Automatic Construction of Hoare Proofs from Abstract Interpretation Results

Sunae Seo[1], Hongseok Yang[2], and Kwangkeun Yi[3]

[1] Department of Computer Science,
Korea Advanced Institute of Science and Technology
saseo@ropas.kaist.ac.kr
[2] MICROS Research Center,
Korea Advanced Institute of Science and Technology
hyang@kaist.ac.kr
[3] School of Computer Science and Engineering,
Seoul National University
kwang@ropas.snu.ac.kr

Abstract. By combining program logic and static analysis, we present an automatic approach to construct program proofs to be used in Proof-Carrying Code. We use Hoare logic in representing the proofs of program properties, and the abstract interpretation in computing the program properties. This combination automatizes proof construction; an abstract interpretation automatically estimates program properties (approximate invariants) of our interest, and our proof-construction method constructs a Hoare-proof for those approximate invariants. The proof-checking side (code consumer's side) is insensitive to a specific static analysis; the assertions in the Hoare proofs are always first-order logic formulas for integers, into which we first compile the abstract interpreters' results. Both the property-compilation and the proof construction refer to the standard safety conditions that are prescribed in the abstract interpretation framework. We demonstrate this approach for a simple imperative language with an example property being the integer ranges of program variables. We prove the correctness of our approach, and analyze the size complexity of the generated proofs.

1 Introduction

Necula and Lee's seminal work [11, 12] on Proof-Carrying Code(PCC) and its subsequent developments [14, 13, 1, 7] have been a convincing technology for certifying the safety of mobile code, yet how to achieve the code's safety proofs is still open for alternatives. The existing proof construction process either assumes that the programmer provides the program invariants [11–13], thus being not fully automatic, or is limited to a class of properties that are automatically inferable by the current type system technologies [7, 2, 10].

In this paper we present a method for automatically constructing the program proofs, to be used in the PCC framework. We use a combination of static analysis and program logic. We use the abstract interpretation [4, 3] for the static analysis and Hoare logic [9] for the program logic. An abstract interpreter first estimates program invariants. For the computed invariants, we construct Hoare proofs using the standard Hoare logic rules. For example, suppose that the program property that we have to establish is the range of integer values of program variables. We employ an abstract interpreter that estimates the range by an integer interval. The estimated integer-interval for every variable at each program point is an invariant for which we will construct Hoare proofs. Since the invariants from abstract interpretations are approximate in general, they sometimes do not exactly fit with the Hoare logic rules. This gap is filled by the safety proofs of the used abstract interpreter. These safety proofs are for the standard safety conditions prescribed in the abstract interpretation framework.

In order to make the proof-checking side (code consumer's side) insensitive to a specific static analysis, we fix the assertion language in Hoare logic to first-order logic for integers,[4] into which we have to translate abstract interpretation results. This translation procedure is nicely defined by referencing the concretization formulas of the used abstract interpreter.

Note that our method still requires the code producer to design an abstract interpreter that estimates the desired properties (program invariants) in a right cost-accuracy balance. Although designing such an abstract interpreter is generally demanding, our method is still appealing because the once-designed abstract interpreter can be used repeatedly for all programs of the same language, as long as their properties to verify and check remain the same.

The code consumer's side remains simple. Checking the Hoare proofs is simply by pattern-matching the proof tree nodes against the corresponding Hoare logic rules. Checking if the proofs are about the accompanied code is straightforward, because the program texts are embedded in the Hoare proofs.

Because the trusted computing base(TCB) is the standard Hoare logic rules with first-order logic for integers, the TCB size amounts to the number of proof rules in Hoare logic and first-order logic for integers. The number of Hoare logic rule is linear to the number of syntactic constructs of the source programming language. The size of first-order logic rules for integers can vary depending on where we strike the balance between the number of rules and proof size. We can reduce this part of the TCB by using the foundational PCC [1, 2, 7] approach.

Our work is based on Cousot and Cousot's insight for the connection between program logic and static analysis [5, 6]. They showed that the set of assertions can be considered as an abstract domain; thus, an abstract interpretation can be used to find assertions that denote approximate invariants, and these assertions can be used to verify a program. Recently, Heintze et al. [8] further developed Cousot and Cousot's insight so that both program logic and static analysis can get benefits from each other. Our work strenghens this connection bewteen program logic and static analysis. We use a static analysis not just to find approximate invariant assertions but also to obtain machine-checkable proofs, which show that those assertions indeed approximate program invariants.

[4] Our method is not necessarily limited to first-order logic. It only requires that the assertion language have first-order quantifiers.

In this paper we demonstrate our method for a simple imperative language with integer variables. In Section 2, we explain the generic abstract interpreter, which can be instantiated to an analysis for a specific program property. In Section 3, we present an algorithm that gets a program annotated with the abstract interpretation results, and gives a Hoare proof for the program. In Section 4, we conclude.

2 Generic Abstract Interpretation

We consider abstract interpretations that are instances of Cousot's generic abstract interpretation [3]. The generic abstract interpretation (Figure 2) is about a simple imperative programming language (Figure 1).

Commands	$C ::= x := E \mid C; C \mid \texttt{if } B \texttt{ then } C \texttt{ else } C \texttt{ fi} \mid \texttt{while } B \texttt{ do } C \texttt{ od}$
Expressions	$E ::= n \mid x \mid E + E$
Boolean Expressions	$B ::= \texttt{tt} \mid \texttt{ff} \mid E = E \mid E < E \mid B \wedge B \mid B \vee B$

Fig. 1. Syntax of a Simple Imperative Language

The abstract interpretation is generic because it is parameterized by an abstract domain $\mathcal{A}$ with a lattice structure $(\sqsubseteq, \bot, \sqcup, \sqcap, \top)$, an abstraction function $\alpha: \mathcal{P}(\mathsf{Ints}) \to \mathcal{A}$, a concretization function $\gamma: \mathcal{A} \to \mathcal{P}(\mathsf{Ints})$, and the following abstract operators:

$$\begin{aligned} \hat{+} &: \mathcal{A} \times \mathcal{A} \to \mathcal{A} \\ \hat{+}^{\triangleleft} &: (\mathcal{A} \times \mathcal{A} \times \mathcal{A}) \to \mathcal{A} \times \mathcal{A} \\ \hat{=}^{\triangleleft} &: \mathcal{A} \times \mathcal{A} \to \mathcal{A} \times \mathcal{A} \\ \hat{<}^{\triangleleft} &: \mathcal{A} \times \mathcal{A} \to \mathcal{A} \times \mathcal{A} \end{aligned}$$

The operator $\hat{+}$ abstracts the addition of integers. The other operators make an abstract interpretation more precise by employing the notion of "backward abstract interpretation" [6, 3]. When $\hat{+}^{\triangleleft}(a, b, c)$ gives the pair (a', b'), the set $\gamma(a')$ of integers contains integers in $\gamma(a)$ excluding some integers n such that "$n+b \neq c$." Similarly, the other operators $\hat{=}^{\triangleleft}$ and $\hat{<}^{\triangleleft}$ exclude integers based on $=$ or $<$: when $\hat{=}^{\triangleleft}(a, b)$ is (a', b'), the set $\gamma(a')$ contains integers in $\gamma(a)$ excluding some integers n that are not equal to any n' in $\gamma(b)$; and when $\hat{<}^{\triangleleft}(a, b) = (a', b')$, the set $\gamma(a')$ excludes some integers n such that "$n \not< b$." For these operators $\hat{+}^{\triangleleft}, \hat{=}^{\triangleleft}, \hat{<}^{\triangleleft}$, we use subscripts $-_0$ and $-_1$ to denote the first and second of the result, respectively.

Given a program, the abstract interpreter associates an *abstract* state with each program point. An abstract state $\hat{s}$ is a map from a finite set Vars of variables to $\mathcal{A}$, and means a set of concrete states, denoted $\gamma(\hat{s})$:

$$s \in \gamma(\hat{s}) \iff \forall x \in \mathsf{Vars}: \ s(x) \in \gamma(\hat{s}(x))$$

When the abstract interpreter associates $\hat{s}$ with a program point, $\gamma(\hat{s})$ contains all the states that are possible at the point during execution.

The interpretation of commands and expressions is standard except the cases of a conditional statement and a loop. In those cases, we use backward semantics $[\![-]\!]_b$ for accurate analysis of branches. Let $\hat{s}$ be an abstract state, and let a be an abstract value. The backward semantics $[\![B]\!]_b\hat{s}$ for boolean expression B makes $\hat{s}$ smaller by excluding some concrete states in $\gamma(\hat{s})$ that violate the condition B. The abstract state $[\![E]\!]_b\hat{s}\,a$ excludes some concrete states s in $\gamma(\hat{s})$ where the concrete value of E is not approximated by a.

In the interpreter definition, we use macro $\neg B$ which expands to a boolean expression without negation. We move $\neg$ inside $\vee$ or $\wedge$ by de-Morgan's laws[5], and then transform the negation of atomic boolean expressions by the usual equivalence: $\neg(E{=}E') \iff E{<}E' \vee E{>}E'$ and $\neg(E{<}E') \iff E{=}E' \vee E{>}E'$.

The generic abstract interpretation in Figure 2 can be instantiated to various program analyses. For instance, when we want to design an analysis (called interval analysis) that estimates program variables' values by integer intervals, we can use, for $\mathcal{A}$, an interval domain

$$\{\bot\} \ \cup \ \{[n,m] \mid \big(n,m \in \mathsf{Ints} \cup \{-\infty,\infty\}\big) \ \wedge \ n \le m\},$$

and the abstract operators defined as in Figure 3.

The instantiated abstract interpretation is sound when the abstract domain and operators are chosen appropriately. The abstract domain $\mathcal{A}$ should be a complete lattice, Galois-connected by abstraction $\alpha{:}\,\mathcal{P}(\mathsf{Ints}) \to \mathcal{A}$ and concretization $\gamma{:}\,\mathcal{A} \to \mathcal{P}(\mathsf{Ints})$. The abstract operators should satisfy the requirements in Figure 4. The Galois connection means that the order in abstract domain $\mathcal{A}$ corresponds to the approximation order among the concrete correspondents. The abstract operators' safety arguments dictate that their abstract results must subsume their concrete correspondents.

3 Construction of Hoare Proofs

The main result of this paper is an algorithm that constructs proofs in Hoare logic from abstract interpretation results. In this section, we will explain this construction algorithm.

3.1 Translation Function

Our algorithm is parameterized by a *translation* function, which compiles abstract interpretation results into formulas in first-order logic. Let tr be a translation function that takes a pair of an abstract value a and an expression E, and gives a first-order logic formula φ about E. Formula $\mathsf{tr}(a,E)$ intuitively means "$E \in \gamma(a)$": $\mathsf{tr}(a,E)$ holds in a concrete state s precisely when the value of E at s belongs to $\gamma(a)$. The formal definition of a translation requires that the function tr satisfy the following conditions:

[5] $\neg(B \wedge B') \Leftrightarrow \neg B \vee \neg B'$ and $\neg(B \vee B') \Leftrightarrow \neg B \wedge \neg B'$

$$\hat{s} \in \mathsf{AbsStates} \stackrel{\Delta}{=} \mathsf{Vars} \to \mathcal{A}$$

Commands

$$\begin{aligned}
[\![C]\!] &: \mathsf{AbsStates} \to \mathsf{AbsStates}\\
[\![x{:=}E]\!]\hat{s} &= \hat{s}[x \mapsto ([\![E]\!]\,\hat{s})]\\
[\![\texttt{if}\ B\ \texttt{then}\ C_0\ \texttt{else}\ C_1\ \texttt{fi}]\!]\hat{s} &= [\![C_0]\!]([\![B]\!]_b\,\hat{s}) \sqcup [\![C_1]\!]([\![\neg B]\!]_b\,\hat{s})\\
[\![\texttt{while}\ B\ \texttt{do}\ C\ \texttt{od}]\!]\hat{s} &= [\![\neg B]\!]_b\big(\mathit{lfp}\ \lambda\hat{s}'.\ \hat{s} \sqcup [\![C]\!]([\![B]\!]_b\,\hat{s}')\big)\\
[\![C_0;C_1]\!]\hat{s} &= [\![C_1]\!]([\![C_0]\!]\hat{s})
\end{aligned}$$

Integer Expressions

$$\begin{aligned}
[\![E]\!] &: \mathsf{AbsStates} \to \mathcal{A}\\
[\![n]\!]\hat{s} &= \alpha(\{n\})\\
[\![x]\!]\hat{s} &= \hat{s}(x)\\
[\![E_0{+}E_1]\!]\hat{s} &= [\![E_0]\!]\hat{s} \mathbin{\hat{+}} [\![E_1]\!]\hat{s}
\end{aligned}$$

Backward Abstract Semantics of Boolean Expressions

$$\begin{aligned}
[\![B]\!]_b &: \mathsf{AbsStates} \to \mathsf{AbsStates}\\
[\![\texttt{tt}]\!]_b\hat{s} &= \hat{s}\\
[\![\texttt{ff}]\!]_b\hat{s} &= \lambda x \in \mathsf{Vars}.\ \bot\\
[\![E_0{=}E_1]\!]_b\hat{s} &= \big([\![E_0]\!]_b\,\hat{s}\,(\hat{=}_0^{\triangleleft}([\![E_0]\!]\hat{s},[\![E_1]\!]\hat{s}))\big) \sqcap \big([\![E_1]\!]_b\,\hat{s}\,(\hat{=}_1^{\triangleleft}([\![E_0]\!]\hat{s},[\![E_1]\!]\hat{s}))\big)\\
[\![E_0{<}E_1]\!]_b\hat{s} &= \big([\![E_0]\!]_b\,\hat{s}\,(\hat{<}_0^{\triangleleft}([\![E_0]\!]\hat{s},[\![E_1]\!]\hat{s}))\big) \sqcap \big([\![E_1]\!]_b\,\hat{s}\,(\hat{<}_1^{\triangleleft}([\![E_0]\!]\hat{s},[\![E_1]\!]\hat{s}))\big)\\
[\![B_0{\wedge}B_1]\!]_b\hat{s} &= [\![B_0]\!]_b\,\hat{s} \sqcap [\![B_1]\!]_b\,\hat{s}\\
[\![B_0{\vee}B_1]\!]_b\hat{s} &= [\![B_0]\!]_b\,\hat{s} \sqcup [\![B_1]\!]_b\,\hat{s}
\end{aligned}$$

Backward Abstract Semantics of Integer Expressions

$$\begin{aligned}
[\![E]\!]_b &: \mathsf{AbsStates} \to \mathcal{A} \to \mathsf{AbsStates}\\
[\![n]\!]_b\,\hat{s}\,a &= \begin{cases} \hat{s} & \text{if } \alpha(\{n\}) \sqsubseteq a\\ \lambda x \in \mathsf{Vars}.\ \bot & \text{otherwise}\end{cases}\\
[\![x]\!]_b\,\hat{s}\,a &= \hat{s}[x \mapsto (\hat{s}(x) \sqcap a)]\\
[\![E_0{+}E_1]\!]_b\,\hat{s}\,a &= \big([\![E_0]\!]_b\,\hat{s}\,(\hat{+}_0^{\triangleleft}([\![E_0]\!]\hat{s},[\![E_1]\!]\hat{s},a))\big) \sqcap \big([\![E_1]\!]_b\,\hat{s}\,(\hat{+}_1^{\triangleleft}([\![E_0]\!]\hat{s},[\![E_1]\!]\hat{s},a))\big)
\end{aligned}$$

Fig. 2. Generic Abstract Interpretation

- Monotonicity: for all a, a', and E, if $a \sqsubseteq a'$, then $\mathsf{tr}(a, E)$ implies $\mathsf{tr}(a', E)$;
- Meet Preservation: for all a, a', and E, the formula $\mathsf{tr}(a \sqcap a', E)$ is equivalent to $\mathsf{tr}(a, E) \wedge \mathsf{tr}(a', E)$;
- Strictness: for all E, the formula $\mathsf{tr}(\bot, E)$ is $\texttt{ff}$;
- Constants Preservation: for all integers n, the formula $\mathsf{tr}(\alpha(\{n\}), n)$ holds;
- Closedness: $\mathsf{Free}(\mathsf{tr}(a, E)) = \mathsf{Free}(E)$ for all a and E; and
- Commutativity: $\mathsf{tr}(a, E)[E'/x] = \mathsf{tr}(a, E[E'/x])$ for all a, E, E', and x.

The first four conditions are from the corresponding properties of the concretization γ: the concretization γ is monotone, preserves $\sqcap$ and $\bot$, and maps $\alpha(\{n\})$

$$
\begin{aligned}
[n_0, m_0] \hat{+} [n_1, m_1] &\stackrel{\Delta}{=} [n_0 + n_1, m_0 + m_1] \\
\hat{+}_i^{\triangleleft}([n_0, m_0], [n_1, m_1], [n_2, m_2]) &\stackrel{\Delta}{=} [n_i, m_i] \sqcap [n_2 - m_{1-i}, m_2 - n_{1-i}] \\
\hat{=}_i^{\triangleleft}([n_0, m_0], [n_1, m_1]) &\stackrel{\Delta}{=} [n_0, m_0] \sqcap [n_1, m_1] \\
\hat{<}^{\triangleleft}([n_0, m_0], [n_1, m_1]) &\stackrel{\Delta}{=} \begin{cases} (\perp, \perp) & \text{if } m_1 \leq n_0 \\ ([n_0, \mathsf{min} X_0], [\mathsf{max} X_1, m_1]) & \text{otherwise} \end{cases} \\
&\ (\text{where } X_0 = \{m_0, m_1 - 1\},\ X_1 = \{n_0 + 1, n_1\})
\end{aligned}
$$

Fig. 3. Abstract Operators for an Interval Analysis

$$
\begin{aligned}
a_0 \hat{+} a_1 &\sqsupseteq \alpha(\{n_0 + n_1 \mid n_i \in \gamma(a_i)\}) \\
\hat{+}_i^{\triangleleft}(a_0, a_1, a_2) &\sqsupseteq \alpha(\{n_i \in \gamma(a_i) \mid \exists n_{i-1} \in \gamma(a_{i-1}),\ n_2 \in \gamma(a_2) :\ n_0 + n_1 = n_2\}) \\
\hat{=}_i^{\triangleleft}(a_0, a_1) &\sqsupseteq \alpha(\{n_i \in \gamma(a_i) \mid \exists n_{1-i} \in \gamma(a_{1-i}) :\ n_i = n_{i-1}\}) \\
\hat{<}_i^{\triangleleft}(a_0, a_1) &\sqsupseteq \alpha(\{n_i \in \gamma(a_i) \mid \exists n_{1-i} \in \gamma(a_{1-i}) :\ n_0 < n_1\})
\end{aligned}
$$

Fig. 4. Safety of Abstract Operators

to a set containing n. The other two conditions say that $\mathsf{tr}(a, E)$ expresses $\gamma(a)$ by a formula with holes, and then fills the hole with E.

For the interval analysis, we can use the translation tr defined as follows:

$$
\mathsf{tr}([n, m], E) \stackrel{\Delta}{=} (n \leq E) \wedge (E \leq m) \qquad \mathsf{tr}(\perp, E) \stackrel{\Delta}{=} \mathtt{ff}
$$

Note that tr for the interval analysis is monotone and preserves $\sqcap$ and $\perp$, and that $\mathsf{tr}([n, m], -)$ is a formula $(n < -) \wedge (- < m)$ with two holes, which are filled by the second argument; thus, tr satisfies the other two conditions for a translation.

Map trst is a natural extension of tr for abstract states:

$$
\mathsf{trst}(\hat{s}) \stackrel{\Delta}{=} \bigwedge_{x \in \mathsf{Vars}} \mathsf{tr}(\hat{s}(x), x)
$$

Formula $\mathsf{trst}(\hat{s})$ means the concretization of $\hat{s}$: $\mathsf{trst}(\hat{s})$ holds for a concrete state s precisely when s is in $\gamma(\hat{s})$. For instance, the abstract state $[x \mapsto [2, 3], y \mapsto [1, 5]]$ from the interval analysis gets translated into a formula as follows:

$$
\begin{aligned}
\mathsf{trst}([x \mapsto [2, 3], y \mapsto [1, 5]]) &= \mathsf{tr}([2, 3], x) \ \wedge\ \mathsf{tr}([1, 5], y) \\
&= 2 \leq x \ \wedge\ x \leq 3 \ \wedge\ 1 \leq y \ \wedge\ y \leq 5
\end{aligned}
$$

3.2 Algorithm

Our algorithm is parameterized by abstract domain and operators, and their soundness proofs. Suppose that we have obtained a program analysis by instantiating the generic abstract interpretation with an abstract domain $\mathcal{A}$ with a lattice structure $(\sqsubseteq, \bot, \sqcup, \sqcap, \top)$, and abstract operators $\hat{+}, \hat{+}^{\triangleleft}, \hat{=}^{\triangleleft}, \hat{<}^{\triangleleft}$. We can specialize our algorithm for this analysis by providing a translation function tr for the domain $\mathcal{A}$, and procedures that prove in first-order logic the soundness of tr and the abstract operators. These procedures must satisfy the specifications in Figure 5. Note that the procedures monTr, meetTr, conTr imply that tr is monotone, preserves $\sqcap$ and maps each $\alpha(\{n\})$ to a set containing n; and that the remaining procedures fAdd, bAdd_i, bEql_i, bInEql_i manifest that the abstract operators $\hat{+}, \hat{+}^{\triangleleft}, \hat{=}^{\triangleleft}, \hat{<}^{\triangleleft}$ satisfy the requirements in Figure 4.

For all expressions E, E_0, E_1, abstract values a, a_0, a_1, b, and integers n

- $\mathsf{monTr}(a, b, E)$ for $a \sqsubseteq b$ is a proof tree for
$$\mathsf{tr}(a, E) \Rightarrow \mathsf{tr}(b, E);$$
- $\mathsf{meetTr}(a, b, E)$ is a proof tree for
$$\big(\mathsf{tr}(a, E) \wedge \mathsf{tr}(b, E)\big) \Rightarrow \mathsf{tr}(a \sqcap b, E);$$
- $\mathsf{conTr}(n)$ is a proof tree for
$$\mathsf{tr}(\alpha(\{n\}), n)$$
- $\mathsf{fAdd}(a_0, a_1, E_0, E_1)$ is a proof tree for
$$\big(\mathsf{tr}(a_0, E_0) \wedge \mathsf{tr}(a_1, E_1)\big) \;\Rightarrow\; \mathsf{tr}(a_0 \hat{+} a_1, E_0 + E_1);$$
- $\mathsf{bEql}_i(a_0, a_1, E)$ is a proof tree for
$$\Big(\exists x.\, \mathsf{tr}(a_i, E) \wedge \mathsf{tr}(a_{1-i}, x) \wedge E = x\Big) \;\Rightarrow\; \mathsf{tr}(\hat{=}_i^{\triangleleft}(a_0, a_1), E);$$
- $\mathsf{bInEql}_0(a_0, a_1, E)$ is a proof tree for
$$\Big(\exists x.\, \mathsf{tr}(a_0, E) \wedge \mathsf{tr}(a_1, x) \wedge E_0 < x\Big) \;\Rightarrow\; \mathsf{tr}(\hat{<}_0^{\triangleleft}(a_0, a_1), E);$$
- $\mathsf{bInEql}_1(a_0, a_1, E)$ is a proof tree for
$$\Big(\exists x.\, \mathsf{tr}(a_0, x) \wedge \mathsf{tr}(a_1, E) \wedge x < E_1\Big) \;\Rightarrow\; \mathsf{tr}(\hat{<}_1^{\triangleleft}(a_0, a_1), E);$$
- $\mathsf{bAdd}_i(a_0, a_1, b, E)$ is a proof tree for
$$\Big(\exists x.\, \mathsf{tr}(a_i, E) \wedge \mathsf{tr}(a_{1-i}, x) \wedge \mathsf{tr}(b, E + x)\Big) \;\Rightarrow\; \mathsf{tr}(\hat{+}_i^{\triangleleft}(a_0, a_1, b), E).$$

Fig. 5. Safety Proofs Generated by Analysis-Specific Procedures

The input of our algorithm is a command in the simple imperative language (Figure 1) annotated with the results of an abstract interpretation. Let $\mathcal{I}$ be an instance of the generic abstract interpretation, and C a command. The analysis $\mathcal{I}$ annotates each point of C with an abstract state $\hat{s}$, so that the output is a term A in the following grammar:

$$A ::= [\hat{s}]R[\hat{s}]$$
$$R ::= x{:=}E \mid A;A \mid \texttt{if}\ B\ \texttt{then}\ A\ \texttt{else}\ A\ \texttt{fi} \mid [\mathsf{inv}\ \hat{s}]\texttt{while}\ B\ \texttt{do}\ A\ \texttt{od}$$

Note that the annotation specifies the pre- and post-conditions for each subcommand of C, and also provides loop invariants $[\mathsf{inv}\ \hat{s}]$. For each annotated command of the form A or R, we denote the corresponding command without annotation using $\overline{A}$ or $\overline{R}$.

Given an annotated program A, our algorithm gives a proof in Hoare logic. This Hoare proof shows that each annotation $\hat{s}$ in A holds: the formula $\mathsf{trst}(\hat{s})$ holds at the annotated place in the command $\overline{A}$. For example, from an annotated assignment $[\hat{s}]x := y[\hat{s}']$, the algorithm gives the following proof tree:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{tr}(\hat{s}(y), y) \Rightarrow \mathsf{tr}(\hat{s}'(x), y)}}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{trst}(\hat{s}')[y/x]} \quad \dfrac{}{\{\mathsf{trst}(\hat{s}')[y/x]\}\ x := y\ \{\mathsf{trst}(\hat{s}')\}}}{\{\mathsf{trst}(\hat{s})\}\ x := y\ \{\mathsf{trst}(\hat{s}')\}}$$

Note that this tree is derivable because $\hat{s}(y) = \hat{s}'(x) = \hat{s}'(y)$: the generic abstract interpretation requires that $\hat{s}'$ be the abstract state $\hat{s}[x \mapsto \hat{s}(y)]$.

Our algorithm, denoted $\mathcal{T}$, calls three subroutines $\mathcal{E}$, $\mathcal{E}_b$, and $\mathcal{B}_b$. Subroutine $\mathcal{E}$ constructs proofs that show that the forward abstract interpretation of expressions is correct. Given an abstract state $\hat{s}$ and an expression E, subroutine $\mathcal{E}$ produces a proof of the form:

$$\dfrac{\vdots}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{tr}([\![E]\!]\hat{s}, E)}$$

The proved implication says that the abstract value $[\![E]\!]\hat{s}$ "contains" all the possible values of E at some state s in $\gamma(\hat{s})$.

Other subroutines $\mathcal{E}_b$ and $\mathcal{B}_b$ are about backward abstract interpretations. Given an abstract state $\hat{s}$, an expression E and an abstract value a, the subroutine $\mathcal{E}_b$ produces the following proof:

$$\dfrac{\vdots}{\mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(a, E) \Rightarrow \mathsf{trst}([\![E]\!]_b\hat{s}\ a)}$$

This proof shows that the backward interpretation $[\![E]\!]_b$ of expression E is correct; the set $\gamma([\![E]\!]_b\hat{s}\ a)$ of states can exclude a state s in $\gamma(\hat{s})$ only when the value of E at s is not in $\gamma(a)$. Subroutine $\mathcal{B}_b$ outputs, for an abstract state $\hat{s}$ and a boolean expression B, the following proof:

$$\dfrac{\vdots}{\mathsf{trst}(\hat{s}) \wedge B \Rightarrow \mathsf{trst}([\![B]\!]_b\hat{s})}$$

The proved implication says that a state s in $\gamma(\hat{s})$ can be pruned in $\gamma([\![B]\!]_b \hat{s})$ only when the boolean expression B is false at the state s. Thus, it implies that the backward abstract interpretation of boolean expressions is correct.

Case $A \equiv [\hat{s}]x := E[\hat{s}']$

$$\dfrac{\dfrac{\dfrac{}{\mathsf{trst}(\hat{s}) \Rightarrow \left(\wedge_{y \in \mathsf{Vars}-\{x\}} \mathsf{tr}(\hat{s}'(y), y)\right)} \quad \mathcal{E}(\hat{s}, E)}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{trst}(\hat{s}')[E/x]} \quad \dfrac{}{\{\mathsf{trst}(\hat{s}')[E/x]\}\ x := E\ \{\mathsf{trst}(\hat{s}')\}}}{\{\mathsf{trst}(\hat{s})\}\ x := E\ \{\mathsf{trst}(\hat{s}')\}}$$

Case $A \equiv [\hat{s}]\texttt{if}\ B\ \texttt{then}\ [\hat{s}_1]R_1[\hat{s}'_1]\ \texttt{else}\ [\hat{s}_2]R_2[\hat{s}'_2]\ \texttt{fi}[\hat{s}']$

$$\dfrac{\dfrac{\mathcal{B}_b(\hat{s}, B) \quad \mathcal{T}([\hat{s}_1]R_1[\hat{s}'_1]) \quad \mathsf{monSt}(\hat{s}'_1, \hat{s}'_1 \sqcup \hat{s}'_2)}{\{\mathsf{trst}(\hat{s}) \wedge B\}\ \overline{R_1}\ \{\mathsf{trst}(\hat{s}')\}} \quad \dfrac{\mathcal{B}_b(\hat{s}, \neg B) \quad \mathcal{T}([\hat{s}_2]R_2[\hat{s}'_2]) \quad \mathsf{monSt}(\hat{s}'_2, \hat{s}'_1 \sqcup \hat{s}'_2)}{\{\mathsf{trst}(\hat{s}) \wedge \neg B\}\ \overline{R_2}\ \{\mathsf{trst}(\hat{s}')\}}}{\{\mathsf{trst}(\hat{s})\}\ \texttt{if}\ B\ \texttt{then}\ \overline{R_1}\ \texttt{else}\ \overline{R_2}\ \texttt{fi}\ \{\mathsf{trst}(\hat{s}')\}}$$

Case $A \equiv [\hat{s}]([\mathsf{inv}\ \hat{s}_0]\texttt{while}\ B\ \texttt{do}\ [\hat{s}_1]R_1[\hat{s}'_1]\ \texttt{od})[\hat{s}']$

$$\dfrac{\mathsf{monSt}(\hat{s}, \hat{s}_0) \quad \dfrac{\dfrac{\mathcal{B}_b(\hat{s}_0, B) \quad \mathcal{T}([\hat{s}_1]R_1[\hat{s}'_1]) \quad \mathsf{monSt}(\hat{s}'_1, \hat{s}_0)}{\{\mathsf{trst}(\hat{s}_0) \wedge B\}\ \overline{R_1}\ \{\mathsf{trst}(\hat{s}_0)\}}}{\{\mathsf{trst}(\hat{s}_0)\}\ \texttt{while}\ B\ \texttt{do}\ \overline{R_1}\ \texttt{od}\ \{\mathsf{trst}(\hat{s}_0) \wedge \neg B\}} \quad \mathcal{B}_b(\hat{s}_0, \neg B)}{\{\mathsf{trst}(\hat{s})\}\ \texttt{while}\ B\ \texttt{do}\ \overline{R_1}\ \texttt{od}\ \{\mathsf{trst}(\hat{s}')\}}$$

Case $A \equiv [\hat{s}]A_1; A_2[\hat{s}']$

$$\dfrac{\mathcal{T}(A_1) \quad \mathcal{T}(A_2)}{\{\mathsf{trst}(\hat{s})\}\ \overline{A_1}; \overline{A_2}\ \{\mathsf{trst}(\hat{s}')\}}$$

Fig. 6. Proof Construction $\mathcal{T}(A)$ for an Annotated Program A

The main algorithm $\mathcal{T}$ is shown in Figure 6, and three subroutines $\mathcal{E}$, $\mathcal{E}_b$, and $\mathcal{B}_b$ are, respectively, in Figure 7, 8, and 9. In the algorithms, we use macros monSt and meetSt, which, respectively, extend monTr and meetTr (Figure 5) to abstract states. Let $x_0, \ldots, x_n$ be the enumeration of all variables in Vars, and let $\hat{s}$ and $\hat{s}'$ be abstract states. The macro $\mathsf{monSt}(\hat{s}, \hat{s}')$ expands to the following tree:

$$\dfrac{\mathsf{monTr}(\hat{s}(x_0), \hat{s}'(x_0), x_0) \quad \ldots \quad \mathsf{monTr}(\hat{s}(x_n), \hat{s}'(x_n), x_n)}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{trst}(\hat{s}')}$$

Note that this tree becomes a proof tree when $\hat{s} \sqsubseteq \hat{s}'$; it is because, if $\hat{s}(x_i) \sqsubseteq \hat{s}'(x_i)$, $\mathsf{monTr}(\hat{s}(x_i), \hat{s}'(x_i), x_i)$ is a proof tree. On the other hand, the macro $\mathsf{meetSt}(\hat{s}, \hat{s}')$ always expands to the proof tree:

$$\dfrac{\mathsf{meetTr}(\hat{s}(x_0), \hat{s}'(x_0), x_0) \quad \ldots \quad \mathsf{meetTr}(\hat{s}(x_n), \hat{s}'(x_n), x_n)}{\mathsf{trst}(\hat{s}) \wedge \mathsf{trst}(\hat{s}') \Rightarrow \mathsf{trst}(\hat{s} \sqcap \hat{s}')}$$

Case $E \equiv n$

$$\dfrac{\mathsf{conTr}(n)}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{tr}(\llbracket n \rrbracket \hat{s}, n)}$$

Case $E \equiv x$

$$\dfrac{}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{tr}(\llbracket x \rrbracket \hat{s}, x)}$$

Case $E \equiv E_0{+}E_1$

$$\dfrac{\dfrac{\mathcal{E}(\hat{s}, E_0) \quad \mathcal{E}(\hat{s}, E_1)}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{tr}(\llbracket E_0 \rrbracket \hat{s}, E_0) \wedge \mathsf{tr}(\llbracket E_1 \rrbracket \hat{s}, E_1)} \quad \mathsf{fAdd}(\llbracket E_0 \rrbracket \hat{s}, \llbracket E_1 \rrbracket \hat{s}, E_0, E_1)}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{tr}(\llbracket E_0{+}E_1 \rrbracket \hat{s}, E_0 + E_1)}$$

Fig. 7. Proof Construction $\mathcal{E}(\hat{s}, E)$

Case $E \equiv n$

if $(\alpha(\{n\}) \sqsubseteq a)$ then

$$\dfrac{\dfrac{}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{trst}(\llbracket n \rrbracket_b \hat{s}\ a)}}{\mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(a, n) \Rightarrow \mathsf{trst}(\llbracket n \rrbracket_b \hat{s}\ a)}$$

else

$$\dfrac{\dfrac{}{\mathsf{tr}(a, n) \Rightarrow \mathsf{trst}(\llbracket n \rrbracket_b \hat{s}\ a)}}{\mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(a, n) \Rightarrow \mathsf{trst}(\llbracket n \rrbracket_b \hat{s}\ a)}$$

Case $E \equiv x$

$$\dfrac{\mathsf{meetTr}(\hat{s}(x), a, x)}{\mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(a, x) \Rightarrow \mathsf{trst}(\llbracket x \rrbracket_b \hat{s}\ a)}$$

Case $E \equiv E_0{+}E_1$

let $(b_0, b_1) \triangleq \hat{+}^{\triangleleft}(\llbracket E_0 \rrbracket \hat{s}, \llbracket E_1 \rrbracket \hat{s}, a)$ and $(\hat{s}_0, \hat{s}_1) \triangleq (\llbracket E_0 \rrbracket_b \hat{s}\, b_0, \llbracket E_1 \rrbracket_b \hat{s}\, b_1)$

$$\dfrac{\dfrac{\dfrac{\tau_0 \quad \mathcal{E}_b(\hat{s}, b_0, E_0)}{\mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(a, E_0 + E_1) \Rightarrow \mathsf{trst}(\hat{s}_0)} \quad \dfrac{\tau_1 \quad \mathcal{E}_b(\hat{s}, b_1, E_1)}{\mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(a, E_0 + E_1) \Rightarrow \mathsf{trst}(\hat{s}_1)}}{\mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(a, E_0 + E_1) \Rightarrow \mathsf{trst}(\hat{s}_0) \wedge \mathsf{trst}(\hat{s}_1)} \quad \mathsf{meetSt}(\hat{s}_0, \hat{s}_1)}{\mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(a, E_0 + E_1) \Rightarrow \mathsf{trst}(\llbracket E_0{+}E_1 \rrbracket_b \hat{s}\ a)}$$

where τ_i is:

$\psi_i \triangleq \exists x.\, \mathsf{tr}(\llbracket E_i \rrbracket \hat{s}, E_i) \wedge \mathsf{tr}(\llbracket E_{1-i} \rrbracket \hat{s}, x) \wedge \mathsf{tr}(a, E_i + x)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{E}(\hat{s}, E_0) \quad \mathcal{E}(\hat{s}, E_1)}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{tr}(\llbracket E_0 \rrbracket \hat{s}, E_0) \wedge \mathsf{tr}(\llbracket E_1 \rrbracket \hat{s}, E_1)}}{\mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(a, E_0{+}E_1) \Rightarrow \mathsf{tr}(\llbracket E_0 \rrbracket \hat{s}, E_0) \wedge \mathsf{tr}(\llbracket E_1 \rrbracket \hat{s}, E_1) \wedge \mathsf{tr}(a, E_0{+}E_1)}}{\mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(a, E_0{+}E_1) \Rightarrow \psi_i}}{\mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(a, E_0{+}E_1) \Rightarrow \mathsf{trst}(\hat{s}) \wedge \psi_i} \quad \dfrac{\mathsf{bAdd}_i(\llbracket E_0 \rrbracket \hat{s}, \llbracket E_1 \rrbracket \hat{s}, a, E_i)}{\mathsf{trst}(\hat{s}) \wedge \psi_i \Rightarrow \mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(b_i, E_i)}}{\mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(a, E_0{+}E_1) \Rightarrow \mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(b_i, E_i)}$$

Fig. 8. Proof Construction $\mathcal{E}_b(\hat{s}, a, E)$

Case $B \equiv \mathtt{tt}$

$$\dfrac{\dfrac{}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{trst}([\![\mathtt{tt}]\!]_b \hat{s})}}{\mathsf{trst}(\hat{s}) \wedge \mathtt{tt} \Rightarrow \mathsf{trst}([\![\mathtt{tt}]\!]_b \hat{s})}$$

Case $B \equiv \mathtt{ff}$

$$\dfrac{\dfrac{}{\mathtt{ff} \Rightarrow \mathsf{trst}([\![\mathtt{ff}]\!]_b \hat{s})}}{\mathsf{trst}(\hat{s}) \wedge \mathtt{ff} \Rightarrow \mathsf{trst}([\![\mathtt{ff}]\!]_b \hat{s})}$$

Case $B \equiv E_0 = E_1$

let $(b_0, b_1) \stackrel{\Delta}{=} \hat{=}^{\triangleleft}([\![E_0]\!]\hat{s}, [\![E_1]\!]\hat{s})$ and $(\hat{s}_0, \hat{s}_1) \stackrel{\Delta}{=} ([\![E_0]\!]_b \hat{s}\, b_0, [\![E_1]\!]_b \hat{s}\, b_1)$

$$\dfrac{\dfrac{\dfrac{\tau_0 \quad \mathcal{E}_b(\hat{s}, b_0, E_0)}{\mathsf{trst}(\hat{s}) \wedge E_0 = E_1 \Rightarrow \mathsf{trst}(\hat{s}_0)} \quad \dfrac{\tau_1 \quad \mathcal{E}_b(\hat{s}, b_1, E_1)}{\mathsf{trst}(\hat{s}) \wedge E_0 = E_1 \Rightarrow \mathsf{trst}(\hat{s}_1)}}{\mathsf{trst}(\hat{s}) \wedge E_0 = E_1 \Rightarrow \mathsf{trst}(\hat{s}_0) \wedge \mathsf{trst}(\hat{s}_1)} \quad \mathsf{meetSt}(\hat{s}_0, \hat{s}_1)}{\mathsf{trst}(\hat{s}) \wedge E_0 = E_1 \Rightarrow \mathsf{trst}([\![E_0 = E_1]\!]_b \hat{s})}$$

where τ_i is:

$\psi_i \stackrel{\Delta}{=} \exists x.\, \mathsf{tr}([\![E_i]\!]\hat{s}, E_i) \wedge \mathsf{tr}([\![E_{1-i}]\!]\hat{s}, x) \wedge E_i = x$

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{E}(\hat{s}, E_0) \quad \mathcal{E}(\hat{s}, E_1)}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{tr}([\![E_0]\!]\hat{s}, E_0) \wedge \mathsf{tr}([\![E_1]\!]\hat{s}, E_1)}}{\mathsf{trst}(\hat{s}) \wedge E_0 = E_1 \Rightarrow \mathsf{tr}([\![E_0]\!]\hat{s}, E_0) \wedge \mathsf{tr}([\![E_1]\!]\hat{s}, E_1) \wedge E_0 = E_1}}{\mathsf{trst}(\hat{s}) \wedge E_0 = E_1 \Rightarrow \mathsf{trst}(\hat{s}) \wedge \psi_i} \quad \dfrac{\mathsf{bEql}_i([\![E_0]\!]\hat{s}, [\![E_1]\!]\hat{s}, E_i)}{\mathsf{trst}(\hat{s}) \wedge \psi_i \Rightarrow \mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(b_i, E_i)}}{\mathsf{trst}(\hat{s}) \wedge E_0 = E_1 \Rightarrow \mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(b_i, E_i)}$$

Case $B \equiv E_0 < E_1$

let $(b_0, b_1) \stackrel{\Delta}{=} \hat{<}^{\triangleleft}([\![E_0]\!]\hat{s}, [\![E_1]\!]\hat{s})$ and $(\hat{s}_0, \hat{s}_1) \stackrel{\Delta}{=} ([\![E_0]\!]_b \hat{s}\, b_0, [\![E_1]\!]_b \hat{s}\, b_1)$

$$\dfrac{\dfrac{\dfrac{\tau_0 \quad \mathcal{E}_b(\hat{s}, b_0, E_0)}{\mathsf{trst}(\hat{s}) \wedge E_0 < E_1 \Rightarrow \mathsf{trst}(\hat{s}_0)} \quad \dfrac{\tau_1 \quad \mathcal{E}_b(\hat{s}, b_1, E_1)}{\mathsf{trst}(\hat{s}) \wedge E_0 < E_1 \Rightarrow \mathsf{trst}(\hat{s}_1)}}{\mathsf{trst}(\hat{s}) \wedge E_0 < E_1 \Rightarrow \mathsf{trst}(\hat{s}_0) \wedge \mathsf{trst}(\hat{s}_1)} \quad \mathsf{meetSt}(\hat{s}_0, \hat{s}_1)}{\mathsf{trst}(\hat{s}) \wedge E_0 < E_1 \Rightarrow \mathsf{trst}([\![E_0 < E_1]\!]_b \hat{s})}$$

where τ_i is:

$\psi_0 \stackrel{\Delta}{=} \exists x.\, \mathsf{tr}([\![E_0]\!]\hat{s}, E_0) \wedge \mathsf{tr}([\![E_1]\!]\hat{s}, x) \wedge E_0 < x \qquad \psi_1 \stackrel{\Delta}{=} \exists x.\, \mathsf{tr}([\![E_0]\!]\hat{s}, x) \wedge \mathsf{tr}([\![E_1]\!]\hat{s}, E_1) \wedge x < E_1$

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{E}(\hat{s}, E_0) \quad \mathcal{E}(\hat{s}, E_1)}{\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{tr}([\![E_0]\!]\hat{s}, E_0) \wedge \mathsf{tr}([\![E_1]\!]\hat{s}, E_1)}}{\mathsf{trst}(\hat{s}) \wedge E_0 < E_1 \Rightarrow \mathsf{tr}([\![E_0]\!]\hat{s}, E_0) \wedge \mathsf{tr}([\![E_1]\!]\hat{s}, E_1) \wedge E_0 < E_1}}{\mathsf{trst}(\hat{s}) \wedge E_0 < E_1 \Rightarrow \mathsf{trst}(\hat{s}) \wedge \psi_i} \quad \dfrac{\mathsf{bInEql}_i([\![E_0]\!]\hat{s}, [\![E_1]\!]\hat{s}, E_i)}{\mathsf{trst}(\hat{s}) \wedge \psi_i \Rightarrow \mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(b_i, E_i)}}{\mathsf{trst}(\hat{s}) \wedge E_0 < E_1 \Rightarrow \mathsf{trst}(\hat{s}) \wedge \mathsf{tr}(b_i, E_i)}$$

Case $B \equiv B_0 \wedge B_1$

$$\dfrac{\dfrac{\mathcal{B}_b(\hat{s}, B_0) \quad \mathcal{B}_b(\hat{s}, B_1)}{\mathsf{trst}(\hat{s}) \wedge (B_0 \wedge B_1) \Rightarrow \mathsf{trst}([\![B_0]\!]_b \hat{s}) \wedge \mathsf{trst}([\![B_1]\!]_b \hat{s})} \quad \mathsf{meetSt}([\![B_0]\!]_b \hat{s}, [\![B_1]\!]_b \hat{s})}{\mathsf{trst}(\hat{s}) \wedge (B_0 \wedge B_1) \Rightarrow \mathsf{trst}([\![B_0 \wedge B_1]\!]_b \hat{s})}$$

Case $B \equiv B_0 \vee B_1$

$$\dfrac{\dfrac{\dfrac{\mathcal{B}_b(\hat{s}, B_0) \quad \mathsf{monSt}([\![B_0]\!]_b \hat{s}, [\![B_0 \vee B_1]\!]_b \hat{s})}{\mathsf{trst}(\hat{s}) \wedge B_0 \Rightarrow \mathsf{trst}([\![B_0 \vee B_1]\!]_b \hat{s})} \quad \dfrac{\mathcal{B}_b(\hat{s}, B_1) \quad \mathsf{monSt}([\![B_1]\!]_b \hat{s}, [\![B_0 \vee B_1]\!]_b \hat{s})}{\mathsf{trst}(\hat{s}) \wedge B_1 \Rightarrow \mathsf{trst}([\![B_0 \vee B_1]\!]_b \hat{s})}}{(\mathsf{trst}(\hat{s}) \wedge B_0) \vee (\mathsf{trst}(\hat{s}) \wedge B_1) \Rightarrow \mathsf{trst}([\![B_0 \vee B_1]\!]_b \hat{s})}}{\mathsf{trst}(\hat{s}) \wedge (B_0 \vee B_1) \Rightarrow \mathsf{trst}([\![B_0 \vee B_1]\!]_b \hat{s})}$$

Fig. 9. Proof Construction $\mathcal{B}_b(\hat{s}, B)$

Lemma 1. *The subroutines $\mathcal{E}$, $\mathcal{E}_b$, and $\mathcal{B}_b$ output proof trees. That is, the output trees are derivable in first-order logic.*

Proof. The lemma can be shown by induction on the structure of the input boolean or integer expression. Each induction step can be shown by the definition of the generic abstraction interpretation, and the specification (Figure 5) for the provided procedures. □

Theorem 1. *If an annotated command A is the result of an abstract interpretation, the tree $\mathcal{T}(A)$ is a proof tree in Hoare logic.*

Proof. We prove by induction on the structure of A.

- A is $[\hat{s}]x{:=}E[\hat{s}']$: In this case, we need to show that the subtree τ of $\mathcal{T}(A)$ for $\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{trst}(\hat{s}')[E/x]$ is derivable in first-order logic. From the generic abstract interpretation of $x := E$, we have

  $$\hat{s}' = \hat{s}[x \mapsto [\![E]\!]\hat{s}].$$

  Thus, $\hat{s}'(y) = \hat{s}(y)$ for all y in $\mathsf{Vars} - \{x\}$. And $\mathsf{tr}([\![E]\!]\hat{s}, E) = \mathsf{tr}([\![x]\!]\hat{s}', E)$, which implies that the tree $\mathcal{E}(\hat{s}, E)$ is for the formula

  $$\mathsf{trst}(\hat{s}) \Rightarrow \mathsf{tr}([\![x]\!]\hat{s}', E).$$

  Since $\mathcal{E}(\hat{s}, E)$ is a proof tree (Lemma 1), the tree τ is derivable.
- A is $[\hat{s}]\texttt{if } B \texttt{ then } [\hat{s}_1]R_1[\hat{s}_1'] \texttt{ else } [\hat{s}_2]R_2[\hat{s}_2'] \texttt{ fi}[\hat{s}']$: From the generic abstraction interpretation, we have

  $$\hat{s}_1' \sqcup \hat{s}_2' = \hat{s}', \quad \hat{s}_1 = [\![B]\!]_b\hat{s}, \quad \text{and} \quad \hat{s}_2 = [\![\neg B]\!]_b\hat{s}.$$

  Note that $\mathcal{B}_b(\hat{s}, B)$ and $\mathcal{B}_b(\hat{s}, \neg B)$ are both derivable (Lemma 1), and that for $i = 0, 1$, the tree $\mathsf{monSt}(\hat{s}_i', \hat{s}_1' \sqcup \hat{s}_2')$ is derivable because $\hat{s}_i' \sqsubseteq \hat{s}_1' \sqcup \hat{s}_2'$. Therefore, the induction hypothesis implies that the tree is derivable.
- A is $[\hat{s}]([\mathsf{inv}\ \hat{s}_0]\texttt{while } B \texttt{ do } [\hat{s}_1]R_1[\hat{s}_1'] \texttt{ od})[\hat{s}']$: From the generic abstraction interpretation, we have

  $$\hat{s} \sqsubseteq \hat{s}_0, \quad \hat{s}_1' \sqsubseteq \hat{s}_0, \quad \hat{s}_1 = [\![B]\!]_b\hat{s}_0, \quad \text{and} \quad \hat{s}' = [\![\neg B]\!]_b\hat{s}_0.$$

  The correctness of $\mathcal{B}_b$, the assumption for monSt, and the induction hypothesis imply that the tree $\mathcal{T}(A)$ is derivable.
- A is $[\hat{s}]A_1;A_2[\hat{s}']$: Let $[\hat{s}_1]R_1[\hat{s}_1']$ be A_1, and let $[\hat{s}_2]R_2[\hat{s}_2']$ be A_2. From the generic abstract interpretation, we have

  $$\hat{s} = \hat{s}_1, \quad \hat{s}_1' = \hat{s}_2, \quad \text{and} \quad \hat{s}_2' = \hat{s}'.$$

  Therefore, the induction hypothesis shows that $\mathcal{T}(A)$ is a proof tree.

□

We illustrate algorithm $\mathcal{T}$ with an example from the interval analysis. Consider the following program A annotated with the analysis results, integer intervals of program variables.

$$
\begin{array}{lll}
[x \mapsto [1,4], y \mapsto [2,5]] & & \\
\texttt{if}\ (x = y+1)\ \texttt{then} & [x \mapsto [3,4], y \mapsto [2,3]] \\
& x := x + y \\
& [x \mapsto [5,7], y \mapsto [2,3]] \\
\qquad\qquad\qquad\quad \texttt{else} & [x \mapsto [1,4], y \mapsto [2,5]] \\
& x := x + 1 \\
& [x \mapsto [2,5], y \mapsto [2,5]] \\
\texttt{fi} & \\
[x \mapsto [2,7], y \mapsto [2,5]] &
\end{array}
$$

When algorithm $\mathcal{T}$ gets the input A, it first recurses for sub-command $x := x+y$, and for $x := x + 1$, and obtains Hoare proofs, one for the Hoare triple H_0

$$\{3 \le x \le 4 \ \wedge\ 2 \le y \le\ 3\}\, x := x + y\, \{5 \le x \le 7 \ \wedge\ 2 \le y \le\ 3\}$$

and the other for the Hoare triple H_1

$$\{1 \le x \le 4 \ \wedge\ 2 \le y \le\ 5\}\, x := x + 1\, \{2 \le x \le 5 \ \wedge\ 2 \le y \le\ 5\}.$$

Note that there is a "gap" between these triples and what's needed to complete a proof for the input A. That is, the pre- and post-conditions of triples H_0 and H_1 do not match with those of the required triples in the following Hoare proof:

$$
\frac{
\begin{array}{c}\vdots\\ \left\{\begin{array}{c}1{\le}x{\le}4\\ \wedge\, 2{\le}y{\le}5\\ \wedge\, x{=}y{+}1\end{array}\right\} x := x + y \left\{\begin{array}{c}2{\le}x{\le}7\\ \wedge\, 2{\le}y{\le}5\end{array}\right\}\end{array}
\quad
\begin{array}{c}\vdots\\ \left\{\begin{array}{c}1{\le}x{\le}4\\ \wedge\, 2{\le}y{\le}5\\ \wedge\, x{\ne}y{+}1\end{array}\right\} x := x + 1 \left\{\begin{array}{c}2{\le}x{\le}7\\ \wedge\, 2{\le}y{\le}5\end{array}\right\}\end{array}
}{
\left\{\begin{array}{c}1{\le}x{\le}4\\ \wedge\, 2{\le}y{\le}5\end{array}\right\} \texttt{if}\ x = y + 1\ \texttt{then}\ x := x + y\ \texttt{else}\ x := x + 1\ \texttt{fi} \left\{\begin{array}{c}2{\le}x{\le}7\\ \wedge\, 2{\le}y{\le}5\end{array}\right\}
}
$$

Algorithm $\mathcal{T}$ fills this gap by calling $\mathcal{B}_b$ and monTr. The calls to $\mathcal{B}_b$ give proofs for implications between pre-conditions:

$$
\begin{array}{lcl}
1 \le x \le 4 \ \wedge\ 2 \le y \le 5 \ \wedge\ x = y + 1 & \Rightarrow & 3 \le x \le 4 \ \wedge\ 2 \le y \le 3 \\
1 \le x \le 4 \ \wedge\ 2 \le y \le 5 \ \wedge\ x \ne y + 1 & \Rightarrow & 1 \le x \le 4 \ \wedge\ 2 \le y \le 5
\end{array}
$$

and the calls to monTr give proofs for implications between post-conditions:

$$
\begin{array}{lcl}
5 \le x \le 7 \ \wedge\ 2 \le y \le 3 & \Rightarrow & 2 \le x \le 7 \ \wedge\ 2 \le y \le 5 \\
2 \le x \le 5 \ \wedge\ 2 \le y \le 5 & \Rightarrow & 2 \le x \le 7 \ \wedge\ 2 \le y \le 5.
\end{array}
$$

3.3 Size of Generated Proof Trees

We measure the size of an output tree from $\mathcal{T}$ by counting the nodes in the tree. While counting these nodes, we will assume that each call to the provided procedures, such as monTr, meetTr and fAdd, returns a proof tree with a single node. Note that the size k of a tree τ, computed under this assumption, still gives an upper bound for the number of nodes in τ; when each call to the provided procedures gives a proof tree with at most k' nodes, the tree τ can have at most $k \times k'$ nodes. Let $|\mathsf{Vars}|$ be the cardinality of Vars. Since we always use for Vars the set of variables in the input program, $|\mathsf{Vars}|$ denotes the number of variables in the input program. Let the size of an expression or a command be the number of its tokens. For instance, the size of the expression $y + z + 1$ is 5, and the size of the command $x := y + z + 1; x := 2$ is 11.

Lemma 2. *For an expression E of size n and an abstract state $\hat{s}$, the tree $\mathcal{E}(\hat{s}, E)$ has $O(n)$ nodes.*

Proof. Subroutine $\mathcal{E}$ recurses only when $E = E_0 + E_1$. In that case, it calls itself for disjoint subparts E_0 and E_1 of E. Thus, there can be only $O(n)$-many recursive calls to $\mathcal{E}$. This number of recursive calls limits the size of the tree $\mathcal{E}(\hat{s}, E)$ to $O(n)$. □

Lemma 3. *For an expression E of size n, an abstract state $\hat{s}$, and an abstract value a, the tree $\mathcal{E}_b(\hat{s}, a, E)$ has $O(n^2 + n \times |\mathsf{Vars}|)$ nodes.*

Proof. Subroutine $\mathcal{E}_b$ can recurse only $O(n)$ times, because of the same reason as in the proof of Lemma 2. Thus, $\mathcal{E}$ is called $O(n)$-times in $\mathcal{E}_b$, and the macro meetSt is expanded $O(n)$ times giving $O(n \times |\mathsf{Vars}|)$-many calls to meetTr. Since each call to $\mathcal{E}$ can add $O(n)$ nodes, the tree $\mathcal{E}_b(\hat{s}, s, E)$ has $O(n) + O(n^2) + O(n \times |\mathsf{Vars}|) = O(n^2 + n \times |\mathsf{Vars}|)$ nodes. □

Lemma 4. *For a boolean expression B of size n and an abstract state $\hat{s}$, the tree $\mathcal{B}_b(\hat{s}, B)$ has $O(n^2 + n \times |\mathsf{Vars}|)$ nodes.*

Proof. Subroutine $\mathcal{B}_b$ recurses only when $B = B_1 \vee B_2$ or $B = B_1 \wedge B_2$. In both cases, the recursive calls are for disjoint subparts of B. So, $\mathcal{B}_b(E)$ recurses only $O(n)$ times. This number of recursive calls bounds the number of macro expansions of meetSt and monSt, so that meetTr and monTr can be called $O(n \times |\mathsf{Vars}|)$ times. As of the calls to $\mathcal{E}$ and $\mathcal{E}_b$ in $\mathcal{B}_b$, we observe that both $\mathcal{E}$ and $\mathcal{E}_b$ are called in $\mathcal{B}_b$ only when $\mathcal{B}_b$ does not recurse; moreover, when $n_0, n_1, \ldots, n_k$ are the size of inputs to all these calls to $\mathcal{E}$ and $\mathcal{E}_b$, the sum $\Sigma_{i=0}^{k} n_i$ is $O(n)$. Therefore, although $\mathcal{E}$ and $\mathcal{E}_b$ are called $O(n)$ times, only $O(n)$ nodes are constructed from all the calls to $\mathcal{E}$, and $O(n^2 + n \times |\mathsf{Vars}|)$ nodes from all the calls to $\mathcal{E}_b$; when $O(n) = \Sigma_{i=0}^{k} n_i$, we have $O(n^l) = \Sigma_{i=0}^{k} (n_i)^l$ for all natural numbers l. Therefore, the tree $\mathcal{B}_b(\hat{s}, B)$ can have $O(n) + O(n \times |\mathsf{Vars}|) + O(n) + O(n^2 + n \times |\mathsf{Vars}|) = O(n^2 + n \times |\mathsf{Vars}|)$ nodes. □

Proposition 1. *For a command $\overline{A}$ of size n, the tree $\mathcal{T}(A)$ has $O(n^2 + n \times |\mathsf{Vars}|)$ nodes.*

Proof. Algorithm $\mathcal{T}$ calls itself $O(n)$ times. Thus, the macro monSt is expanded $O(n)$ times, giving $O(n \times |\mathsf{Vars}|)$ calls to monTr. For the calls to $\mathcal{B}_b$ and $\mathcal{E}$, we note that when $n_0, \ldots, n_k$ are the sizes of inputs to all the calls to $\mathcal{B}_b$ and $\mathcal{E}$, the sum $\Sigma_{i=0}^{k} n_i$ is $O(n)$. We can use the argument in the proof of Lemma 4 to prove that all the calls to $\mathcal{B}_b$ can construct $O(n^2 + n \times |\mathsf{Vars}|)$ nodes, and all the calls to $\mathcal{E}$ $O(n)$ nodes. The tree $\mathcal{T}(A)$, therefore, has $O(n) + O(n \times |\mathsf{Vars}|) + O(n^2 + n \times |\mathsf{Vars}|) + O(n) = O(n^2 + n \times |\mathsf{Vars}|)$ nodes. □

We expect that in practice, the sizes of generated proofs are significantly smaller than the worst case $O(n^2 + n \times |\mathsf{Vars}|)$, because boolean or integer expressions in programs are usually short: their sizes are practically constant compared to the size of a program.

4 Conclusion

We have presented an algorithm that automatically constructs Hoare proofs for program's approximate invariants annotated by abstract interpreters. The gap between the approximate invariants and the Hoare-logic rules is filled by the safety proofs of the used abstract interpreter. Although our algorithm still requires a well-designed abstract interpreter and its safety proofs, it reduces the complexity of proof construction, because 1) the same abstract interpreter can be used repeatedly for multiple programs; 2) the needed safety proofs of the used abstract interpreter are standard ones prescribed by the abstract interpretation framework.

The method reported in this paper suggests a yet another framework of PCC, where the proof construction process is fully automatic, and the code properties it can verify and check are more general than types. We will employ this method in our planned PCC compiler system. The compiler uses abstract interpretations and our method to construct safety proofs of the input programs. The compiler then compiles the obtained proofs for the source code into proofs for the compiled target code. This compiled proof and code pairs are to be checked by the code consumer. Developing such a "proof compiler" technology is our next goal.

Currently we are implementing our algorithm for a simple imperative language extended with arrays and procedures. An abstract interpreter verifies that all array references are within bounds. Through this implementation, we expect to use similar ideas as [13] in engineering the proof sizes.

Acknowledgements. Seo and Yang were supported by grant No. R08-2003-000-10370-0 from the Basic Research Program of the Korea Science & Engineering Foundation. Yi was supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

References

1. A. W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

2. A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, January 2000.
3. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
5. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence and Programming Languages*, volume 12, pages 1–12, Rochester, NY, ACM SIGPLAN Notices, August 1977.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.
7. N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 89–100, June 2002.
8. Nevin Heintze, Joxan Jaffar, and Razvan Voicu. A framework for combining analysis and verification. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 26–39, Boston, MA, USA, January 2000.
9. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
10. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, January 1998.
11. G. C. Necula. Proof-carrying code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.
12. G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, 1997.
13. G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154, January 2001.
14. G. C. Necula and R. Schneck. Proof-carrying code with untrusted proof rules. In *Software Security – Theories and Systems*, volume 2609 of *Lecture Notes in Computer Science*, pages 283–298. Springer-Verlag, November 2002.