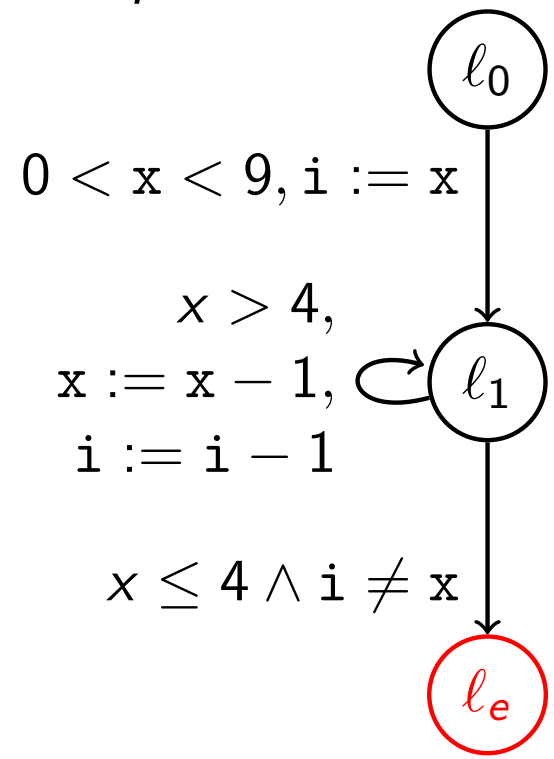
Init: $reach = \lambda x.\bot$, $worklist = \{(\ell_0, \top)\}$
Choose a state: $(\ell_0, \top)$ *(only choice)*
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(\top \Rightarrow reach(\ell_0))$ *is sat,*
$worklist := worklist \cup \{(\ell_1, 0 < x = i < 9)\}$
$reach(\ell_0) := reach(\ell_0) \vee \top := \top$
Again choose a state: $\{(\ell_1, 0 < x = i < 9)\}$
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(0 < x = i < 9 \Rightarrow reach(\ell_1))$ *is sat,*
$worklist := worklist \cup \{(\ell_1, 3 < x = i < 9), (\ell_e, \bot)\}$

Let $V = [\mathtt{x}, \mathtt{i}]$

Example: symbolic model checking

Example 3.11

Consider the following example



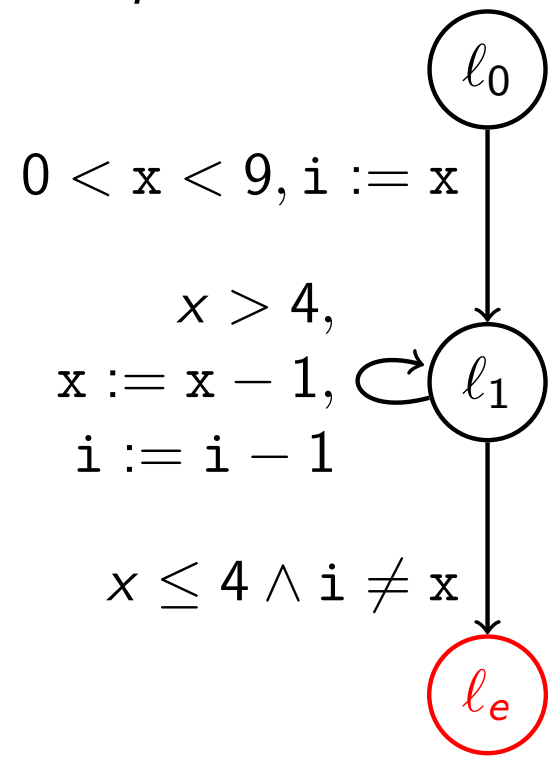
Let $V = [\mathtt{x}, \mathtt{i}]$

Init: $reach = \lambda x.\bot$, $worklist = \{(\ell_0, \top)\}$
Choose a state: $(\ell_0, \top)$ *(only choice)*
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(\top \Rightarrow reach(\ell_0))$ *is sat,*
$worklist := worklist \cup \{(\ell_1, 0 < x = i < 9)\}$
$reach(\ell_0) := reach(\ell_0) \vee \top := \top$
Again choose a state: $\{(\ell_1, 0 < x = i < 9)\}$
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(0 < x = i < 9 \Rightarrow reach(\ell_1))$ *is sat,*
$worklist := worklist \cup \{(\ell_1, 3 < x = i < 9), (\ell_e, \bot)\}$
$reach(\ell_1) := reach(\ell_1) \vee 0 < x = i < 9$
$reach(\ell_e) := reach(\ell_e) \vee \bot$

Example: symbolic model checking

Example 3.11

Consider the following example



Let $V = [\mathtt{x}, \mathtt{i}]$

Init: $reach = \lambda x.\bot$, $worklist = \{(\ell_0, \top)\}$
Choose a state: $(\ell_0, \top)$ *(only choice)*
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(\top \Rightarrow reach(\ell_0))$ *is sat,*
  $worklist := worklist \cup \{(\ell_1, 0 < x = i < 9)\}$
  $reach(\ell_0) := reach(\ell_0) \vee \top := \top$
Again choose a state: $\{(\ell_1, 0 < x = i < 9)\}$
Update worklist: $worklist := \emptyset$
Add successors in worklist:
Since $\neg(0 < x = i < 9 \Rightarrow reach(\ell_1))$ *is sat,*
  $worklist := worklist \cup \{(\ell_1, 3 < x = i < 9), (\ell_e, \bot)\}$
  $reach(\ell_1) := reach(\ell_1) \vee 0 < x = i < 9$
  $reach(\ell_e) := reach(\ell_e) \vee \bot$

Exercise 3.8

complete the run of the algorithm

Proof generation

If the symbolic model checker terminates with the answer SAFE, then it must also report a proof of the safety, which is the *reach* map.

Proof generation

If the symbolic model checker terminates with the answer SAFE, then it must also report a proof of the safety, which is the *reach* map.

It has implicitly computed a Hoare style proof of $P = (V, L, \ell_0, \ell_e, E)$.

$$(\ell, \rho(V, V'), \ell') \in E \quad \{reach(\ell)\}\rho(V, V')\{reach(\ell')\}$$

Proof generation

If the symbolic model checker terminates with the answer SAFE, then it must also report a proof of the safety, which is the *reach* map.

It has implicitly computed a Hoare style proof of $P = (V, L, \ell_0, \ell_e, E)$.

$$(\ell, \rho(V, V'), \ell') \in E \quad \{reach(\ell)\}\rho(V, V')\{reach(\ell')\}$$

If an LTS program has been obtained from a simple language program then one may generate a Hoare style proof system.

Proof generation

If the symbolic model checker terminates with the answer SAFE, then it must also report a proof of the safety, which is the *reach* map.

It has implicitly computed a Hoare style proof of $P = (V, L, \ell_0, \ell_e, E)$.

$$(\ell, \rho(V, V'), \ell') \in E \quad \{reach(\ell)\}\rho(V, V')\{reach(\ell')\}$$

If an LTS program has been obtained from a simple language program then one may generate a Hoare style proof system.

Exercise 3.9

Describe the construction for the above translation

Topic 3.6

Constraint based invariant generation

Invariant generation using constraint solving

Invariant generation: find a safe inductive invariant map I

Invariant generation using constraint solving

Invariant generation: find a safe inductive invariant map I

- This is our first method that computes the fixed point automatically without resorting to some kind of enumeration

Templates

Let $L = \{l_0, \ldots, l_n, l_e\}$,
Let $V = \{x_1, \ldots, x_m\}$

Templates

Let $L = \{l_0, \ldots, l_n, l_e\}$,
Let $V = \{x_1, \ldots, x_m\}$

We assume the following templates for each invariant in the invariant map.

$$\begin{aligned} \mathrm{I}(l_0) &= 0 \leq 0 \\ \forall i \in 1..n.\ \mathrm{I}(l_i) &= (p_{i1}x_1 + \ldots p_{im}x_m \leq p_{i0}) \\ \mathrm{I}(l_e) &= 0 \leq -1 \end{aligned}$$

p_{ij} are called parameters to the templates and they define a set of candidate invariants.

Constraint generation

A safe inductive invariant map I must satisfy for all $(l_i, \rho, l_{i'}) \in E$

$$sp(\mathrm{I}(l_i), \rho) \Rightarrow \mathrm{I}(l_{i'}).$$

The above condition translates to

$$\forall V, V'.\ (p_{i1}x_1 + \ldots p_{im}x_m \leq p_{i0}) \wedge \rho(V, V') \Rightarrow (p_{i'1}x'_1 + \ldots p_{i'm}x'_m \leq p_{i'0})$$

Our goal is to find p_{ij}s such that the above constraints are satisfied. Unfortunately there is quantifier alternation in the constraints. Therefore, they are hard to solve.

Constraint solving using Farkas lemma

If all ρs are linear constraints then we can use Farkas lemma to turn the validity question into a "conjunctive satisfiablity question"

Lemma 3.1

For a rational matrix A, vectors a and b, and constant c,
$\forall X.\ AX \leq b \Rightarrow aX \leq c$ *iff*
$\exists \lambda \geq 0.\ \lambda^T A = a$ *and* $\lambda^T b \leq c$

Application of farkas lemma

Consider $(l_i, (AV + A'V \leq b), l_{i'}) \in E$

After applying Farkas lemma on

$$\forall V, V'.\ (p_{i1}x_1 + \ldots p_{im}x_m \leq p_{i0}) \wedge \rho(V, V') \Rightarrow (p_{i'1}x'_1 + \ldots p_{i'm}x'_m \leq p_{i'0}),$$

we obtain

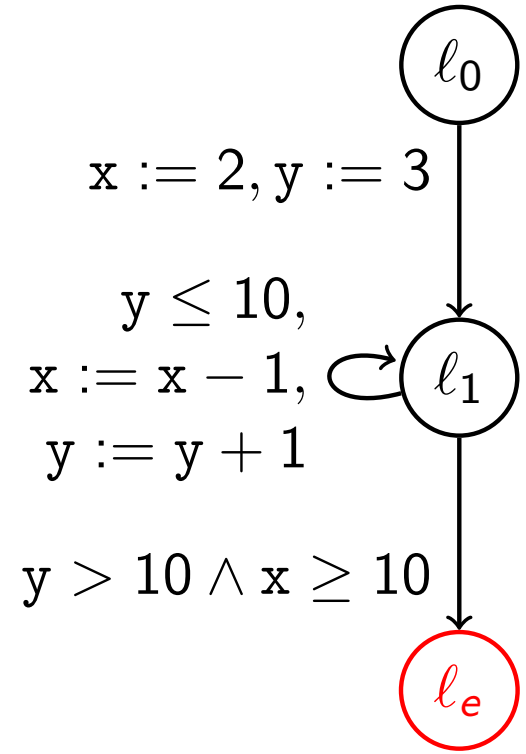
$$\exists \lambda_0, \lambda.\ (\lambda_0[p_{i1}, \ldots, p_{im}] + \lambda^T A) = 0 \wedge \lambda^T A' = [p_{i'1}, \ldots, p_{i'm}] \wedge \lambda_0 p_{i0} + \lambda^T b \leq p_{i'0}$$

All the variables p_{ij}s and λs are existentially quantified, which can be solved by a quadratic constraints solver.

Example: invariant generation

Example 3.12

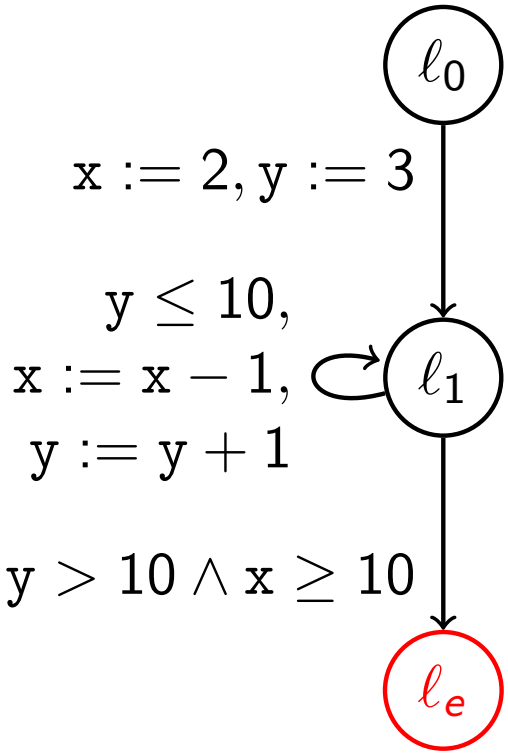
Consider the following example



Example: invariant generation

Example 3.12

Consider the following example

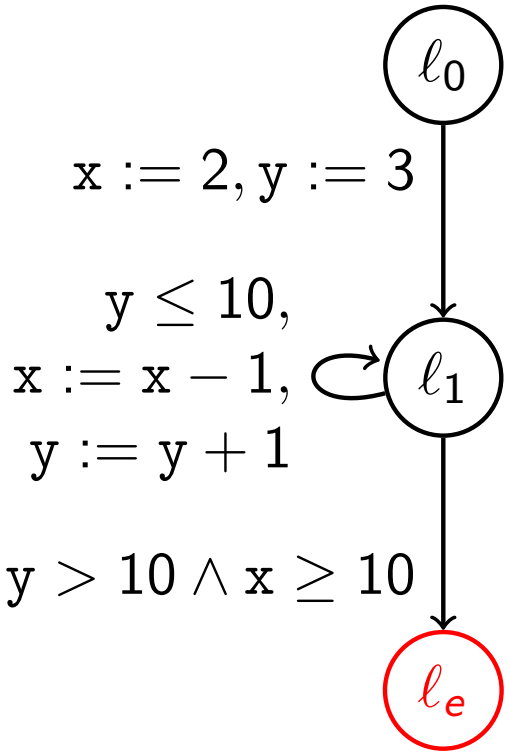


Let $V = [\mathrm{x}, \mathrm{y}]$

Example: invariant generation

Example 3.12

Consider the following example



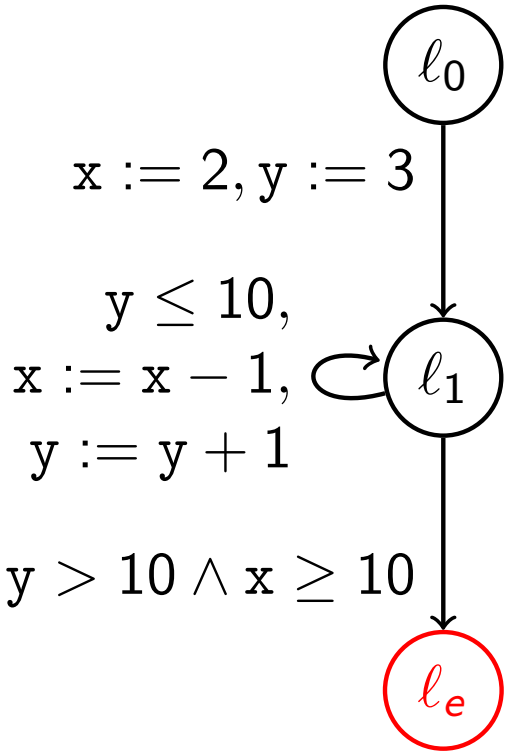
We assume the following invariant template at ℓ_1:
$\mathrm{I}(\ell_1) = (p_1\mathrm{x} + p_2\mathrm{y} \leq p_0)$

Let $V = [\mathrm{x}, \mathrm{y}]$

Example: invariant generation

Example 3.12

Consider the following example



We assume the following invariant template at ℓ_1:
$\mathrm{I}(\ell_1) = (p_1\mathrm{x} + p_2\mathrm{y} \leq p_0)$

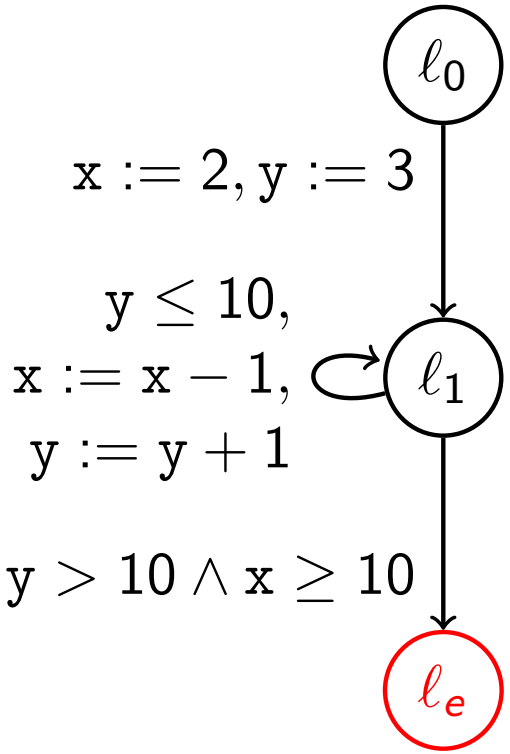
We generate the following constraints for program transitions:

Let $V = [\mathrm{x}, \mathrm{y}]$

Example: invariant generation

Example 3.12

Consider the following example



We assume the following invariant template at ℓ_1:
$\mathrm{I}(\ell_1) = (p_1\mathrm{x} + p_2\mathrm{y} \leq p_0)$

We generate the following constraints for program transitions:

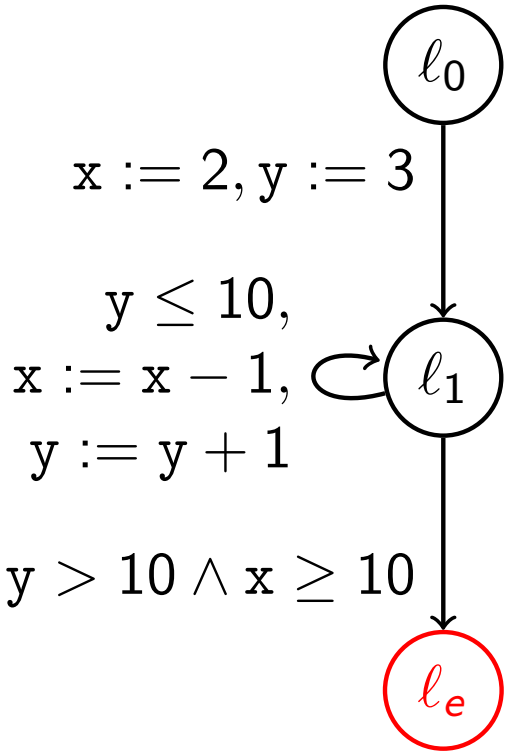
For ℓ_0 to ℓ_1,
$\forall \mathrm{x}', \mathrm{y}'.\ \mathrm{x}' = 2 \wedge \mathrm{y}' = 3 \Rightarrow (p_1\mathrm{x}' + p_2\mathrm{y}' \leq p_0)$

Let $V = [\mathrm{x}, \mathrm{y}]$

Example: invariant generation

Example 3.12

Consider the following example



We assume the following invariant template at ℓ_1:
$\mathrm{I}(\ell_1) = (p_1\mathrm{x} + p_2\mathrm{y} \leq p_0)$

We generate the following constraints for program transitions:

For ℓ_0 to ℓ_1,
$\forall \mathrm{x}', \mathrm{y}'.\ \mathrm{x}' = 2 \wedge \mathrm{y}' = 3 \Rightarrow (p_1\mathrm{x}' + p_2\mathrm{y}' \leq p_0)$

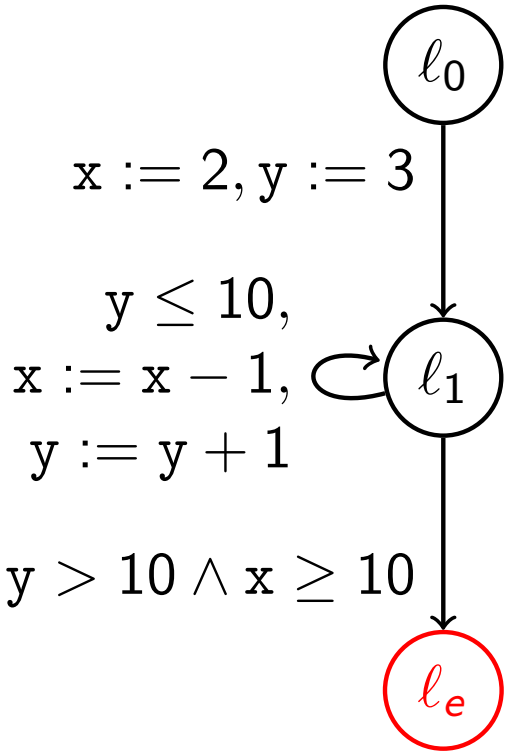
For ℓ_1 to ℓ_1,
$\forall \mathrm{x}, \mathrm{y}, \mathrm{x}', \mathrm{y}'.\ (p_1\mathrm{x} + p_2\mathrm{y} \leq p_0) \wedge \mathrm{y} \leq 10 \wedge \mathrm{x}' = \mathrm{x} - 1 \wedge \mathrm{y}' = \mathrm{y} + 1 \Rightarrow (p_1\mathrm{x}' + p_2\mathrm{y}' \leq p_0)$

Let $V = [\mathrm{x}, \mathrm{y}]$

Example: invariant generation

Example 3.12

Consider the following example



Let $V = [\mathrm{x}, \mathrm{y}]$

We assume the following invariant template at ℓ_1*:*
$\mathrm{I}(\ell_1) = (p_1\mathrm{x} + p_2\mathrm{y} \leq p_0)$

We generate the following constraints for program transitions:

For ℓ_0 *to* ℓ_1*,*
$\forall \mathrm{x}', \mathrm{y}'.\ \mathrm{x}' = 2 \wedge \mathrm{y}' = 3 \Rightarrow (p_1\mathrm{x}' + p_2\mathrm{y}' \leq p_0)$

For ℓ_1 *to* ℓ_1*,*
$\forall \mathrm{x}, \mathrm{y}, \mathrm{x}', \mathrm{y}'.\ (p_1\mathrm{x} + p_2\mathrm{y} \leq p_0) \wedge \mathrm{y} \leq 10 \wedge \mathrm{x}' = \mathrm{x} - 1 \wedge \mathrm{y}' = \mathrm{y} + 1 \Rightarrow (p_1\mathrm{x}' + p_2\mathrm{y}' \leq p_0)$

For ℓ_1 *to* ℓ_e*,*
$\forall \mathrm{x}, \mathrm{y}.\ (p_1\mathrm{x} + p_2\mathrm{y} \leq p_0) \wedge \mathrm{y} > 10 \wedge \mathrm{x} \geq 10 \Rightarrow \bot$

Example: invariant generation(contd.)

Now consider the second constraint:

$\forall \mathrm{x}, \mathrm{y}, \mathrm{x}', \mathrm{y}'.$

$(p_1\mathrm{x} + p_2\mathrm{y} \leq p_0) \wedge \mathrm{y} \leq 10 \wedge \mathrm{x}' = \mathrm{x} - 1 \wedge \mathrm{y}' = \mathrm{y} + 1 \Rightarrow (p_1\mathrm{x}' + p_2\mathrm{y}' \leq p_0)$

Example: invariant generation(contd.)

Now consider the second constraint:
$\forall \mathrm{x}, \mathrm{y}, \mathrm{x}', \mathrm{y}'.$
$(p_1\mathrm{x} + p_2\mathrm{y} \leq p_0) \wedge \mathrm{y} \leq 10 \wedge \mathrm{x}' = \mathrm{x} - 1 \wedge \mathrm{y}' = \mathrm{y} + 1 \Rightarrow (p_1\mathrm{x}' + p_2\mathrm{y}' \leq p_0)$

Matrix view of the transition relation $\mathrm{y} \leq 10 \wedge \mathrm{x}' = \mathrm{x} - 1 \wedge \mathrm{y}' = \mathrm{y} + 1$

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathrm{x} \\ \mathrm{y} \\ \mathrm{x}' \\ \mathrm{y}' \end{bmatrix} \leq \begin{bmatrix} 10 \\ 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

Example: invariant generation(contd.)

Applying farkas lemma on the constraint, we obtain

$$\begin{bmatrix} \lambda_0 & \lambda_1 & \lambda_2 & \lambda_3 & \lambda_4 & \lambda_5 \end{bmatrix} \begin{bmatrix} p_1 & p_2 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & p_1 & p_2 \end{bmatrix}$$

$$\begin{bmatrix} \lambda_0 & \lambda_1 & \lambda_2 & \lambda_3 & \lambda_4 & \lambda_5 \end{bmatrix} \begin{bmatrix} p_0 \\ 10 \\ 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} \leq \begin{bmatrix} p_0 \end{bmatrix}$$

Exercise 3.10

Apply farkas lemma on the other two implications

$\forall \mathrm{x}', \mathrm{y}'.\ \mathrm{x}' = 2 \wedge \mathrm{y}' = 3 \Rightarrow (p_1\mathrm{x}' + p_2\mathrm{y}' \leq p_0)$

$\forall \mathrm{x}, \mathrm{y}.\ (p_1\mathrm{x} + p_2\mathrm{y} \leq p_0) \wedge \mathrm{y} > 10 \wedge \mathrm{x} \geq 10 \Rightarrow \bot$

Does this method work?

- Quadratic constraint solving does not scale
- For small tricky problems, this method may prove to be useful

Topic 3.7

Assignment

Problem 1

1. (1) Prove the following Hoare triple is valid

```
{true}
assume( n > 1);
i = n;
x = 0;
while(i > 0) {
  x = x + i;
  i = i - 1;
}
{ 2x = n*(n+1) }
```

Problem 2

2. (1) Fill the annotations to prove following program correct via Vcc

```
#include <vcc.h>
int main()
{
  int x = 0, y = 2;
 _(assume 1==1 )
  while( x < 3 ) _(invariant ... ) {
    x = x + 1;
    y = 3;
   }
  _(assert y == 3)
  return 0;
}
```

Problem 3

3. (2) extend your tool in the last assignment in the following ways
   - define classes for
     - locations,
     - variables,
     - guarded commands,
     - transitions (give names to the transitions), and
     - programs
   - encode the program in example 3.12 using the class
   - Write a function that computes path constraints for a given path
   - Read path from command line as space separated transition names and output the path constraints