Precondition Inference from Intermittent Assertions and Application to Contracts on Collections

Patrick Cousot [2,3], Radhia Cousot [1,3,4], and Francesco Logozzo [4]

[1] Centre National de la Recherche Scientifique, Paris
[2] Courant Institute of Mathematical Sciences, New York University
[3] École Normale Supérieure, Paris
[4] MSR, Redmond

Abstract. Programmers often insert assertions in their code to be optionally checked at runtime, at least during the debugging phase. In the context of design by contracts, these assertions would better be given as a precondition of the method/procedure which can detect that a caller has violated the procedure's contract in a way which definitely leads to an assertion violation (*e.g.*, for separate static analysis). We define precisely and formally the contract inference problem from intermittent assertions inserted in the code by the programmer. Our definition excludes no good run even when a non-deterministic choice (e.g., an interactive input) could lead to a bad one (so this is not the weakest precondition, nor its strengthening by abduction, since a terminating successful execution is not guaranteed). We then introduce new abstract interpretation-based methods to automatically infer both the static contract precondition of a method/procedure and the code to check it at runtime on scalar and collection variables.

1 Introduction

In the context of compositional/structural static program analysis for design by contract [23,24], it is quite frequent that preconditions for the code (*i.e.* a program/module/method/procedure/function/assembly/etc) have been only partially specified by the programmer (or even not at all for legacy code) and need to be automatically strengthened or inferred by taking into account the implicit *language assertions* (*e.g.*, runtime errors) and the explicit *programmer assertions* (*e.g.*, assertions and contracts of called methods/procedures). Besides the methodological advantage of anticipating future inevitable requirements when running a code, precise contracts are necessary in the context of a separate program analysis as *e.g.*, in `Clousot`, an abstract interpretation-based static contract checker for `.NET` [18]. We work in the context of contracts embedded in the program code [4] so that specification conditions are expressed in the programming language itself (and extracted by the compiler for use in contract related tools). The precondition inference problem for a code is twofold [4]:

R. Jhala and D. Schmidt (Eds.): VMCAI 2011, LNCS 6538, pp. 150–168, 2011.

– *Static analysis problem:* infer the entry semantic precondition from control flow dependent language and programmer assertions embedded in the code to guard, whenever possible, against inevitable errors;
– *Code synthesis problem:* generate visible side-effect free code checking for that precondition. This checking code must be separable from the checked code and should only involve elements visible to all callers of the checked code.

Example 1 The problem is illustrated by the following `AllNotNull` procedure where the precondition that the array `A` and all array elements should not be null $\texttt{A} \neq \texttt{null} \wedge \forall i \in [0, \texttt{A.length}) : \texttt{A}[i] \neq \texttt{null}$ is checked by the implicit language assertions while iterating over the array.

```
         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 language assertion `A[i] != null` for a *given* value of `i` is *intermittent* at program point `4:` but not *invariant* since the array content is modified at program point `5:`.

□

On one hand, a solution to the contract inference problem could be to infer the precondition as a set of states, logical formula, or abstract property ensuring proper termination without any language or programmer assertion failure (as proposed *e.g.*, in [10, Sect. 10-4.6]) or [9, Sect. 3.4.5]). But this does not guarantee the precondition to be easily understandable and that efficient code can be generated to check it. Moreover this is stronger than strictly required (*e.g.*, the code `x = random(); assert(x ==0)` is not guaranteed to terminate properly, but has at least one execution without failure, so should not be rejected). On the other hand, the precondition checking code could be a copy of the method body where all code with random or visible side effect (including input) as well as all further dependent code is removed.

Example 2 Continuing Ex. 1, we get the straw man

```
bool CheckAllNotNull(Ptr[] A) {
   int i = 0;
   while (if (A == null) { return false }; i < A.length) {
      if ((A == null) || (0 > i) || (i >= A.length) || (A[i] == null))
         { return false };
      i++ }
   return true }
```

Modifications of `i` have no visible side effects while those of elements of `A` do have, so the assignment `A[i].f` is dropped. There is no code that depends on this value, so no other code needs to be removed. □

However, this simple solution may not provide a simple precondition both easily understandable by the programmer, easily reusable for separate modular static analysis, and efficiently checkable at runtime, if necessary.

Example 3 Continuing Ex. 1 and 2, we would like to automatically infer the precondition `ForAll(0,A.length,i => A[i] != null)` using `ForAll` quantifiers [4] over integer ranges and collections. Iterative checking code is then easy to generate. □

The semantics of code is formalized in Sect. 2 and that of specifications by runtime assertions in Sect. 3. The contract precondition inference problem is defined in Sect. 4 and compared with weakest preconditions computation. Elements of abstract interpretation are recalled in Sect. 5 and used in Sect. 6 to provide a fixpoint solution to the contract precondition inference problem. Several effective contract precondition inference are then proposed, by data flow analysis in Sect. 7, for scalar variables both by forward symbolic analysis in Sect. 8 and by backward symbolic analysis in Sect. 9, for collections by forward analysis in Sect. 10. Sect. 11 has a comparison with related work, suggestions for future work, and conclusions.

2 Program semantics

Small-step operational semantics. Following [9], the small-step operational semantics of code is assumed to be given by a *transition system* $\langle \Sigma, \tau, \mathfrak{I} \rangle$ where Σ is a set of *states*, $\tau \in \wp(\Sigma \times \Sigma)$ is a non-deterministic *transition relation* between a state and its possible successors, and $\mathfrak{I} \in \wp(\Sigma)$ is the set of *initial states* (on code entry, assuming the precondition, if any, to be true). We write $\tau(s, s')$ for $\langle s,\ s' \rangle \in \tau$. The *final or blocking states* without any possible successor (on code exit or violation of a language assertion with unpredictable consequences) are $\mathfrak{B} \triangleq \{s \in \Sigma \mid \forall s' : \neg\tau(s, s')\}$. If the code must satisfy a *global invariant* $\mathfrak{G} \in \wp(\Sigma)$ (*e.g.*, class invariant for a method), we assume this to be included in the definition of the transition relation τ (*e.g.*, $\tau \subseteq \mathfrak{G} \times \mathfrak{G}$). We use a map $\boldsymbol{\pi} \in \Sigma \to \Gamma$ of states of Σ into *control points* in Γ which is assumed to be of finite cardinality. The program has *scalar variables* $\mathtt{x} \in \mathbb{x}$, *collection variables* $\mathtt{X} \in \mathbb{X}$ and visible side effect free expressions $\mathtt{e} \in \mathbb{E}$, including *Boolean expressions* $\mathtt{b} \in \mathbb{B} \subseteq \mathbb{E}$. Collection variables `X` have elements `X[i]` ranging from 0 to `X.count` -1 (`A.length` -1 for arrays `A`). The value of $\mathtt{e} \in \mathbb{E}$ in state $s \in \Sigma$ is $[\![\mathtt{e}]\!]s \in \mathcal{V}$. The *values* $\mathcal{V}$ include the Booleans $\mathcal{B} \triangleq \{true, false\}$ where the complete Boolean algebra $\langle \mathcal{B}, \Rightarrow \rangle$ is ordered by $false \Rightarrow true$. The value $[\![\mathtt{X}]\!]s$ of a collection `X` in a state $s \in \Sigma$ is a pair $[\![\mathtt{X}]\!]s = \langle n,\ X \rangle$ where $n = [\![\mathtt{X.count}]\!]s \geqslant 0$ is a non-negative integer and $X \in [0, n) \to \mathcal{V}$ denotes the value $X(i)$ of i-th element, $i \in [0, n)$, in the collection. When $i \in [0, n)$, we define $[\![\mathtt{X}]\!]s[i] \triangleq X(i)$ $(= [\![\mathtt{X[e]}]\!]s$ where $[\![\mathtt{e}]\!]s = i)$ to denote the i-th element in the collection.

Traces. We let traces be sequences of states in Σ. $\vec{\Sigma}^n$ is the set of non-empty *finite traces* $\vec{s} = \vec{s}_0 \ldots \vec{s}_{n-1}$ of *length* $|\vec{s}| \triangleq n \geqslant 0$ including the *empty trace* $\vec{\epsilon}$ of length $|\vec{\epsilon}| \triangleq 0$. $\vec{\Sigma}^+ \triangleq \bigcup_{n \geqslant 1} \vec{\Sigma}^n$ is the set of *non-empty finite traces* and $\vec{\Sigma}^* \triangleq \vec{\Sigma}^+ \cup \{\vec{\epsilon}\}$. As usual, *concatenation* is denoted by juxtaposition and extended to sets of traces. Moreover, the *sequential composition* of traces is $\vec{s}s \circ s\vec{s}' \triangleq \vec{s}s\vec{s}'$ when $\vec{s}, \vec{s}' \in \vec{\Sigma}^*$ and $s \in \Sigma$, and is otherwise undefined. $\vec{S} \circ \vec{S}' \triangleq \{\vec{s}s\vec{s}' \mid \vec{s}s \in$

$\vec{S} \cap \vec{\Sigma}^+ \wedge s\vec{s}' \in \vec{S}'\}$. The *partial execution traces* or *runs* of $\langle \Sigma,\ \tau,\ \mathfrak{I} \rangle$ are prefix traces generated by transitions, as follows

$$\begin{array}{lcll}
\vec{\tau}^n & \triangleq & \{\vec{s} \in \vec{\Sigma}^n \mid \forall i \in [0, n-1) : \tau(\vec{s}_i, \vec{s}_{i+1})\} & \textit{partial runs of length } n \geqslant 0 \\
\vec{\tau}^+ & \triangleq & \bigcup_{n \geqslant 1} \vec{\tau}^n & \textit{non-empty finite partial runs} \\
\vec{\tau}^n & \triangleq & \{\vec{s} \in \vec{\tau}^n \mid \vec{s}_{n-1} \in \mathfrak{B}\} & \textit{complete runs of length } n \geqslant 0 \\
\vec{\tau}^+ & \triangleq & \bigcup_{n \geqslant 1} \vec{\tau}^n & \textit{non-empty finite complete runs.}
\end{array}$$

The partial (resp. complete/maximal) runs starting from an initial state are $\vec{\tau}^+_{\mathfrak{I}} \triangleq \{\vec{s} \in \vec{\tau}^+ \mid \vec{s}_0 \in \mathfrak{I}\}$ (resp. $\vec{\tau}^+_{\mathfrak{I}} \triangleq \{\vec{s} \in \vec{\tau}^+ \mid \vec{s}_0 \in \mathfrak{I}\}$). Given $\mathfrak{S} \subseteq \Sigma$, we let $\vec{\mathfrak{S}}^n \triangleq \{\vec{s} \in \vec{\Sigma}^n \mid \vec{s}_0 \in \mathfrak{S}\}$, $n \geqslant 1$. Partial and maximal finite runs have the following fixpoint characterization [11]

$$\begin{array}{lcl}
\vec{\tau}^+_{\mathfrak{I}} & = & \mathsf{lfp}^{\subseteq}_{\emptyset}\ \boldsymbol{\lambda}\vec{T} \bullet \vec{\mathfrak{I}}^1 \cup \vec{T} \circ \vec{\tau}^2 \\
\vec{\tau}^+ & = & \mathsf{lfp}^{\subseteq}_{\emptyset}\ \boldsymbol{\lambda}\vec{T} \bullet \vec{\mathfrak{B}}^1 \cup \vec{\tau}^2 \circ \vec{T} \ = \ \mathsf{gfp}^{\subseteq}_{\vec{\Sigma}^+}\ \boldsymbol{\lambda}\vec{T} \bullet \vec{\mathfrak{B}}^1 \cup \vec{\tau}^2 \circ \vec{T}.
\end{array} \qquad \text{(1-a,1-b)}$$

3 Specification semantics

The *specification* includes the existing precondition and postcondition, if any, the language and programmer assertions, made explicit in the form

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

whenever a runtime check `assert(`$\mathtt{b}_j$`)` is attached to a control point $\mathtt{c}_j \in \Gamma$, $j \in \Delta$. $\mathbb{A}$ is computed by a syntactic pre-analysis of the code. The Boolean expressions $\mathtt{b}_j$ are assumed to be both visible side effect free and always well-defined when evaluated in a shortcut manner, which may have to be checked by a prior `assert` (*e.g.*, `assert((A!= null) && (A[i] == 0))`). For simplicity, we assume that $\mathtt{b}_j$ either refers to a scalar variable (written $\mathtt{b}_j$`(x)`) or to an element of a collection (written $\mathtt{b}_j$`(X, i)`). This defines

$$\begin{array}{lcll}
\mathfrak{E}_{\mathbb{A}} & \triangleq & \{s \in \Sigma \mid \exists \langle c,\ \mathtt{b} \rangle \in \mathbb{A} : \boldsymbol{\pi} s = c \wedge \neg [\![\mathtt{b}]\!] s\} & \textit{erroneous or bad states} \\
\vec{\mathfrak{E}}_{\mathbb{A}} & \triangleq & \{\vec{s} \in \vec{\Sigma}^+ \mid \exists i < |\vec{s}| : \vec{s}_i \in \mathfrak{E}_{\mathbb{A}}\} & \textit{erroneous or bad runs.}
\end{array}$$

As part of the implicit specification, and for the sake of brevity, we consider that program executions should terminate. Otherwise the results are similar after revisiting (1-a,1-b) for infinite runs as considered in [11].

4 The 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*

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

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

The following lemma shows that, according to Def. 4, 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, so we would like $P_{\mathbb{A}}$ to be minimal, whenever possible (so that $\vec{\tau}^+_{\mathfrak{I}\setminus P_{\mathbb{A}}}$ is maximal). Please note that this is <u>not</u> the weakest (liberal) precondition [17], which yields the weakest condition under which the code (either does not terminate or) terminates without assertion failure, whichever non-deterministic choice is chosen. So this is not either the problem of strengthening a precondition to a weaker one by abduction for specification synthesis [7].

Theorem 6 *The strongest*[5] *solution to the precondition inference problem in Def. 4 is*

$$\mathfrak{P}_{\mathbb{A}} \triangleq \{s \mid \exists s\vec{s} \in \vec{\tau}^+ \cap \neg\vec{\mathfrak{E}}_{\mathbb{A}}\}. \quad (4) \quad \square$$

Instead of reasoning on the set $\mathfrak{P}_{\mathbb{A}}$ of states from which there exists a good run without any error, we can reason on the complement $\overline{\mathfrak{P}}_A$ that is the set of states from which all runs are bad in that they always lead to an error. Define $\overline{\mathfrak{P}}_{\mathbb{A}}$ to be the set of states from which any complete run in $\vec{\tau}^+$ does fail.

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

5 Basic elements of abstract interpretation

Galois connections. A *Galois connection* $\langle L, \leqslant\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \overline{L}, \sqsubseteq\rangle$ consists of posets $\langle L, \leqslant\rangle$, $\langle \overline{L}, \sqsubseteq\rangle$ and maps $\alpha \in L \to \overline{L}$, $\gamma \in \overline{L} \to L$ such that $\forall x \in L, y \in \overline{L} : \alpha(x) \sqsubseteq y \Leftrightarrow x \leqslant \gamma(y)$. The dual is $\langle \overline{L}, \sqsupseteq\rangle \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \langle L, \geqslant\rangle$. In a Galois connection, the *abstraction* α preserves existing least upper bounds (lubs) hence is monotonically increasing so, by duality, the *concretization* γ preserves existing greatest lower bounds (glbs) and is monotonically increasing. If $\langle L, \leqslant\rangle$ is a complete Boolean lattice with unique complement $\neg$ then the self-dual *complement isomorphism* is $\langle L, \leqslant\rangle \underset{\neg}{\overset{\neg}{\leftrightarrows}} \langle L, \geqslant\rangle$ (since $\neg x \leqslant y \Leftrightarrow x \geqslant \neg y$).

Fixpoint abstraction. Recall from [13, 7.1.0.4] that

Lemma 7 *If* $\langle L, \leqslant, \bot\rangle$ *is a complete lattice or a cpo,* $F \in L \to L$ *is monotonically 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.* $\alpha(\mathsf{lfp}^{\leqslant}_{\bot} F) \sqsubseteq \mathsf{lfp}^{\sqsubseteq}_{\alpha(\bot)} \overline{F}$*).*

(5) Following [17], P is said to be *stronger* than Q and Q *weaker* than P if and only if $P \subseteq Q$.

(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 $F^0 \triangleq \bot$, $F^{n+1} \triangleq F(F^n)$, $F^\omega \triangleq \bigsqcup_{n\geqslant} F^n$ of the least fixpoint of F.

Applying Lem. 7 to $\langle L, \leqslant\rangle \mathrel{\substack{\xleftarrow{\neg}\\ \xrightarrow[\neg]{}}} \langle L, \geqslant\rangle$, we get Cor. 8 and by duality Cor. 9 below.

Corollary 8 (David Park [26, Sect. 2.3]) *If $F \in L \to L$ is monotonically increasing on a complete Boolean lattice $\langle L, \leqslant, \bot, \neg\rangle$ then $\neg\, \mathsf{lfp}^{\leqslant}_{\bot} F = \mathsf{gfp}^{\leqslant}_{\neg\bot}\, \neg \circ F \circ \neg$.*

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 monotonically 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 Fixpoint strongest contract precondition

Following [11], let us define the abstraction generalizing [17] to traces

$$\begin{aligned} \mathsf{wlp}[\vec{T}] &\triangleq \boldsymbol{\lambda}\, \vec{Q} \bullet \left\{ s \;\middle|\; \forall s\vec{s} \in \vec{T} : s\vec{s} \in \vec{Q} \right\} \\ \mathsf{wlp}^{-1}[\vec{Q}] &\triangleq \boldsymbol{\lambda}\, P \bullet \left\{ s\vec{s} \in \vec{\Sigma}^+ \;\middle|\; (s \in P) \Rightarrow (s\vec{s} \in \vec{Q}) \right\} \end{aligned}$$

such that $\langle \wp(\vec{\Sigma}^+), \subseteq\rangle \mathrel{\substack{\xleftarrow{\mathsf{wlp}^{-1}[\vec{Q}]}\\ \xrightarrow[\boldsymbol{\lambda}\, \vec{T} \bullet \mathsf{wlp}[\vec{T}]\vec{Q}]{}}} \langle \wp(\Sigma), \supseteq\rangle$ and $\overline{\mathfrak{P}}_{\mathbb{A}} = \mathsf{wlp}[\vec{\tau}^+](\vec{\mathfrak{E}}_{\mathbb{A}})$. By fixpoint abstraction, it follows from (1-a) and Cor. 8 that

Theorem 10 $\overline{\mathfrak{P}}_{\mathbb{A}} = \mathsf{gfp}^{\subseteq}_{\Sigma}\, \boldsymbol{\lambda}\, P \bullet \mathfrak{E}_{\mathbb{A}} \cup (\neg\mathfrak{B} \cap \widetilde{\mathsf{pre}}[\tau]P)$ *and* $\mathfrak{P}_{\mathbb{A}} = \mathsf{lfp}^{\subseteq}_{\emptyset}\, \boldsymbol{\lambda}\, P \bullet \neg\mathfrak{E}_{\mathbb{A}} \cap (\mathfrak{B} \cup \mathsf{pre}[\tau]P)$ *where* $\mathsf{pre}[\tau]Q \triangleq \{s \mid \exists s' \in Q : \langle s, s'\rangle \in \tau\}$ *and* $\widetilde{\mathsf{pre}}[\tau]Q \triangleq \neg\mathsf{pre}[\tau](\neg Q) = \{s \mid \forall s' : \langle s, s'\rangle \in \tau \Rightarrow s' \in Q\}$. □

If the set Σ of states is finite, as assumed in model-checking [2], the fixpoint definition of $\mathfrak{P}_{\mathbb{A}}$ in Th. 10 is computable iteratively, up to combinatorial explosion. The code to check the precondition $s \in \mathfrak{P}_{\mathbb{A}}$ can proceed by exhaustive enumeration. In case this does not scale up or for infinite state systems, bounded model-checking [5] is an alternative using $\bigcup_{i=0}^{k} \vec{\tau}^i$ instead of $\vec{\tau}^+$ but, by Th. 6, the bounded prefix abstraction $\alpha_k(\vec{T}) \triangleq \left\{ \vec{s}_0 \ldots \vec{s}_{\min(k, |\vec{s}|)-1} \;\middle|\; \vec{s} \in \vec{T} \right\}$ is unsound for approximating both $\mathfrak{P}_{\mathbb{A}}$ and $\overline{\mathfrak{P}}_{\mathbb{A}}$.

7 Contract precondition inference by symbolic flow analysis

Instead of state-based reasonings, as in Sect. 4 and 6, we can consider symbolic (or even syntactic) reasonings moving the code assertions to the code entry, when the effect is the same. This can be done by a sound data flow analysis [21] when

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.

We propose a backward data flow analysis to check for both sufficient conditions 1 and 2.

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

Backward expression propagation. Let $c \in \Gamma$ be a control point and b be a Boolean expression. For example b can contain `ForAll` or `Exists` assertions on unmodified collections without free scalar variables and no visible side effect (see Sect. 10 otherwise). $P(c, b)$ 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. $P = \mathsf{gfp}^{\dot{\Rightarrow}} B[\![\tau]\!]$ is the $\dot{\Rightarrow}$-greatest solution of the backward system of equations [9],[10]

$$\begin{cases} P(c,b) \;=\; B[\![\tau]\!](P)(c,b) \\ c \in \Gamma, \quad b \in \mathbb{A}_{\mathsf{b}} \end{cases}$$

where the expressions of `asserts` are $\mathbb{A}_{\mathsf{b}} \triangleq \{b \mid \exists c : \langle c,\, b\rangle \in \mathbb{A}\}$ and the transformer $B \in (\Gamma \times \mathbb{A}_{\mathsf{b}} \to \mathcal{B}) \to (\Gamma \times \mathbb{A}_{\mathsf{b}} \to \mathcal{B})$ is

$$\begin{array}{lcll} B[\![\tau]\!](P)(c,b) & = & \mathit{true} \quad \text{when} \quad \langle c,\, b\rangle \in \mathbb{A} & (\texttt{assert(}b\texttt{)} \text{ at } c) \\ B[\![\tau]\!](P)(c,b) & = & \mathit{false} \quad \text{when} \quad \exists s \in \mathfrak{B} : \boldsymbol{\pi} s = c \wedge \langle c,\, b\rangle \notin \mathbb{A} & (\text{exit at } c) \\ B[\![\tau]\!](P)(c,b) & = & \displaystyle\bigwedge_{c' \,\in\, \mathsf{succ}[\![\tau]\!](c)} \mathsf{unchanged}[\![\tau]\!](c,c',b) \wedge P(c',b) & (\text{otherwise}) \end{array}$$

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

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

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

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

$\mathsf{unchanged}[\![\tau]\!](c, c', b)$ can be a syntactic underapproximation of its semantic definition [3]. Define

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

and the abstraction

$$\begin{array}{rcl} \vec{\alpha}_D(\vec{T})(c,b) & \triangleq & \forall \vec{s} \in \vec{T} : \boldsymbol{\pi}\vec{s}_0 = c \Rightarrow \vec{s} \in \vec{\mathfrak{R}}_{\mathbb{A}}(b) \\ \vec{\gamma}_D(P) & \triangleq & \{\vec{s} \mid \forall b \in \mathbb{A}_{\mathsf{b}} : P(\boldsymbol{\pi}\vec{s}_0, b) \Rightarrow \vec{s} \in \vec{\mathfrak{R}}_{\mathbb{A}}(b)\} \end{array}$$

such that $\langle \vec{\Sigma}^+,\, \subseteq\rangle \underset{\vec{\alpha}_D}{\overset{\vec{\gamma}_D}{\leftrightarrows}} \langle \Gamma \times \mathbb{A}_{\mathsf{b}} \to \mathcal{B},\, \dot{\Leftarrow}\rangle$. By (1-a) and Lem. 7, we have

Theorem 11 $\vec{\alpha}_D(\vec{\tau}^+) \;\dot{\Leftarrow}\; \mathsf{lfp}^{\dot{\Leftarrow}} B[\![\tau]\!] \;=\; \mathsf{gfp}^{\dot{\Rightarrow}} B[\![\tau]\!] \;\triangleq\; P.$ □

---

[9] $\dot{\Rightarrow}$ is the pointwise extension of logical implication $\Rightarrow$

[10] The system of equations $\vec{X} = \vec{F}(\vec{X})$ where $\vec{X} = X_1, \ldots, X_n$ is written $\begin{cases} X_i \;=\; F_i(X_1, \ldots, X_n)\ . \\ i = 1, \ldots, n \end{cases}$

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 $\&\&\,\emptyset \triangleq \texttt{true}$)

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

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) $P_{\mathfrak{I}} \triangleq \{s \in \Sigma \mid \exists i \in \mathfrak{I}_{\boldsymbol{\pi}} : s \in P_i\}$. We have

Theorem 12 $\mathfrak{P}_{\mathbb{A}} \cap \mathfrak{I} \subseteq P_{\mathfrak{I}}$. □

By Th. 6 and 12, the precondition generation is sound: a rejected initial state would inevitably have lead to an assertion failure.

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. □

However, the data flow abstraction is rather imprecise because a precondition is checked on code entry only if

1. the exact same precondition is checked in an `assert` (since scalar and collection variable modifications are not taken into account, other than annihilating the backward propagation);
2. and this, whichever execution path is taken (conditions are not taken into account).

We propose remedies to 1 and 2 in the following Sect. 8 and 9.

8 Contract precondition inference for scalar variables by forward symbolic analysis

Let us define the cmd, succ and pred functions mapping control points to their command, successors and predecessors ($\forall \mathsf{c}, \mathsf{c}' \in \Gamma : \mathsf{c}' \in \mathsf{pred}(\mathsf{c}) \Leftrightarrow \mathsf{c} \in \mathsf{succ}(\mathsf{c}')$).

— c: x:=e; c′:...	$\mathsf{cmd}(\mathsf{c},\mathsf{c}') \triangleq$ x:=e	$\mathsf{succ}(\mathsf{c}) \triangleq \{\mathsf{c}'\}$	$\mathsf{pred}(\mathsf{c}') \triangleq \{\mathsf{c}\}$
— c: assert(b); c′:...	$\mathsf{cmd}(\mathsf{c},\mathsf{c}') \triangleq$ b	$\mathsf{succ}(\mathsf{c}) \triangleq \{\mathsf{c}'\}$	$\mathsf{pred}(\mathsf{c}') \triangleq \{\mathsf{c}\}$
— c: if b then	$\mathsf{cmd}(\mathsf{c},\mathsf{c}'_t) \triangleq$ b	$\mathsf{succ}(\mathsf{c}) \triangleq \{\mathsf{c}'_t,\mathsf{c}'_f\}$	
c′t:...c″t:	$\mathsf{cmd}(\mathsf{c},\mathsf{c}'_f) \triangleq \neg$b		$\mathsf{pred}(\mathsf{c}'_t) \triangleq \{\mathsf{c}\}$
else	$\mathsf{cmd}(\mathsf{c}''_t,c') \triangleq$ skip	$\mathsf{succ}(\mathsf{c}''_t) \triangleq \{\mathsf{c}'\}$	
c′f:...c″f:	$\mathsf{cmd}(\mathsf{c}''_f,c') \triangleq$ skip	$\mathsf{succ}(\mathsf{c}''_f) \triangleq \{\mathsf{c}'\}$	$\mathsf{pred}(\mathsf{c}'_f) \triangleq \{\mathsf{c}\}$
fi; c′...			$\mathsf{pred}(\mathsf{c}') \triangleq \{\mathsf{c}''_t,\mathsf{c}''_f\}$
— c :while c′: b do	$\mathsf{cmd}(\mathsf{c},\mathsf{c}') \triangleq$ skip	$\mathsf{succ}(\mathsf{c}) \triangleq \{\mathsf{c}'\}$	$\mathsf{pred}(\mathsf{c}') \triangleq \{\mathsf{c},\mathsf{c}''_b\}$
c′b:...c″b:	$\mathsf{cmd}(\mathsf{c}',\mathsf{c}'_b) \triangleq$ b	$\mathsf{succ}(\mathsf{c}') \triangleq \{\mathsf{c}'_b,\mathsf{c}''\}$	$\mathsf{pred}(\mathsf{c}'_b) \triangleq \{\mathsf{c}'\}$
od; c″...	$\mathsf{cmd}(\mathsf{c}',\mathsf{c}'') \triangleq \neg$b	$\mathsf{succ}(\mathsf{c}''_b) \triangleq \{\mathsf{c}'\}$	$\mathsf{pred}(\mathsf{c}'') \triangleq \{\mathsf{c}'\}$
	$\mathsf{cmd}(\mathsf{c}''_b,\mathsf{c}) \triangleq$ skip		

For programs with scalar variables $\vec{\mathtt{x}}$, we denote by $\underline{\vec{x}}$ (or $\overline{\mathtt{x0}}$) their initial values and by $\vec{x}$ their current values. Following [9, Sect. 3.4.5], the symbolic execution

[22] attaches invariants $\Phi(c)$ to program points $\mathsf{c} \in \Gamma$ defined as the pointwise $\dot{\Rightarrow}$-least fixpoint of the system of equations $\Phi = F(\Phi)$ with

$$\begin{cases} F(\Phi)\mathsf{c} \;=\; \bigvee\limits_{\mathsf{c}' \in \mathsf{pred}(\mathsf{c})} \mathcal{F}(\mathsf{cmd}(\mathsf{c}', \mathsf{c}), \Phi(c')) \vee \bigvee\limits_{\mathsf{c} \in \mathfrak{I}_{\boldsymbol{\pi}}} (\vec{x} = \underline{\vec{x}}) \\ \mathsf{c} \in \Gamma \end{cases}$$

where $\mathsf{pred}(\mathsf{c}) = \emptyset$ for program entry points $\mathsf{c} \in \mathfrak{I}_{\boldsymbol{\pi}}$ and the forward transformers are in Floyd's style (the predicates ϕ depends only on the symbolic initial $\underline{\vec{x}}$ and current $\vec{x}$ values of the program variables $\vec{\mathtt{x}}$)

$$\begin{array}{rcl} \mathcal{F}(\mathtt{skip}, \phi) & \triangleq & \phi \\ \mathcal{F}(\mathtt{x:=e}, \phi) & \triangleq & \exists \vec{x}\,' : \phi[\vec{x} := \vec{x}\,'] \wedge \mathsf{dom}(\mathsf{e}, \vec{x}\,') \wedge \vec{x} = \vec{x}\,'[x := \mathsf{e}[\vec{\mathtt{x}} := \vec{x}\,']] \\ \mathcal{F}(\mathtt{b}, \phi) & \triangleq & \phi \wedge \mathsf{dom}(\mathtt{b}, \vec{x}) \wedge \mathtt{b}[\vec{\mathtt{x}} := \vec{x}] \end{array}$$

where $\mathsf{dom}(\mathsf{e}, \vec{x})$ is the condition on $\vec{x}$ for evaluating e as a function of $\vec{x}$ without runtime error. By allowing infinitary disjunctions, we have [9, Sect. 3.4.5]

Theorem 14 *$\Phi = \mathsf{lfp}^{\dot{\Rightarrow}} F$ has the form $\Phi(c) = \bigvee_{i \in \Delta_c} p_{c,i}(\underline{\vec{x}}) \wedge \vec{x} = \vec{e}_{c,i}(\underline{\vec{x}})$ where $p_{c,i}(\underline{\vec{x}})$ is a Boolean expression defining the condition for control to reach the current program point c as a function of the initial values $\underline{\vec{x}}$ of the scalar variables $\vec{\mathtt{x}}$ and $\vec{e}_i(\underline{\vec{x}})$ defines the current values $\vec{x}$ of the scalar variables $\vec{\mathtt{x}}$ as a function of their initial values $\underline{\vec{x}}$ when reaching program point c with path condition $p_{c,i}(\underline{\vec{x}})$ true.* □

The soundness follows from $\forall \vec{s} \in \vec{\tau}^+ : \forall j < |\vec{s}| : \phi(c)[\underline{\vec{x}} := [\![\vec{\mathtt{x}}]\!]\vec{s}_0][\vec{x} := [\![\vec{\mathtt{x}}]\!]\vec{s}_j] = \forall \vec{s} \in \vec{\tau}^+ : \forall j < |\vec{s}| : \forall i \in \Delta_{\boldsymbol{\pi}\vec{s}_j} : p_{\boldsymbol{\pi}\vec{s}_j,i}[\underline{\vec{x}} := [\![\vec{\mathtt{x}}]\!]\vec{s}_0] \Rightarrow [\![\vec{\mathtt{x}}]\!]\vec{s}_j = \vec{e}_{\boldsymbol{\pi}\vec{s}_j,i}[\underline{\vec{x}} := [\![\vec{\mathtt{x}}]\!]\vec{s}_0]$ where $[\![\vec{\mathtt{x}}]\!]s$ is the value of the vector $\vec{\mathtt{x}}$ of scalar variables in state s.

This suggests a method for calculating the precondition by adding for each assertion `c:assert(b)` the condition $\bigwedge_{i \in \Delta_c} p_{c,i}[\underline{\vec{x}} := \vec{\mathtt{x}}] \Rightarrow \mathtt{b}[\vec{\mathtt{x}} := \vec{e}_{c,i}[\underline{\vec{x}} := \vec{\mathtt{x}}]]$ which is checked on the initial values of variables.

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))`. □

Of course the iterative computation of $\mathsf{lfp}^{\dot{\Rightarrow}} F$ will in general not terminate so that a widening [12] is needed. A simple one would bound the number of iterations and widen $\bigvee_{i \in \Delta_c} p_{c,i}(\underline{\vec{x}}) \wedge \vec{x} = \vec{e}_{c,i}(\underline{\vec{x}})$ to $\bigwedge_{i \in \Delta_c} p_{c,i}(\underline{\vec{x}}) \Rightarrow \vec{x} = \vec{e}_{c,i}(\underline{\vec{x}})$.

9 Contract precondition inference by backward symbolic analysis

Backward symbolic precondition analysis of simple assertions. The symbolic relation between entry and `assert` conditions can be also established

backwards, starting from the `assert` conditions and propagating towards the entry points taking assignments and tests into account with widening around unbounded loops. We first consider simple assertions involving only scalar variables (including *e.g.*, the size of collections as needed in Sect. 10).

Abstract domain. Given the set $\mathbb{B}$ of visible side effect free Boolean expressions on scalar variables, we consider the abstract domain $\mathbb{B}/_{\equiv}$ containing the infimum `false` (unreachable), the supremum `true` (unknown) and equivalence classes of expressions $[\mathtt{b}]/_{\equiv}$ for the abstract equivalence of expressions $\equiv$ abstracting semantic equality that is $\mathtt{b} \equiv \mathtt{b}' \Rightarrow \forall s \in \Sigma : [\![\mathtt{b}]\!]s = [\![\mathtt{b}']\!]s$. The equivalence classes are encoded by choosing an arbitrary representative $\mathtt{b}' \in [\mathtt{b}]/_{\equiv}$. The abstract equivalence $\equiv$ can be chosen within a wide range of possibilities, from syntactic equality, to the use of a simplifier, of abstract domains, or that of a SMT solver. This provides an abstract implication $\mathtt{b} \Mapsto \mathtt{b}'$ underapproximating the concrete implication $\Rightarrow$ in that $\mathtt{b} \Mapsto \mathtt{b}'$ implies that $\forall s \in \Sigma : [\![\mathtt{b}]\!]s \Rightarrow [\![\mathtt{b}']\!]s$. The equivalence is defined as $\mathtt{b} \equiv \mathtt{b}' \triangleq \mathtt{b} \Mapsto \mathtt{b}' \wedge \mathtt{b}' \Mapsto \mathtt{b}$. The basic abstract domain is therefore $\langle \mathbb{B}/_{\equiv}, \Mapsto \rangle$.

We now define the abstract domain functor

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

Notice that $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a$ denotes the pair $\langle [\mathtt{b}_p]/_{\equiv}, [\mathtt{b}_a]/_{\equiv} \rangle$ of $\mathbb{B}/_{\equiv} \times \mathbb{B}/_{\equiv}$. The interpretation of $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a$ is that 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. We exclude the elements such that $\mathtt{b}_p \Mapsto \mathtt{b}_a$ which implies $\mathtt{b}_p \Rightarrow \mathtt{b}_a$ so that no precondition is needed. An example is `if (`$\mathtt{b}_p$`) { assert(`$\mathtt{b}_a$`) }` where the assertion has already been checked on the paths leading to that assertion. The abstract ordering on $\langle \overline{\mathbb{B}}^2, \Mapsto \rangle$ is $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a \Mapsto \mathtt{b}'_p \rightsquigarrow \mathtt{b}'_a \triangleq \mathtt{b}'_p \Mapsto \mathtt{b}_p \wedge \mathtt{b}_a \Mapsto \mathtt{b}'_a$.

Different paths to different assertions are abstracted by elements of $\langle \wp(\overline{\mathbb{B}}^2), \subseteq \rangle$, each $\mathtt{b}_p \rightsquigarrow \mathtt{b}_a$ corresponding to a different path to an assertion. The number of paths can grow indefinitely so $\langle \wp(\overline{\mathbb{B}}^2), \subseteq \rangle$ must be equipped with a widening.

Finally our abstract domain will be $\langle \mathit{\Gamma} \to \wp(\overline{\mathbb{B}}^2), \dot{\subseteq} \rangle$ ordered pointwise so as to attach an abstract property $\rho(c) \in \wp(\overline{\mathbb{B}}^2)$ to each program point $c \in \mathit{\Gamma}$.

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: */
```

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

Because the abstraction is syntactic, there may be no best abstraction, so we define the concretization (recall that $\mathbb{A}$ is the set of pairs $\langle \mathtt{c}, \mathtt{b} \rangle$ such that `assert(b)` is checked at program point `c` and define $\mathbb{A}(\mathtt{c}) \triangleq \bigwedge_{\langle \mathtt{c},\, \mathtt{b} \rangle \in \mathbb{A}} \mathtt{b}$)

$$\dot\gamma \in (\Gamma \to \wp(\overline{\mathbb{B}}^2)) \to \wp(\vec\Sigma^+), \qquad \dot\gamma(\rho) \triangleq \bigcup_{c \in \Gamma} \{\vec s \in \overline\gamma_c(\rho(c)) \mid \boldsymbol\pi \vec s_0 = c\}$$

$$\overline\gamma_c \in \wp(\overline{\mathbb{B}}^2) \to \wp(\{\vec s \in \vec\Sigma^+ \mid \boldsymbol\pi \vec s_0 = c\}), \qquad \overline\gamma_c(C) \triangleq \bigcap_{b_p \rightsquigarrow b_a \in C} \gamma_c(b_p \rightsquigarrow b_a)$$

$$\gamma_c \in \overline{\mathbb{B}}^2 \to \wp(\{\vec s \in \vec\Sigma^+ \mid \boldsymbol\pi \vec s_0 = c\})$$

$$\gamma_c(\mathtt{b}_p \rightsquigarrow \mathtt{b}_a) \triangleq \{\vec s \in \vec\Sigma^+ \mid \boldsymbol\pi \vec s_0 = c \wedge [\![\mathtt{b}_p]\!]\vec s_0 \Rightarrow (\exists j < |\vec s| : [\![\mathtt{b}_a]\!]\vec s_0 = [\![\mathbb{A}(\boldsymbol\pi \vec s_j)]\!]\vec s_j)\}.$$

Observe that $\dot\gamma$ is decreasing which corresponds to the intuition that an analysis finding no path precondition $b_p \rightsquigarrow b_a$ defines all possible executions in $\vec\Sigma^+$.

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)

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

By Cor. 9 and (1-b), the analysis is sound, *i.e.*

Theorem 17 *If* $\rho \dot\subseteq \mathsf{lfp}^{\dot\subseteq} B$ *then* $\vec\tau^+ \subseteq \dot\gamma(\rho)$. □

Observe that B can be $\not\Rrightarrow$-overapproximated (*e.g.*, to allow for simplifications of the Boolean expressions).

Example 18 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:` $\rho^0(\mathtt{1}) = \emptyset$, $\rho^1(\mathtt{1}) = \{\mathtt{x} \neq 0 \rightsquigarrow \mathtt{x} > 0\}$, which is stable since the next iterate is $(\mathtt{x} \neq 0 \wedge \mathtt{x} > 0 \wedge \mathtt{x} - 1 \neq 0) \rightsquigarrow (\mathtt{x} - 1 > 0) \equiv \mathtt{x} > 1 \rightsquigarrow \mathtt{x} > 1$, which is trivially satisfied hence not added to $\rho^2(\mathtt{1}) = \rho^1(\mathtt{1})$. □

Example 19 The backward symbolic analysis of Ex. 1 moves the checks `(A != null)` to the precondition. □

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.

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

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

Example 20 For Ex. 18, the precondition generated at program point `1` will be `!(x != 0) || (x > 0)` since the static analysis was able to show that only the first assert in the loop does matter because when passed successfully it implies all the following ones. □

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 21 $\mathfrak{P}_{\mathbb{A}} \cap \mathfrak{I} \subseteq P_{\mathfrak{I}}$. □

So, by Th. 6, the data flow analysis is sound, a rejected initial state would inevitably have lead to an assertion failure.

10 Contract precondition inference for collections by forward static analysis

Symbolic execution as considered in Sect. 8 and 9 for scalars is harder for data structures since all the elements of the data structure must be handled individually without loss of precision. We propose a simple solution for collections (including arrays). The idea is to move to the precondition the assertions on elements of the collection which can be proved to be unmodified before reaching the condition.

Abstract domain for scalar variables. For scalar variables $\mathtt{x} \in \mathbb{x}$, we assume that we are given abstract properties in $\eta \in \Gamma \to \overline{\mathcal{R}}$ with concretization $\gamma_{\mathbb{x}}(\eta) \in \wp(\Sigma)$. Moreover, we consider a dataflow analysis with abstract properties $\zeta \in \Gamma \to \mathbb{x} \to \overline{\mathcal{A}}$ and pointwise extension of the order $0 \preceq 0 \prec 1 \preceq 1$ on $\overline{\mathcal{A}} \triangleq \{0, 1\}$ where 0 means "unmodified" and 1 "unknown". The concretization is

$$\gamma(\eta, \zeta) \triangleq \{\vec{s} \in \vec{\Sigma}^+ \mid \forall j < |\vec{s}| : \vec{s}_j \in \gamma_{\mathbb{x}}(\eta) \wedge \forall \mathtt{x} \in \mathbb{x} : \zeta(\boldsymbol{\pi}\vec{s}_j)(\mathtt{x}) = 0 \Rightarrow [\![\mathtt{x}]\!]\vec{s}_0 = [\![\mathtt{x}]\!]\vec{s}_j\}$$

Segmentation abstract domain. For collections $\mathtt{X} \in \mathbb{X}$, we propose to use segmentation as introduced by [16]. A segmentation abstract property in $\overline{\mathcal{S}}(\overline{\mathcal{A}})$ depends on abstract properties in $\overline{\mathcal{A}}$ holding for elements of segments. So

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

and the segmentation abstract properties have the form

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

where

– $\overline{\mathcal{E}}$ is a set of symbolic expressions in normal form depending on variables. Here, the abstract expressions $\overline{\mathcal{E}}$ are restricted to the normal form $\mathtt{v}+k$ where $\mathtt{v} \in \mathbb{x} \cup \{\mathtt{v}_0\}$ is an integer variable plus an integer constant $k \in \mathbb{Z}$ (an auxiliary variable $\mathtt{v}_0 \notin \mathbb{x}$ is assumed to be always 0 and is used to represent the integer constant k as $\mathtt{v}_0 + k$);
– the segment bounds $\{e^i_1 \ ... \ e^i_{m^i}\} \in \overline{\mathcal{B}}$, $i \in [1, n]$, $n > 1$, are finite non-empty sets of symbolic expressions in normal form $e^i_j \in \overline{\mathcal{E}}$;
– the abstract predicates $A_i \in \overline{\mathcal{A}}$ denote properties that are valid for all the elements in the collection between the bounds; and
– the optional question mark $[?^i]$ follows the upper bound of a segment. Its presence `?` means that the segment might be empty. Its absence ␣ means that the segment cannot be empty. Because this information is attached to the segment upper bound (which is also the lower bound of the next segment), the lower bound $\{e^1_1 \ \ldots \ e^1_{m^1}\}$ of the first segment never has a question mark. $\langle \{␣, \mathtt{?}\}, \preccurlyeq, \curlyvee, \curlywedge \rangle$ is a complete lattice with $␣ \prec \mathtt{?}$.

Segmentation modification and checking analyses. We consider a *segmentation modification analysis* with abstract domain $\overline{\mathcal{S}}(\overline{\mathcal{M}})$ where $\overline{\mathcal{M}} \triangleq \{\mathfrak{e}, \mathfrak{d}\}$ with $\mathfrak{e} \sqsubseteq \mathfrak{e} \sqsubset \mathfrak{d} \sqsubseteq \mathfrak{d}$. The abstract property $\mathfrak{e}$ states that all the elements in the segment must be equal to their initial value (so $\gamma_{\mathcal{M}}(\mathfrak{e}) \triangleq \{\langle v, \, v\rangle \mid v \in \mathcal{V}\}$) and the abstract property $\mathfrak{d}$ means that some element in the segment might have been modified hence might be different from its initial value (in which case we define $\gamma_{\mathcal{M}}(\mathfrak{d}) \triangleq \mathcal{V} \times \mathcal{V}$).

For each `assert` in the program, we also use a *segmentation checking analysis* with abstract domain $\overline{\mathcal{C}} \triangleq \{\bot, \mathfrak{n}, \mathfrak{c}.\top\}$ where $\bot \sqsubset \mathfrak{n} \sqsubset \top$ and $\bot \sqsubset \mathfrak{c} \sqsubset \top$ to collect the set of elements of a collection that have been checked by this `assert`. The abstract property $\bot$ is unreachability, $\mathfrak{c}$ states that all the elements in the segment have definitely been checked by the relevant `assert`, $\mathfrak{n}$ when none of the elements in the segment have been checked, and $\top$ is unknown.

Example 22 The analysis of Ex. 1 proceeds as follows (the first segmentation in $\overline{\mathcal{S}}(\overline{\mathcal{M}})$ collects element modifications for `A` while the second in segmentation $\overline{\mathcal{S}}(\overline{\mathcal{C}})$ collects the set of elements `A[i]` of `A` checked by the assertion at program point `4:` while equal to its initial value. The classical analyses for `A` (not `null` whenever used) and `i` are not shown.).

(a) `1:` $\mathtt{\{0\}\mathfrak{e}\{A.length\}?}$ $\mathtt{-}$ $\mathtt{\{0\}\mathfrak{n}\{A.length\}?}$
no element yet modified ($\mathfrak{e}$) and none checked ($\mathfrak{n}$), array may be empty
(b) `2:` $\mathtt{\{0,i\}\mathfrak{e}\{A.length\}?}$ $\mathtt{-}$ $\mathtt{\{0,i\}\mathfrak{n}\{A.length\}?}$ $\mathtt{i} = 0$
(c) `3:` $\bot \sqcup (\mathtt{\{0,i\}\mathfrak{e}\{A.length\}?}$ $\mathtt{-}$ $\mathtt{\{0,i\}\mathfrak{n}\{A.length\}?})$ join
$= \mathtt{\{0,i\}\mathfrak{e}\{A.length\}?}$ $\mathtt{-}$ $\mathtt{\{0,i\}\mathfrak{n}\{A.length\}?}$
(d) `4:` $\mathtt{\{0,i\}\mathfrak{e}\{A.length\}}$ $\mathtt{-}$ $\mathtt{\{0,i\}\mathfrak{n}\{A.length\}}$
last and only segment hence array not empty (since $\mathtt{A.length} > \mathtt{i} = 0$)
(e) `5:` $\mathtt{\{0,i\}\mathfrak{e}\{A.length\}}$ $\mathtt{-}$ $\mathtt{\{0,i\}\mathfrak{c}\{1,i+1\}\mathfrak{n}\{A.length\}?}$
`A[i]` checked while unmodified

(f) `6: {0,i}𝔡{1,i+1}𝔢{A.length}? - {0,i}𝔠{1,i+1}𝔫{A.length}?`
`A[i]` appears on the left handside of an assignment, hence is potentially modified
(g) `7: {0,i-1}𝔡{1,i}𝔢{A.length}? - {0,i-1}𝔠{1,i}𝔫{A.length}?`
invertible assignment $\mathtt{i}_{\text{old}} = \mathtt{i}_{\text{new}} - 1$
(h) `3: {0,i}𝔢{A.length}? ⊔ {0,i-1}𝔡{1,i}𝔢{A.length}? -` join
`{0,i}𝔫{A.length}? ⊔ {0,i-1}𝔠{1,i}𝔫{A.length}?`
= `{0}𝔢{i}?𝔢{A.length}? ⊔ {0}𝔡{i}𝔢{A.length}? -` segment unification
`{0}⊥{i}?𝔫{A.length}? ⊔ {0}𝔠{i}𝔫{A.length}?`
= `{0}𝔡{i}?𝔢{A.length}? - {0}𝔠{i}?𝔫{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: {0}𝔡{i}?𝔢{A.length} - {0}𝔠{i}?𝔫{A.length}`last segment not empty
(j) `5: {0}𝔡{i}?𝔢{A.length} - {0}𝔠{i}?𝔠{i+1}𝔫{A.length}?`
`A[i]` checked while unmodified
(k) `6: {0}𝔡{i}?𝔡{i+1}𝔢{A.length}? - {0}𝔠{i}?𝔠{i+1}𝔫{A.length}?`
`A[i]` potentially modified
(l) `7: {0}𝔡{i-1}?𝔡{i}𝔢{A.length}? - {0}𝔠{i-1}?𝔠{i}𝔫{A.length}?`
invertible assignment $\mathtt{i}_{\text{old}} = \mathtt{i}_{\text{new}} - 1$
(m) `3: {0}𝔡{i}?𝔢{A.length}? ⊔ {0}𝔡{i-1}𝔡{i}𝔢{A.length}? -` join
`{0}𝔠{i}?𝔫{A.length}? ⊔ {0}𝔠{i-1}𝔠{i}𝔫{A.length}?`
= `{0}𝔡{i}?𝔢{A.length}? ⊔ {0}𝔡{i}?𝔢{A.length}? -`segment unification
`{0}𝔠{i}?𝔫{A.length}? ⊔ {0}𝔠{i}?𝔫{A.length}?`
= `{0}𝔡{i}?𝔢{A.length}? - {0}𝔠{i}?𝔫{A.length}?`
segmentwise join, convergence
(n) `8: {0}𝔡{i,A.length}? - {0}𝔠{i,A.length}?`
$i \leqslant$ `A.length` in segmentation and $\geqslant$ in test negation so `i` = `A.length`.

To generate code for the precondition, the information `{0}𝔠{i,A.length}?` in (n) is valid at program `8:` dominating the end of the program, so `assert(A[i] != null)` has been checked on all the elements of the array before they where changed in the program. Hence the generated precondition is `Forall(0,A.length,k => A[k] != null)` where `k` is a dummy variable from which iterative code follows immediately.

Notice that the size of a collection can change and that the values of the symbolic bounds in a collection can change from one program point to another. So these expressions in the final segmentation must be expressed in terms of values on entry, a problem solved in Sect. 8. □

Abstract domain for collections. The abstract properties are

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

At program point $c \in \mathit{\Gamma}$, the collection $\mathtt{X} \in \mathbb{X}$ has the collection segmentation abstract property $\xi(c)(\mathtt{X})$ which is a pair $\langle \xi(c)(\mathtt{X})_{\overline{\mathcal{M}}}, \xi(c)(\mathtt{X})_{\overline{\mathcal{C}}} \rangle$. The abstract relational invariance property $\xi(c)(\mathtt{X})_{\overline{\mathcal{M}}}$ specifies which elements of the collection are for sure equal to their initial values. 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` [(11)]), the abstract trace-based property $\xi(c)(\mathtt{X})_{\overline{C}}\langle \mathtt{c},\ \mathtt{b(X,i)}\rangle$ specifies which elements of the collection have been checked for sure by `b` at point `c` while equal to their initial values.

Collection segmentation concretization. (a) The concretization $\gamma^{\mathtt{X}}_S$ of a segmentation $B_1A_1B_2[?^2]A_2\ \ldots A_{n-1}B_n[?^n] \in \overline{\mathcal{S}}(\overline{\mathcal{A}})$ for a collection `X` is the set of prefixes $\vec{s} = \vec{s}_0 \ldots \vec{s}_\ell$ of the program run describing how the elements $\mathtt{A}[k]$, $k \in [0, \mathtt{A.count})$ of the collection `X` have been organized into consecutive, non-overlapping segments, covering the whole collection.

(b) All the elements of the collection in each segment $B_kA_kB_{k+1}[?^k]$ have the property described by A_k. The values of expressions in segment bounds $B_1, \ldots, B_n$ should be understood as evaluated in this last state $\vec{s}_\ell$ while the properties A_k may refer to some or all of the states $\vec{s}_0, \ldots, \vec{s}_\ell$.

(c) The segmentation should fully cover all the elements of the collection `X`. So all the expressions in B_1 should be equal and have value 0, $\forall \mathtt{e}_1 \in B_1 : [\![\mathtt{e}_1]\!]\vec{s}_\ell = 0$ while all the expressions in B_n should be equal to the number $[\![\mathtt{X.count}]\!]\vec{s}_\ell$ of the elements in the collection, so $\forall \mathtt{e}_n \in B_n : [\![\mathtt{e}_n]\!]\vec{s}_\ell = [\![\mathtt{X.count}]\!]\vec{s}_\ell$.

(d) The segment bounds B_k, $k \in [0, n]$ are sets of equal expressions when evaluated in the last state $\vec{s}_\ell$ of the prefix trace, so $\forall \mathtt{e}_1, \mathtt{e}_2 \in B_k : [\![\mathtt{e}_1]\!]\vec{s}_\ell = [\![\mathtt{e}_2]\!]\vec{s}_\ell$.

(e) In a segment segment $B_k[?^k]M_kB_{k+1}[?^{k+1}]$, $k \in [0, n)$, the marker $[?^k]$, $k \in [1, n)$ is relevant to the previous segment, if any. $[?^{k+1}]$ specifies the possible emptiness of the segment. If $[?^{k+1}] = ?$ then the segment is possibly empty (in which case $<?$ stands for $\leqslant$). If $[?^{k+1}] = \lrcorner$ then the segment is definitely not empty (in which case there is no question mark and $<_\lrcorner$ stands for $<$). The upper bound h of the segment is therefore greater (for non-empty segments) or greater or equal (for possibly empty segments) than its lower bound l and within the limit of the collection size so $\forall \mathtt{e}_1, \mathtt{e}_2 \in B_k$, $\forall \mathtt{e}'_1, \mathtt{e}'_2 \in B_{k+1}$, $0 \leq l = [\![\mathtt{e}_1]\!]\vec{s}_\ell = [\![\mathtt{e}_2]\!]\vec{s}_\ell <[?^{k+1}] [\![\mathtt{e}'_1]\!]\vec{s}_\ell = [\![\mathtt{e}'_2]\!]\vec{s}_\ell = h < [\![\mathtt{X.count}]\!]\vec{s}_\ell$. (a—e) explains the following definition of segmentation concretization.

$$\gamma^{\mathtt{X}}_S(B_1A_1B_2[?^2]A_2 \ldots A_{n-1}B_n[?^n]) \triangleq \{\vec{s} \mid \ell = |\vec{s}| - 1 : \forall \mathtt{e}_1 \in B_1 : [\![\mathtt{e}_1]\!]\vec{s}_\ell = 0 \wedge$$
$$\forall \mathtt{e}_n \in B_n : [\![\mathtt{e}_n]\!]\vec{s}_\ell = [\![\mathtt{X.count}]\!]\vec{s}_\ell \wedge \forall k \in [0, n) : \forall \mathtt{e}_1, \mathtt{e}_2 \in B_k : \forall \mathtt{e}'_1, \mathtt{e}'_2 \in B_{k+1} :$$
$$0 \leq l = [\![\mathtt{e}_1]\!]\vec{s}_\ell = [\![\mathtt{e}_2]\!]\vec{s}_\ell <[?^{k+1}] [\![\mathtt{e}'_1]\!]\vec{s}_\ell = [\![\mathtt{e}'_2]\!]\vec{s}_\ell = h < [\![\mathtt{X.count}]\!]\vec{s}_\ell\}$$

and $\gamma^{\mathtt{X}}_S(\bot) = \emptyset$.

Segmented modification analysis concretization. The concretization $\gamma^{\mathtt{X}}_M$ of a segmentation $B_1M_1B_2[?^2]M_2\ \ldots M_{n-1}B_n[?^n] \in \overline{\mathcal{S}}(\overline{\mathcal{M}})$ for a collection `X` is the set of prefixes $\vec{s} = \vec{s}_0 \ldots \vec{s}_\ell$ of the program run describing how the collection `X` has been modified in the last state $\vec{s}_\ell$ compared to the initial state $\vec{s}_0$.

The abstract value $M_k = \mathfrak{d}$ of segment $B_k[?^k]M_kB_{k+1}[?^{k+1}]$ provides no information while $M_k = \mathfrak{e}$ states that the values of the all the elements $\mathtt{X}[i]$ of

(11) 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.

the collection $\mathtt{X}$ have not changed between the initial state $\vec{s}_0$ and the current state $\vec{s}_\ell$ (which is the last of the prefix trace). So $\forall i \in [\,\ell, h\,) : [\![\mathtt{X}]\!]\vec{s}_0[i] = [\![\mathtt{X}]\!]\vec{s}_j[i]$.

The size of the collections may change monotonically. If the collection size only decreased, then all the elements $\mathtt{X}[i]$, $i \in [\,\ell, h\,)$ did exist in the initial collection. If the collection size only increased, its current size $[\![\mathtt{X.count}]\!]\vec{s}_\ell$ is larger than the initial size $[\![\mathtt{X.count}]\!]\vec{s}_0$ so the comparison of elements can only be done for the elements $\mathtt{X}[i]$, $i \in \min(h, [\![\mathtt{X.count}]\!]\vec{s}_0)$ existing in both in the initial and current states.

$$\begin{array}{l}\gamma^{\mathtt{X}}_M(B_1M_1B_2[?^2]M_2\dots M_{n-1}B_n[?^n]) \triangleq \{\vec{s} \in \gamma^{\mathtt{X}}_S(B_1M_1B_2[?^2]M_2\dots M_{n-1}B_n[?^n]) \mid \\ \quad \ell = |\vec{s}| - 1 \wedge \forall k \in [0,n) : \exists \mathtt{e} \in B_k, l : l = [\![\mathtt{e}]\!]\vec{s}_\ell \wedge \exists \mathtt{e}' \in B_{k+1}, h : h = [\![\mathtt{e}']\!]\vec{s}_\ell \wedge \\ \quad\quad \forall i \in [\,l, \min(h, [\![\mathtt{X.count}]\!]\vec{s}_0)\,) : \langle [\![\mathtt{X}]\!]\vec{s}_0[i],\ [\![\mathtt{X}]\!]\vec{s}_j[i]\rangle \in \gamma_{\mathcal{M}}(M_k)\}.\end{array}$$

Segmented checking analysis concretization. The concretization $\gamma^{\mathtt{X}}_C$ of a segmentation $B_1C_1B_2[?^2]C_2\ \dots C_{n-1}B_n[?^n] \in \overline{\mathcal{S}}(\overline{\mathcal{C}})$ of a collection $\mathtt{X}$ for an assertion check $\langle \mathtt{c},\ \mathtt{b(X,i)}\rangle \in \mathbb{A}(\mathtt{X})$ is the set of prefixes of traces such that, at the end of the prefix, the elements of the collection in all segments $[B_k, B_{k+1})$, $k = 1, \dots, n-1$ with $C_k = \mathtt{c}$ have been submitted to the check $\langle \mathtt{c}, \mathtt{b(X,i)}\rangle \in \mathbb{A}(\mathtt{X})$ when $C_k = \mathtt{c}$ while the situation is unknown when $C_k = \mathtt{n}$.

$$\begin{array}{l}\gamma^{\mathtt{X}}_C(B_1C_1B_2[?^2]C_2\dots C_{n-1}B_n[?^n])(\langle \mathtt{c},\ \mathtt{b(X,i)}\rangle) \triangleq \{\vec{s} \in \gamma^{\mathtt{X}}_S(B_1C_1B_2[?^2]C_2\dots B_n[?^n]) \mid \\ \quad \ell = |\vec{s}| - 1 \wedge \forall k \in [0,n) : \exists \mathtt{e} \in B_k, l : l = [\![\mathtt{e}]\!]\vec{s}_\ell \wedge \exists \mathtt{e}' \in B_{k+1}, h : h = [\![\mathtt{e}']\!]\vec{s}_\ell \wedge \\ \quad\quad (C_k = \mathtt{c}) \Rightarrow (\forall i \in [\,l, \min(h, [\![\mathtt{X.count}]\!]\vec{s}_0)\,) : \exists j \leqslant \ell : \\ \quad\quad\quad\quad \boldsymbol{\pi}\vec{s}_j = \mathtt{c} \wedge [\![\mathtt{i}]\!]\vec{s}_j = i \wedge [\![\mathtt{X}]\!]\vec{s}_0[i] = [\![\mathtt{X}]\!]\vec{s}_j[i])\}.\end{array}$$

The modification analysis must be used to determine that $[\![\mathtt{X}]\!]\vec{s}_0[i] = [\![\mathtt{X}]\!]\vec{s}_j[i]$.

Segmented modification and checking analysis concretization. The concretization is

$$\begin{array}{rcl}\gamma & \in & (\mathit{\Gamma} \to \mathtt{X} \in \mathbb{X} \mapsto \overline{\mathcal{S}}(\overline{\mathcal{M}}) \times \mathbb{A}(\mathtt{X}) \to \overline{\mathcal{S}}(\overline{\mathcal{C}})) \mapsto \vec{\Sigma}^+ \\ \gamma(\xi) & \triangleq & \big\{\vec{s} \in \vec{\Sigma}^+ \;\big|\; \forall j < |\vec{s}| : \forall \mathtt{X} \in \mathbb{X} : \forall \langle \mathtt{c},\ \mathtt{b(X,i)}\rangle \in \mathbb{A}(\mathtt{X}) : \\ & & \qquad \vec{s}_0 \dots \in \vec{s}_j \in \gamma^{\mathtt{X}}_C\big(\xi(\boldsymbol{\pi}\vec{s}_j)(\mathtt{X})_{\overline{\mathcal{C}}}(\langle \mathtt{c},\ \mathtt{b(X,i)}\rangle)\big)\big\}\end{array}$$

The soundness of the result $\xi \in \mathit{\Gamma} \to \mathtt{X} \in \mathbb{X} \mapsto \overline{\mathcal{S}}(\overline{\mathcal{M}}) \times \mathbb{A}(\mathtt{X}) \to \overline{\mathcal{S}}(\overline{\mathcal{C}})$ of collection segmentation modification and checking static analysis is stated by $\vec{\tau}^+ \subseteq \gamma(\xi)$. The details of the segmentation analysis are those of [16] for the specific abstract domains $\overline{\mathcal{M}}$ and $\overline{\mathcal{C}}$.

Precondition generation. Let $\mathtt{f}$ be the exit program point (assumed to be unique for simplicity and corresponding to a blocking state $\forall s \in \Sigma : \boldsymbol{\pi}s = \mathtt{f} \Rightarrow s \in \mathfrak{B}$). Let $\mathtt{X} \in \mathbb{X}$ be any of the collection variables in the program. Let $\langle \mathtt{c},\ \mathtt{b(X,i)}\rangle \in \mathbb{A}(\mathtt{X})$ by any assertion check for element $\mathtt{X[i]}$ of collection $\mathtt{X}$. let $\xi(\mathtt{f})(\mathtt{X})_{\overline{\mathcal{C}}}(\langle \mathtt{c},\ \mathtt{b(X,i)}\rangle) = B_1C_1B_2[?^2]C_2\ \dots C_{n-1}B_n[?^n] \in \overline{\mathcal{S}}(\overline{\mathcal{C}})$ be the information collected by the checking analysis (using the modification analysis no longer useful for the precondition generation). Let $\mathit{\Delta} \subseteq [1, n)$ be the set of indices $k \in \mathit{\Delta}$ for which $C_k = \mathtt{c}$. The precondition code is

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

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 `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.*

11 Related work, future work, and conclusions

The problem of calculating (weakest)-preconditions has been intensively studied since [17] e.g., [19] using assisted theorem proving. In the context of static analysis by abstract interpretation, the problem can be handled by backward analysis [13, Sect. 3.2] including in symbolic form [15,8], a combination of forward and backward analyzes [9] (see also [14]), and overapproximation of negated properties to get underapproximations [10] followed by [6,27]. Most often the precondition inference problem is considered in the context of partial or total correctness, including for procedure summary [7,12,20], contract inference [1] or specification abduction [7], where no bad behavior is allowed at all [17] so one has to consider under-approximations to ensure that any assertion that exists in the code holds when reached [25,28]. Our point of view for non-deterministic programs is different and, to our knowledge, our formalization of the precondition inference problem is the first in the context of design by contracts. The derived precondition never excludes a bad run when a non-deterministic choice could alternatively yield a good run. So the program is not checked for partial/-total correctness, but the intentions of the programmer, as only expressed by his code and assertions within this code, are preserved, since only definite failures are prohibited. Future work includes the implementation, the combination of Sect. 10 with path-conditions as in Sect. 8, the study of the relation between forward and backward analyzes (using [10, Th. 10.13]), of infinite behaviors and of expressive abstract domains than segmentation to express relations between values of components of data structures in `asserts` and on code entry while preserving scalability.

References

[1] Arnout, K., Meyer, B.: Uncovering hidden contracts: The .NET example. IEEE Computer 36(11), 48–55 (2003)

[2] Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)

[3] Barnett, M., Fähndrich, M., Garbervetsky, D., Logozzo, F.: Annotations for (more) precise points-to analysis. IWACO '07. Stockholm U. and KTH (2007)

[4] Barnett, M., Fähndrich, M., Logozzo, F.: Embedded contract languages. SAC'10. 2103–2110. ACM (2010)

[5] Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)

[6] Bourdoncle, F.: Abstract debugging of higher-order imperative languages. PLDI '93. 46–55. ACM (1993)

[7] Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. 36$^{\text{th}}$ POPL. 289–300. ACM (2009)

[8] Chandra, S., Fink, S., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. PLDI. 363–374. ACM (2009)

[9] 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)

[10] Cousot, P.: Semantic foundations of program analysis. Muchnick, S., Jones, N. (eds.) Program Flow Analysis: Theory and Applications, ch. 10, 303–342. Prentice-Hall (1981)

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

[12] Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. Neuhold, E. (ed.) IFIP Conf. on Formal Description of Programming Concepts. 237–277. North-Holland (1977)

[13] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. 6$^{\text{th}}$ POPL. 269–282. ACM (1979)

[14] Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. Journal of Logic Programming 13(2–3), 103–179 (1992),

[15] Cousot, P., Cousot, R.: Modular static program analysis. CC 2002. 159–178. LNCS 2304, Springer, Grenoble, France (2002)

[16] Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. POPL '2011. ACM Press (2011)

[17] Dijkstra, E.: Guarded commands, nondeterminacy and formal derivation of programs. CACM 18(8), 453–457 (1975)

[18] Fähndrich, M., Logozzo, F.: Clousot: Static contract checking with abstract interpretation. FoVeOOS. Springer (2010)

[19] Flanagan, C., Leino, K., Lillibridge, M., Nelson, G., Saxe, J., Stata, R.: Extended Static Checking for Java. PLDI. 234–245. ACM (2002)

[20] Gulwani, S., Tiwari, A.: Computing procedure summaries for interprocedural analysis. ESOP '07, 253–267. LNCS 4421, Springer (2007)

[21] Hecht, M.: Flow Analysis of Computer Programs. Elsevier North-Holland (1977)

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

[23] Meyer, B.: Eiffel: The Language. Prentice Hall (1991)

[24] Meyer, B.: Applying "Design by Contract". IEEE Computer 25(10), 40–51 (1992)

[25] Moy, Y.: Sufficient preconditions for modular assertion checking. VMCAI 08. 188–202. LNCS 4905, Springer (2008)

[26] Park, D.: Fixpoint induction and proofs of program properties. Meltzer, B., Michie, D. (eds.) Machine Intelligences, vol. 5, 59–78. Edinburgh University Press (1969)

[27] Rival, X.: Understanding the origin of alarms in Astrée. SAS '05, 303–319. LNCS 3672, Springer (2005)

[28] T.Lev-Ami, Sagiv, M., Reps, T., Gulwani, S.: Backward analysis for inferring quantified preconditions. Tr-2007-12-01, Tel Aviv University (2007)