Precondition Inference from Intermittent Assertions and Application to Contract on Collections

Patrick Cousot (ENS & NYU)
Radhia Cousot (CNRS, ENS & MSR Redmond)
Francesco Logozzo (MSR Redmond)

VMCAI 2011, Austin, TX

January 25, 2011

Objective

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

Usefullness

- Anticipate errors at runtime (e.g. change to trace execution mode before actual error does occur)
- *Main motivation:* use contracts for separate static analysis of modules (in Clousot)

Example

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

Example

language assertions

```
         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: */ }
```

Example

From the language assertions

```
              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: */ }
```

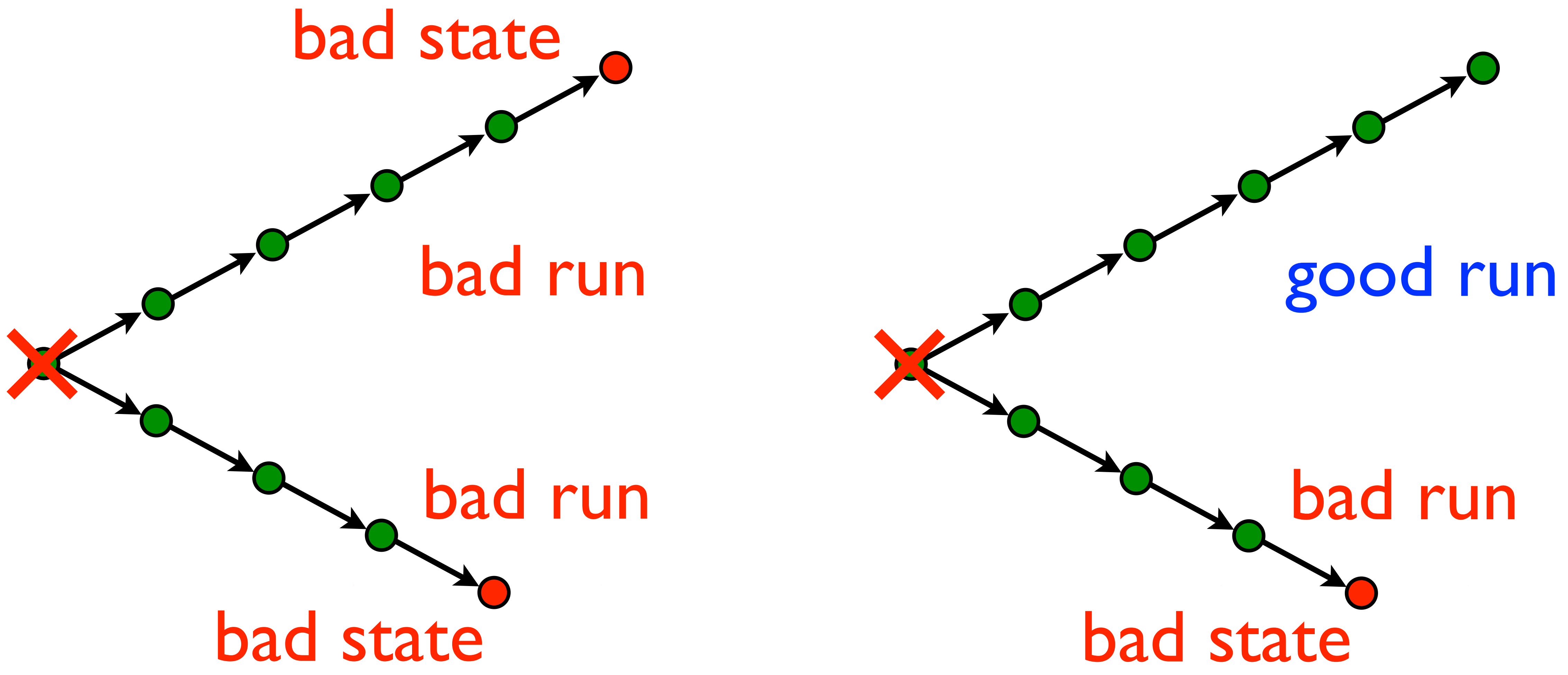
infer the precondition

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

Understanding the problem

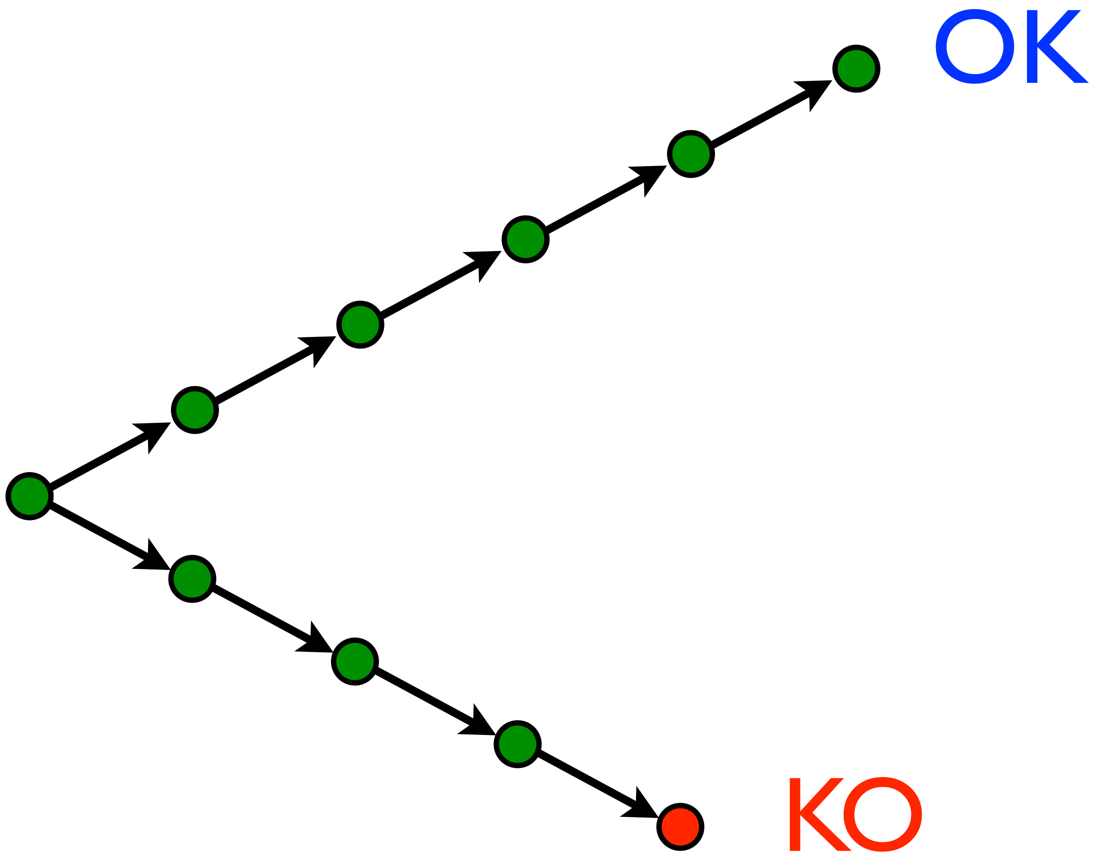
First alternative: eliminating *potential* errors

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



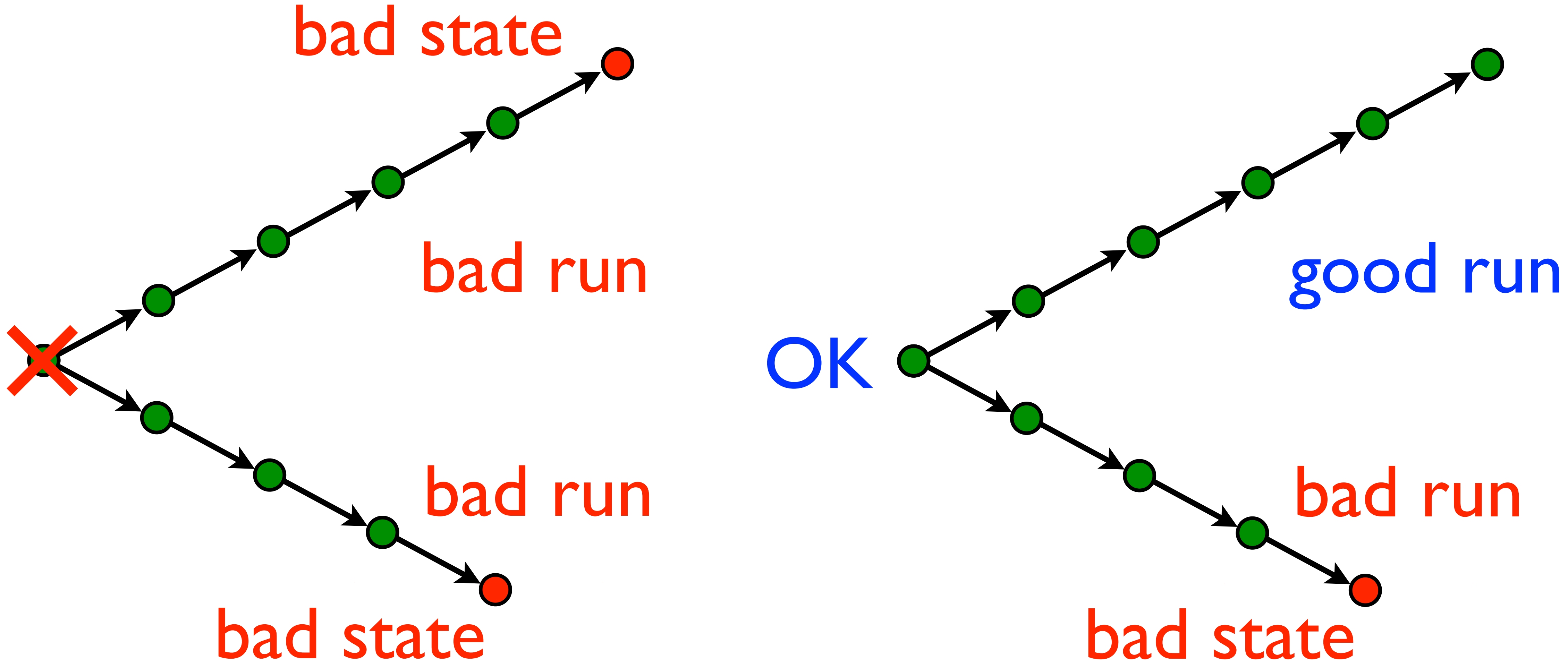
Defects of *potential* error elimination

- A priori correctness point of view
- Makes hypotheses on the programmer's intentions



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)



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

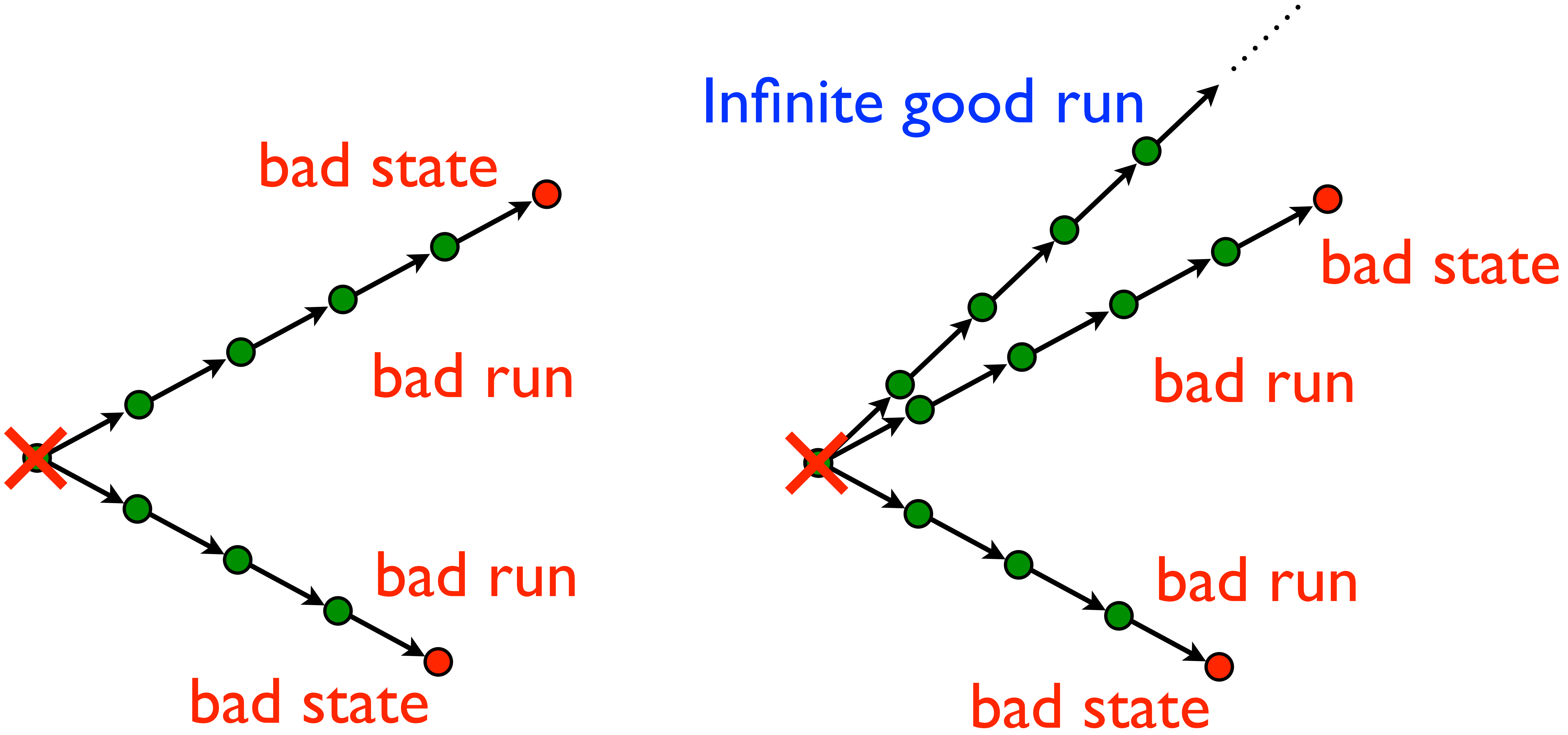
```
Collatz(p);
assert(false);
```

- 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



Semantics

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 run semantics

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

$$n \geqslant 0$$

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

finite partial runs

Program maximal run semantics

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

$n \geqslant 0$

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

finite maximal runs

$$\vec{\tau}^{\,+}_{\mathfrak{I}} \triangleq \{\vec{s} \in \vec{\tau}^{\,+} \mid \vec{s}_0 \in \mathfrak{I}\}$$

$\mathfrak{I}$ initial states

Fixpoint maximal run semantics

$$\begin{aligned}\vec{\tau}^{+} &= \mathsf{lfp}^{\subseteq}_{\emptyset}\ \boldsymbol{\lambda} \vec{T} \boldsymbol{\cdot} \vec{\mathfrak{B}}^{1} \cup \overset{\rightleftarrows}{\tau}^{2} \fatsemi \vec{T} \\ &= \mathsf{gfp}^{\subseteq}_{\vec{\Sigma}^{+}}\ \boldsymbol{\lambda} \vec{T} \boldsymbol{\cdot} \vec{\mathfrak{B}}^{1} \cup \overset{\rightleftarrows}{\tau}^{2} \fatsemi \vec{T}\end{aligned}$$

where

- *sequential composition* of traces is $\vec{s}s \fatsemi s\vec{s}\,' \triangleq \vec{s}s\vec{s}\,'$
- $\vec{S} \fatsemi \vec{S}\,' \triangleq \{\vec{s}s\vec{s}\,' \mid \vec{s}s \in \vec{S} \cap \vec{\Sigma}^{+} \wedge s\vec{s}\,' \in \vec{S}\,'\}$
- 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

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

where

- $\mathtt{assert(b}_j\mathtt{)}$ is attached to a control point $\mathtt{c}_j \in \Gamma, j \in \Delta$
- $\mathtt{b}_j$: well defined and visible side effect free

Evaluation of expressions

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

Control

- Map $\pi \in \Sigma \to \mathit{\Gamma}$ of states of Σ into *control points* in $\mathit{\Gamma}$

  (of finite cardinality)

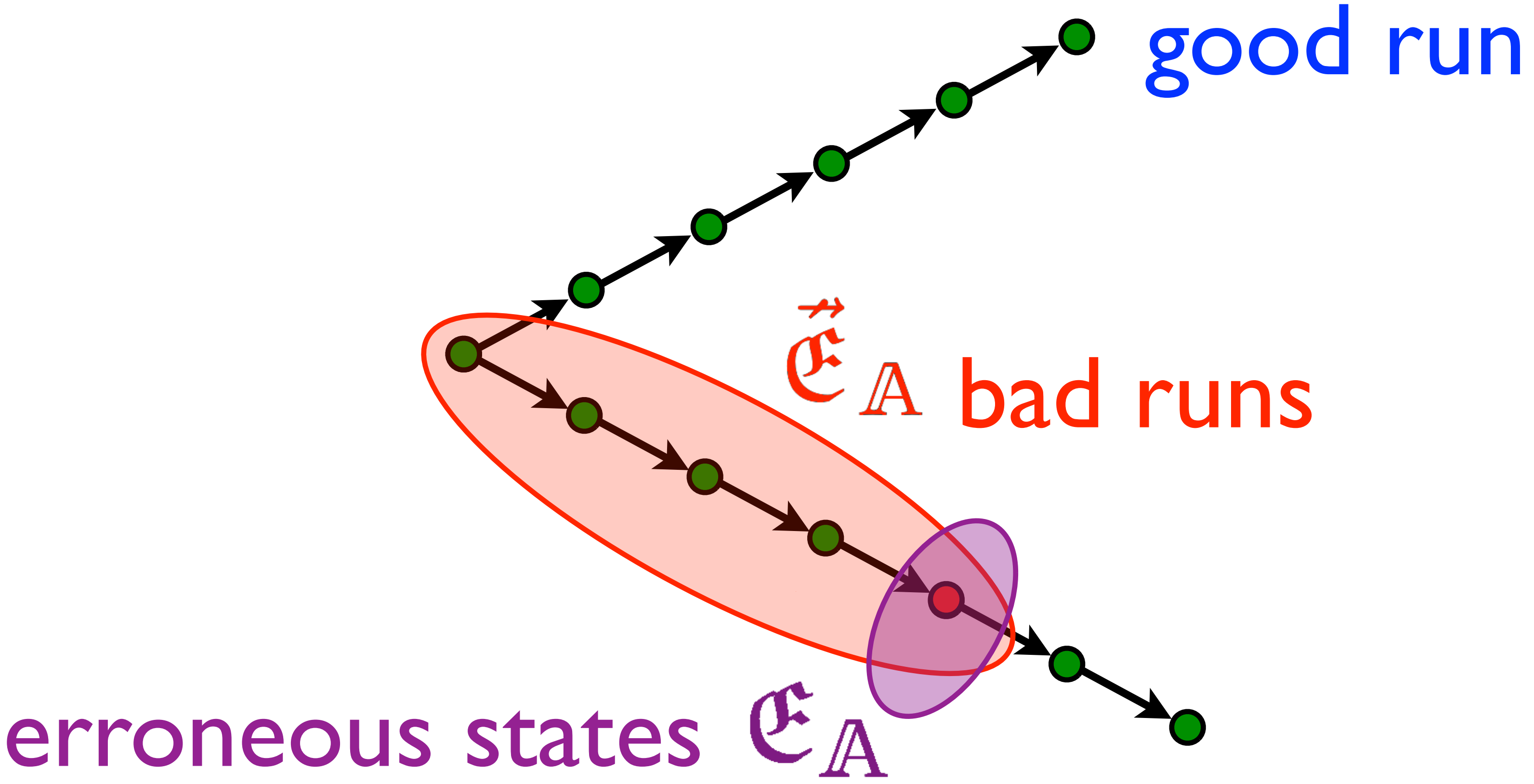
Bad states and bad traces

- Erroneous/bad states

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

- Erroneous/bad traces

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



Formal specification of the contract inference problem

The contract inference problem

- Effectively compute a condition $P_{\mathbb{A}}$ restricting the initial states $\mathfrak{I}$ such that
  - no new run is introduced

$$\vec{\tau}^{\,+}_{P_{\mathbb{A}}\cap\mathfrak{I}} \subseteq \vec{\tau}^{\,+}_{\mathfrak{I}}$$

  - all eliminated runs are bad runs

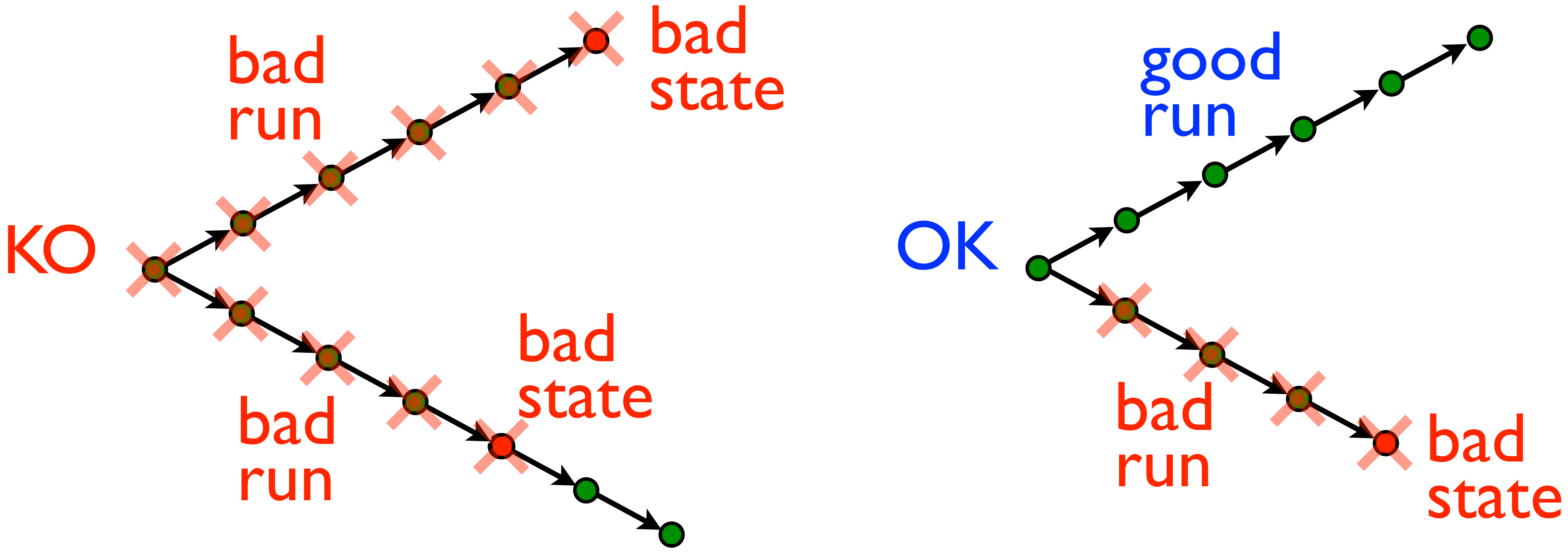
$$\vec{\tau}^{\,+}_{\mathfrak{I}\setminus P_{\mathbb{A}}} = \vec{\tau}^{\,+}_{\mathfrak{I}} \setminus \vec{\tau}^{\,+}_{P_{\mathbb{A}}} \subseteq \vec{\vec{\mathfrak{E}}}_{\mathbb{A}}$$

so that no finite maximal good run is ever eliminated

- Trivial solution: $P_{\mathbb{A}} = \mathfrak{I}$ so that $\mathfrak{I} \setminus P_{\mathbb{A}} = \emptyset$ hence $\vec{\tau}^{\,+}_{\mathfrak{I}\setminus P_{\mathbb{A}}} = \emptyset$

The *strongest* (5) solution

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

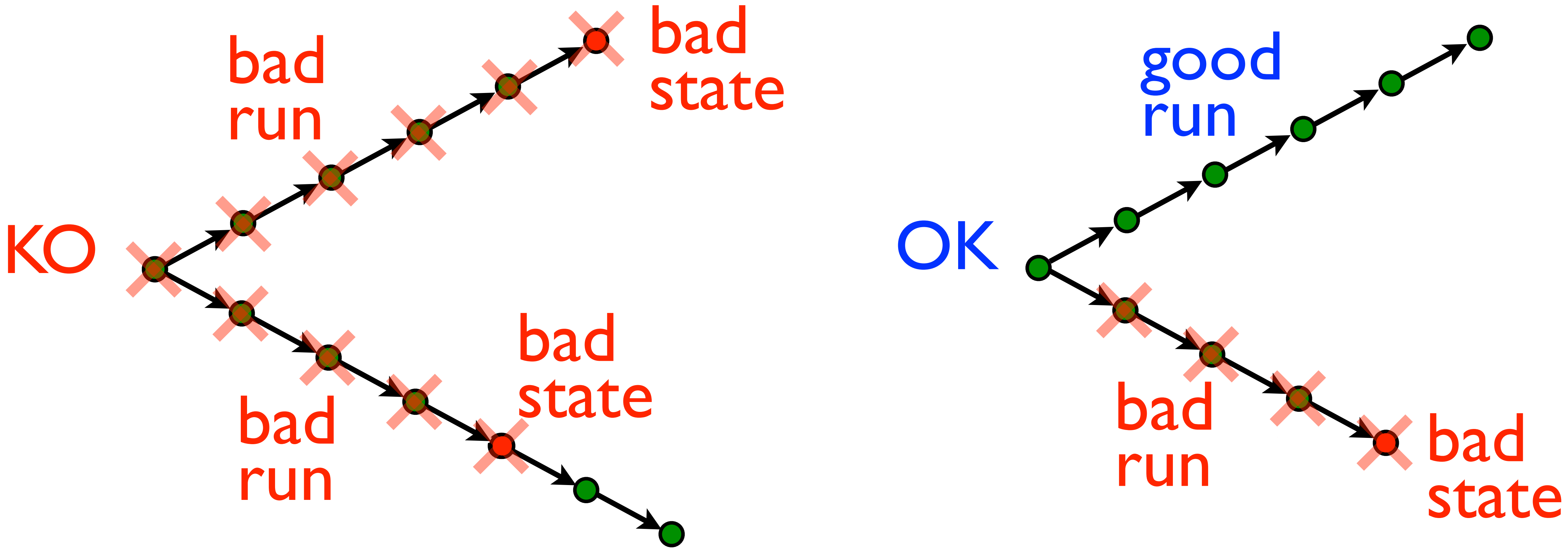


---

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

The *strongest* (5) solution

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



It is correct to under-approximate $\mathfrak{P}_{\mathbb{A}}$, but incorrect to over-approximate!

---

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

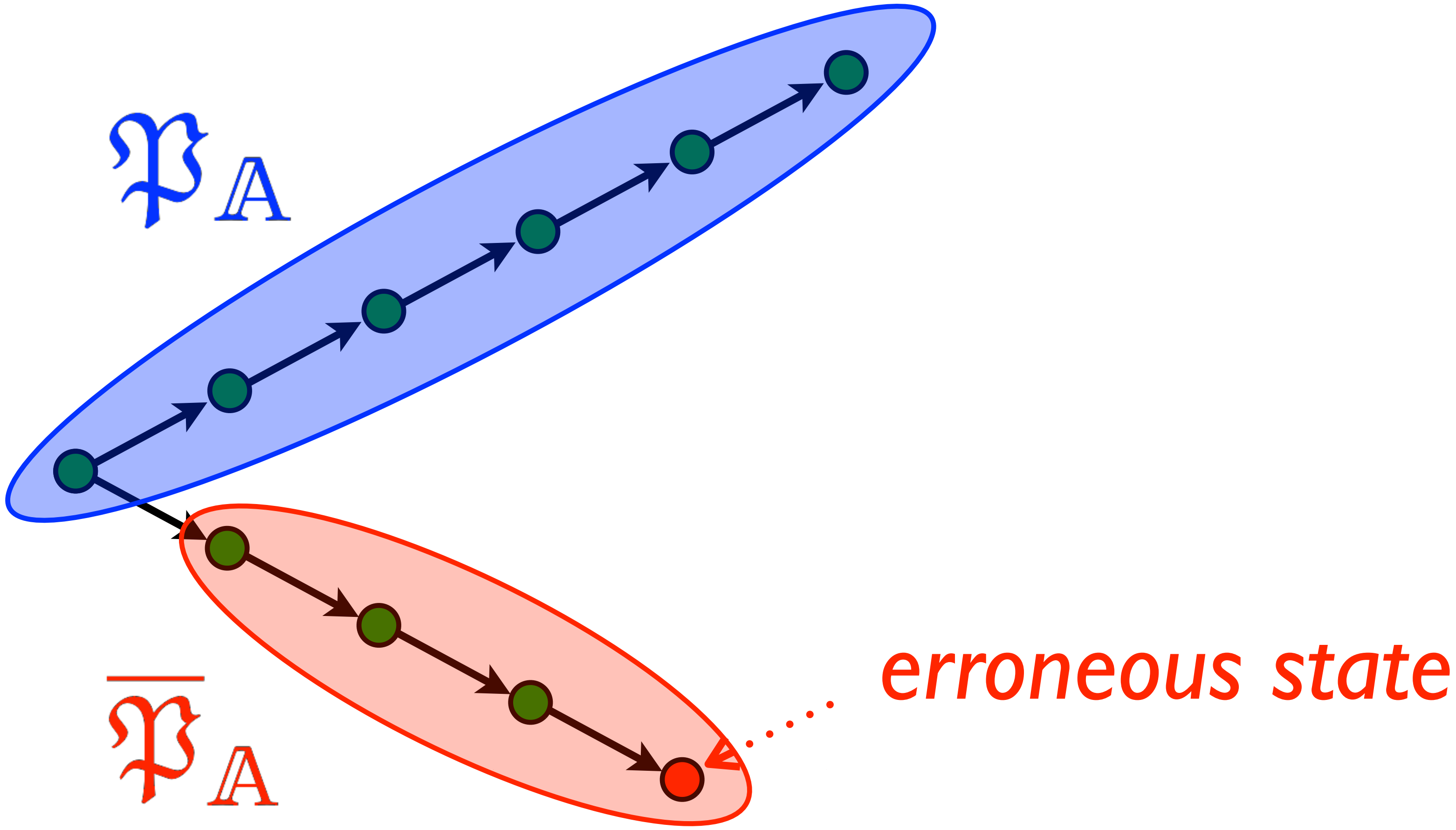
Good and bad states

- **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

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



Fixpoint strongest contract precondition (collecting semantics)

Trace predicate transformers are abstractions

- Trace predicate transformers (*)

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

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

- Galois connection

$$\begin{tikzcd}[column sep=huge] \langle \wp(\vec{\Sigma}^{+}), \subseteq \rangle \arrow[r, shift right, "\boldsymbol{\lambda}\, \vec{T} \bullet \text{wlp}[\vec{T}]\vec{Q}"'] & \langle \wp(\Sigma), \supseteq \rangle \arrow[l, shift right, "\text{wlp}^{-1}[\vec{Q}]"'] \end{tikzcd}$$

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

$$\overline{\mathfrak{P}}_{\mathbb{A}} = \text{wlp}[\vec{\tau}^{+}](\vec{\vec{\mathfrak{E}}}_{\mathbb{A}})$$

$$= \{s \mid \forall s\vec{s} \in \vec{\tau}^{+} : s\vec{s} \in \vec{\vec{\mathfrak{E}}}_{\mathbb{A}}\}$$

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

Fixpoint abstraction(8)

Lemma 7 *If* $\langle L, \leqslant, \perp\rangle$ *is a complete lattice or a cpo,* $F \in L \rightarrow L$ *is increasing,* $\langle \overline{L}, \sqsubseteq\rangle$ *is a poset,* $\alpha \in L \rightarrow \overline{L}$ *is continuous*$^{(6),(7)}$, $\overline{F} \in \overline{L} \rightarrow \overline{L}$ *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.* $\alpha(\mathsf{lfp}^{\leqslant}_{\perp} F) \sqsubseteq \mathsf{lfp}^{\sqsubseteq}_{\alpha(\perp)} \overline{F}$*).*

---

(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) Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6th POPL. pp. 269–282. ACM Press (1979)

Fixpoint abstraction (8)

Lemma 7 *If* $\langle L, \leqslant, \perp\rangle$ *is a complete lattice or a cpo,* $F \in L \rightarrow L$ *is increasing,* $\langle \overline{L}, \sqsubseteq\rangle$ *is a poset,* $\alpha \in L \rightarrow \overline{L}$ *is continuous*(6),(7)*,* $\overline{F} \in \overline{L} \rightarrow \overline{L}$ *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.* $\alpha(\mathsf{lfp}^{\leqslant}_{\perp} F) \sqsubseteq \mathsf{lfp}^{\sqsubseteq}_{\alpha(\perp)} \overline{F}$*).*

Example: Park theorem

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

so $$\neg\, \mathsf{lfp}^{\leqslant}_{\perp} F = \mathsf{gfp}^{\leqslant}_{\neg\perp}\, \neg \circ F \circ \neg$$

---

(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.

(8) Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6th POPL. pp. 269–282. ACM Press (1979)

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:

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

$\square$

Contract precondition inference by abstract interpretation

Under-approximations

- Extremely hard not to be trivial:
    - Tests
    - Bounded model checking

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

- All our proposed solutions:

  symbolic under-approximations by program expression propagation

(I) Forward symbolic execution

General 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)

(II) Backward expression propagation

General idea

- Try to move the condition code in assertions at the beginning of the program/method/...
- This is possible under sufficient conditions:
  - The checked condition has the same value on entry and when checked in `asserts`
  - It is checked in an `assert` on all possible paths from entry
- We derive a sound backward dataflow analysis by abstraction of the trace semantics
- Too *imprecise*

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: */ }
```

Example 13 Continuing Ex. 1, 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. ☐

(III) Backward *path condition* and expression propagation

General idea

- Try to move the condition code in assertions at the beginning of the program/method/... keeping track of the path condition

- Example:

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

If this condition is true now then control must lead to an `assert(b)` where...

...testing this condition now is the same as testing the `assert(b)` condition `b` later

Example

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

forward analysis
from precondition

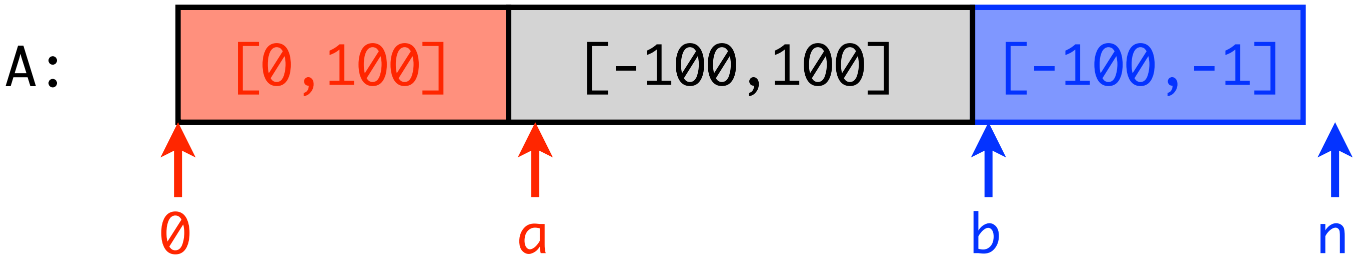
(IV) Forward analysis for collections

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

Segmentation (*)

- Example



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

(*) see POPL'2011.

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, may be different

- Checking analysis

$$\overline{\mathcal{C}} \triangleq \{\bot, \mathfrak{n}, \mathfrak{c}. \top\} \qquad \bot \sqsubset \mathfrak{n} \sqsubset \top \quad \bot \sqsubset \mathfrak{c} \sqsubset \top$$

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

$\mathfrak{n}$: none of the elements have been checked yet

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}?
```

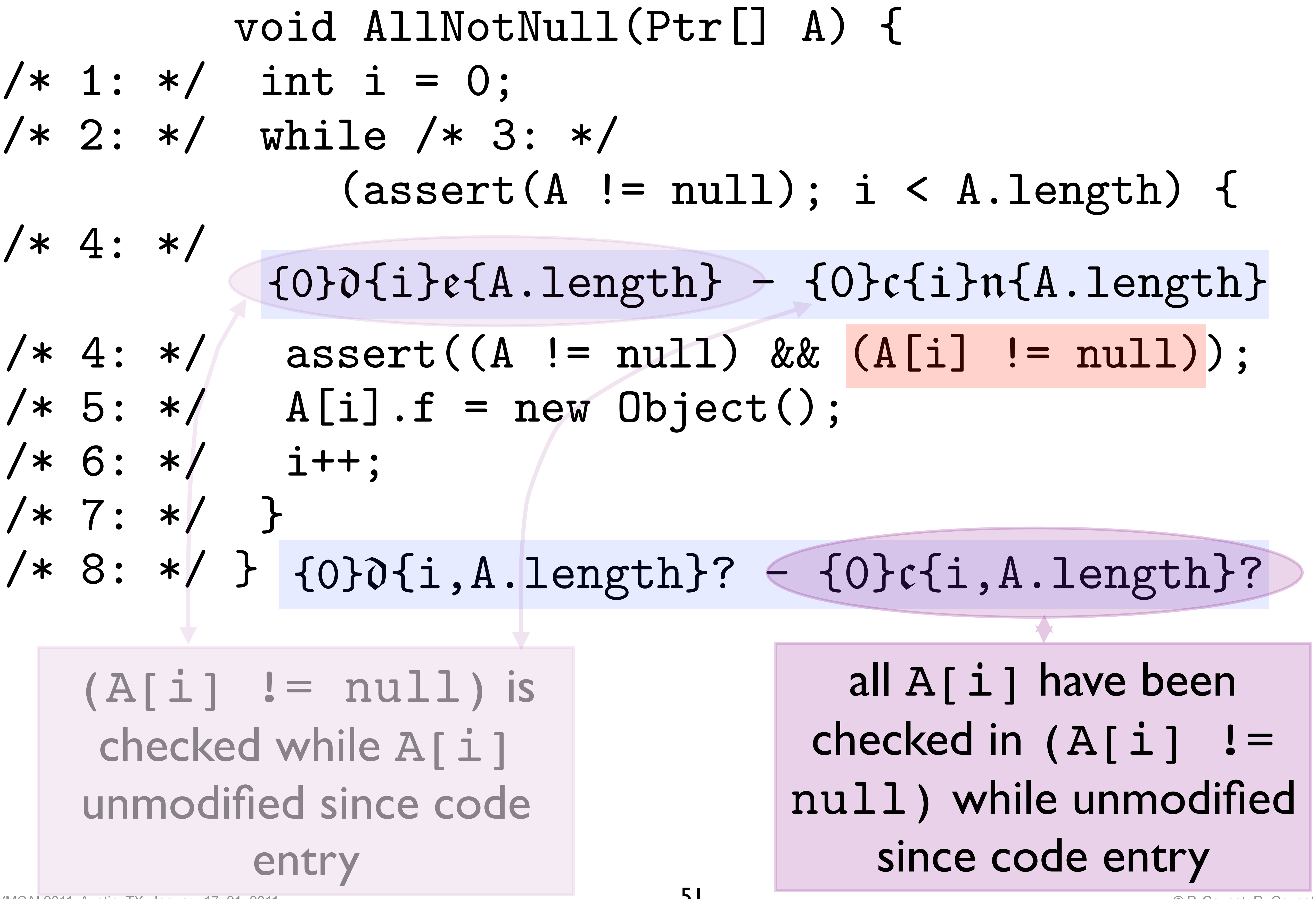
Example : (IIb) modification analysis

```
          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*) $1\text{: } \{0\}\mathfrak{e}\{\mathtt{A.length}\}?\ \text{-}\ \{0\}\mathfrak{n}\{\mathtt{A.length}\}?$

no element yet modified ($\mathfrak{e}$) and none checked ($\mathfrak{n}$), array may be empty

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

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

$= \{0,\mathtt{i}\}\mathfrak{e}\{\mathtt{A.length}\}?\ \text{-}\ \{0,\mathtt{i}\}\mathfrak{n}\{\mathtt{A.length}\}?$

(*d*) $4\text{: } \{0,\mathtt{i}\}\mathfrak{e}\{\mathtt{A.length}\}\ \text{-}\ \{0,\mathtt{i}\}\mathfrak{n}\{\mathtt{A.length}\}$

last and only segment hence array not empty (since $\mathtt{A.length} > \mathtt{i} = 0$)

(*e*) $5\text{: } \{0,\mathtt{i}\}\mathfrak{e}\{\mathtt{A.length}\}\ \text{-}\ \{0,\mathtt{i}\}\mathfrak{c}\{1,\mathtt{i+1}\}\mathfrak{n}\{\mathtt{A.length}\}?$

`A[i]` checked while unmodified

(*f*) $6\text{: } \{0,\mathtt{i}\}\mathfrak{d}\{1,\mathtt{i+1}\}\mathfrak{e}\{\mathtt{A.length}\}?\ \text{-}\ \{0,\mathtt{i}\}\mathfrak{c}\{1,\mathtt{i+1}\}\mathfrak{n}\{\mathtt{A.length}\}?$

`A[i]` appears on the left handside of an assignment, hence is potentially modified

(*g*) $7\text{: } \{0,\mathtt{i-1}\}\mathfrak{d}\{1,\mathtt{i}\}\mathfrak{e}\{\mathtt{A.length}\}?\ \text{-}\ \{0,\mathtt{i-1}\}\mathfrak{c}\{1,\mathtt{i}\}\mathfrak{n}\{\mathtt{A.length}\}?$

invertible assignment $\mathtt{i}_{\text{old}} = \mathtt{i}_{\text{new}} - 1$

(*h*) $3\text{: } \{0,\mathtt{i}\}\mathfrak{e}\{\mathtt{A.length}\}? \sqcup \{0,\mathtt{i-1}\}\mathfrak{d}\{1,\mathtt{i}\}\mathfrak{e}\{\mathtt{A.length}\}?\ \text{-}$ join

$\{0,\mathtt{i}\}\mathfrak{n}\{\mathtt{A.length}\}? \sqcup \{0,\mathtt{i-1}\}\mathfrak{c}\{1,\mathtt{i}\}\mathfrak{n}\{\mathtt{A.length}\}?$

$= \{0\}\mathfrak{e}\{\mathtt{i}\}?\mathfrak{e}\{\mathtt{A.length}\}? \sqcup \{0\}\mathfrak{d}\{\mathtt{i}\}\mathfrak{e}\{\mathtt{A.length}\}?\ \text{-}$ segment unification

$\{0\}\bot\{\mathtt{i}\}?\mathfrak{n}\{\mathtt{A.length}\}? \sqcup \{0\}\mathfrak{c}\{\mathtt{i}\}\mathfrak{n}\{\mathtt{A.length}\}?$

$= \{0\}\mathfrak{d}\{\mathtt{i}\}?\mathfrak{e}\{\mathtt{A.length}\}?\ \text{-}\ \{0\}\mathfrak{c}\{\mathtt{i}\}?\mathfrak{n}\{\mathtt{A.length}\}?$

segmentwise join $\mathfrak{e} \sqcup \mathfrak{d} = \mathfrak{d}$, $\mathfrak{e} \sqcup \mathfrak{e} = \mathfrak{e}$, $\bot \sqcup \mathfrak{c} = \mathfrak{c}$, $\mathfrak{n} \sqcup \mathfrak{n} = \mathfrak{n}$

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

(*j*) $5\text{: } \{0\}\mathfrak{d}\{\mathtt{i}\}?\mathfrak{e}\{\mathtt{A.length}\}\ \text{-}\ \{0\}\mathfrak{c}\{\mathtt{i}\}?\mathfrak{c}\{\mathtt{i+1}\}\mathfrak{n}\{\mathtt{A.length}\}?$

`A[i]` checked while unmodified

(*k*) $6\text{: } \{0\}\mathfrak{d}\{\mathtt{i}\}?\mathfrak{d}\{\mathtt{i+1}\}\mathfrak{e}\{\mathtt{A.length}\}?\ \text{-}\ \{0\}\mathfrak{c}\{\mathtt{i}\}?\mathfrak{c}\{\mathtt{i+1}\}\mathfrak{n}\{\mathtt{A.length}\}?$

`A[i]` potentially modified

(*l*) $7\text{: } \{0\}\mathfrak{d}\{\mathtt{i-1}\}?\mathfrak{d}\{\mathtt{i}\}\mathfrak{e}\{\mathtt{A.length}\}?\ \text{-}\ \{0\}\mathfrak{c}\{\mathtt{i-1}\}?\mathfrak{c}\{\mathtt{i}\}\mathfrak{n}\{\mathtt{A.length}\}?$

invertible assignment $\mathtt{i}_{\text{old}} = \mathtt{i}_{\text{new}} - 1$

(*m*) $3\text{: } \{0\}\mathfrak{d}\{\mathtt{i}\}?\mathfrak{e}\{\mathtt{A.length}\}? \sqcup \{0\}\mathfrak{d}\{\mathtt{i-1}\}\mathfrak{d}\{\mathtt{i}\}\mathfrak{e}\{\mathtt{A.length}\}?\ \text{-}$ join

$\{0\}\mathfrak{c}\{\mathtt{i}\}?\mathfrak{n}\{\mathtt{A.length}\}? \sqcup \{0\}\mathfrak{c}\{\mathtt{i-1}\}\mathfrak{c}\{\mathtt{i}\}\mathfrak{n}\{\mathtt{A.length}\}?$

$= \{0\}\mathfrak{d}\{\mathtt{i}\}?\mathfrak{e}\{\mathtt{A.length}\}? \sqcup \{0\}\mathfrak{d}\{\mathtt{i}\}?\mathfrak{e}\{\mathtt{A.length}\}?$ -segment unification

$\{0\}\mathfrak{c}\{\mathtt{i}\}?\mathfrak{n}\{\mathtt{A.length}\}? \sqcup \{0\}\mathfrak{c}\{\mathtt{i}\}?\mathfrak{n}\{\mathtt{A.length}\}?$

$= \{0\}\mathfrak{d}\{\mathtt{i}\}?\mathfrak{e}\{\mathtt{A.length}\}?\ \text{-}\ \{0\}\mathfrak{c}\{\mathtt{i}\}?\mathfrak{n}\{\mathtt{A.length}\}?$

segmentwise join, convergence

(*n*) $8\text{: } \{0\}\mathfrak{d}\{\mathtt{i},\mathtt{A.length}\}?\ \text{-}\ \{0\}\mathfrak{c}\{\mathtt{i},\mathtt{A.length}\}?$

$i \leqslant \mathtt{A.length}$ in segmentation and $\geqslant$ in test negation so $\mathtt{i} = \mathtt{A.length}$.

Just to show that the analysis is very fast!

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
- Currently being implemented

THE END,
THANK YOU