Contract Precondition Inference from Intermittent Assertions on Collections (*)

Patrick Cousot Radhia Cousot Francesco Logozzo

(*) Tech. Rept. no. MSR-TR-2010-117, Sep. 2010, submitted.

Motivation

The problem of contract precondition inference

- Infer a contract precondition from the language and programmer assertions
- Generate code to check that precondition

Usefullness

- Anticipate errors (e.g. change to trace execution mode before actual error does occur)
- Use contract for separate static analysis of modules

Example

From

```
             void AllNotNull(Ptr[] A) {
  /* 1: */    int i = 0;
  /* 2: */    while /* 3: */
                 (assert(A != null); i < A.length) {
  /* 4: */     assert((A != null) && (A[i] != null));
  /* 5: */     A[i].f = new Object();
  /* 6: */     i++;
  /* 7: */    }
  /* 8: */ }
```

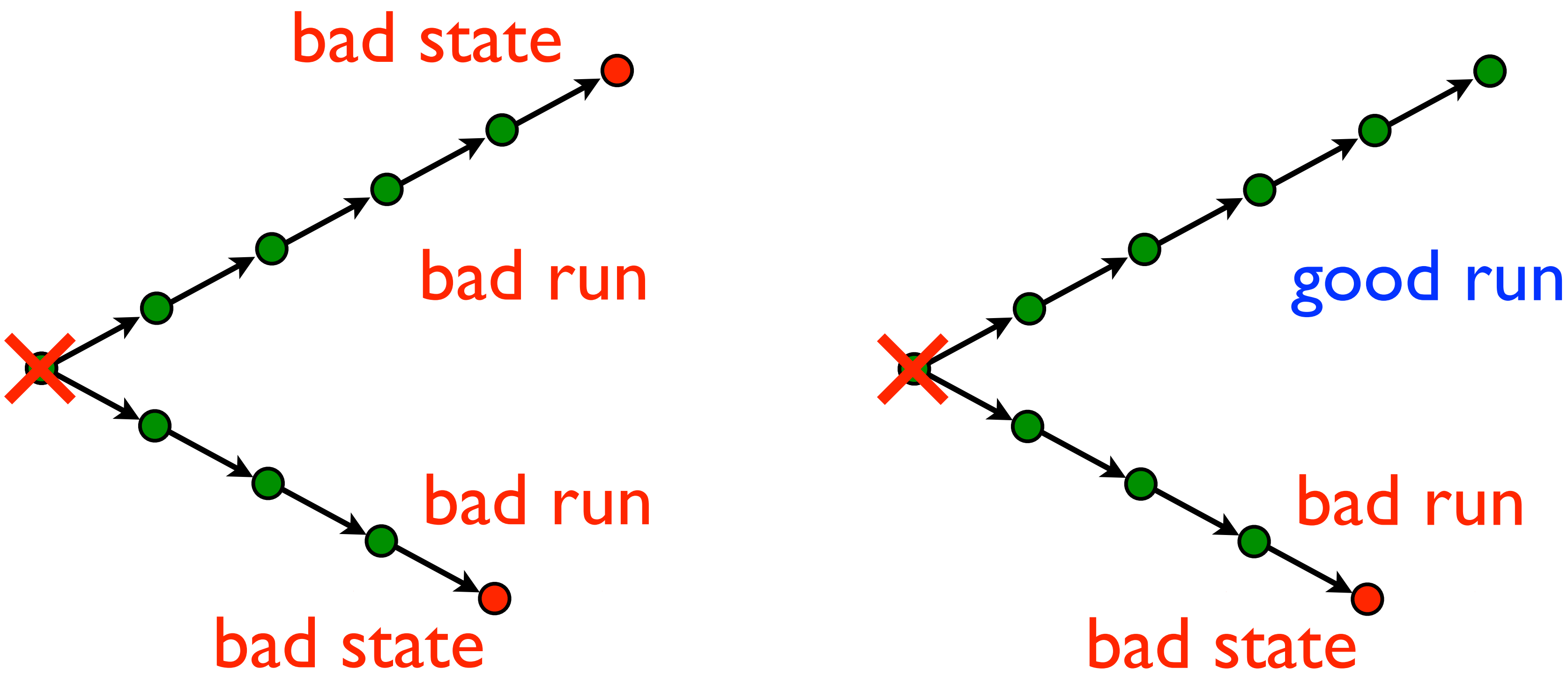
infer the precondition

$$\mathtt{A} \neq \mathtt{null} \land \forall i \in [0, \mathtt{A.length}) : \mathtt{A}[i] \neq \mathtt{null}$$

Problem specification

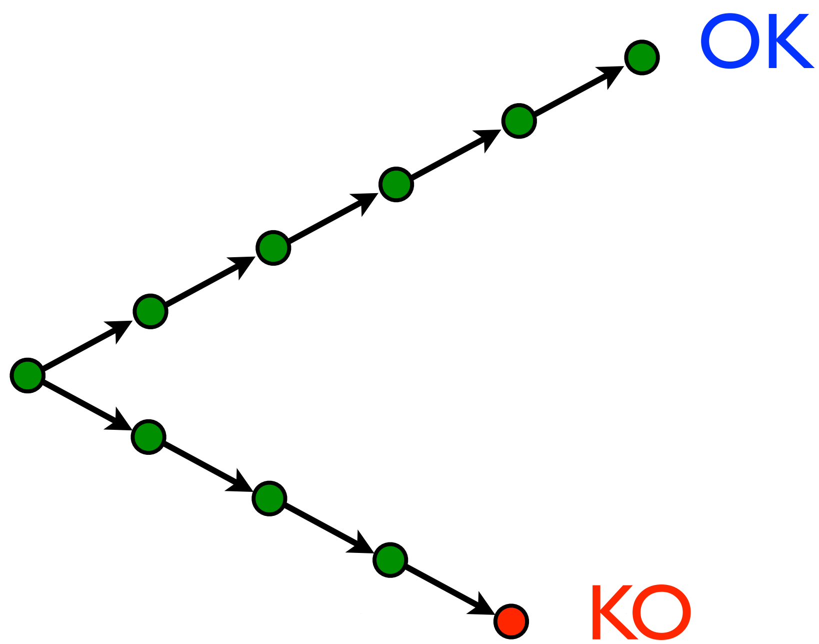
First alternative: eliminating potential errors

- The precondition should eliminate any initial state from which a nondeterministic execution may lead to a bad state (violating an assertion)



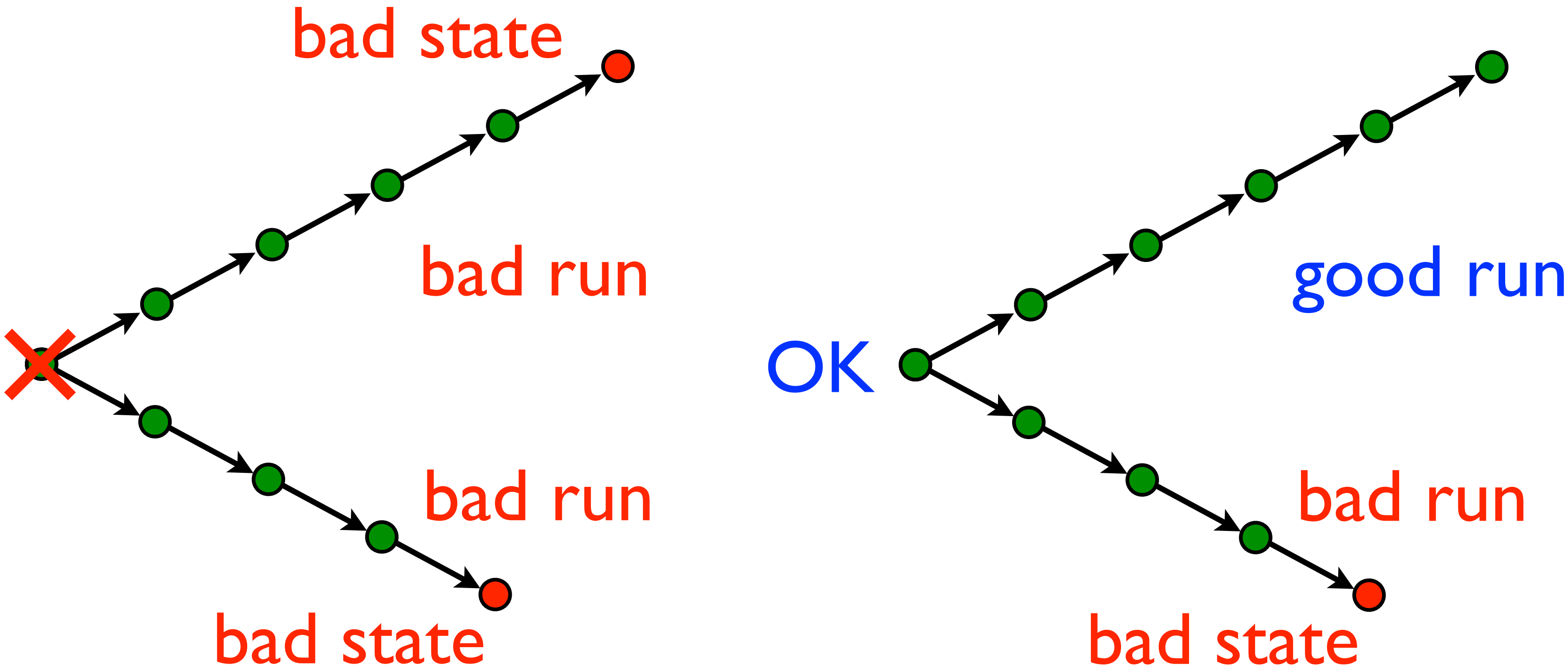
Defects of potential error elimination

- A priori correctness point of view
- We should not make any hypothesis on the programmer's intention



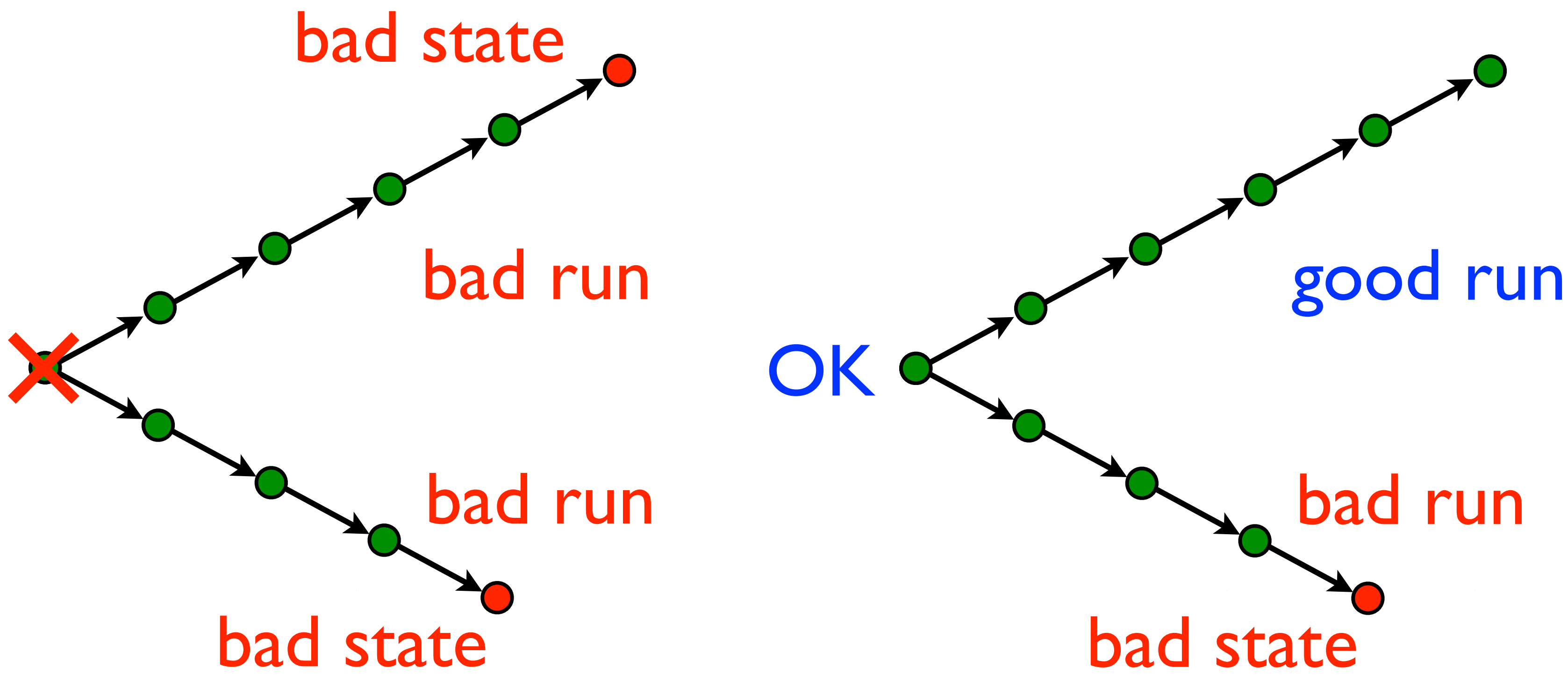
Second alternative: eliminating definite errors

- The precondition should eliminate any initial state from which all nondeterministic executions must lead to a bad state (violating an assertion)



Advantage of eliminating only definite errors

- We check states from which all executions can only go wrong as specified by the `asserts`



On non-termination

- Up to now, no human or machine could prove (or disprove) the conjecture that the following program always terminates

```
void Collatz(int n) {
   requires (n >= 1);
   while (n != 1) {
      if (odd (n)) {
         n = 3*n+1
      } else {
         n = n / 2
      }
   }
}
```

On non-termination (cont'd)

- Consider

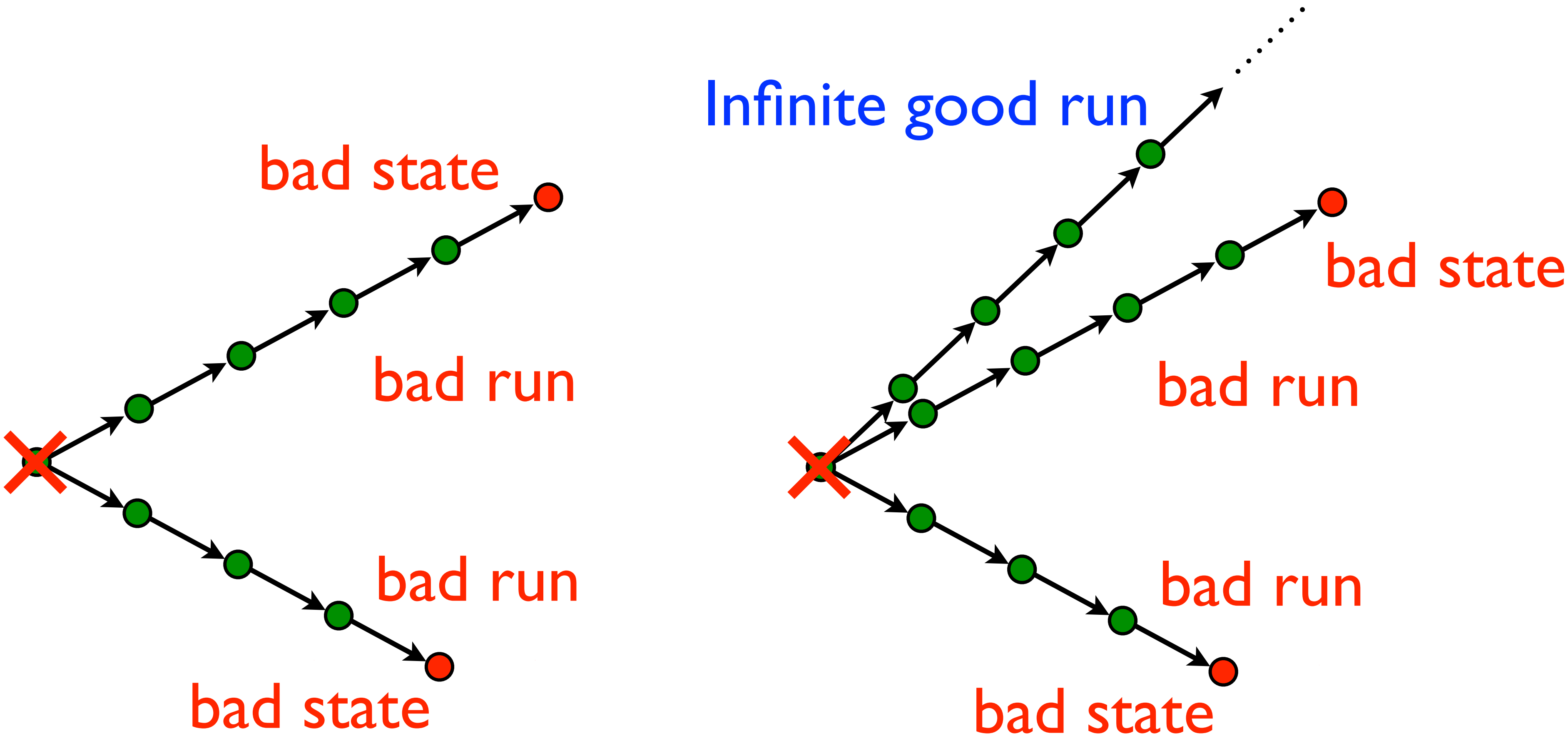
```
Collatz(p);
assert(false);
```

- The precondition is
  - `assert(false)` if `Collatz` always terminates
  - `assert(p >= 1)` if `Collatz` may not terminate
  - or even better

    `assert(NecessaryConditionForCollatzNotToTerminate(p))`

A compromise on non-termination

- We do not want to have to solve the program termination problem
- We ignore non-terminating executions, if any



Problem formalization

Program small-step operational semantics

- Transition system

$$\langle \Sigma,\ \tau,\ \mathfrak{I} \rangle$$

Set of states

Transition relation $\tau \in \wp(\Sigma \times \Sigma)$

Initial states $\mathfrak{I} \in \wp(\Sigma)$

- Blocking states

$$\mathfrak{B} \triangleq \{s \in \Sigma \mid \forall s' : \neg\tau(s, s')\}$$

Traces

- $\vec{\Sigma}^n$ traces of length n

  $\vec{s} = \vec{s}_0 \ldots \vec{s}_{n-1}$ of *length* $|\vec{s}| \triangleq n \geqslant 0$

- $\vec{\Sigma}^+ \triangleq \bigcup_{n \geqslant 1} \vec{\Sigma}^n$ non-empty finite traces

- $\vec{\Sigma}^* \triangleq \vec{\Sigma}^+ \cup \{\vec{\epsilon}\}$ finite traces

Program partial trace semantics

- Partial runs of length $n \geqslant 0$

$$\vec{\tau}^{\,n} \triangleq \{\vec{s} \in \vec{\Sigma}^n \mid \forall i \in [0, n-1) : \tau(\vec{s}_i, \vec{s}_{i+1})\}$$

- Non-empty finite partial runs

$$\vec{\tau}^{\,+} \triangleq \bigcup_{n \geqslant 1} \vec{\tau}^{\,n}$$

Program complete/maximal trace semantics

- Complete runs of length $n \geqslant 0$

$$\vec{\tau}^{n} \triangleq \{\vec{s} \in \vec{\tau}^{n} \mid \vec{s}_{n-1} \in \mathfrak{B}\}$$

- Non-empty finite complete runs

$$\vec{\tau}^{+} \triangleq \bigcup_{n \geqslant 1} \vec{\tau}^{n}$$

- Non-empty finite complete runs from initial states $\mathfrak{I}$

$$\vec{\tau}^{+}_{\mathfrak{I}} \triangleq \{\vec{s} \in \vec{\tau}^{+} \mid \vec{s}_{0} \in \mathfrak{I}\}$$

Fixpoint program trace semantics

$$\vec{\tau}^{+}_{\mathfrak{I}} = \mathsf{lfp}^{\subseteq}_{\emptyset}\ \boldsymbol{\lambda}\vec{T}\bullet\vec{\mathfrak{I}}^{1}\cup\vec{T}\,\mathbin{;}\,\vec{\tau}^{2}$$
$$\vec{\tau}^{+} = \mathsf{lfp}^{\subseteq}_{\emptyset}\ \boldsymbol{\lambda}\vec{T}\bullet\vec{\mathfrak{B}}^{1}\cup\vec{\tau}^{2}\,\mathbin{;}\,\vec{T} = \mathsf{gfp}^{\subseteq}_{\vec{\Sigma}^{+}}\ \boldsymbol{\lambda}\vec{T}\bullet\vec{\mathfrak{B}}^{1}\cup\vec{\tau}^{2}\,\mathbin{;}\,\vec{T}.$$

where

- *sequential composition* of traces is $\vec{s}s\,\mathbin{;}\,s\vec{s}' \triangleq \vec{s}s\vec{s}'$
- $\vec{S}\,\mathbin{;}\,\vec{S}' \triangleq \{\vec{s}s\vec{s}' \mid \vec{s}s \in \vec{S}\cap\vec{\Sigma}^{+} \wedge s\vec{s}' \in \vec{S}'\}$
- Given $\mathfrak{S}\subseteq\Sigma$, we let $\vec{\mathfrak{S}}^{n} \triangleq \{\vec{s}\in\vec{\Sigma}^{n} \mid \vec{s}_0\in\mathfrak{S}\}$, $n \geqslant 1$

Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. TCS 277(1—2), 47–103 (2002)

Collecting asserts

- All language and programmer assertions are collected by a syntactic pre-analysis of the code

- : $\texttt{assert(b}_j\texttt{)}$ is attached to a control point $\texttt{c}_j \in \Gamma$, $j \in \Delta$

- $\mathbb{A} \; = \; \{\langle \texttt{c}_j,\, \texttt{b}_j \rangle \mid j \in \Delta\}$

- $\texttt{b}_j$: well defined and visible side effect free

Evaluation of expressions

- Expressions $\mathsf{e} \in \mathbb{E}$ include Boolean expressions (over scalar variables or quantifcations over collections)
- $\text{The value of } \mathsf{e} \in \mathbb{E} \text{ in state } s \in \Sigma \text{ is } [\![\mathsf{e}]\!]s$
- Values include
  - Booleans $\mathcal{B} \triangleq \{true, false\}$
  - Collections (arrays, sets, hash tables, etc.) ,
  - etc

Control

- Map $\boldsymbol{\pi} \in \Sigma \rightarrow \Gamma$ of states of Σ into *control points* in Γ (of finite cardinality)

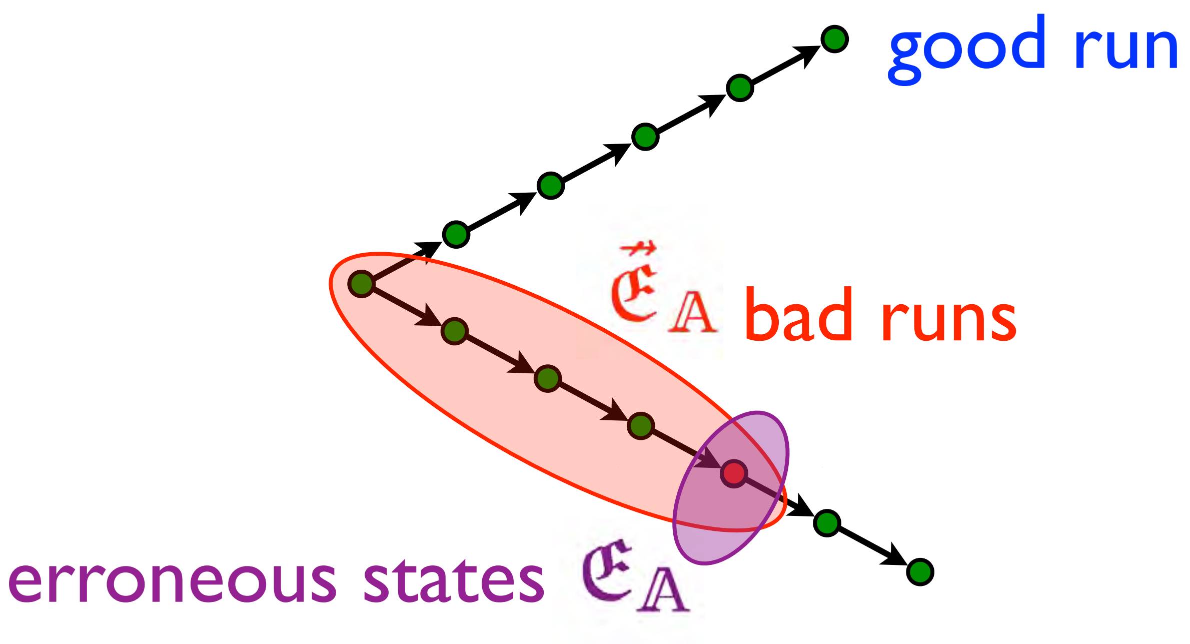
Bad states and bad traces

- Erroneous/bad states

$$\mathfrak{E}_{\mathbb{A}} \triangleq \{s \in \Sigma \mid \exists \langle c,\ \mathsf{b}\rangle \in \mathbb{A} : \boldsymbol{\pi} s = c \wedge \neg [\![\mathsf{b}]\!] s\}$$

- Erroneous/bad traces

$$\vec{\mathfrak{E}}_{\mathbb{A}} \triangleq \{\vec{s} \in \vec{\Sigma}^{+} \mid \exists i < |\vec{s}| : \vec{s}_i \in \mathfrak{E}_{\mathbb{A}}\}$$



Formal specification of the contract inference problem

Contract precondition inference problem

Definition 4 *Given a transition system* $\langle \Sigma, \tau, \mathfrak{I} \rangle$ *and a specification* $\mathbb{A}$, *the* contract precondition inference problem *consists in computing* $P_{\mathbb{A}} \in \wp(\Sigma)$ *such that when replacing the initial states* $\mathfrak{I}$ *by* $P_{\mathbb{A}} \cap \mathfrak{I}$, *we have*

$$\vec{\tau}^{+}_{P_{\mathbb{A}} \cap \mathfrak{I}} \subseteq \vec{\tau}^{+}_{\mathfrak{I}} \qquad \textit{(no new run is introduced)} \tag{2}$$

$$\vec{\tau}^{+}_{\mathfrak{I} \setminus P_{\mathbb{A}}} = \vec{\tau}^{+}_{\mathfrak{I}} \setminus \vec{\tau}^{+}_{P_{\mathbb{A}}} \subseteq \vec{\mathfrak{E}}_{\mathbb{A}} \qquad \textit{(all eliminated runs are bad runs).} \tag{3}$$

$\square$

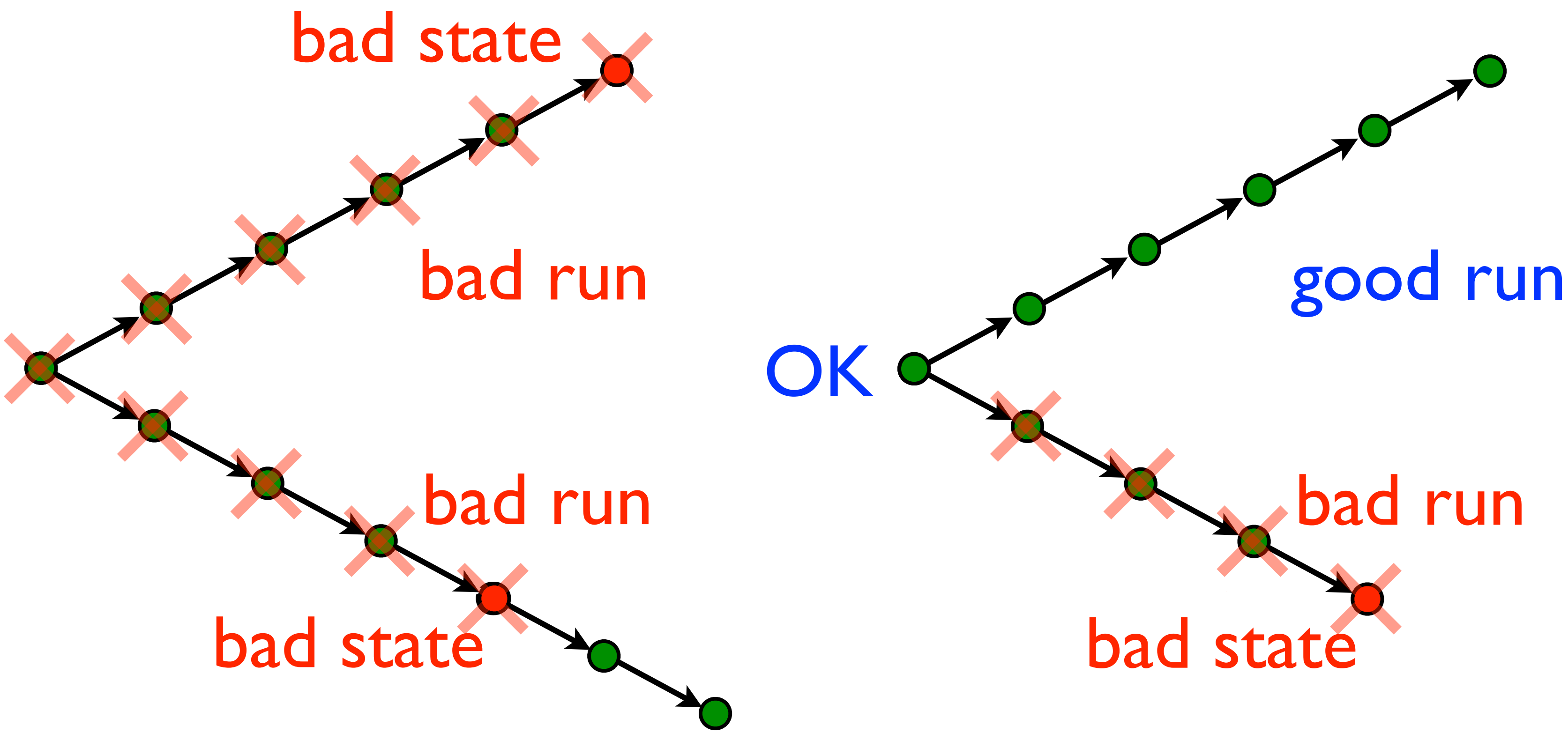
So no finite maximal good run is ever eliminated:

Lemma 5 (3) *implies* $\vec{\tau}^{+}_{\mathfrak{I}} \cap \neg\vec{\mathfrak{E}}_{\mathbb{A}} \subseteq \vec{\tau}^{+}_{P_{\mathbb{A}}}$.

Choosing $P_{\mathbb{A}} = \mathfrak{I}$ so that $\mathfrak{I} \setminus P_{\mathbb{A}} = \emptyset$ hence $\vec{\tau}^{+}_{\mathfrak{I} \setminus P_{\mathbb{A}}} = \emptyset$ is a trivial solution

The strongest solution

Theorem 6 *The strongest*[5] *solution to the precondition inference problem in Def. 4 is*
$$\mathfrak{P}_{\mathbb{A}} \triangleq \{s \mid \exists s\vec{s} \in \vec{\tau}^{+} \cap \neg\vec{\mathfrak{E}}_{\mathbb{A}}\}. \quad (4)$$
$\square$



---

(5) P is said to be *stronger* than Q and Q *weaker* than P if and only if $P \subseteq Q$.

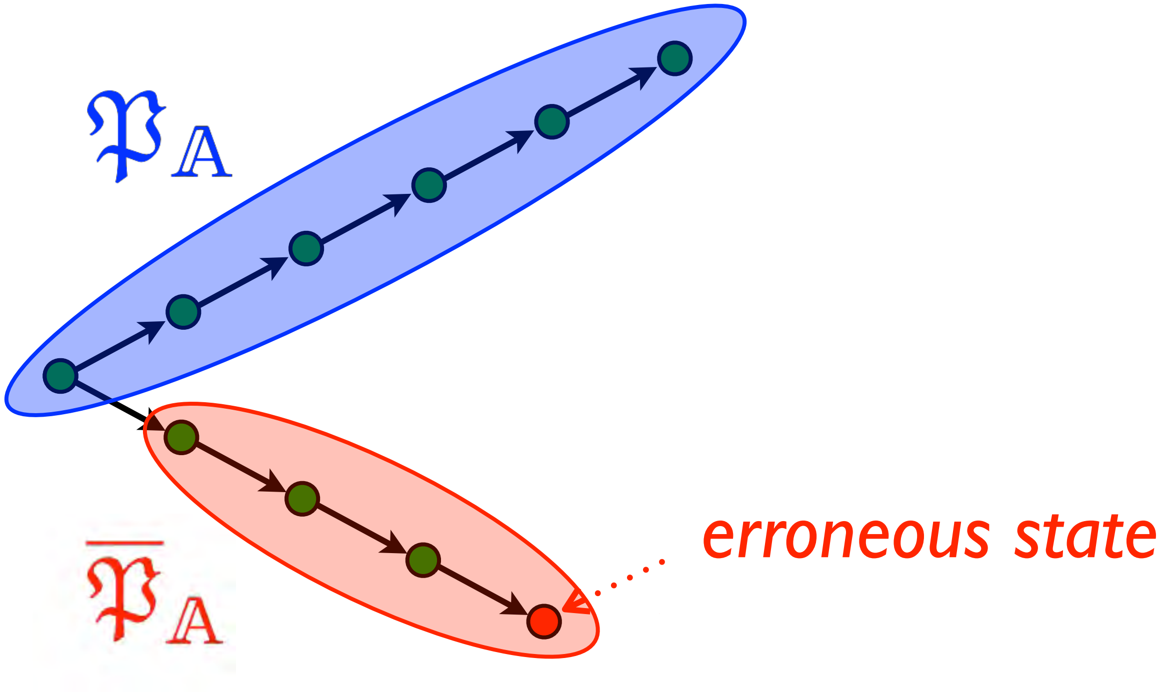
Good and bad states

- Good states : start at least one good run

$$\mathfrak{P}_{\mathbb{A}} \triangleq \{s \mid \exists s\vec{s} \in \vec{\tau}^{+} \cap \neg\vec{\mathfrak{E}}_{\mathbb{A}}\}.$$

- Bad states : start only bad runs

$$\overline{\mathfrak{P}}_{\mathbb{A}} \triangleq \neg\mathfrak{P}_{\mathbb{A}} = \{s \mid \forall s\vec{s} \in \vec{\tau}^{+} : s\vec{s} \in \vec{\mathfrak{E}}_{\mathbb{A}}\}.$$



Trace predicate transformers

- Trace predicate transformers (*)

$$\mathsf{wlp}[\vec{T}] \triangleq \boldsymbol{\lambda}\vec{Q}\bullet\{s \mid \forall s\vec{s} \in \vec{T} : s\vec{s} \in \vec{Q}\}$$

$$\mathsf{wlp}^{-1}[\vec{Q}] \triangleq \boldsymbol{\lambda}P\bullet\{s\vec{s} \in \vec{\Sigma}^{+} \mid (s \in P) \Rightarrow (s\vec{s} \in \vec{Q})\}$$

- Galois connection

$$\langle\wp(\vec{\Sigma}^{+}), \subseteq\rangle \underset{\boldsymbol{\lambda}\vec{T}\bullet\mathsf{wlp}[\vec{T}]\vec{Q}}{\overset{\mathsf{wlp}^{-1}[\vec{Q}]}{\leftrightarrows}} \langle\wp(\Sigma), \supseteq\rangle$$

- Bad initial states (all runs from these states are bad)

$$\begin{aligned}\overline{\mathfrak{P}}_{\mathbb{A}} &= \mathsf{wlp}[\vec{\tau}^{+}](\vec{\vec{\mathfrak{E}}}_{\mathbb{A}}) \\ &= \{s \mid \forall s\vec{s} \in \vec{\tau}^{+} : s\vec{s} \in \vec{\vec{\mathfrak{E}}}_{\mathbb{A}}\}\end{aligned}$$

(*) Denoted as, but different from, and enjoying properties similar to Dijkstra's syntactic WLP predicate transformer

A very brief recap of abstract interpretation

Galois connections

concretization

$$\langle L,\ \leqslant\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \overline{L},\ \sqsubseteq\rangle$$

concrete domain

concrete implication

abstraction

abstract domain

abstract implication

$\Longleftarrow$ best abstraction

$$\forall x \in L, y \in \overline{L} : \alpha(x) \sqsubseteq y \Leftrightarrow x \leqslant \gamma(y)$$

$\Longrightarrow$ soundness

Duality

$$\langle \overline{L},\ \sqsupseteq\rangle \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \langle L,\ \geqslant\rangle$$

Example: complement isomorphism

- $\langle L, \leqslant\rangle$ is a complete Boolean lattice with unique complement $\neg$

$$\langle L, \leqslant\rangle \underset{\neg}{\overset{\neg}{\leftrightarrows}} \langle L, \geqslant\rangle \text{ (since } \neg x \leqslant y \Leftrightarrow x \geqslant \neg y).$$

- self-dual

Fixpoint abstraction

Lemma 7 *If* $\langle L, \leqslant, \perp\rangle$ *is a complete lattice or a cpo,* $F \in L \to L$ *is increasing,* $\langle \overline{L}, \sqsubseteq\rangle$ *is a poset,* $\alpha \in L \to \overline{L}$ *is continuous*[(6),(7)], $\overline{F} \in \overline{L} \to \overline{L}$ *commutes (resp. semi-commutes) with* F *that is* $\alpha \circ F = \overline{F} \circ \alpha$ *(resp.* $\alpha \circ F \sqsubseteq \overline{F} \circ \alpha$*) then* $\alpha(\mathsf{lfp}_{\perp}^{\leqslant} F) = \mathsf{lfp}_{\alpha(\perp)}^{\sqsubseteq} \overline{F}$ *(resp.* $\alpha(\mathsf{lfp}_{\perp}^{\leqslant} F) \sqsubseteq \mathsf{lfp}_{\alpha(\perp)}^{\sqsubseteq} \overline{F}$*).*

---

(6) α is *continuous* if and only if it preserves existing lubs of increasing chains.

(7) The continuity hypothesis for α can be restricted to the iterates of the least fixpoint of F.

Fixpoint abstraction (cont'd)

Lemma 7 *If* $\langle L, \leqslant, \bot\rangle$ *is a complete lattice or a cpo,* $F \in L \to L$ *is increasing,* $\langle \overline{L}, \sqsubseteq\rangle$ *is a poset,* $\alpha \in L \to \overline{L}$ *is continuous*[(6),(7)], $\overline{F} \in \overline{L} \to \overline{L}$ *commutes (resp. semi-commutes) with* F *that is* $\alpha \circ F = \overline{F} \circ \alpha$ *(resp.* $\alpha \circ F \sqsubseteq \overline{F} \circ \alpha$*) then* $\alpha(\mathsf{lfp}^{\leqslant}_{\bot} F) = \mathsf{lfp}^{\sqsubseteq}_{\alpha(\bot)} \overline{F}$ *(resp.* $\alpha(\mathsf{lfp}^{\leqslant}_{\bot} F) \sqsubseteq \mathsf{lfp}^{\sqsubseteq}_{\alpha(\bot)} \overline{F}$*).*

Applying Lem. 7 to $\langle L, \leqslant\rangle \underset{\neg}{\overset{\neg}{\leftrightarrows}} \langle L, \geqslant\rangle$, we get

Corollary 8 (David Park) *If* $F \in L \to L$ *is increasing on a complete Boolean lattice* $\langle L, \leqslant, \bot, \neg\rangle$ *then* $\neg\,\mathsf{lfp}^{\leqslant}_{\bot} F = \mathsf{gfp}^{\leqslant}_{\neg\bot} \neg \circ F \circ \neg$.

---

(6) α is *continuous* if and only if it preserves existing lubs of increasing chains.

(7) The continuity hypothesis for α can be restricted to the iterates of the least fixpoint of F.

Fixpoint abstraction (cont'd)

Lemma 7 *If $\langle L, \leqslant, \perp\rangle$ is a complete lattice or a cpo, $F \in L \to L$ is increasing, $\langle \overline{L}, \sqsubseteq\rangle$ is a poset, $\alpha \in L \to \overline{L}$ is continuous*[(6),(7)], *$\overline{F} \in \overline{L} \to \overline{L}$ commutes (resp. semi-commutes) with F that is $\alpha \circ F = \overline{F} \circ \alpha$ (resp. $\alpha \circ F \sqsubseteq \overline{F} \circ \alpha$) then $\alpha(\mathsf{lfp}^{\leqslant}_{\perp} F) = \mathsf{lfp}^{\sqsubseteq}_{\alpha(\perp)} \overline{F}$ (resp. $\alpha(\mathsf{lfp}^{\leqslant}_{\perp} F) \sqsubseteq \mathsf{lfp}^{\sqsubseteq}_{\alpha(\perp)} \overline{F}$).*

Applying Lem. 7 to $\langle L, \leqslant\rangle \underset{\neg}{\overset{\neg}{\leftrightarrows}} \langle L, \geqslant\rangle$, we get Cor. 8 and by duality Cor. 9 below.

Corollary 8 (David Park) *If $F \in L \to L$ is increasing on a complete Boolean lattice $\langle L, \leqslant, \perp, \neg\rangle$ then $\neg\,\mathsf{lfp}^{\leqslant}_{\perp} F = \mathsf{gfp}^{\leqslant}_{\neg\perp} \neg \circ F \circ \neg$.*

Corollary 9 *If $\langle \overline{L}, \sqsubseteq, \top\rangle$ is a complete lattice or a dcpo, $\overline{F} \in \overline{L} \to \overline{L}$ is increasing, $\gamma \in \overline{L} \to L$ is co-continuous*[(8)], *$F \in L \to L$ commutes with F that is $\gamma \circ \overline{F} = F \circ \gamma$ then $\gamma(\mathsf{gfp}^{\sqsubseteq}_{\top} \overline{F}) = \mathsf{gfp}^{\leqslant}_{\gamma(\top)} F$.*

---

(6) α is *continuous* if and only if it preserves existing lubs of increasing chains.

(7) The continuity hypothesis for α can be restricted to the iterates of the least fixpoint of F.

(8) γ is *co-continuous* if and only if it preserves existing glbs of decreasing chains.

Fixpoint strongest contrat precondition (collecting semantics)

Fixpoint strongest contract precondition

Theorem 10 $\overline{\mathfrak{P}}_{\mathbb{A}} = \mathsf{gfp}^{\subseteq}_{\Sigma}\ \boldsymbol{\lambda} P \bullet \mathfrak{E}_{\mathbb{A}} \cup (\neg\mathfrak{B} \cap \widetilde{\mathsf{pre}}[t]P)$ *and* $\mathfrak{P}_{\mathbb{A}} = \mathsf{lfp}^{\subseteq}_{\emptyset}\ \boldsymbol{\lambda} P \bullet \neg\mathfrak{E}_{\mathbb{A}} \cap (\mathfrak{B} \cup \mathsf{pre}[t]P)$ *where* $\mathsf{pre}[t]Q \triangleq \{s \mid \exists s' \in Q : \langle s,\ s'\rangle \in t\}$ *and* $\widetilde{\mathsf{pre}}[t]Q \triangleq \neg\mathsf{pre}[t](\neg Q) = \{s \mid \forall s' : \langle s,\ s'\rangle \in t \Rightarrow s' \in Q\}$. $\square$

Fixpoint strongest contract precondition (proof)

Theorem 10 $\overline{\mathfrak{P}}_{\mathbb{A}} = \mathsf{gfp}^{\subseteq}_{\Sigma}\ \boldsymbol{\lambda} P \bullet \mathfrak{E}_{\mathbb{A}} \cup (\neg\mathfrak{B} \cap \widetilde{\mathsf{pre}}[t]P)$ *and* $\mathfrak{P}_{\mathbb{A}} = \mathsf{lfp}^{\subseteq}_{\emptyset}\ \boldsymbol{\lambda} P \bullet \neg\mathfrak{E}_{\mathbb{A}} \cap (\mathfrak{B} \cup \mathsf{pre}[t]P)$ *where* $\mathsf{pre}[t]Q \triangleq \{s \mid \exists s' \in Q : \langle s,\ s'\rangle \in t\}$ *and* $\widetilde{\mathsf{pre}}[t]Q \triangleq \neg\mathsf{pre}[t](\neg Q) = \{s \mid \forall s' : \langle s,\ s'\rangle \in t \Rightarrow s' \in Q\}$. $\square$

Proof sketch:

- $\vec{\tau}^{+} = \mathsf{lfp}^{\subseteq}_{\emptyset}\ \boldsymbol{\lambda}\vec{T} \bullet \vec{\mathfrak{B}}^{1} \cup \vec{\tau}^{2} \,\text{⨾}\, \vec{T}$
- $\langle \wp(\vec{\Sigma}^{+}),\ \subseteq\rangle \underset{\boldsymbol{\lambda}\vec{T} \bullet \mathsf{wlp}[\vec{T}]\vec{Q}}{\overset{\mathsf{wlp}^{-1}[\vec{Q}]}{\leftrightarrows}} \langle \wp(\Sigma),\ \supseteq\rangle$
- $\mathsf{wlp}[\vec{\mathfrak{B}}^{1} \cup \vec{\tau}^{2} \,\text{⨾}\, \vec{T}](\vec{\mathfrak{E}}_{\mathbb{A}}) = \mathfrak{E}_{\mathbb{A}} \cup (\neg\mathfrak{B} \cap \widetilde{\mathsf{pre}}[t](\mathsf{wlp}[\vec{T}](\vec{\mathfrak{E}}_{\mathbb{A}})))$
- $\overline{\mathfrak{P}}_{\mathbb{A}} = \mathsf{wlp}[\vec{\tau}^{+}](\vec{\mathfrak{E}}_{\mathbb{A}}) = \mathsf{wlp}[\mathsf{lfp}^{\subseteq}_{\emptyset}\ \boldsymbol{\lambda}\vec{T} \bullet \vec{\mathfrak{B}}^{1} \cup \vec{\tau}^{2} \,\text{⨾}\, \vec{T}](\vec{\mathfrak{E}}_{\mathbb{A}})$
  $= \mathsf{lfp}^{\supseteq}_{\Sigma}\ \boldsymbol{\lambda} P \bullet \mathfrak{E}_{\mathbb{A}} \cup (\neg\mathfrak{B} \cap \widetilde{\mathsf{pre}}[t]P) = \mathsf{gfp}^{\subseteq}_{\Sigma}\ \boldsymbol{\lambda} P \bullet \mathfrak{E}_{\mathbb{A}} \cup (\neg\mathfrak{B} \cap \widetilde{\mathsf{pre}}[t]P)$
- $\mathfrak{P}_{\mathbb{A}} = \neg\overline{\mathfrak{P}}_{\mathbb{A}} = \mathsf{lfp}^{\subseteq}_{\emptyset}\ \boldsymbol{\lambda} P \bullet \neg\mathfrak{E}_{\mathbb{A}} \cap (\mathfrak{B} \cup \mathsf{pre}[t]P)$ (Park)

$\square$

Model-checking

- Computers are finite
- Compute $\mathfrak{P}_{\mathbb{A}} = \mathsf{lfp}^{\subseteq}_{\emptyset}\ \boldsymbol{\lambda} P \bullet \neg\mathfrak{E}_{\mathbb{A}} \cap (\mathfrak{B} \cup \mathsf{pre}[t]P)$ iteratively
- Might not scale up (pure conjecture, not implemented)

Bounded model-checking

$$\alpha_k(\vec{T}) \triangleq \left\{ \vec{s}_0 \ldots \vec{s}_{\min(k,|\vec{s}|)-1} \;\middle|\; \vec{s} \in \vec{T} \right\}$$

is unsound both for $\mathfrak{P}_{\mathbb{A}}$ and $\overline{\mathfrak{P}}_{\mathbb{A}}$

Contract precondition inference by abstract interpretation

(I) Backward expression propagation

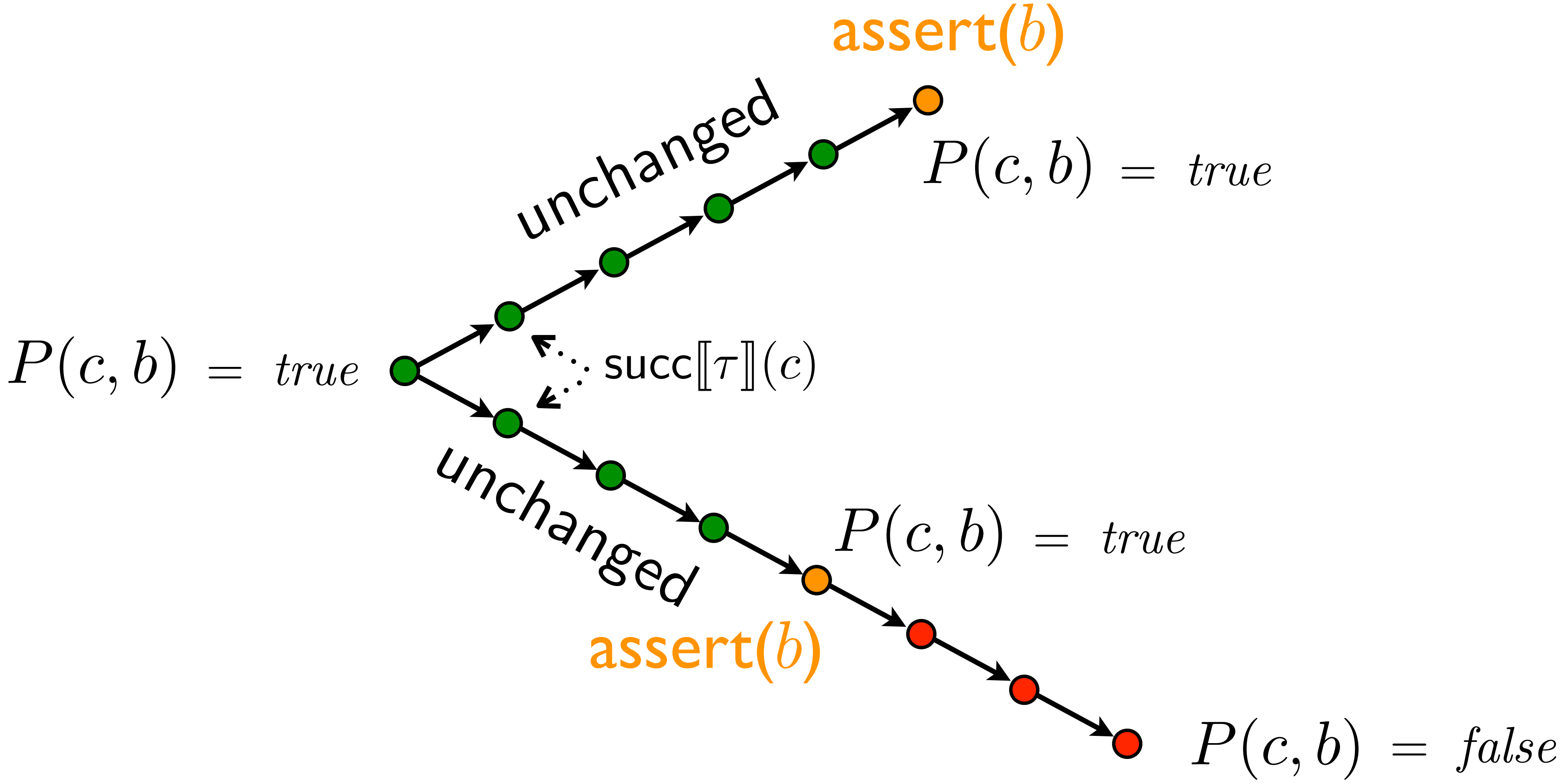
General idea

- Replace state-based reasonings by symbolic reasonings
- Idea: try to move the condition code in assertions at the beginning of the program/method/...
- This is possible under the sufficient conditions:

1. the value of the visible side effect free Boolean expression on scalar or collection variables in the `assert` is exactly the same as the value of this expression when evaluated on entry;
2. the value of the expression checked on program entry is checked in an `assert` on all paths that can be taken from the program entry.

Dataflow analysis

- $P(c,b)$ holds at program point c when Boolean expression b will definitely be checked in an `assert(`b`)` on all paths from c without being changed up to this check.



Dataflow analysis (cont'd)

- $P \;=\; \mathsf{gfp}^{\dot{\Rightarrow}}\; B[\![\tau]\!]$ $\qquad B \in (\Gamma \times \mathbb{A}_{\mathsf{b}} \to \mathcal{B}) \to (\Gamma \times \mathbb{A}_{\mathsf{b}} \to \mathcal{B})$

- $\begin{cases} P(c,b) \;=\; B[\![\tau]\!](P)(c,b) \\ c \in \Gamma, \quad b \in \mathbb{A}_{\mathsf{b}} \end{cases}$ $\qquad \mathbb{A}_{\mathsf{b}} \triangleq \{b \mid \exists c : \langle c,\ b\rangle \in \mathbb{A}\}$

- $B[\![\tau]\!](P)(c,b) \;=\; \mathit{true} \quad \text{when} \quad \langle c,\ b\rangle \in \mathbb{A}$ $\qquad (\mathtt{assert}(b)\text{ at } c)$

  $B[\![\tau]\!](P)(c,b) \;=\; \mathit{false} \quad \text{when} \quad \exists s \in \mathfrak{B} : \boldsymbol{\pi} s = c \wedge \langle c,\ b\rangle \notin \mathbb{A}$ $\qquad (\text{exit at } c)$

  $B[\![\tau]\!](P)(c,b) \;=\; \bigwedge_{c' \in \mathsf{succ}[\![\tau]\!](c)} \mathsf{unchanged}[\![\tau]\!](c,c',b) \wedge P(c',b)$ $\qquad (\text{otherwise})$

- the set $\mathsf{succ}[\![\tau]\!](c)$ of successors of the program point $c \in \Gamma$ satisfies

  $$\mathsf{succ}[\![\tau]\!](c) \;\supseteq\; \{c' \in \Gamma \mid \exists s, s' : \boldsymbol{\pi} s = c \wedge \tau(s,s') \wedge \boldsymbol{\pi} s' = c'\}$$

- $\mathsf{unchanged}[\![\tau]\!](c,c',b)$ implies than a transition by τ from program point c to program point c' can never change the value of Boolean expression b

  $$\mathsf{unchanged}[\![\tau]\!](c,c',b) \;\Rightarrow\; \forall s,s' : (\boldsymbol{\pi} s = c \wedge \tau(s,s') \wedge \boldsymbol{\pi} s' = c') \Rightarrow ([\![\mathsf{b}]\!] s = [\![\mathsf{b}]\!] s').$$

Soundness of the dataflow analysis (cont'd)

- Define

$$\begin{aligned}\mathfrak{R}_{\mathbb{A}} &\triangleq \boldsymbol{\lambda} b \bullet \{\langle s,\ s'\rangle \mid \langle \boldsymbol{\pi} s',\ b\rangle \in \mathbb{A} \wedge [\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!]s'\}\\ \vec{\mathfrak{R}}_{\mathbb{A}} &\triangleq \boldsymbol{\lambda} b \bullet \{\vec{s} \in \vec{\Sigma}^{+} \mid \exists i < |\vec{s}| : \langle \vec{s}_0,\ \vec{s}_i\rangle \in \mathfrak{R}_{\mathbb{A}}(b)\}\end{aligned}$$

and the abstraction

$$\vec{\alpha}_D(\vec{T})(c, b) \triangleq \forall \vec{s} \in \vec{T} : \boldsymbol{\pi}\vec{s}_0 = c \Rightarrow \vec{s} \in \vec{\mathfrak{R}}_{\mathbb{A}}(b)$$

$$\vec{\gamma}_D(P) \triangleq \{\vec{s} \mid \forall b \in \mathbb{A}_{\mathsf{b}} : P(\boldsymbol{\pi}\vec{s}_0, b) \Rightarrow \vec{s} \in \vec{\mathfrak{R}}_{\mathbb{A}}(b)\}$$

such that $\langle \vec{\Sigma}^{+},\ \subseteq\rangle \underset{\vec{\alpha}_D}{\overset{\vec{\gamma}_D}{\leftrightarrows}} \langle \Gamma \times \mathbb{A}_{\mathsf{b}} \to \mathcal{B},\ \dot{\Leftarrow}\rangle$.

- **Theorem 12** $\vec{\alpha}_D(\vec{\tau}^{+}) \dot{\Leftarrow} \mathsf{lfp}^{\dot{\Leftarrow}}\, B[\![\tau]\!] = \mathsf{gfp}^{\dot{\Rightarrow}}\, B[\![\tau]\!] \triangleq P$. $\square$

Proof $\vec{\tau}^{+} = \mathsf{lfp}_{\emptyset}^{\subseteq}\, \boldsymbol{\lambda} \vec{T} \bullet \vec{\mathfrak{B}}^{1} \cup \vec{\tau}^{2} \,{}_{9}^{\circ}\, \vec{T}$ and fixpoint abstraction (Lem. 8)

Calculational design of the dataflow analysis

Proof By (1-a), we have $\vec{\tau}^{+} = \mathsf{lfp}^{\subseteq}_{\emptyset}\, \boldsymbol{\lambda} \vec{T} \bullet \vec{\mathfrak{B}}^{1} \cup \vec{\tau}^{2} \mathbin{;} \vec{T}$ so, by Lem. 8, it is sufficient to prove the semi-commutativity property

$$\vec{\alpha}_D(\vec{\mathfrak{B}}^{1} \cup \vec{\tau}^{2} \mathbin{;} \vec{T}) = \vec{\alpha}_D(\vec{\mathfrak{B}}^{1}) \dot{\wedge}\, \vec{\alpha}_D(\vec{\tau}^{2} \mathbin{;} \vec{T}) \mathrel{\dot{\Leftarrow}} B[\![\tau]\!](\vec{\alpha}_D(\vec{T})).$$

$$
\begin{aligned}
&- \ \vec{\alpha}_D(\vec{\mathfrak{B}}^{1})(c,b)\\
&= \ \forall \vec{s} \in \vec{\mathfrak{B}}^{1} : \boldsymbol{\pi}\vec{s}_0 = c \Rightarrow \vec{s} \in \vec{\mathfrak{R}}_{\mathbb{A}}(b) && \wr \text{def. } \vec{\alpha}_D \int\\
&= \ \forall s \in \mathfrak{B} : \boldsymbol{\pi} s = c \Rightarrow \langle s,\ s\rangle \in \mathfrak{R}_{\mathbb{A}}(b) && \wr \text{def. } \vec{\mathfrak{B}}^{1} \text{ and } \vec{\mathfrak{R}}_{\mathbb{A}}(b) \int\\
&= \ \forall s \in \mathfrak{B} : \boldsymbol{\pi} s = c \Rightarrow \langle c,\ b\rangle \in A && \wr \text{def. } \mathfrak{R}_{\mathbb{A}} \int\\
&= \ \mathit{true} && \wr \text{when } \langle c,\ b\rangle \in A \int\\
&= \ \mathit{false} && \wr \text{when } \exists s \in \mathfrak{B} : \boldsymbol{\pi} s = c \wedge \langle c,\ b\rangle \notin A \int\\
&= \ B[\![\tau]\!](\vec{\alpha}_D(\vec{T})(c,b) && \wr \text{def. } B[\![\tau]\!] \int
\end{aligned}
$$

$$
\begin{aligned}
&- \ \vec{\alpha}_D(\vec{\tau}^{2} \mathbin{;} \vec{T})(c,b)\\
&= \ \forall \vec{s} \in \vec{\tau}^{2} \mathbin{;} \vec{T} : \boldsymbol{\pi}\vec{s}_0 = c \Rightarrow \vec{s} \in \vec{\mathfrak{R}}_{\mathbb{A}}(b) && \wr \text{def. } \vec{\alpha}_D \int\\
&= \ \forall s, s', \vec{s} : (\tau(s,s') \wedge s'\vec{s} \in \vec{T} \wedge \boldsymbol{\pi} s = c) \Rightarrow ss'\vec{s} \in \vec{\mathfrak{R}}_{\mathbb{A}}(b) && \wr \text{def. } ; \text{ and } \vec{\tau}^{2} \int\\
&= \ \forall s, s', \vec{s} : (\tau(s,s') \wedge s'\vec{s} \in \vec{T} \wedge \boldsymbol{\pi} s = c) \Rightarrow (\exists j < |ss'\vec{s}| : \langle s,\ (ss'\vec{s})_j\rangle \in \mathfrak{R}_{\mathbb{A}}(b))\\
& && \wr \text{def. } \vec{\mathfrak{R}}_{\mathbb{A}} \int\\
&= \ \forall s, s', \vec{s} : (\tau(s,s') \wedge s'\vec{s} \in \vec{T} \wedge \boldsymbol{\pi} s = c) \Rightarrow (\exists j < |ss'\vec{s}| : \langle \boldsymbol{\pi}(ss'\vec{s})_j,\ b\rangle \in\\
&\quad\ A \wedge [\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!](ss'\vec{s})_j) && \wr \text{def. } \mathfrak{R}_{\mathbb{A}} \int\\
&= \ \forall s, s', \vec{s} : (\tau(s,s') \wedge s'\vec{s} \in \vec{T} \wedge \boldsymbol{\pi} s = c) \Rightarrow (\langle \boldsymbol{\pi} s,\ b\rangle \in A \vee (\exists j < |s'\vec{s}| : \langle \boldsymbol{\pi}(s'\vec{s})_j,\\
&\quad\ b\rangle \in A \wedge [\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!](s'\vec{s})_j)) && \wr \text{separating the case } j = 0 \int\\
&\Leftarrow \langle c,\ b\rangle \in A \vee \forall s, s', \vec{s} : (\tau(s,s') \wedge s'\vec{s} \in \vec{T} \wedge \boldsymbol{\pi} s = c) \Rightarrow (\exists j < |s'\vec{s}| : \langle \boldsymbol{\pi}(s'\vec{s})_j,\\
&\quad\ b\rangle \in A \wedge [\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!](s'\vec{s})_j) && \wr \text{def. } \Rightarrow \int\\
&= \ \langle c,\ b\rangle \in A \vee \forall s, s' : \big(\tau(s,s') \wedge \boldsymbol{\pi} s = c\big) \Rightarrow \big(\forall s'\vec{s} \in \vec{T} : \exists j < |s'\vec{s}| : \langle \boldsymbol{\pi}(s'\vec{s})_j,\\
&\quad\ b\rangle \in A \wedge [\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!](s'\vec{s})_j\big) && \wr \text{def. } \Rightarrow \int\\
&\Leftarrow \langle c,\ b\rangle \in A \vee \forall s, s' : \big(\tau(s,s') \wedge \boldsymbol{\pi} s = c\big) \Rightarrow \big([\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!]s' \wedge \forall s'\vec{s}' \in \vec{T} : (\exists j <\\
&\quad\ |s'\vec{s}'| : \langle \boldsymbol{\pi}(s'\vec{s}')_j,\ b\rangle \in A \wedge [\![\mathsf{b}]\!]s' = [\![\mathsf{b}]\!](s'\vec{s}')_j)\big) && \wr \text{transitivity of} = \text{and } \vec{s'} = \vec{s} \int\\
&= \ \langle c, b\rangle \in A \vee \forall s, s' : \big(\tau(s,s') \wedge \boldsymbol{\pi} s = c\big) \Rightarrow \big([\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!]s' \wedge \forall s'\vec{s}' \in \vec{T} : \boldsymbol{\pi}(s'\vec{s}')_0 =\\
&\quad\ \boldsymbol{\pi} s' \Rightarrow (\exists j < |s'\vec{s}'| : \langle \boldsymbol{\pi}(s'\vec{s}')_j,\ b\rangle \in A \wedge [\![\mathsf{b}]\!](s'\vec{s}')_0 = [\![\mathsf{b}]\!](s'\vec{s}')_j)\big)\\
& && \wr (s'\vec{s}')_0 = s' \int\\
&= \ \langle c,\ b\rangle \in A \vee \forall s, s' : \big(\tau(s,s') \wedge \boldsymbol{\pi} s = c\big) \Rightarrow \big([\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!]s' \wedge \forall \vec{s} \in \vec{T} : \boldsymbol{\pi}\vec{s}_0 =\\
&\quad\ \boldsymbol{\pi} s' \Rightarrow (\exists j < |\vec{s}| : \langle \boldsymbol{\pi}\vec{s}_j,\ b\rangle \in A \wedge [\![\mathsf{b}]\!]\vec{s}_0 = [\![\mathsf{b}]\!]\vec{s}_j)\big) && \wr \text{letting } \vec{s} = s'\vec{s}' \int\\
&= \ \langle c,\ b\rangle \in A \vee \forall c' : \forall s, s' : \big(\tau(s,s') \wedge \boldsymbol{\pi} s = c \wedge \boldsymbol{\pi} s' = c'\big) \Rightarrow \big([\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!]s' \wedge \forall \vec{s} \in\\
&\quad\ \vec{T} : \boldsymbol{\pi}\vec{s}_0 = c' \Rightarrow (\exists j < |\vec{s}| : \langle \boldsymbol{\pi}\vec{s}_j,\ b\rangle \in A \wedge [\![\mathsf{b}]\!]\vec{s}_0 = [\![\mathsf{b}]\!]\vec{s}_j)\big) && \wr \text{letting } c' = \boldsymbol{\pi} s' \int\\
&\Leftarrow \langle c,\ b\rangle \in A \vee \forall c' : \forall s, s' : \big(\tau(s,s') \wedge \boldsymbol{\pi} s = c \wedge \boldsymbol{\pi} s' = c'\big) \Rightarrow \big(\forall s, s' : (\boldsymbol{\pi} s =\\
&\quad\ c \wedge \tau(s,s') \wedge \boldsymbol{\pi} s' = c') \Rightarrow ([\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!]s') \wedge \forall \vec{s} \in \vec{T} : \boldsymbol{\pi}\vec{s}_0 = c' \Rightarrow (\exists j < |\vec{s}| : \langle \boldsymbol{\pi}\vec{s}_j,\\
&\quad\ b\rangle \in A \wedge [\![\mathsf{b}]\!]\vec{s}_0 = [\![\mathsf{b}]\!]\vec{s}_j)\big) && \wr \text{since } A \Rightarrow (A \Rightarrow B \wedge C) \text{ implies } A \Rightarrow (B \wedge C) \int\\
&\Leftarrow \langle c,\ b\rangle \in A \vee \forall c' : \big(\exists s, s' : \tau(s,s') \wedge \boldsymbol{\pi} s = c \wedge \boldsymbol{\pi} s' = c'\big) \Rightarrow \big(\forall s, s' : (\boldsymbol{\pi} s =\\
&\quad\ c \wedge \tau(s,s') \wedge \boldsymbol{\pi} s' = c') \Rightarrow ([\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!]s') \wedge \forall \vec{s} \in \vec{T} : \boldsymbol{\pi}\vec{s}_0 = c' \Rightarrow (\exists j < |\vec{s}| : \langle \boldsymbol{\pi}\vec{s}_j,\\
&\quad\ b\rangle \in A \wedge [\![\mathsf{b}]\!]\vec{s}_0 = [\![\mathsf{b}]\!]\vec{s}_j)\big) && \wr (\exists x : A) \Rightarrow B \text{ iff } \forall x : (A \Rightarrow B) \int\\
&= \ \langle c,\ b\rangle \in A \vee \forall c' : \big(\exists s, s' : \tau(s,s') \wedge \boldsymbol{\pi} s = c \wedge \boldsymbol{\pi} s' = c'\big) \Rightarrow \big(\forall s, s' : (\boldsymbol{\pi} s =\\
&\quad\ c \wedge \tau(s,s') \wedge \boldsymbol{\pi} s' = c') \Rightarrow ([\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!]s') \wedge \forall \vec{s} \in \vec{T} : \boldsymbol{\pi}\vec{s}_0 = c' \Rightarrow (\exists j < |\vec{s}| : \langle \vec{s}_0,\\
&\quad\ \vec{s}_j\rangle \in \mathfrak{R}_{\mathbb{A}}(b))\big) && \wr \text{def. } \mathfrak{R}_{\mathbb{A}} \triangleq \boldsymbol{\lambda} b \bullet \{\langle s,\ s'\rangle \mid \langle \boldsymbol{\pi} s',\ b\rangle \in A \wedge [\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!]s'\} \int\\
&= \ \langle c,\ b\rangle \in A \vee \forall c' : \big(\exists s, s' : \tau(s,s') \wedge \boldsymbol{\pi} s = c \wedge \boldsymbol{\pi} s' = c'\big) \Rightarrow \big(\forall s, s' : (\boldsymbol{\pi} s =\\
&\quad\ c \wedge \tau(s,s') \wedge \boldsymbol{\pi} s' = c') \Rightarrow ([\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!]s') \wedge \forall \vec{s} \in \vec{T} : \boldsymbol{\pi}\vec{s}_0 = c' \Rightarrow \vec{s} \in \vec{\mathfrak{R}}_{\mathbb{A}}(b)\big)\\
& && \wr \text{def. } \vec{\mathfrak{R}}_{\mathbb{A}}(b) \int\\
&\Leftarrow \langle c,\ b\rangle \in A \vee \forall c' \in \mathsf{succ}[\![\tau]\!](c) : (\forall s, s' : (\boldsymbol{\pi} s = c \wedge \tau(s,s') \wedge \boldsymbol{\pi} s' = c') \Rightarrow ([\![\mathsf{b}]\!]s =\\
&\quad\ [\![\mathsf{b}]\!]s') \wedge \forall \vec{s} \in \vec{T} : \boldsymbol{\pi}\vec{s}_0 = c' \Rightarrow \vec{s} \in \vec{\mathfrak{R}}_{\mathbb{A}}(b))\\
& && \wr \text{def. } \mathsf{succ}[\![\tau]\!](c) \supseteq \{c' \in \Gamma \mid \exists s, s' : \tau(s,s') \wedge \boldsymbol{\pi} s = c \wedge \boldsymbol{\pi} s' = c'\} \int\\
&= \ \langle c,\ b\rangle \in A \vee \forall c' \in \mathsf{succ}[\![\tau]\!](c) : (\forall s, s' : (\boldsymbol{\pi} s = c \wedge \tau(s,s') \wedge \boldsymbol{\pi} s' = c') \Rightarrow ([\![\mathsf{b}]\!]s =\\
&\quad\ [\![\mathsf{b}]\!]s') \wedge \vec{\alpha}_D(\vec{T})(c', b)) && \wr \text{def. } \vec{\alpha}_D(\vec{T})(c,b) \triangleq \forall \vec{s} \in \vec{T} : \boldsymbol{\pi}\vec{s}_0 = c \Rightarrow \vec{s} \in \vec{\mathfrak{R}}_{\mathbb{A}}(b) \int\\
&\Leftarrow \langle c,\ b\rangle \in A \vee \forall c' \in \mathsf{succ}[\![\tau]\!](c) : \mathsf{unchanged}[\![\tau]\!](c, c', b) \wedge \vec{\alpha}_D(\vec{T})(c', b) && \wr \text{def.}\\
&\quad\ \mathsf{unchanged}[\![\tau]\!](c, c', b) \Rightarrow \forall s, s' : (\boldsymbol{\pi} s = c \wedge \tau(s,s') \wedge \boldsymbol{\pi} s' = c') \Rightarrow ([\![\mathsf{b}]\!]s = [\![\mathsf{b}]\!]s') \int\\
&= \ B[\![\tau]\!](\vec{\alpha}_D(\vec{T}))(c,b) && \wr \text{def. } B[\![\tau]\!] \int \quad \square
\end{aligned}
$$

Just to show that is is machine-checkable

Backward expression propagation-based precondition generation

- **Precondition generation.** The syntactic precondition generated at entry control point $i \in \mathfrak{I}_{\boldsymbol{\pi}} \triangleq \{i \in \Gamma \mid \exists s \in \mathfrak{I} : \boldsymbol{\pi} s = i\}$ is (assuming $\texttt{\&\&}\,\emptyset \triangleq \texttt{true}$)

$$\mathtt{P}_i \;\triangleq\; \mathop{\texttt{\&\&}}_{b \in \mathbb{A}_{\mathtt{b}},\; P(i,b)} b$$

The set of states for which the syntactic precondition $\mathtt{P}_i$ is evaluated to *true* at program point $i \in \Gamma$ is

$$P_i \;\triangleq\; \{s \in \Sigma \mid \boldsymbol{\pi} s = i \wedge [\![\mathtt{P}_i]\!] s\}$$

and so for all program entry points (in case there is more than one)

$$P_{\mathfrak{I}} \;\triangleq\; \{s \in \Sigma \mid \exists i \in \mathfrak{I}_{\boldsymbol{\pi}} : s \in P_i\}$$

- **Theorem 13** $\mathfrak{P}_{\mathbb{A}} \cap \mathfrak{I} \subseteq P_{\mathfrak{I}}$. □

Example

```
               void AllNotNull(Ptr[] A) {
/* 1: */  int i = 0;
/* 2: */  while /* 3: */
             (assert(A != null); i < A.length) {
/* 4: */   assert((A != null) && (A[i] != null));
/* 5: */   A[i].f = new Object();
/* 6: */   i++;
/* 7: */  }
/* 8: */ }
```

the assertion `A != null` is checked on all paths and `A` is not changed (only its elements are), so the data flow analysis is able to move the assertion as a precondition. □

- The dataflow analysis is a sound abstraction of the trace semantics but too imprecise

(II) Forward symbolic execution

Just the idea:

- Perform a symbolic execution [19]
- Move asserts symbolically to the program entry

Example 15 For the program

```
/* 1: x=x0 & y=y0 */                if (x == 0 ) {
/* 2: x0=0 & x=x0 & y=y0 */             x++;
/* 3: x0=0 & x=x0+1 & y=y0 */           assert(x==y);
                                     }
```

the precondition at program point `1:` is `(!(x==0)||(x+1==y))`. □

- Fixpoint approximation thanks to the formalization of symbolic execution as an abstract interpretation [8, Sect. 3.4.5] (a widening enforces convergence)

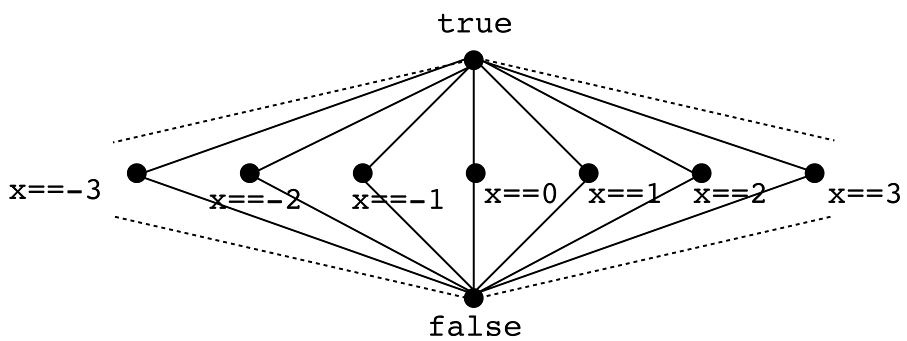
[8] Cousot, P.: Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French). Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble (1978)

[19] King, J.: Symbolic execution and program testing. CACM 19(7), 385–394 (1976)

(III) Backward symbolic execution

Abstract domain $\mathbb{B}/_{\equiv}$

- $\mathbb{B}$: visible side-effect and error free Boolean expressions on scalar variables
- $\mathsf{b} \mathrel{\dot{\Rightarrow}} \mathsf{b}'$ implies that $\forall s \in \Sigma : [\![\mathsf{b}]\!]s \Rightarrow [\![\mathsf{b}']\!]s.$ abstract implication
- $\mathsf{b} \equiv \mathsf{b}' \triangleq \mathsf{b} \mathrel{\dot{\Rightarrow}} \mathsf{b}' \wedge \mathsf{b}' \mathrel{\dot{\Rightarrow}} \mathsf{b}.$ abstract equivalence
- $\mathsf{b}' \in [\mathsf{b}]/_{\equiv}$ encoding of equivalence class by a representant
- $\langle \mathbb{B}/_{\equiv}, \mathrel{\dot{\Rightarrow}} \rangle$ abstract domain of Boolean expressions
- (Trivial) example:



Abstract domain $\langle \overline{\mathbb{B}}^2, \Rrightarrow \rangle$

- $\overline{\mathbb{B}}^2 \triangleq \{\mathtt{b}_p \rightsquigarrow \mathtt{b}_a \mid \mathtt{b}_p \in \mathbb{B} \wedge \mathtt{b}_a \in \mathbb{B} \wedge \mathtt{b}_p \not\Rrightarrow \mathtt{b}_a\}$

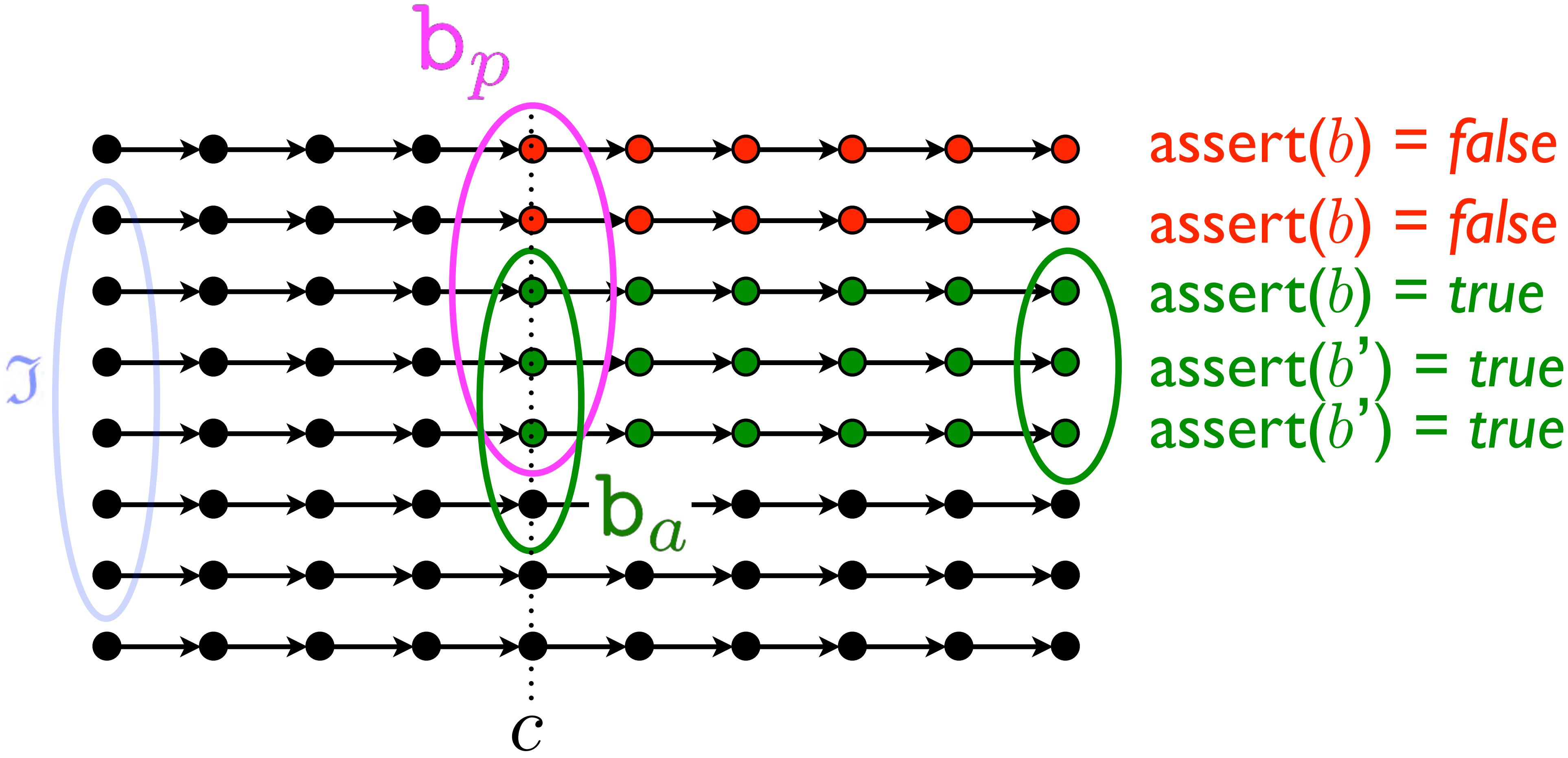
interpretation of $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a$: when the path condition $\mathtt{b}_p$ holds, an execution path will be followed to some `assert(b)` and checking $\mathtt{b}_a$ at the beginning of the path is the same as checking this `b` later in the path when reaching the assertion.

- Example $\quad$ `odd(x)` $\rightsquigarrow$ `y >= 0`

```
if ( odd(x) ) {
   y++;
   assert(y > 0);
} else {
   assert(y < 0); }
```

- $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a \Rrightarrow \mathtt{b}'_p \rightsquigarrow \mathtt{b}'_a \triangleq \mathtt{b}'_p \Rrightarrow \mathtt{b}_p \wedge \mathtt{b}_a \Rrightarrow \mathtt{b}'_a$. $\qquad$ order

Intuitive meaning of $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a$



Abstract domains $\langle \wp(\overline{\mathbb{B}}^2), \subseteq \rangle$ and $\Gamma \to \wp(\overline{\mathbb{B}}^2)$

- each $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a$ corresponding to a different path to an assertion
- a <u>set</u> of conditions , $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a$ attached to each program point
- **Example 16** The program on the left has abstract properties given on the right.

```
/* 1: */  if ( odd(x) ) {
/* 2: */    y++;
/* 3: */    assert(y > 0);
          } else {
/* 4: */    assert(y < 0); }
/* 5: */
```

$$\begin{aligned}
\rho(1) &= \{\mathtt{odd(x)} \rightsquigarrow \mathtt{y >= 0}, \neg\mathtt{odd(x)} \rightsquigarrow \mathtt{y < 0}\}\\
\rho(2) &= \{\mathtt{true} \rightsquigarrow \mathtt{y >= 0}\}\\
\rho(3) &= \{\mathtt{true} \rightsquigarrow \mathtt{y > 0}\}\\
\rho(4) &= \{\mathtt{true} \rightsquigarrow \mathtt{y < 0}\}\\
\rho(5) &= \emptyset
\end{aligned}$$

□

- **Infinitely many paths: widening**

A simple widening to enforce convergence would limit the size of the elements of $\wp(\overline{\mathbb{B}}^2)$, which is sound since eliminating a pair $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a$ would just lead to ignore some assertion in the precondition, which is always correct.

Concretization

- Concretization of $\mathtt{b}_p \leadsto \mathtt{b}_a$ for a given program point c

$$\gamma_c \in \overline{\mathbb{B}}^2 \rightarrow \wp(\{\vec{s} \in \vec{\Sigma}^+ \mid \boldsymbol{\pi}\vec{s}_0 = c\})$$

$$\gamma_c(\mathtt{b}_p \leadsto \mathtt{b}_a) \triangleq \{\vec{s} \in \vec{\Sigma}^+ \mid \boldsymbol{\pi}\vec{s}_0 = c \wedge [\![\mathtt{b}_p]\!]\vec{s}_0 \Rightarrow (\exists j < |\vec{s}| : [\![\mathtt{b}_a]\!]\vec{s}_0 = [\![\mathbb{A}(\boldsymbol{\pi}\vec{s}_j)]\!]\vec{s}_j)\}.$$

$$\mathbb{A}(\mathtt{c}) \triangleq \bigwedge\nolimits_{\langle \mathtt{c},\, \mathtt{b}\rangle \in \mathbb{A}} \mathtt{b}$$

- Concretization of a set of $\mathtt{b}_p \leadsto \mathtt{b}_a$ for a given program point c

$$\overline{\gamma}_c \in \wp(\overline{\mathbb{B}}^2) \rightarrow \wp(\{\vec{s} \in \vec{\Sigma}^+ \mid \boldsymbol{\pi}\vec{s}_0 = c\})$$

$$\overline{\gamma}_c(C) \triangleq \bigcap_{b_p \leadsto b_a \,\in C} \gamma_c(b_p \leadsto b_a)$$

- Concretization for all program points c

$$\dot{\gamma} \in (\Gamma \rightarrow \wp(\overline{\mathbb{B}}^2)) \rightarrow \wp(\vec{\Sigma}^+) \qquad \dot{\gamma} \text{ is decreasing}$$

$$\dot{\gamma}(\rho) \triangleq \bigcup_{c \in \Gamma} \{\vec{s} \in \overline{\gamma}_c(\rho(c)) \mid \boldsymbol{\pi}\vec{s}_0 = c\}$$

Command, successor and predecessor of a program point

— `c: x:=e; c':...`	$\mathsf{cmd}(c, c') \triangleq \texttt{x:=e}$	$\mathsf{succ}(c) \triangleq \{c'\}$	$\mathsf{pred}(c') \triangleq \{c\}$
— `c: assert(b); c':...`	$\mathsf{cmd}(c, c') \triangleq \texttt{b}$	$\mathsf{succ}(c) \triangleq \{c'\}$	$\mathsf{pred}(c') \triangleq \{c\}$
— `c: if b then`	$\mathsf{cmd}(c, c'_t) \triangleq \texttt{b}$	$\mathsf{succ}(c) \triangleq \{c'_t, c'_f\}$	
$\quad c'_t$`: ...`c''_t`:`	$\mathsf{cmd}(c, c'_f) \triangleq \neg\texttt{b}$		$\mathsf{pred}(c'_t) \triangleq \{c\}$
`else`	$\mathsf{cmd}(c''_t, c') \triangleq \texttt{skip}$	$\mathsf{succ}(c''_t) \triangleq \{c'\}$	
$\quad c'_f$`: ...`c''_f`:`	$\mathsf{cmd}(c''_f, c') \triangleq \texttt{skip}$	$\mathsf{succ}(c''_f) \triangleq \{c'\}$	$\mathsf{pred}(c'_f) \triangleq \{c\}$
`fi; c'...`			$\mathsf{pred}(c') \triangleq \{c''_t, c''_f\}$
— `c :while c': b do`	$\mathsf{cmd}(c, c') \triangleq \texttt{skip}$	$\mathsf{succ}(c) \triangleq \{c'\}$	$\mathsf{pred}(c') \triangleq \{c, c''_b\}$
$\quad c'_b$`: ...`c''_b`:`	$\mathsf{cmd}(c', c'_b) \triangleq \texttt{b}$	$\mathsf{succ}(c') \triangleq \{c'_b, c''\}$	$\mathsf{pred}(c'_b) \triangleq \{c'\}$
`od; c''...`	$\mathsf{cmd}(c', c'') \triangleq \neg\texttt{b}$	$\mathsf{succ}(c''_b) \triangleq \{c'\}$	$\mathsf{pred}(c'') \triangleq \{c'\}$
	$\mathsf{cmd}(c''_b, c) \triangleq \texttt{skip}$		

Backward symbolic execution

- We compute iteratively the under-approximation $\rho \mathrel{\dot{\subseteq}} \mathsf{lfp}^{\dot{\subseteq}}\, B$

- **Backward path condition and checked expression propagation.** The system of backward equations $\rho = B(\rho)$ is (recall that $\bigcup \emptyset = \emptyset$)

$$\begin{cases} B(\rho)\mathtt{c} \;=\; \displaystyle\bigcup_{\mathtt{c}'\in\mathsf{succ}(\mathtt{c}),\ b\rightsquigarrow b'\in\rho(\mathtt{c}')} B(\mathsf{cmd}(\mathtt{c},\mathtt{c}'), b\rightsquigarrow b') \cup \{\mathtt{true} \rightsquigarrow \mathtt{b} \mid \langle\mathtt{c},\ \mathtt{b}\rangle \in \mathbb{A}\} \\ \mathtt{c} \in \Gamma \end{cases}$$

where (writing $e[x := e']$ for the substitution of e' for x in e)

$$\begin{aligned}
B(\mathtt{skip}, \mathtt{b}_p \rightsquigarrow \mathtt{b}_a) &\triangleq \{\mathtt{b}_p \rightsquigarrow \mathtt{b}_a\} \\
B(\mathtt{x:=e}, \mathtt{b}_p \rightsquigarrow \mathtt{b}_a) &\triangleq \{\mathtt{b}_p[\mathtt{x} := \mathtt{e}] \rightsquigarrow \mathtt{b}_a[\mathtt{x} := \mathtt{e}]\} && \text{if } \mathtt{b}_p[\mathtt{x} := \mathtt{e}] \in \mathbb{B} \wedge \mathtt{b}_a[\mathtt{x} := \mathtt{e}] \in \mathbb{B} \\
& && \quad \wedge\, \mathtt{b}_p[\mathtt{x} := \mathtt{e}] \not\Rightarrow \mathtt{b}_c[\mathtt{x} := \mathtt{e}] \\
&\triangleq \emptyset && \text{otherwise} \\
B(\mathtt{b}, \mathtt{b}_p \rightsquigarrow \mathtt{b}_a) &\triangleq \{\mathtt{b}\ \&\&\ \mathtt{b}_p \rightsquigarrow \mathtt{b}_a\} && \text{if } \mathtt{b}\ \&\&\ \mathtt{b}_p \in \mathbb{B} \wedge \mathtt{b}\ \&\&\ \mathtt{b}_p \not\Rightarrow \mathtt{b}_a \\
&\triangleq \emptyset && \text{otherwise}
\end{aligned}$$

Soundness of the backward symbolic execution

Theorem 18 *If* $\rho \mathrel{\dot\subseteq} \mathsf{lfp}^{\dot\subseteq} B$ *then* $\vec{\tau}^{+} \subseteq \dot\gamma(\rho)$. $\square$

Observe that B can be $\dot\Rightarrow$-overapproximated (*e.g.* to allow for simplifications of the Boolean expressions).

Proof Apply Cor. 10 to $\vec{\tau}^{+} = \mathsf{gfp}^{\subseteq}_{\vec{\Sigma}^{+}} \boldsymbol{\lambda} \vec{T} \bullet \vec{\mathfrak{B}}^{1} \cup \vec{\tau}^{2} \mathbin{;} \vec{T}$ (1-b). $\square$

Example

Example 22 The analysis of the following program

```
/* 1: */   while (x != 0) {
/* 2: */      assert(x > 0);
/* 3: */      x--;
/* 4: */   }  /* 5: */
```

leads to the following iterates at program point `1`:

$$
\begin{array}{lll}
\rho^0(\mathtt{1}) & = & \emptyset \qquad\qquad\qquad\qquad \text{Initialization} \\
\rho^1(\mathtt{1}) & = & \{\mathtt{x} \neq 0 \leadsto \mathtt{x} > 0\} \\
\rho^2(\mathtt{1}) & = & \rho^1(\mathtt{1}) \qquad\qquad\qquad \text{since } (\mathtt{x} \neq 0 \wedge \mathtt{x} > 0 \wedge \mathtt{x} - 1 \neq 0) \leadsto (\mathtt{x} - 1 > 0) \\
 & & \qquad\qquad\qquad\qquad\quad \equiv \mathtt{x} > 1 \leadsto \mathtt{x} > 1
\end{array}
$$

□

Backward symbolic execution-based precondition generation

Given an analysis $\rho \mathrel{\dot{\subseteq}} \mathsf{lfp}^{\subseteq} B$, the syntactic precondition generated at entry control point $i \in \mathfrak{I}_{\boldsymbol{\pi}} \triangleq \{i \in \Gamma \mid \exists s \in \mathfrak{I} : \boldsymbol{\pi} s = i\}$ is

$$\mathtt{P}_i \triangleq \mathop{\&\&}_{\mathtt{b}_p \rightsquigarrow \mathtt{b}_a \in \rho(i)} (\mathtt{!}(\mathtt{b}_p)\ \mathtt{||}\ (\mathtt{b}_a)) \qquad \text{(again, assuming } \&\&\,\emptyset \triangleq \mathit{true}\text{)}$$

Example

```
                  !(x != 0) || (x > 0)
/* 1: */    while (x != 0) {
/* 2: */        assert(x > 0);
/* 3: */        x--;
/* 4: */    }   /* 5: */
```

forward analysis
from precondition

(IV) Forward analysis for collections

General idea

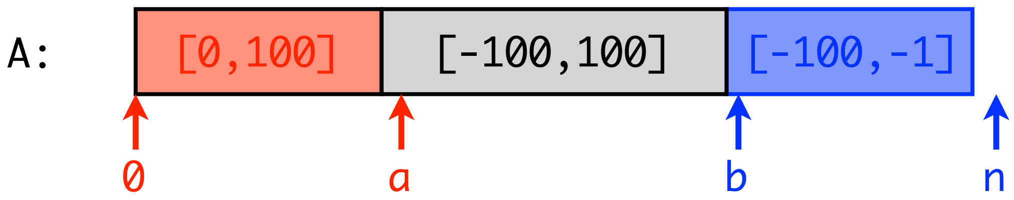
- The previous analyzes for scalar variables can be applied elementwise to collections
  $\Longrightarrow$ much too costly

- Apply segmentwise to collections!

- Forward or backward symbolic execution might be costly, an efficient solution is needed
  $\Longrightarrow$ segmented forward dataflow analysis

Recall on segmentation (from last year talk[(*)])

- Example



A: <{0},[0,100],{a}?,[-100,100],{b}?,[-100,-1],{n}?>

- Formally, the abstract domain functor is

$$\overline{\mathcal{S}}(\overline{\mathcal{A}}) \triangleq \{(\overline{\mathcal{B}} \times \overline{\mathcal{A}}) \times (\overline{\mathcal{B}} \times \overline{\mathcal{A}} \times \{\textvisiblespace, ?\})^k \times (\overline{\mathcal{B}} \times \{\textvisiblespace, ?\}) \mid k \geqslant 0\} \cup \{\perp\}$$

$$\{e_1^1 \ \ldots\ e_{m^1}^1\}\, A_1\, \{e_1^2 \ \ldots\ e_{m^2}^2\}[?^2]\, A_2 \ \ldots\ A_{n-1}\, \{e_1^n \ \ldots\ e_{m^n}^n\}[?^n]$$

expressions on scalar variables (all have equal values)	lower bound of segment (included)	abstract property of all elements in segment	upper bound of segment (excluded)	possible emptyness of segment

?: segment may be empty, ␣ segment is not empty

(*) Tech. Rept. no. MSR-TR-2009-194, Sep. 2009, submitted.

Basic abstract domains for segments

- Modification analysis

$$\overline{\mathcal{M}} \triangleq \{\mathfrak{e}, \mathfrak{d}\} \qquad \mathfrak{e} \sqsubseteq \mathfrak{e} \sqsubset \mathfrak{d} \sqsubseteq \mathfrak{d}.$$

$\mathfrak{e}$: all elements in the segment must be equal to their initial value

$\mathfrak{d}$: otherwise

- Checking analysis

$$\overline{\mathcal{C}} \triangleq \{\mathfrak{n}, \mathfrak{c}\} \qquad \mathfrak{n} \sqsubseteq \mathfrak{n} \sqsubset \mathfrak{c} \sqsubseteq \mathfrak{c}$$

$\mathfrak{c}$: all elements `A[`i`]` in the segment must have been checked in assert(`b(A[`i`])`) while equal to their initial value (as determined by the above modification analysis)

$\mathfrak{n}$: otherwise

Abstract domain for collections

Segment modification analysis

Segment checking analysis

$$\xi \in \Gamma \rightarrow \mathtt{X} \in \mathbb{X} \mapsto \overline{\mathcal{S}}(\overline{\mathcal{M}}) \times \mathbb{A}(\mathtt{X}) \rightarrow \overline{\mathcal{S}}(\overline{\mathcal{C}})$$

Program point

Collection variable

Assertions on x

For each assertion in $\langle \mathtt{c}, \mathtt{b(X,i)} \rangle \in \mathbb{A}(\mathtt{X})$ (where `c` is a program point designating an `assert(b)` and `b(X,i)` is a side effect free Boolean expression checking a property of element `X[i]` of collection `X` [9])

---

(9) If more than one index is used, like in `assert(A[i]<A[i+1])` or `assert(A[i]<A[A.length-i])`, the modification analysis must check that the array `A` has not been modified for all these indexes.

Example : (I) program

```
         void AllNotNull(Ptr[] A) {
/* 1: */  int i = 0;
/* 2: */  while /* 3: */
             (assert(A != null); i < A.length) {
/* 4: */

/* 4: */   assert((A != null) && (A[i] != null));
/* 5: */   A[i].f = new Object();
/* 6: */   i++;
/* 7: */  }
/* 8: */ }
```

Example : (IIa) analysis

```
          void AllNotNull(Ptr[] A) {
/* 1: */   int i = 0;
/* 2: */   while /* 3: */
              (assert(A != null); i < A.length) {
/* 4: */
           {0}𝔡{i}𝔢{A.length} - {0}𝔠{i}𝔫{A.length}
/* 4: */     assert((A != null) && (A[i] != null));
/* 5: */     A[i].f = new Object();
/* 6: */     i++;
/* 7: */   }
/* 8: */ }  {0}𝔡{i,A.length}? - {0}𝔠{i,A.length}?
```

Example : (IIb) modification analysis

```
             void AllNotNull(Ptr[] A) {
/* 1: */       int i = 0;
/* 2: */       while /* 3: */
                   (assert(A != null); i < A.length) {
/* 4: */
                {0}𝔡{i}𝔢{A.length} - {0}𝔠{i}𝔫{A.length}
/* 4: */          assert((A != null) && (A[i] != null));
/* 5: */          A[i].f = new Object();
/* 6: */          i++;
/* 7: */        }
/* 8: */     }  {0}𝔡{i,A.length}? - {0}𝔠{i,A.length}?
```

`(A[i] != null)` is checked while `A[i]` unmodified since code entry

Example : (III) result

```
          void AllNotNull(Ptr[] A) {
/* 1: */   int i = 0;
/* 2: */   while /* 3: */
              (assert(A != null); i < A.length) {
/* 4: */
           {0}𝔡{i}𝔢{A.length} - {0}𝔠{i}𝔫{A.length}
/* 4: */    assert((A != null) && (A[i] != null));
/* 5: */    A[i].f = new Object();
/* 6: */    i++;
/* 7: */   }
/* 8: */ }  {0}𝔡{i,A.length}? - {0}𝔠{i,A.length}?
```

`(A[i] != null)` is checked while `A[i]` unmodified since code entry

all `A[i]` have been checked in `(A[i] != null)` while unmodified since code entry

Details of the analysis

(*a*) 1: $\{0\}\mathfrak{e}\{\texttt{A.length}\}?$ - $\{0\}\mathfrak{n}\{\texttt{A.length}\}?$
no element yet modified ($\mathfrak{e}$) and none checked ($\mathfrak{n}$), array may be empty

(*b*) 2: $\{0,\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}?$ - $\{0,\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}?$ $\quad \texttt{i} = 0$

(*c*) 3: $\bot \sqcup (\{0,\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}?$ - $\{0,\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}?)$ join
$= \{0,\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}?$ - $\{0,\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}?$

(*d*) 4: $\{0,\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}$ - $\{0,\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}$
last and only segment hence array not empty (since $\texttt{A.length} > \texttt{i} = 0$)

(*e*) 5: $\{0,\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}$ - $\{0,\texttt{i}\}\mathfrak{c}\{1,\texttt{i+1}\}\mathfrak{n}\{\texttt{A.length}\}?$
A[i] checked while unmodified

(*f*) 6: $\{0,\texttt{i}\}\mathfrak{d}\{1,\texttt{i+1}\}\mathfrak{e}\{\texttt{A.length}\}?$ - $\{0,\texttt{i}\}\mathfrak{c}\{1,\texttt{i+1}\}\mathfrak{n}\{\texttt{A.length}\}?$
A[i] has been modified

(*g*) 7: $\{0,\texttt{i-1}\}\mathfrak{d}\{1,\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}?$ - $\{0,\texttt{i-1}\}\mathfrak{c}\{1,\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}?$
invertible assignment $\texttt{i}_{\text{old}} = \texttt{i}_{\text{new}} - 1$

(*h*) 3: $\{0,\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}? \sqcup \{0,\texttt{i-1}\}\mathfrak{d}\{1,\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}?$ - join
$\{0,\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}? \sqcup \{0,\texttt{i-1}\}\mathfrak{c}\{1,\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}?$
$= \{0\}\mathfrak{e}\{\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}? \sqcup \{0\}\mathfrak{d}\{\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}?$ - segment unification
$\{0\}\mathfrak{n}\{\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}? \sqcup \{0\}\mathfrak{c}\{\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}?$
$= \{0\}\mathfrak{d}\{\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}?$ - $\{0\}\mathfrak{c}\{\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}?$
segmentwise join $\mathfrak{e} \sqcup \mathfrak{e} = \mathfrak{e}$, $\mathfrak{e} \sqcup \mathfrak{d} = \mathfrak{d}$, $\mathfrak{n} \sqcup \mathfrak{n} = \mathfrak{n}$, $\mathfrak{n} \sqcup \mathfrak{c} = \mathfrak{c}$

(*i*) 4: $\{0\}\mathfrak{d}\{\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}$ - $\{0\}\mathfrak{c}\{\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}$ last segment not empty

(*j*) 5: $\{0\}\mathfrak{d}\{\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}$ - $\{0\}\mathfrak{c}\{\texttt{i}\}\mathfrak{c}\{\texttt{i+1}\}\mathfrak{n}\{\texttt{A.length}\}?$
A[i] checked while unmodified

(*k*) 6: $\{0\}\mathfrak{d}\{\texttt{i}\}\mathfrak{d}\{\texttt{i+1}\}\mathfrak{e}\{\texttt{A.length}\}?$ - $\{0\}\mathfrak{c}\{\texttt{i}\}\mathfrak{c}\{\texttt{i+1}\}\mathfrak{n}\{\texttt{A.length}\}?$
A[i] has been modified

(*l*) 7: $\{0\}\mathfrak{d}\{\texttt{i-1}\}\mathfrak{d}\{\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}?$ - $\{0\}\mathfrak{c}\{\texttt{i-1}\}\mathfrak{c}\{\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}?$
invertible assignment $\texttt{i}_{\text{old}} = \texttt{i}_{\text{new}} - 1$

(*m*) 3: $\{0\}\mathfrak{d}\{\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}? \sqcup \{0\}\mathfrak{d}\{\texttt{i-1}\}\mathfrak{d}\{\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}?$ - join
$\{0\}\mathfrak{c}\{\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}? \sqcup \{0\}\mathfrak{c}\{\texttt{i-1}\}\mathfrak{c}\{\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}?$
$= \{0\}\mathfrak{d}\{\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}? \sqcup \{0\}\mathfrak{d}\{\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}?$ - segment unification
$\{0\}\mathfrak{c}\{\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}? \sqcup \{0\}\mathfrak{c}\{\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}?$
$= \{0\}\mathfrak{d}\{\texttt{i}\}\mathfrak{e}\{\texttt{A.length}\}?$ - $\{0\}\mathfrak{c}\{\texttt{i}\}\mathfrak{n}\{\texttt{A.length}\}?$
segmentwise join, convergence

(*m*) 8: $\{0\}\mathfrak{d}\{\texttt{i},\texttt{A.length}\}?$ - $\{0\}\mathfrak{c}\{\texttt{i},\texttt{A.length}\}?$
$i \leqslant \texttt{A.length}$ in segmentation and $\geqslant$ in test negation so $\texttt{i} = \texttt{A.length}$.

Just to show that the analysis is very fast!

Code generated for the precondition

- Result of the checking analysis (at any point dominating the code exit) for an `assert(b(X,i))` on collection `X` at a program point `c`

$$B_1 C_1 B_2[?^2] C_2 \ \ldots C_{n-1} B_n[?^n] \in \overline{\mathcal{S}}(\overline{\mathcal{C}})$$

- Let $\Delta \subseteq [1, n)$ be the set of indices $k \in \Delta$ for which $C_k = \mathtt{c}$.

- The precondition is

$$\mathop{\&\&}_{\mathtt{X} \in \mathbb{X}} \ \mathop{\&\&}_{\langle \mathtt{c},\, \mathtt{b(X,i)} \rangle \in \mathbb{A}(\mathtt{X})} \ \mathop{\&\&}_{k \in \Delta} \ \mathtt{ForAll}(\mathtt{l}_k, \mathtt{h}_k, \mathtt{i} \Rightarrow \mathtt{b}(\mathtt{X}, \mathtt{i})) \tag{4}$$

where $\exists e_k \in B_k, e'_k \in B_{k+1}$ such that the value of e_k (resp. e'_k) at program point $\mathtt{f}$ is always equal to that of $\mathtt{l}_k$ (resp. $\mathtt{h}_k$) on program entry and is less that the size of the collection on program entry.

Theorem 23 *The precondition* (4) *based on a sound modification and checking static analysis* ξ *is sound.*

Related work

Related work

- Static contract checking

  ...

  - Barnett, M., Fähndrich, M., Garbervetsky, D., Logozzo, F.: Annotations for (more) precise points-to analysis. In: IWACO '07. DSV Report series No. 07-010, Stockholm University and KTH (2007)
  - Barnett, M., Fähndrich, M., Logozzo, F.: Embedded contract languages. In: SAC'10. pp. 2103–2110. ACM Press (2010)

- Abstract interpretation

  ...

  - Fähndrich, M., Logozzo, F.: Clousot: Static contract checking with abstract interpretation. In: FoVeOOS: Conference on Formal Verification of Object-Oriented software. Springer-Verlag (2010)
  - Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. Tech. rep., MSR-TR-2009-194, MSR Redmond (Sep 2009)

Related work (cont'd)

- Of course, (set-based, weakest) precondition for correctness (and termination):
  - Dijkstra, E.: Guarded commands, nondeterminacy and formal derivation of programs. CACM 18(8), 453–457 (1975)
- Many analyzes to determine sufficient conditions for the code to satisfy the assertions (and terminate)
  - Cousot, P.: Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French). Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble (1978)
  - Cousot, P.: Semantic foundations of program analysis. In: Muchnick, S., Jones, N. (eds.) Program Flow Analysis: Theory and Applications, chap. 10, pp. 303–342. Prentice-Hall (1981)
  - Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In: Neuhold, E. (ed.) IFIP Conf. on Formal Description of Programming Concepts. pp. 237–277. North-Holland (1977)
  - Bourdoncle, F.: Abstract debugging of higher-order imperative languages. In: PLDI '93. pp. 46–55. ACM Press (1993)
- etc, etc.

Conclusion

Precondition inference from assertions

- Our point of view that only definite (and not potential) assertion violations should be checked in preconditions looks original
- The analyzes for scalar and collection variables have been chosen to be simple
  - for scalability of the analyzes
  - for understandability of the automatic program annotation
- Remains to be implemented

Thanks to all for this very nice visit