Difficulty of String Analysis, Reachability & Fixpoints

292C
Tevfik Bultan
A simple string manipulation language

- Language syntax

  \[
  \begin{align*}
  \text{prog} & \rightarrow (\text{stmt})^+ \\
  \text{stmt} & \rightarrow l : \text{stmt} \\
  \text{stmt} & \rightarrow v := \text{sexp}; \\
  \text{stmt} & \rightarrow \text{if } bexp \text{ then goto } l; \\
  \text{stmt} & \rightarrow \text{goto } l; \\
  \text{stmt} & \rightarrow \text{read } v; \\
  \text{stmt} & \rightarrow \text{print } \text{sexp}; \\
  \text{stmt} & \rightarrow \text{assert } bexp; \\
  \text{stmt} & \rightarrow \text{halt}; \\
  bexp & \rightarrow v = \text{sexp} | bexp \land bexp | bexp \lor bexp | \neg bexp \\
  \text{sexp} & \rightarrow v | "c" | \text{sexp} . \text{sexp}
  \end{align*}
  \]

- Example code

  1: read x1;
  2: read x2;
  3: x1 := x1 . "a";
  4: x2 := x2 . "a";
  5: if (x1 = x2) goto 7;
  6: print x1 . x2;
  7: print x1;
Reachability problem

• Reachability problem in string programs:
  – Given a string program P and a program state s
    • where a program state s is defined with the instruction label of an instruction in the program and the values of all the variables,
    • determine if at some point during the execution of the program P, the program state s will be reached.

• Reachability problem for string programs is undecidable (even if we allow only 3 string variables)
Counter machines

- Counter machines are a simple and powerful computational model that can simulate Turing Machines.
- A counter machine consists of a finite number of counters (unbounded integer variables) and a finite set of instructions.
- Counter machines have a very small instruction set that includes an increment, a decrement, a conditional branch instruction that tests if a counter value is equal to zero, and a halt instruction.
- The counters can only assume nonnegative values.
- It is well-known that the halting problem for two-counter machines, where both counters are initialized to 0, is undecidable.
- Two counter machines can simulate Turing Machines.
String programs can simulate counter machines

- A string program $P$ with three string variables ($X_1, X_2, X_3$) can simulate a counter machine $M$ with two counters ($C_1, C_2$)

- We will use the lengths of the strings $X_1, X_2$ and $X_3$ to simulate the values of the counters $C_1$ and $C_2$

Where

\[
C_1 = |X_1| - |X_3|
\]
\[
C_2 = |X_2| - |X_3|
\]
String programs can simulate counter machines

- M starts from the initial configuration $(q_0, 0, 0)$ where $q_0$ denotes the initial instruction and the two integer values represent the initial values of counters $C_1$ and $C_2$, respectively.

- The initial state of the string program $P$ will be $(q_0, \varepsilon, \varepsilon, \varepsilon)$ where $q_0$ is the label of the first instruction, and the string variables $X_1$, $X_2$, and $X_3$, are initialized to empty string: $\varepsilon$
Translation of counter-machine instructions to string program instructions

<table>
<thead>
<tr>
<th>Counter machine instruction</th>
<th>String program simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>inc $C_1$</td>
<td>$X_1 := X_1.a;$</td>
</tr>
<tr>
<td>inc $C_2$</td>
<td>$X_2 := X_2.a;$</td>
</tr>
<tr>
<td>dec $C_1$</td>
<td>$X_2 := X_2.a;$</td>
</tr>
<tr>
<td>dec $C_2$</td>
<td>$X_3 := X_3.a;$</td>
</tr>
<tr>
<td>if ($C_1 = 0$)</td>
<td>if ($X_1 = X_3$)</td>
</tr>
<tr>
<td>if ($C_2 = 0$)</td>
<td>if ($X_2 = X_3$)</td>
</tr>
<tr>
<td>halt</td>
<td>halt;</td>
</tr>
</tbody>
</table>
Reachability problem

- Halting problem for counter machines is undecidable
- String programs can simulate counter machines
- Hence, halting problem for string programs is undecidable.
- Hence, reachability problem for string programs is undecidable.
A richer string manipulating language

\[
\begin{align*}
prog & \rightarrow block \\
block & \rightarrow stmt^+ \\
stmt & \rightarrow l : stmt \\
stmt & \rightarrow v := \exp; \\
& \mid \text{read} v; \\
& \mid \text{print} \exp; \\
& \mid \text{assert} \exp; \\
& \mid \text{halt}; \\
& \mid \text{if} (bexp) \text{ then } \{ \text{block} \} \\
& \mid \text{if} (bexp) \text{ then } \{ \text{block} \} \text{ else } \{ \text{block} \} \\
& \mid \text{while} (bexp) \{ \text{block} \} \\
exp & \rightarrow \exp | iexp \\
bexp & \rightarrow \exp = \exp \\
& \mid \text{match} (\exp, \exp) \\
& \mid \text{contains} (\exp, \exp) \\
& \mid \text{begins} (\exp, \exp) \\
& \mid \text{ends} (\exp, \exp) \\
& \mid iexp = iexp | iexp < iexp | iexp > iexp \\
& \mid bexp \land bexp | bexp \lor bexp | \neg bexp \\
iexp & \rightarrow v | n | iexp + iexp | iexp - iexp \\
& \mid \text{length} (\exp) \\
& \mid \text{indexof} (\exp, \exp) \\
sexp & \rightarrow v | \text{"c"} | \exp.\exp | \exp^* | \exp|\exp \\
& \mid \text{replace} (\exp, \exp, \exp) \\
& \mid \text{substring} (\exp, iexp, iexp) \\
& \mid \text{charat} (\exp, iexp) \\
& \mid \text{reverse} (\exp)
\end{align*}
\]
Semantics

\[
\text{match}(s, r) \iff s \in \mathcal{L}(r)
\]

\[
\text{contains}(s, t) \iff \exists s_1, s_2 \in \Sigma^* : s = s_1ts_2
\]

\[
\text{begins}(s, t) \iff \exists s_1 \in \Sigma^* : s = ts_1
\]

\[
\text{ends}(s, t) \iff \exists s_1 \in \Sigma^* : s = s_1t
\]
Semantics

\[ t = \text{substring}(s, i, j) \iff \exists s_1, s_2 \in \Sigma^* : s = s_1ts_2 \land |s_1| = i \land |t| = j - i \]

\[ t = \text{charat}(s, i) \iff \exists s_0, s_1, \ldots, s_n \in \Sigma : s = s_0s_1\ldots s_n \land 0 \leq i \leq n \land t = s_i \]

\[ t = \text{reverse}(s) \iff \exists s_0, s_1, \ldots, s_i \in \Sigma : s = s_0s_1\ldots s_i \land t = s_i\ldots s_1s_0 \]
Semantics

\[(\text{length}(s) = 0 \iff s = \epsilon) \land (\text{length}(s) = n \iff \exists c_1, c_2, \ldots, c_n \in \Sigma : s = c_1c_2\ldots c_n)\]

\[(\text{indexof}(s,t) = -1 \iff \neg \text{contains}(s,t)) \land\]
\[(\text{indexof}(s,t) = n \iff (\exists s_1, s_2 \in \Sigma^* : s = s_1ts_2 \land |s_1| = n) \land (\forall i < n : \neg (\exists s_1, s_2 \in \Sigma^* : s = s_1ts_2 \land |s_1| = i)))\]
Semantics

\[ r = \text{replace}(s, p, t) \iff ((\neg \text{contains}(s, p) \land r = s) \lor \\
(\exists s_3, s_4, s_5 \in \Sigma^*: s = s_3ps_4 \land r = s_3ts_5 \land s_5 = \text{replace}(s_4, p, t) \land \\
(\forall s_6, s_7 \in \Sigma^*: s = s_6ps_7 \Rightarrow |s_6| \geq |s_3|))) \]
Semantics of a string program

• Semantics of a string program can be defined as a transition system

• A *transition system* \( T = (S, I, R) \) consists of
  – a set of states \( S \)
  – a set of initial states \( I \subseteq S \)
  – and a transition relation \( R \subseteq S \times S \)
Semantics of a string program

Let $L$ denote the labels of program statements, and assume $n$ string and $m$ integer variables, then the set of states of the string program can be defined as:

$$S = L \times (\Sigma^*)^n \times (\mathbb{Z})^m$$

and the initial state is (where $l_1$ is the label of the first statement):

$$I = \{ \langle l_1, \epsilon, \ldots, \epsilon, 0, \ldots, 0 \rangle \}$$
Semantics of a string program

- Given a statement labeled \( l \), its transition relation can be defined as a set of tuples:

\[
  r_l \subseteq S \times S
\]

where \( (s_1, s_2) \in r_l \) means that executing statement \( l \) in state \( s_1 \) results in state \( s_2 \).

- Then, the transition relation of the whole program can be defined as:

\[
  R = \bigcup_{l \in L} r_l
\]
Post condition function

• Using the transition relation, we can define the post condition function that identifies, given a state which state the program will transition.

\[
\begin{align*}
s_2 = \text{POST}(s_1, l) & \iff (s_1, s_2) \in r_l \\
s_2 = \text{POST}(s_1) & \iff \exists l \in L : s_2 = \text{POST}(s_1, r_l) \\
s_2 = \text{POST}(s_1) & \iff (s_1, s_2) \in R
\end{align*}
\]
Computing reachable states

- The set of states that are reachable from the initial states of the program can be defined as:

\[ RS = \{ s \mid \exists s_0, s_1, \ldots, s_n : \forall i < n : (s_i, s_{i+1}) \in R \land s_0 \in I \land s_n = s \} \]

- Reachable states can be computed using a simple depth-first-search
Computing reachable states with DFS

**Algorithm 1** REACHABILITYDFS

1: Stack := I;
2: RS := I;
3: while Stack ≠ ∅ do
4:     s := POP(Stack);
5:     s′ := POST(s);
6:     if s′ ∉ RS then
7:         RS := RS ∪ {s′};
8:         PUSH(Stack, s′);
9:     end if
10: end while
11: return RS;
Pre-condition function

\[ s_2 \in \text{PRE}(s_1, l) \iff (s_2, s_1) \in r_l \]
\[ s_2 \in \text{PRE}(s_1) \iff \exists l \in L : s_2 \in \text{PRE}(s_1, r_l) \]
\[ s_2 \in \text{PRE}(s_1) \iff (s_2, s_1) \in R \]
Backward reachability using DFS

Algorithm 2 BACKWARDREACHABILITYDFS(P)

1: Stack := P;
2: BRS := P;
3: while Stack ≠ ∅ do
4:     s := POP(Stack);
5:     for s' ∈ PRE(s) do
6:         if s' ∉ BRS then
7:             BRS := BRS ∪ {s'};
8:             PUSH(Stack, s');
9:         end if
10:     end for
11: end while
12: return BRS;
Explicit vs. Symbolic reachability analysis

• The DFS algorithms that we showed work on one state at a time. This is called explicit state (or enumerative, or concrete) reachability analysis.

• It is not feasible to enumerate each state since state space of a program is exponential in the number of variables.

• Symbolic reachability analysis works on sets of states, rather than a single state at a time.

• We need to generalize pre and post condition functions so that they work on sets of states.
Post and pre condition

\[
\text{POST}(P, l) = \{ s \mid \exists s' \in P : (s', s) \in r_l \} \\
\text{POST}(P) = \{ s \mid \exists s' \in P : (s', s) \in R \} \\
\text{PRE}(P, l) = \{ s \mid \exists s' \in P : (s, s') \in r_l \} \\
\text{PRE}(P) = \{ s \mid \exists s' \in P : (s, s') \in R \}
\]
Symbolic Reachability Analysis

Algorithm 3 \textsc{ReachabilityFixpoint}

1: \( RS := I; \)
2: \ repeat
3: \quad RS' := RS; \\
4: \quad RS := RS \cup \text{post}(RS); \\
5: \ until \ RS = RS' \\
6: \ return RS;

Algorithm 4 \textsc{BackwardReachabilityFixpoint}(P)

1: \( BRS := P; \)
2: \ repeat
3: \quad BRS' := BRS; \\
4: \quad BRS := BRS \cup \text{pre}(BRS); \\
5: \ until \ BRS = BRS' \\
6: \ return BRS;
Reachability and fixpoints

• We will demonstrate that reachability analysis corresponds to computing the least fixpoint of a function.

• In order to do that we need to introduce the concept of a lattice
Pre and post condition functions on sets of states

- Given a transition system \( T=(S, I, R) \), we define functions from sets of states to sets of states
  \[ F : 2^S \rightarrow 2^S \]
- For example, one such function is the post function (which computes the post-condition of a set of states)
  \[ \text{post} : 2^S \rightarrow 2^S \]
  which can be defined as (where \( P \subseteq S \)):
  \[ \text{Post}(P) = \{ s' \mid (s,s') \in R \text{ and } s \in P \} \]
- We can similarly define the pre function (which computes the pre-condition of a set of states)
  \[ \text{pre} : 2^S \rightarrow 2^S \]
  which can be defined as:
  \[ \text{Pre}(P) = \{ s \mid (s,s') \in R \text{ and } s' \in P \} \]
Lattices

The set of states of the transition system forms a lattice:

• lattice \( 2^S \)
• partial order \( \subseteq \)
• bottom element \( \emptyset \) (alternative notation: \( \bot \))
• top element \( S \) (alternative notation: \( T \))
• Least upper bound (lub) \( \cup \)
  (aka join) operator
• Greatest lower bound (glb) \( \cap \)
  (aka meet) operator
Lattices

In general, a lattice is a partially ordered set with a least upper bound operation and a greatest lower bound operation.

- Least upper bound $a \cup b$ is the smallest element where $a \subseteq a \cup b$ and $b \subseteq a \cup b$
- Greatest lower bound $a \cap b$ is the biggest element where $a \cap b \subseteq a$ and $a \cap b \subseteq b$

A partial order is a
- reflexive (for all $x$, $x \subseteq x$),
- transitive (for all $x, y, z$, $x \subseteq y \land y \subseteq z \Rightarrow x \subseteq z$), and
- antisymmetric (for all $x, y$, $x \subseteq y \land y \subseteq x \Rightarrow x = y$) relation.
Complete Lattices

$2^S$ forms a lattice with the partial order defined as the subset-or-equal relation and the least upper bound operation defined as the set union and the greatest lower bound operation defined as the set intersection.

In fact, $(2^S, \subseteq, \emptyset, S, \cup, \cap)$ is a complete lattice since for each set of elements from this lattice there is a least upper bound and a greatest lower bound.

Also, note that the top and bottom elements can be defined as:

$\bot = \emptyset = \cap \{ y \mid y \in 2^S \}$
$\top = S = \cup \{ y \mid y \in 2^S \}$

This definition is valid for any complete lattice.
An Example Lattice

\{\emptyset, \{0\}, \{1\}, \{2\}, \{0,1\},\{0,2\},\{1,2\},\{0,1,2\}\}

partial order: \(\subseteq\) (subset relation)

bottom element: \(\emptyset = \bot\)  top element: \(\{0,1,2\} = T\)

lub: \(\cup\) (union)  glb: \(\cap\) (intersection)

\{0,1,2\} = T (top element)

\emptyset = \bot (bottom element)

The Hasse diagram for the example lattice (shows the transitive reduction of the corresponding partial order relation)
What is a Fixpoint (aka, Fixed Point)

Given a function

\[ \mathcal{F} : D \rightarrow D \]

\( x \in D \) is a fixpoint of \( \mathcal{F} \) if and only if \( \mathcal{F}(x) = x \)
Reachability

Let $RS(I)$ denote the set of states reachable from the initial states $I$ of the transition system $T = (S, I, R)$

In general, given a set of states $P \subseteq S$, we can define the reachability function as follows:

$RS(P) = \{ s_n \mid s_n \in P, \text{ or there exists } s_0s_1\ldots s_n \in S,$
\hspace{1cm} where for all $0 \leq i < n \ (s_i, s_{i+1}) \in R, \text{ and } s_0 \in P \}$

We can also define the backward reachability function $BRS$ as follows:

$BRS(P) = \{ s_0 \mid s_0 \in P, \text{ or there exists } s_0s_1\ldots s_n \in S,$
\hspace{1cm} where for all $0 \leq i < n \ (s_i, s_{i+1}) \in R, \text{ and } s_n \in P \}$
Reachability ≡ Fixpoints

Here is an interesting property

\[ RS(P) = P \cup \text{post}(RS(P)) \]

we observe that \( RS(P) \) is a fixpoint of the following function:

\[ Fy = P \cup \text{post}(y) \] (we can also write it as \( \lambda y . P \cup \text{post}(y) \))

\[ F(RS(P)) = RS(P) \]

In fact, \( RS(P) \) is the least fixpoint of \( F \), which is written as:

\[ RS(P) = \mu y . Fy = \mu y . P \cup \text{post}(y) \]

(\( \mu \) means least fixpoint)
Reachability $\equiv$ Fixpoints

We have the same property for backward reachability

$$\text{BRS}(P) = P \cup \text{pre}(\text{RS}(P))$$

i.e., BRS(P) is a fixpoint of the following function:

$$F y = P \cup \text{pre}(y) \quad \text{(we can also write it as } \lambda y . P \cup \text{pre}(y))$$

$$F (\text{RS}(P)) = \text{RS}(P)$$

In fact, BRS(P) is the least fixpoint of $F$, which is written as:

$$\text{BRS}(P) = \mu y . F y = \mu y . P \cup \text{pre}(y)$$
RS(P) = \( \mu y . P \cup RS(y) \)

- Let’s prove this.

- First we have the equivalence \( RS(P) = P \cup post(RS(P)) \)
  - Why? Because according to the definition of \( RS(P) \), a state is in \( RS(P) \) if that state is in \( P \), or if that state has a previous state which is in \( RS(P) \).
  - From this equivalence we know that \( RS(P) \) is a fixpoint of the function \( \lambda y . P \cup post(y) \) and since the least fixpoint is the smallest fixpoint we have:
    \( \mu y . P \cup post(y) \subseteq RS(P) \)
\[ RS(P) = \mu y . P \cup RS(y) \]

- Next we need to prove that \( RS(P) \subseteq \mu y . P \cup RS(y) \) to complete the proof.
- Suppose \( z \) is a fixpoint of \( \lambda y . P \cup RS(y) \), then we know that \( z = P \cup RS(z) \) which means that \( RS(z) \subseteq z \) and this means that no state that is reachable from \( z \) is outside of \( z \).
- Since we also have \( P \subseteq z \), any path that is reachable from \( P \) must be in \( z \).

Hence, we can conclude that \( RS(P) \subseteq z \).

Since we showed that \( RS(P) \) is contained in any fixpoint of the function \( \lambda y . P \cup RS(y) \), we get
\[ RS(P) \subseteq \mu y . P \cup RS(y) \]
which completes the proof.
Monotonicity

• Function $\mathcal{F}$ is monotonic if and only if, for any $x$ and $y$, $x \subseteq y \Rightarrow \mathcal{F}x \subseteq \mathcal{F}y$

Note that, 
$\lambda y \cdot P \cup \text{post}(y)$ 
$\lambda y \cdot P \cup \text{pre}(y)$
are monotonic.

For both these functions, if you give a bigger $y$ as input you will get a bigger result as output.
Monotonicity

- One can define non-monotonic functions:
  For example: \( \lambda y . P \cup \text{post}(S - y) \)
  This function is not monotonic. If you give a bigger \( y \) as input you will get a smaller result.

- For the functions that are non-monotonic the fixpoint computation techniques we are going to discuss will not work. For such functions a fixpoint may not even exist.

- The functions we defined for reachability are monotonic because we are applying monotonic operations (like post and \( \cup \)) to the input variable \( y \).

- Set complement – is not monotonic. However, if you have an even number of negations in front of the input variable \( y \), then you will get a monotonic function.
Least Fixpoint

Given a monotonic function \( \mathcal{F} \), its least fixpoint exists, and it is the greatest lower bound (glb) of all the reductive elements:

\[
\mu y . \mathcal{F} y = \cap \{ y | \mathcal{F} y \subseteq y \}
\]
\[ \mu y. \mathcal{F} y = \cap \{ y \mid \mathcal{F} y \subseteq y \} \]

- Let’s prove this property.
- Let us define \( z \) as \( z = \cap \{ y \mid \mathcal{F} y \subseteq y \} \)

We will first show that \( z \) is a fixpoint of \( \mathcal{F} \) and then we will show that it is the least fixpoint which will complete the proof.

- Based on the definition of \( z \), we know that:
  - for any \( y \), \( \mathcal{F} y \subseteq y \), we have \( z \subseteq y \).

Since \( \mathcal{F} \) is monotonic, \( z \subseteq y \Rightarrow \mathcal{F} z \subseteq \mathcal{F} y \).

But since \( \mathcal{F} y \subseteq y \), then \( \mathcal{F} z \subseteq y \).

I.e., for all \( y \), \( \mathcal{F} y \subseteq y \), we have \( \mathcal{F} z \subseteq y \).

This implies that, \( \mathcal{F} z \subseteq \cap \{ y \mid \mathcal{F} y \subseteq y \} \),
and based on the definition of \( z \), we get \( \mathcal{F} z \subseteq z \).
\[ \mu y. \mathcal{F} y = \cap \{ y | \mathcal{F} y \subseteq y \} \]

- Since \( \mathcal{F} \) is monotonic and since \( \mathcal{F} z \subseteq z \), we have \( \mathcal{F}(\mathcal{F} z) \subseteq \mathcal{F} z \) which means that \( \mathcal{F} z \in \{ y | \mathcal{F} y \subseteq y \} \).
  Then by definition of \( z \) we get, \( z \subseteq \mathcal{F} z \)

- Since we showed that \( \mathcal{F} z \subseteq z \) and \( z \subseteq \mathcal{F} z \), we conclude that \( \mathcal{F} z = z \), i.e., \( z \) is a fixpoint of the function \( \mathcal{F} \).

- For any fixpoint of \( \mathcal{F} \) we have \( \mathcal{F} y = y \) which implies \( \mathcal{F} y \subseteq y \)
  So any fixpoint of \( \mathcal{F} \) is a member of the set \( \{ y | \mathcal{F} y \subseteq y \} \) and \( z \) is smaller than any member of the set \( \{ y | \mathcal{F} y \subseteq y \} \) since it is the greatest lower bound of all the elements in that set.
  Hence, \( z \) is the least fixpoint of \( \mathcal{F} \).
Computing the Least Fixpoint

The least fixpoint \( \mu \ y \ . \ F \ y \) is the limit of the following sequence (assuming \( F \) is \( \cup \)-continuous):

\[ \emptyset, F \emptyset, F^2 \emptyset, F^3 \emptyset, \ldots \]

\( F \) is \( \cup \)-continuous if and only if
\( p_1 \subseteq p_2 \subseteq p_3 \subseteq \ldots \) implies that \( F(\bigcup_i p_i) = \bigcup_i F(p_i) \)

If \( S \) is finite, then we can compute the least fixpoint using the sequence \( \emptyset, F \emptyset, F^2 \emptyset, F^3 \emptyset, \ldots \) This sequence is guaranteed to converge if \( S \) is finite and it will converge to the least fixpoint.
Computing the Least Fixpoint

Given a monotonic and union continuous function \( \mathcal{F} \)

\[
\mu y . \mathcal{F} y = \bigcup_i \mathcal{F}^i(\emptyset)
\]

We can prove this as follows:

- First, we can show that for all \( i \), \( \mathcal{F}^i(\emptyset) \subseteq \mu y . \mathcal{F} y \) using induction

  for \( i=0 \), we have \( \mathcal{F}^0(\emptyset) = \emptyset \subseteq \mu y . \mathcal{F} y \)

Assuming \( \mathcal{F}^i(\emptyset) \subseteq \mu y . \mathcal{F} y \)

and applying the function \( \mathcal{F} \) to both sides and using monotonicity of \( \mathcal{F} \) we get: \( \mathcal{F}(\mathcal{F}^i(\emptyset)) \subseteq \mathcal{F}(\mu y . \mathcal{F} y) \)

and since \( \mu y . \mathcal{F} y \) is a fixpoint of \( \mathcal{F} \) we get:

\( \mathcal{F}^{i+1}(\emptyset) \subseteq \mu y . \mathcal{F} y \)

which completes the induction.
Computing the Least Fixpoint

• So, we showed that for all i, $F^i (\emptyset) \subseteq \mu y . F y$

• If we take the least upper bound of all the elements in the sequence $F^i (\emptyset)$ we get $\bigcup_i F^i (\emptyset)$ and using above result, we have:

$$\bigcup_i F^i (\emptyset) \subseteq \mu y . F y$$

• Now, using union-continuity we can conclude that

$$F (\bigcup_i F^i (\emptyset)) = \bigcup_i F (F^i (\emptyset)) = \bigcup_i F^{i+1} (\emptyset)$$

$$= \emptyset \bigcup_i F^{i+1} (\emptyset) = \bigcup_i F^i (\emptyset)$$

• So, we showed that $\bigcup_i F^i (\emptyset)$ is a fixpoint of $F$ and $\bigcup_i F^i (\emptyset) \subseteq \mu y . F y$, then we conclude that $\mu y . F y = \bigcup_i F^i (\emptyset)$
Computing the Least Fixpoint

If there exists a j, where $F^j(\emptyset) = F^{j+1}(\emptyset)$, then

$\mu \ y . \ F \ y = F^j(\emptyset)$

• We have proved earlier that for all i, $F^i(\emptyset) \subseteq \mu \ y . \ F \ y$

• If $F^j(\emptyset) = F^{j+1}(\emptyset)$, then $F^j(\emptyset)$ is a fixpoint of $F$ and since we know that $F^j(\emptyset) \subseteq \mu \ y . \ F \ y$ then we conclude that

$\mu \ y . \ F \ y = F^j(\emptyset)$
RS(P) Fixpoint Computation

RS(P) = \( \mu y . P \cup RS(y) \) is the limit of the sequence:

\( \emptyset, \)
\( P \cup post(\emptyset), \)
\( P \cup post(P \cup post(\emptyset)), \)
\( P \cup post(P \cup post(P \cup post(\emptyset))), \)
\( \ldots \)

which is equivalent to

\( \emptyset, P, P \cup post(P), P \cup post(P \cup post(P)), \ldots \)
RS(P) Fixpoint Computation

RS(P) ≡ states that are reachable from P ≡ P ∪ post(P) ∪ post(post(P)) ∪ ...