Rapid Semantics

Pamina Georgiou, Bernhard Gleiss

TU Wien

1 Running Example

```
func main() {
  const Int[] a;
  Int[] b;
  Int i = 0;
  Int j = 0;
  while (i < a.length) {
    if (a[i] ≥ 0) {
      b[j] = a[i];
      j = j + 1:
    }
      i = i + 1;
  }
}
assert (∀k_𝕀.∃l_𝕀.((0 ≤ k <j ∧ a.length ≥ 0) → b(k) = a(l)))
```

Fig. 1. Program copying positive elements from array `a` to `b`.

2 Programming Model $\mathcal{W}$

We consider programs written in an imperative while-like programming language $\mathcal{W}$. We do not consider multiple program traces in $\mathcal{W}$. In Section 3, we then introduce a generalized program semantics in trace logic $\mathcal{L}$, extended with reachability predicates.
Figure 2 shows the (partial) grammar of our programming model $\mathcal{W}$, emphasizing the use of contexts to capture lists of statements. An input program in $\mathcal{W}$ has a single `main`-function, with arbitrary nestings of if-then-else conditionals and while-statements. We consider *mutable and constant variables*, where variables are either integer-valued numeric variables or arrays of such numeric variables. We include standard *side-effect free expressions over booleans and integers.*

2.1 Locations and Timepoints

A program in $\mathcal{W}$ is considered as sets of locations, with each location corresponding to positions/lines of program statements in the program. Given a program statement `s`, we denote by l_s its (program) location. We reserve the location l_{end} to denote the end of a program. For programs with loops, some program locations might be revisited multiple times. We therefore model locations l_s corresponding to a statement `s` as functions of *iterations* when the respective location is visited. For simplicity, we write l_s also for the functional representation of the location

```
   program := function
  function := func main(){ context }
subprogram := statement | context
 statement := atomicStatement
            | if( condition ){ context } else { context }
            | while( condition ){ context }
   context := statement; ... ; statement
```

Fig. 2. Grammar of $\mathcal{W}$.

l_s of `s`. We thus consider locations as timepoints of a program and treat them l_s as being functions l_s over iterations. The target sort of locations l_s is $\mathbb{L}$. For each enclosing loop of a statement `s`, the function symbol l_s takes arguments of sort $\mathbb{N}$, corresponding to loop iterations. Further, when `s` is a loop itself, we also introduce a function symbol n_s with argument and target sort $\mathbb{N}$; intuitively, n_s corresponds to the last loop iteration of `s`. We denote the set of all function symbols l_s as S_{Tp}, whereas the set of all function symbols n_s is written as S_n.

Example 1. We refer to program statements `s` by their (first) line number in Figure 1. Thus, l_5 encodes the timepoint corresponding to the first assignment of `i` in the program (line 5). We write $l_7(\mathtt{0})$ and $l_7(n_7)$ to denote the timepoints of the first and last loop iteration, respectively. The timepoints $l_8(\mathtt{s}(\mathtt{0}))$ and $l_8(it)$ correspond to the beginning of the loop body in the second and the it-th loop iterations, respectively. □

2.2 Expressions over Timepoints

We next introduce commonly used expressions over timepoints. For each while-statement `w` of $\mathcal{W}$, we introduce a function $it^{\mathtt{w}}$ that returns a unique variable of sort $\mathbb{N}$ for `w`, denoting loop iterations of `w`. Let $w_1, \ldots, w_k$ be the enclosing loops for statement `s` and consider an arbitrary term it of sort $\mathbb{N}$. We define $tp_{\mathtt{s}}$ to be the expressions denoting the timepoints of statements `s` as

$$\begin{aligned} tp_{\mathtt{s}} &:= l_s(it^{w_1}, \ldots, it^{w_k}) && \text{if } \mathtt{s} \text{ is non-while statement} \\ tp_{\mathtt{s}}(it) &:= l_s(it^{w_1}, \ldots, it^{w_k}, it) && \text{if } \mathtt{s} \text{ is while-statement} \\ lastIt_{\mathtt{s}} &:= n_s(it^{w_1}, \ldots, it^{w_k}) && \text{if } \mathtt{s} \text{ is while-statement} \end{aligned}$$

If `s` is a while-statement, we also introduce $lastIt_{\mathtt{s}}$ to denote the last iteration of `s`. Further, consider an arbitrary subprogram `p`, that is, `p` is either a statement or a context. The timepoint $start_{\mathtt{p}}$ (parameterized by an iteration of each enclosing loop) denotes the timepoint when the execution of `p` has started and is defined as

$$start_{\mathtt{p}} := \begin{cases} tp_{\mathtt{p}}(\mathtt{0}) & \text{if } \mathtt{p} \text{ is while-statement} \\ tp_{\mathtt{p}} & \text{if } \mathtt{p} \text{ is non-while statement} \\ start_{\mathtt{s_1}} & \text{if } \mathtt{p} \text{ is context } \mathtt{s_1};\ldots;\mathtt{s_k} \end{cases}$$

We also introduce the timepoint $end_{\mathtt{p}}$ to denote the timepoint upon which a subprogram `p` has been completely evaluated and define it as

$$end_{\mathtt{p}} := \begin{cases} start_{\mathtt{s}} & \text{if } \mathtt{s} \text{ occurs after } \mathtt{p} \text{ in a context} \\ end_{\mathtt{c}} & \text{if } \mathtt{p} \text{ is last statement in context } \mathtt{c} \\ end_{\mathtt{s}} & \text{if } \mathtt{p} \text{ is context of if-branch or} \\ & \text{else-branch of } \mathtt{s} \\ tp_{\mathtt{s}}(\mathtt{s}(it^s)) & \text{if } \mathtt{p} \text{ is context of body of } \mathtt{s} \\ l_{end} & \text{if } \mathtt{p} \text{ is top-level context} \end{cases}$$

Finally, if s is the topmost statement of the top-level context in `main()`, we define

$$start := start_s.$$

2.3 Program Variables

We express values of program variables `v` at various timepoints of the program execution. To this end, we model (numeric) variables `v` as functions $v : \mathbb{L} \mapsto \mathbb{I}$, where $v(tp)$ gives the value of `v` at timepoint tp. For array variables `v`, we add an additional argument of sort $\mathbb{I}$, corresponding to the position where the array is accessed; that is, $v : \mathbb{L} \times \mathbb{I} \mapsto \mathbb{I}$. The set of such function symbols corresponding to program variables is denoted by S_V.
Our framework for constant, non-mutable variables can be simplified by omitting the timepoint argument in the functional representation of such program variables, as illustrated below.

Example 2. For Figure 1, we denote by $i(l_5)$ the value of program variable `i` before being assigned in line 5. As the array variable `a` is non-mutable (specified by `const` in the program), we write $a(i(l_8(it)))$ for the value of array `a` at the position corresponding to the current value of `i` at timepoint $l_8(it)$. For the mutable array `b`, we consider timepoints where `b` has been updated and write $b(l_9(it), j(l_9(it)))$ for the array `b` at position `j` at the timepoint $l_9(it)$ during the loop. □

We emphasize that we consider (numeric) program variables `v` to be of sort $\mathbb{I}$, whereas loop iterations it are of sort $\mathbb{N}$.

2.4 Program Expressions

Arithmetic constants and program expressions are modeled using integer functions and predicates. Let `e` be an arbitrary program expression and write $[\![\mathtt{e}]\!](tp)$

to denote the value of the evaluation of `e` at timepoint tp. Let $v \in S_V$, that is a function v denoting a program variable `v`. Consider `e`, $\mathtt{e}_1, \mathtt{e}_2$ to be program expressions and let tp_1, tp_2 denote two timepoints. We define

$$\begin{aligned}&Eq(v, tp_1, tp_2) := \\ &\qquad \begin{cases} \forall pos_{\mathbb{I}}.\ v(tp_1, pos) \simeq v(tp_2, pos), & \text{if } \mathtt{v} \text{ is an array} \\ v(tp_1) \simeq v(tp_2), \text{otherwise} \end{cases}\end{aligned}$$

to denote that the program variable `v` has the same values at tp_1 and tp_2. We further introduce

$$EqAll(tp_1, tp_2) := \bigwedge_{v \in S_V} Eq(v, tp_1, tp_2)$$

to define that all program variables have the same values at timepoints tp_1 and tp_2. We also define

$$\begin{aligned}&Update(v, e, tp_1, tp_2) := \\ &\qquad v(tp_2) \simeq [\![\mathtt{e}]\!](tp_1) \wedge \textstyle\bigwedge_{v' \in S_V \setminus \{v\}} Eq(v', tp_1, tp_2),\end{aligned}$$

asserting that the numeric program variable `v` has been updated while all other program variables `v'` remain unchanged. This definition is further extended to array updates as

$$\begin{aligned}&UpdateArr(v, e_1, e_2, tp_1, tp_2) := \\ &\qquad \forall pos_{\mathbb{I}}.\ (pos \not\simeq [\![e_1]\!](tp_1) \rightarrow v(tp_2, pos) \simeq v(tp_1, pos)) \\ &\qquad \wedge\ v(tp_2, [\![e_1]\!](tp_1)) \simeq [\![e_2]\!](tp_1) \\ &\qquad \textstyle\bigwedge_{v' \in S_V \setminus \{v\}} Eq(v', tp_1, tp_2).\end{aligned}$$

Example 3. In Figure 1, we refer to the value of `i+1` at timepoint $l_{12}(it)$ as $i(l_{12}(it)) + 1$. Let S_V^1 be the set of function symbols representing the program variables of Figure 1. For an update of `j` in line 10 at some iteration it, we derive

$$\begin{aligned}Update(j, \mathtt{j+1}, l_9(it), l_{10}(it)) &:= j(l_{10}(it)) \simeq (j(l_9(it)) + 1) \\ \wedge \bigwedge_{v' \in S_V^1 \setminus \{j\}} & Eq(v', l_9(it), l_{10}(it)).\end{aligned}$$

□

3 Axiomatic Semantics in Trace Logic $\mathcal{L}$

3.1 Trace Logic $\mathcal{L}$

Trace logic $\mathcal{L}$ is an instance of many-sorted first-order logic with equality. We define the signature $\Sigma(\mathcal{L})$ of trace logic as

$$\Sigma(\mathcal{L}) := S_{\mathbb{N}} \cup S_{\mathbb{I}} \cup S_{Tp} \cup S_V \cup S_n,$$

containing the signatures of the theory of natural numbers (term algebra) $\mathbb{N}$ and integers $\mathbb{I}$, as well the respective sets of timepoints, program variables and last iteration symbols as defined in section 2.
We next define the semantics of $\mathcal{W}$ in trace logic $\mathcal{L}$.

3.2 Reachability and its Axiomatization

We introduce a predicate $\mathit{Reach} : \mathbb{L} \mapsto \mathbb{B}$ to capture the set of timepoints reachable in an execution and use *Reach* to define the axiomatic semantics of $\mathcal{W}$ in trace logic $\mathcal{L}$. We define reachability *Reach* as a predicate over timepoints, in contrast to defining reachability as a predicate over program configurations such as in [?,?,?,?].
We axiomatize *Reach* using trace logic formulas as follows.

Definition 1 **(*Reach*-predicate).** *For any context* c*, any statement* s*, let* Cond_s *be the expression denoting a potential branching condition in* s*. We define*

$$\mathit{Reach}(\mathit{start}_c) := \begin{cases} \mathit{true}, \\ \quad \text{if } \mathtt{c} \text{ is top-level context} \\ \mathit{Reach}(\mathit{start}_s) \wedge \mathit{Cond}_s(\mathit{start}_s), \\ \quad \text{if } \mathtt{c} \text{ is context of if-branch of } \mathtt{s} \\ \mathit{Reach}(\mathit{start}_s) \wedge \neg \mathit{Cond}_s(\mathit{start}_s), \\ \quad \text{if } \mathtt{c} \text{ is context of else-branch of } \mathtt{s} \\ \mathit{Reach}(\mathit{start}_s) \wedge \mathit{it}^s < \mathit{lastIt}_s, \\ \quad \text{if } \mathtt{c} \text{ is context of body of } \mathtt{s}. \end{cases}$$

For any non-while statement $\mathtt{s}'$ *occurring in context* c*, let*

$$\mathit{Reach}(\mathit{start}_{s'}) := \mathit{Reach}(\mathit{start}_c),$$

and for any while-statement $\mathtt{s}'$ *occurring in context* c*, let*

$$\mathit{Reach}(\mathit{tp}_{s'}(\mathit{it}^{s'})) := \mathit{Reach}(\mathit{start}_c) \wedge \mathit{it}^{s'} \leq \mathit{lastIt}_{s'}.$$

Finally let $\mathit{Reach}(\mathit{end}) := \mathit{true}$. □

Note that our reachability predicate *Reach* allows specifying properties about intermediate timepoints (since those properties can only hold if the referred timepoints are reached) and supports reasoning about which locations are reached.

3.3 Axiomatic Semantics of $\mathcal{W}$

We axiomatize the semantics of each program statement in $\mathcal{W}$, and define the semantics of a program in $\mathcal{W}$ as the conjunction of all these axioms.

Main-function Let $\mathtt{p}_0$ be an arbitrary, but fixed program in $\mathcal{W}$; we give our definitions relative to $\mathtt{p}_0$. The semantics of $\mathtt{p}_0$, denoted by $[\![\mathtt{p}_0]\!]$, consists of a conjunction of one implication per statement, where each implication has the reachability of the start-timepoint of the statement as premise and the semantics of the statement as conclusion:

$$[\![\mathtt{p}_0]\!] := \bigwedge_{\mathtt{s} \text{ statement of } \mathtt{p}_0} \forall \mathit{enclIts}. \big(\mathit{Reach}(\mathit{start}_s) \rightarrow [\![\mathtt{s}]\!]\big)$$

where *enclIts* is the set of iterations $\{it^{w_1}, \ldots, it^{w_n}\}$ of all enclosing loops $w_1, \ldots, w_n$ of some statement $\mathtt{s}$ in $\mathtt{p}_0$, and the semantics $[\![\mathtt{s}]\!]$ of program statements $\mathtt{s}$ is defined as follows.

Skip Let `s` be a statement `skip`. Then

$$[\![\mathtt{s}]\!] := \mathit{EqAll}(\mathit{end}_{\mathtt{s}}, \mathit{start}_{\mathtt{s}}) \tag{1}$$

Break Let `s` be a statement `break`. Then

$$\mathit{TODO} \tag{2}$$

Continue Let `s` be a statement `continue`. Then

$$\mathit{TODO} \tag{3}$$

Early-Return Let `s` be a statement `return`. Then

$$\mathit{TODO} \tag{4}$$

Integer assignments Let `s` be an assignment `v = e`, where `v` is an integer-valued program variable and `e` is an expression. The evaluation of `s` is performed in one step such that, after the evaluation, the variable `v` has the same value as `e` before the evaluation. All other variables remain unchanged and thus

$$[\![\mathtt{s}]\!] := \mathit{Update}(v, e, \mathit{end}_s, \mathit{start}_s) \tag{5}$$

Array assignments Consider `s` of the form $\mathtt{a[e_1]\ =\ e_2}$, with `a` being an array variable and $\mathtt{e}_1, \mathtt{e}_2$ being expressions. The assignment is evaluated in one step. After the evaluation of `s`, the array `a` contains the value of $\mathtt{e}_2$ before the evaluation at position *pos* corresponding to the value of $\mathtt{e}_1$ before the evaluation. The values at all other positions of `a` and all other program variables remain unchanged and hence

$$[\![\mathtt{s}]\!] := \mathit{UpdateArr}(v, e_1, e_2, \mathit{end}_s, \mathit{start}_s) \tag{6}$$

Conditional if-then-else Statements Let `s` be $\mathtt{if(Cond)\{c_1\}\ else\ \{c_2\}}$. The semantics of `s` states that entering the if-branch and/or entering the else-branch does not change the values of the variables and we have

$$[\![\mathtt{s}]\!] := \quad [\![\mathtt{Cond}]\!](\mathit{start}_s) \rightarrow \mathit{EqAll}(\mathit{start}_{\mathtt{c}_1}, \mathit{start}_s) \tag{7a}$$

$$\wedge \quad \neg[\![\mathtt{Cond}]\!](\mathit{start}_s) \rightarrow \mathit{EqAll}(\mathit{start}_{\mathtt{c}_2}, \mathit{start}_s) \tag{7b}$$

where the semantics $[\![\mathtt{Cond}]\!]$ of the expression `Cond` is according to Section 2.4.

While-Statements Let `s` be the while-statement `while(Cond){c}`. We refer to `Cond` as the *loop condition*. The semantics of `s` is captured by conjunction of the following three properties: (8a) the iteration $lastIt_s$ is the first iteration where `Cond` does not hold, (8b) entering the loop body does not change the values of the variables, (8c) the values of the variables at the end of evaluating `s` are the same as the variable values at the loop condition location in iteration $lastIt_s$. As such, we have

$$\begin{aligned}
[\![\mathtt{s}]\!] := \quad & \forall it^s_{\mathbb{N}}.\ (it^s < lastIt_{\mathtt{s}} \to [\![\mathtt{Cond}]\!](tp_{\mathtt{s}}(it^s))) \\
\wedge \quad & \neg[\![\mathtt{Cond}]\!](tp(lastIt_{\mathtt{s}})) && \text{(8a)} \\
\wedge \quad & \forall it^s_{\mathbb{N}}.\ (it^s < lastIt_{\mathtt{s}} \to EqAll(start_{\mathtt{c}}, tp_{\mathtt{s}}(it^s)) && \text{(8b)} \\
\wedge \quad & EqAll(end_{\mathtt{s}}, tp_s(lastIt_{\mathtt{s}})) && \text{(8c)}
\end{aligned}$$

3.4 Soundness and Completeness.

The axiomatic semantics of $\mathcal{W}$ in trace logic is sound. That is, given a program `p` in $\mathcal{W}$ and a trace logic property $F \in \mathcal{L}$, we have that any interpretation in $\mathcal{L}$ is a model of F according to the small-step operational semantics of $\mathcal{W}$. We conclude the next theorem - and refer to [?] for details.

Theorem 1 ($\mathcal{W}$-Soundness). *Let `p` be a program. Then the axiomatic semantics $[\![p]\!]$ is sound with respect to standard small-step operational semantics.* □ *[-.5em]*

Next, we show that the axiomatic semantics of $\mathcal{W}$ in trace logic $\mathcal{L}$ is complete with respect to Hoare logic [?], as follows.
Intuitively, a Hoare Triple $\{F_1\}\mathtt{p}\{F_2\}$ corresponds to the trace logic formula

$$\forall enclIts.\big(Reach(start_p) \to ([F_1](start_p) \to [F_2](end_p))\big) \tag{9}$$

where the expressions $[F_1](start_p)$ and $[F_2](end_p)$ denote the result of adding to each program variable in F_1 and F_2 the timepoints $start_p$ respectively end_p as first arguments. We therefore define that the axiomatic semantics of $\mathcal{W}$ is *complete with respect to Hoare logic*, if for any Hoare triple $\{F_1\}\mathtt{p}\{F_2\}$ valid relative to the background theory $\mathcal{T}$, the corresponding trace logic formula (9) is derivable from the axiomatic semantics of $\mathcal{W}$ in the background theory $\mathcal{T}$. With this definition at hand, we get the following result, proved formally in [?].

Theorem 2 ($\mathcal{W}$-Completeness with respect to Hoare logic). *The axiomatic semantics of $\mathcal{W}$ in trace logic is complete with respect to Hoare logic.* □

4 Small-step operational semantics

Here, we give a small-step operational semantics of $\mathcal{W}$. Our presentation is semantically equivalent to standard small-step operational semantics, but differs

syntactically in several points, in order to simplify later definitions and theorems: (i) we annotate while-statements with counters to ensure the uniqueness of timepoints during the execution, (ii) we reference nodes in the program-tree to keep track of the current location during the execution instead of using strings to denote the remaining program, (iii) we avoid additional constructs like states or configurations, (iv) we keep the timepoints in the execution separated from the values of the program variables at these timepoints, and (v) we evaluate expressions on the fly. We start by formalizing single steps of the execution of the program as transition rules, as defined in Figure 3. Intuitively, the rules describe (i) how we move the location-pointer around on the program-tree and (ii) how the state changes while moving the location-pointer around. Each rule consists of (i) a premise $\mathit{Reach}(tp_1)$ for some timepoint tp_1, (ii) an additional premise F (omitted if F is $\top$), the so-called *side-condition*, which is an arbitrary trace-logic formula referencing only the timepoint tp_1, (iii) the first conjunct of the conclusion of the form $\mathit{Reach}(tp_2)$ for some timepoint tp_2, and (iv) the second conjunct of the conclusion, which again is an arbitrary trace-logic formula G referencing only the timepoints tp_1 and tp_2.
Next, we formalize the possible executions of the program as a set of first-order interpretations, so-called *execution interpretations*. In a nutshell, execution interpretations can be described as follows. Each possible execution of the program induces an interpretation. For each such execution, the predicate symbol *Reach* is interpreted as the set of timepoints which are reached during the execution. The function symbols denoting values of program variables are interpreted according to the transition rules at the timepoints which are reached during the execution, and are interpreted arbitrarily at all other timepoints.
We construct execution interpretation iteratively, as follows: We move around the program as defined by the transition rules. Whenever we reach a new timepoint, we choose a program state J', such that the side-conditions of the transition rule are fulfilled, and extend the current interpretation J with J'. We furthermore collect all timepoints that we already reached in I. We stop as soon as we reach end. We then construct an execution interpretation as follows: we interpret *Reach* as I, extend J to an interpretation of S_V by choosing an arbitrary state at any timepoint which we did not reach, and choose an arbitrary interpretation of the theory symbols according to the background theory.

Definition 2 (Program state). *A* program state at timepoint *tp is a partial interpretation, which exactly contains (i) for each non-array variable $\mathtt{v}$ an interpretation of $v(tp)$ and (ii) for each array variable $\mathtt{a}$ and for each element pos of the domain $S_{\mathbb{I}}$ an interpretation of $a(tp, pos)$.*

Definition 3 (Execution interpretation). *Let p_0 be a fixed program. Let I, J be any possible result returned by the algorithm in Algorithm 1. Let M be any interpretation, such that (i) $\mathit{Reach}(tp)$ is true iff $tp \in I$, (ii) M is an extension of J, and (iii) M interprets the symbols of the background theory according to the theory. Then M is called an* execution interpretation of p_0.

$$[init^{sos}]\ \frac{}{Reach(start)}$$

Let s be a `skip`.

$$[skip^{sos}]\ \frac{Reach(start_s)}{Reach(end_s) \wedge EqAll(start_s, end_s)}$$

Let s be an assignment `v = e`.

$$[asg^{sos}]\ \frac{Reach(start_s)}{Reach(end_s) \wedge Update(v, e, start_s, end_s)}$$

Let s be an array-assignment `v[e₁] = e₂`.

$$[asg^{sos}_{arr}]\ \frac{Reach(start_s)}{Reach(end_s) \wedge UpdateArr(v, e_1, e_2, start_s, end_s)}$$

Let `s` be `if(Cond){c₁}else{c₂}`.

$$[ite^{sos}_T]\ \frac{Reach(start_s) \qquad [\![Cond]\!](start_s)}{Reach(start_{c_1}) \wedge EqAll(start_s, start_{c_1})}$$

$$[ite^{sos}_F]\ \frac{Reach(start_s) \qquad \neg[\![Cond]\!](start_s)}{Reach(start_{c_2}) \wedge EqAll(start_s, start_{c_2})}$$

Let `s` be `while(Cond){c}`.

$$[while^{sos}_T]\ \frac{Reach(tp_s(it^s)) \qquad [\![Cond]\!](tp_s(it^s))}{Reach(start_c) \wedge EqAll(tp_s(it^s), start_c)}$$

$$[while^{sos}_F]\ \frac{Reach(tp_s(it^s)) \qquad \neg[\![Cond]\!](tp_s(it^s))}{Reach(end_s) \wedge EqAll(tp_s(it^s), end_s)}$$

Fig. 3. Small-step operational semantics using *tp*, *start*, *end*.

Algorithm 1 Algorithm to compute execution interpretation.

```
$curr = start$
$I = \{curr\}$
$J =$ choose program state at $curr$
while $curr \neq end$ do
    choose $r := \frac{\sigma Reach(curr) \qquad \sigma F}{\sigma Reach(next) \wedge \sigma G}$, with $J \vDash \sigma F$
    if $r$ is $[while^{sos}_F]$ for some statement $s$ then
        $J = J \cup \{\sigma lastIt_s \mapsto \sigma it^s\}$
    choose a program state $J'$ such that $J \cup J' \vDash \sigma G$.
    $J = J \cup J'$
    $I = I \cup \{next\}$
    $curr = next$
return $I, J$
```

With the definition of execution interpretations at hand, we are now able to define the valid properties of a program as the properties which hold in each execution interpretation.

Definition 4. *Let p_0 be a fixed program. Let F be a trace logic formula. Then F is called* valid *with respect to p_0, if F holds in each execution interpretation of p_0.*

We conclude this subsection with stating simple properties of executions. The (i) first property states that whenever we reach the start of the execution of a subprogram `p`, we also reach the end of the execution of `p`. The (ii) second property states that whenever we reach the start of the execution of a context `c`, we also reach the start of the execution of each statement occurring in `c`. The (iii) third property states that whenever we reach the start of the execution of a while-statement `s`, then (a) we also reach the loop-condition check of `s` in each iteration up to and including the last iteration, and (b) we also reach the start of the execution of the context of the loop body of `s` in each iteration before the last iteration. Formally, we have the following result.

Lemma 1. *Let $\mathtt{p}_0$ be a fixed program and let M be an execution interpretation of $\mathtt{p}_0$. Let further `p` be an arbitrary subprogram of $\mathtt{p}_0$ and σ be an arbitrary grounding of the enclosing iterations of `p` such that $\sigma Reach(start_p)$ holds. Then:*
1. *$\sigma Reach(end_p)$ holds in M.*
2. *If `p` is a context, $\sigma Reach(start_{s_i})$ holds in M for any statement $\mathtt{s_i}$ occurring in `p`.*
3. *If `p` is a while-statement `while(Cond){c}`, then*
   a. *$\sigma Reach(tp_p(it^p))$ holds in M for any iteration $it^p \leq \sigma(lastIt_p)$.*
   b. *$\sigma\sigma' Reach(start_c)$ holds in M for any iteration with $\sigma' it^p < \sigma(lastIt_p)$, where σ' is any grounding of it^p.*

Proof. We prove all three properties using a single induction proof. We proceed by structural induction over the program structure with the induction hypothesis

$$\forall enclIts.\big(Reach(start_p) \rightarrow Reach(end_p)\big).$$

Let `p` be an arbitrary subprogram of $\mathtt{p}_0$. For an arbitrary grounding σ of the enclosing iterations assume that $\sigma Reach(start_p)$ holds in M. In order to show that $\sigma Reach(end_p)$ holds in M, we perform a case distinction on the type of `p`:
- Assume `p` is `skip`, or an integer- or array-assignment: Since $\sigma Reach(start_p)$ holds in M, the rule $skip^{sos}$ resp. asg^{sos} resp. asg^{sos}_{arr} applies, so $\sigma Reach(end_p)$ holds in M too.
- Assume `p` is a context $s_1; \ldots ; s_k$. By definition $start_p = start_{s_1}$, therefore $\sigma Reach(start_{s_1})$ holds in M. By the induction hypothesis, we know that $\sigma Reach(start_{s_i}) \rightarrow \sigma Reach(end_{s_i})$ holds in M for any $1 \leq i \leq k$. Using a trivial induction, we conclude that $\sigma Reach(end_{s_i})$ holds in M for any $1 \leq i \leq k$.
- Assume `p` is `if(Cond){c`$_1$`} else {c`$_2$`}`. Assume w.l.o.g. that $\sigma[\![Cond]\!](start_p)$ holds in M. Then the rule ite^{sos}_T applies, so $\sigma Reach(start_{c_1})$ holds in M.

Using the induction hypothesis, we get $\sigma Reach(start_{c_1}) \to \sigma Reach(end_{c_1})$, so $\sigma Reach(end_{c_1})$ holds in M. By definition, $end_c = end_p$, so $\sigma Reach(end_p)$ holds in M.

– Assume `p` is `while(Cond){c}`. We perform bounded induction over it^p from `0` to $\sigma(lastIt_p)$ with the induction hypothesis $P(it^p) = \mathtt{0} \leq \sigma(it^p) < \sigma(lastIt_p) \to \sigma Reach(tp_p(it^p))$.
The base case holds, since $\sigma Reach(start_p)$ is the same as $\sigma Reach(tp_p(\mathtt{0}))$.
For the inductive case, assume that both $\sigma\sigma' Reach(tp_p(it^p))$ and $\sigma'(it^p) < \sigma(lastIt_p)$ holds for some grounding σ' of it^p with the goal of deriving $\sigma\sigma' Reach(tp_p(\mathtt{s}(it^p)))$. Then rule $while_T^{sos}$ applies, so $\sigma\sigma' Reach(start_c)$ holds in M. From the induction hypothesis, we conclude $\sigma\sigma' Reach(start_c) \to \sigma\sigma' Reach(end_c)$, so $\sigma\sigma' Reach(end_c)$ holds. By definition, $end_c = tp_p(\mathtt{s}(it^p))$, so we conclude that $\sigma\sigma' Reach(tp_p(\mathtt{s}(it^p)))$ holds in M.
We have established the base case and the inductive case, so we apply bounded induction to derive that

$$\forall it^p. \big(\sigma(it^p) \leq \sigma(lastIt_p) \to \sigma Reach(tp_p(it^p))\big)$$

holds in M. In particular, $\sigma Reach(lastIt_p)$ holds in M. Since by definition also $\sigma\neg[\![Cond]\!](lastIt_p)$ holds in M, we deduce that $while_F^{sos}$ applies, so $\sigma Reach(end_p)$ holds. □

5 Trace Lemmas

5.1 Trace Lemmas $\mathcal{T}_\mathcal{L}$ for Verification

Trace logic properties support arbitrary quantification over timepoints and describe values of program variables at arbitrary loop iterations and timepoints. We therefore can relate timepoints with values of program variables in trace logic $\mathcal{L}$, allowing us to describe the value distributions of program variables as functions of timepoints throughout program executions. As such, trace logic $\mathcal{L}$ supports

(1) reasoning about the *existence* of a specific loop iteration, allowing us to split the range of loop iterations at a particular timepoint, based on the safety property we want to prove. For example, we can express and derive loop iterations corresponding to timepoints where one program variable takes a specific value for *the first time during loop execution*;
(2) universal quantification over the array content and range of loop iterations bounded by two arbitrary left and right bounds, allowing us to apply instances of the induction scheme (**??**) within a range of loop iterations bounded, for example, by it and $lastIt_s$ for some while-statement `s`.

Addressing these benefits of trace logic, we express generic patterns of inductive program properties as *trace lemmas*. Identifying a suitable set $\mathcal{T}_\mathcal{L}$ of trace lemmas to automate inductive reasoning in trace logic $\mathcal{L}$ is however challenging and domain-specific. We propose three trace lemmas for inductive reasoning over arrays and integers, by considering

(A1) one trace lemma describing how values of program variables change during an interval of loop iterations;

(B1-B2) two trace lemmas to describe the behavior of loop counters.

We prove soundness of our trace lemmas - below we include only one proof and refer to [**?**] for further details.

(A1) Value Evolution Trace Lemma Let `w` be a while-statement, let `v` be a mutable program variable and let $\circ$ be a reflexive and transitive relation - that is $\simeq$ or $\leq$ in the setting of trace logic. The *value evolution trace lemma of* `w`*,* `v`*, and* $\circ$ is defined as

$$\begin{aligned}&\forall bl_{\mathbb{N}}, br_{\mathbb{N}}.\\ &\Big(\forall it_{\mathbb{N}}.\Big((bl \leq it < br \land v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it)))\\ &\qquad\qquad \rightarrow v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(\mathtt{s}(it)))\Big)\\ &\quad \rightarrow \big(bl \leq br \rightarrow v(tp_{\mathtt{w}}(br)) \circ v(tp_{\mathtt{w}}(br))\big)\Big)\end{aligned} \tag{A1}$$

In our work, the value evolution trace lemma is mainly instantiated with the equality predicate $\simeq$ to conclude that the value of a variable does not change during a range of loop iterations, provided that the variable value does not change at any of the considered loop iterations.

Example 4. For Figure 1, the value evaluation trace lemma (A1) yields the property

$$\begin{aligned}&\forall j_{\mathbb{I}}.\ \forall bl_{\mathbb{N}}.\ \forall br_{\mathbb{N}}.\\ &\Big(\forall it_{\mathbb{N}}.\Big((bl \leq it < br\ \land\ b(l_8(bl), j) = b(l_8(it), j))\\ &\qquad\qquad \rightarrow b(l_8(bl), j) = b(l_8(s(it)), j)\Big)\\ &\quad \rightarrow \big(bl \leq br \rightarrow b(l_8(bl), j) = b(l_8(br), j)\big)\Big),\end{aligned}$$

which allows to prove that the value of `b` at some position `j` remains the same from the timepoint *it* the value was first set until the end of program execution. That is, we derive $b(l_9(end), j(l_9(it))) = a(i(l_8(it)))$. □

We next prove soundness of our trace lemma (A1).

Proof *(Soundness Proof of Value Evolution Trace Lemma* (A1)*)* Let *bl* and *br* be arbitrary but fixed and assume that the premise of the outermost implication of (A1) holds. That is,

$$\begin{aligned}\forall it_{\mathbb{N}}.\big((bl \leq it < br \land v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it)))\\ \rightarrow v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(\mathtt{s}(it)))\big)\end{aligned} \tag{10}$$

We use the induction axiom scheme (**??**) and consider its instance with $P(it) := v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it))$, yielding the following instance of (**??**):

$$\Big(v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it)) \quad \wedge \tag{11a}$$

$$\forall it_{\mathbb{N}}.\big((bl \leq it < br \wedge v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it))) \tag{11b}$$

$$\to v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(\mathtt{s}(it))))\Big)$$

$$\to \forall it_{\mathbb{N}}.\Big(bl \leq it \leq br \to v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it))\Big) \tag{11c}$$

Note that the base case property (11a) holds since $\circ$ is reflexive. Further, the inductive case (11b) holds also since it is implied by (10). We thus derive property (11c), and in particular $bl \leq br \leq br \to v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(br))$. Since $\leq$ is reflexive, we conclude $bl \leq br \to v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(br))$, proving thus our trace lemma (A1). $\square$

(B1) Intermediate Value Trace Lemma Let w be a while-statement and let v be a mutable program variable. We call v to be *dense* if the following holds:

$$Dense_{w,v} := \forall it_{\mathbb{N}}.\Big(it < lastIt_{\mathtt{w}} \to \big(v(tp_{\mathtt{w}}(\mathtt{s}(it))) = v(tp_{\mathtt{w}}(it)) \vee v(tp_{\mathtt{w}}(\mathtt{s}(it))) = v(tp_{\mathtt{w}}(it)) + 1\big)\Big)$$

The *intermediate value trace lemma of* w *and* v is defined as

$$\forall x_{\mathbb{I}}.\Big(\big(Dense_{w,v} \wedge v(tp_{\mathtt{w}}(\mathtt{0})) \leq x < v(tp_{\mathtt{w}}(lastIt_{\mathtt{w}}))\big) \to \exists it_{\mathbb{N}}.\big(it < lastIt_{\mathtt{w}} \wedge\ v(tp_{\mathtt{w}}(it)) \simeq x \wedge v(tp_{\mathtt{w}}(\mathtt{s}(it))) \simeq v(tp_{\mathtt{w}}(it)) + 1\big)\Big) \tag{B1}$$

The intermediate value trace lemma (B1) allows us conclude that if the variable v is dense, and if the value x is between the value of v at the beginning of the loop and the value of v at the end of the loop, then there is an iteration in the loop, where v has exactly the value x and is incremented. This trace lemma is mostly used to find specific iterations corresponding to positions x in an array.

Example 5. In Figure 1, using trace lemma (B1) we synthesize the iteration it such that $b(l_9(it), j(l_9(it))) = a(i(l_8(it)))$. $\square$

(B2) Iteration Injectivity Trace Lemma Let w be a while-statement and let v be a mutable program variable. The *iteration injectivity trace lemma of* w *and* v is

$$\forall it^1_{\mathbb{N}}, it^2_{\mathbb{N}}.\Big(\big(Dense_{w,v} \wedge v(tp_{\mathtt{w}}(\mathtt{s}(it^1))) = v(tp_{\mathtt{w}}(it^1)) + 1 \wedge it^1 < it^2 \leq lastIt_{\mathtt{w}}\big) \to v(tp_{\mathtt{w}}(it^1)) \not\simeq v(tp_{\mathtt{w}}(it^2))\Big) \tag{B2}$$

The trace lemma (B2) states that a strongly-dense variable visits each array-position at most once. As a consequence, if each array position is visited only once in a loop, we know that its value has not changed after the first visit, and in particular the value at the end of the loop is the value after the first visit.

Example 6. Trace lemma (B2) is necessary in Figure 1 to apply the value evolution trace lemma (A1) for `b`, as we need to make sure we will never reach the same position of `j` twice. □

Based on the soundness of our trace lemmas, we conclude the next result.

Theorem 3 (Trace Lemmas and Induction). *Let $\mathtt{p}$ be a program. Let L be a trace lemma for some while-statement $\mathtt{w}$ of $\mathtt{p}$ and some variable $\mathtt{v}$ of $\mathtt{p}$. Then L is a consequence of the bounded induction scheme* (**??**) *and of the axiomatic semantics of $[\![\mathtt{p}]\!]$ in trace logic $\mathcal{L}$.* □

5.2 Correctness of trace lemmas

We already proved soundness of trace lemma (A1) in Section **??**. In this section, we prove the remaining two trace lemmas (B1-B2).

Proof (Soundness of Intermediate Value Trace Lemma (B1)). We prove the following equivalent formula obtained from the intermediate value trace lemma (B1) by modus tollens.

$$\begin{aligned}\forall x_{\mathbb{I}}.\Bigg(\bigg(&Dense_{w,v} \wedge v(tp_{\mathtt{w}}(\mathtt{0})) \leq x \wedge \\ &\forall it_{\mathbb{N}}.\big((it < lastIt_{\mathtt{w}} \wedge v(tp_{\mathtt{w}}(\mathtt{s}(it))) \simeq v(tp_{\mathtt{w}}(it)) + 1) \\ &\rightarrow v(tp_{\mathtt{w}}(it)) \not\simeq x\big)\bigg) \\ &\ \rightarrow v(tp_{\mathtt{w}}(lastIt_{\mathtt{w}})) \leq x\Bigg)\end{aligned} \tag{12}$$

The proof proceeds by deriving the conclusion of formula (12) from the premises of formula (12).
Consider the instance of the induction axiom scheme with

$$\text{Base case: } v(tp_{\mathtt{w}}(\mathtt{0})) \leq x \tag{13a}$$

$$\begin{aligned}\text{Inductive case: } \forall it_{\mathbb{N}}.\Big(&\big(\mathtt{0} \leq it < lastIt_{\mathtt{w}} \wedge v(tp_{\mathtt{w}}(it)) \leq x\big) \\ &\rightarrow v(tp_{\mathtt{w}}(\mathtt{s}(it))) \leq x\Big)\end{aligned} \tag{13b}$$

$$\text{Conclusion: } \forall it_{\mathbb{N}}.\Big(\mathtt{0} \leq it \leq lastIt_{\mathtt{w}} \rightarrow v(tp_{\mathtt{w}}(it)) \leq x\Big), \tag{13c}$$

obtained from the bounded induction axiom scheme (**??**) with $P(it) := v(tp_{\mathtt{w}}(it)) \leq x$.
The base case (13a) holds, since it occurs as second premise of formula (12). For the inductive case (13b), assume $\mathtt{0} \leq it < lastIt_{\mathtt{w}}$ and $v(tp_{\mathtt{w}}(it)) \leq x$. By density of v, we obtain two cases:

– Assume $v(tp_{\mathtt{w}}(\mathtt{s}(it))) = v(tp_{\mathtt{w}}(it))$. Since we also assume $v(tp_{\mathtt{w}}(it)) \leq x$, we immediately get $v(tp_{\mathtt{w}}(\mathtt{s}(it))) \leq x$.
– Assume $v(tp_{\mathtt{w}}(\mathtt{s}(it))) = v(tp_{\mathtt{w}}(it)) + 1$. From the assumption $it < lastIt_{\mathtt{w}}$ and the third premise of formula 12, we get $v(tp_{\mathtt{w}}(it)) \not\simeq x$, which combined with $v(tp_{\mathtt{w}}(it)) \leq x$ and the totality-axiom of $<$ for integers gives $v(tp_{\mathtt{w}}(it)) < x$. Finally we combine this fact with $v(tp_{\mathtt{w}}(\mathtt{s}(it))) = v(tp_{\mathtt{w}}(it)) + 1$ and the integer-theory-lemma $x < y \rightarrow x + 1 \leq y$ to derive $v(tp_{\mathtt{w}}(\mathtt{s}(it))) \leq x$.

Hence, we conclude that the inductive case (13b) holds. Thus, the conclusion (13c) also holds. Since the theory axiom $\forall it_{\mathbb{N}}.\ \mathtt{0} \leq it$ holds, formula (13c) implies the conclusion of formula (12), which concludes the proof. □

Proof (Soundness of Iteration Injectivity Trace Lemma (B2)). For arbitrary but fixed iterations it^1 and it^2, assume that the premises of the lemma hold. Now consider the instance of the induction axiom scheme with

$$\text{Base case: } v(tp_{\mathtt{w}}(it^1)) < v(tp_{\mathtt{w}}(\mathtt{s}(it^1))) \tag{14a}$$

$$\begin{aligned}\text{Inductive case: } \forall it_{\mathbb{N}}.\Big(\big(\mathtt{s}(it^1) \leq it < lastIt_{\mathtt{w}} \\ \wedge\, v(tp_{\mathtt{w}}(it^1)) < v(tp_{\mathtt{w}}(it))\big) \\ \rightarrow v(tp_{\mathtt{w}}(it^1)) < v(tp_{\mathtt{w}}(\mathtt{s}(it)))\Big)\end{aligned} \tag{14b}$$

$$\begin{aligned}\text{Conclusion: } \forall it_{\mathbb{N}}.\Big(\mathtt{s}(it^1) \leq it \leq lastIt_{\mathtt{w}} \rightarrow \\ v(tp_{\mathtt{w}}(it^1)) < v(tp_{\mathtt{w}}(it))\Big),\end{aligned} \tag{14c}$$

obtained from the bounded induction axiom scheme (**??**) with $P(it) := v(tp_{\mathtt{w}}(it^1)) < v(tp_{\mathtt{w}}(it))$, by instantiating *bl* and *br* to $\mathtt{s}(it^1)$, respectively $lastIt_{\mathtt{w}}$.
The base case (14a) holds since by integer theory we have $\forall x_{\mathbb{I}}.\ x < x + 1$ and by assumption $v(tp_{\mathtt{w}}(\mathtt{s}(it^1))) = v(tp_{\mathtt{w}}(it^1)) + 1$ holds.
For the inductive case, we assume for arbitrary but fixed it that $v(tp_{\mathtt{w}}(it^1)) < v(tp_{\mathtt{w}}(it))$ holds. Combined with $Dense_{w,v}$ and $\forall x_{\mathbb{I}}.(x < y \rightarrow x < y{+}1)$ this yields $v(tp_{\mathtt{w}}(it^1)) < v(tp_{\mathtt{w}}(\mathtt{s}(it)))$, so (14b) holds. Since both premises (14a) and (14b) hold, also the conclusion (14c) holds. Next, $it^1 < it^2$ implies $\mathtt{s}(it^1) \leq it^2$ (using the monotonicity of $\mathtt{s}$). We therefore have $\mathtt{s}(it^1) \leq it^2 < lastIt_{\mathtt{w}}$, so we are able to instantiate the conclusion(14c) to obtain $v(tp_{\mathtt{w}}(it^1)) < v(tp_{\mathtt{w}}(it^2))$. Finally, we use the arithmetic property $\forall x_{\mathbb{I}}, y_{\mathbb{I}}.(x < y \rightarrow x \not\simeq y)$ to conclude $v(tp_{\mathtt{w}}(it^1)) \not\simeq v(tp_{\mathtt{w}}(it^2))$. □