Complexity and algorithms for monomial and clausal predicate abstraction

Shuvendu K. Lahiri and Shaz Qadeer

Microsoft Research

Abstract. In this paper, we investigate the asymptotic complexity of various predicate abstraction problems relative to the asymptotic complexity of checking an annotated program in a given assertion logic. Unlike previous approaches, we pose the predicate abstraction problem as a decision problem, instead of the traditional inference problem. For assertion logics closed under weakest (liberal) precondition and Boolean connectives, we show two restrictions of the predicate abstraction problem where the two complexities match. The restrictions correspond to the case of *monomial* and *clausal* abstraction. For these restrictions, we show a symbolic encoding that reduces the predicate abstraction problem to checking the satisfiability of a single formula whose size is polynomial in the size of the program and the set of predicates. We also provide a new iterative algorithm for solving the *clausal* abstraction problem that can be seen as the dual of the `Houdini` algorithm for solving the *monomial* abstraction problem.

1 Introduction

Predicate abstraction [8] is a method for constructing inductive invariants for programs or transition systems over a given set of predicates $\mathcal{P}$. It has been an enabling technique for several automated hardware and software verification tools. SLAM [1], BLAST [11] use predicate abstraction to construct invariants for sequential software programs. Predicate abstraction has also been used in tools for verifying hardware descriptions [12] and distributed algorithms [14]. Although predicate abstraction is an instance of the more general theory of abstract interpretation [5], it differs from most other abstract interpretation techniques (e.g. for numeric domains [6], shape analysis [17]) in that it does not require a fixed abstract domain; it is parameterized by decision procedures for the assertion logic in which the predicates are expressed.

Most previous work on predicate abstraction has been concerned with *constructing* an inductive invariant over a set of predicates $\mathcal{P}$ using abstract interpretation techniques. A (finite) abstract domain over the predicates is defined and the program is executed over the abstract domain until the set of abstract states do not change. At that point, the abstract state can be shown to be an inductive invariant. These techniques are usually *eager* in that the inductive invariant is computed without recourse to the property of interest. The property directedness is somewhat recovered using various forms of refinements (e.g.

counterexample-guided [13, 3, 11], or proof-guided [10] refinements) by varying the set of predicates. However, these techniques do not usually address the problem for the case of a fixed set of predicates.

In this paper, we study the problem of predicate abstraction as a decision problem. For a given program $Prog(pre, post, body)$

$$\begin{array}{l}\{pre\}\\ \mathsf{while}\ (*)\ \mathsf{do}\ body\\ \{post\}\end{array}$$

where *pre* and *post* are assertions, *body* is loop-free code, and a set of predicates $\mathcal{P}$, we are interested in the question:

> Does there exist a loop invariant I over $\mathcal{P}$ such that program *Prog* can be proved correct?

We define this decision problem to be $InferPA(Prog, \mathcal{P})$. By posing the problem as a decision problem, we do not have to adhere to any particular way to *construct* the loop invariant I (say using abstract interpretation), and it allows us to study the complexity of the problem. Besides, the problem formulation requires us to search for only those I that can prove the program. This problem formulation has been first proposed in the context of the annotation inference problem in `ESC/Java` [7] for a restricted case, and more recently by Gulwani et al. [9]. Although the latter has the same motivation as our work, they do not study the complexity of the predicate abstraction problems.

In this paper, we study the asymptotic complexity of the decision problem $InferPA(Prog, \mathcal{P})$ relative to the decision problem $Check(Prog, I)$ which checks if I is a loop invariant that proves the program correct. Throughout the paper, we assume that the assertion logic of assertions in *pre*, *post* and predicates in $\mathcal{P}$ is closed under *weakest (liberal) precondition* predicate transformer *wp*, and also closed under Boolean connectives. The assertion logic determines the logic in which $Check(Prog, I)$ is expressed. We are also most interested in logics for which the decision problem is **Co-NP** complete— this captures a majority of the assertion logics for which efficient decision procedures have been implemented using Satisfiability Modulo Theories (SMT) solvers [20]. In addition to propositional logic, this includes the useful theories of uninterpreted functions, arithmetic, select-update arrays, inductive datatypes, and more recently logics about linked lists [15] and types in programs [4].

For such assertion logics, we show that if checking $Check(Prog, I)$ is in **PSPACE**, then checking $InferPA(Prog, \mathcal{P})$ is **PSPACE** complete. We also study the problem of template abstraction [19, 21] where the user provides a formula J with some free Boolean variables $\mathcal{W}$, and is interested in finding whether there is a valuation $\sigma_{\mathcal{W}}$ of $\mathcal{W}$ such that $Check(Prog, J[\sigma_{\mathcal{W}}/\mathcal{W}])$ is true. We call this decision problem $InferTempl(Prog, J, \mathcal{W})$, and show $InferTempl(Prog, J, \mathcal{W})$ is $\mathbf{\Sigma_2^P}$ complete ($\mathbf{NP^{NP}}$ complete), when $Check(Prog, I)$ is in **Co-NP**.

Given that the general problem of predicate or template abstraction can be much more complex than checking $Check(Prog, I)$, we focus on two restrictions:

$$
\begin{array}{lll}
\mathsf{x}, \mathsf{y} & \in \mathit{Scalars} & \\
\mathsf{X}, \mathsf{Y} & \in \mathit{Maps} & \\
e & \in \mathit{Expr} & ::= \mathsf{x} \mid c \mid e \pm e \mid \mathbf{sel}(E, e) \\
E & \in \mathit{MapExpr} & ::= \mathsf{X} \mid \mathbf{upd}(E, e, e) \\
s, t & \in \mathit{Stmt} & ::= \mathsf{skip} \mid \mathsf{assert}\ \phi \mid \mathsf{assume}\ \phi \mid \mathsf{x} := e \mid \mathsf{X} := E \\
 & & \quad \mathsf{havoc}\ \mathsf{x} \mid \mathsf{havoc}\ \mathsf{X} \mid s; s \mid s \diamond s \\
 & & \\
\phi, \psi & \in \mathit{Formula} & ::= e \leq e \mid \phi \wedge \phi \mid \neg\phi \mid \ldots
\end{array}
$$

Fig. 1. A simple programming language SimpPL and an extensible assertion logic *Formula*.

- *InferMonome*$(Prog, \mathcal{P})$: Given a set of predicates $\mathcal{P}$, does there exist a $\mathcal{R} \subseteq \mathcal{P}$ such that *Check*$(Prog, \bigwedge_{p \in \mathcal{R}} p)$ is true, and
- *InferClause*$(Prog, \mathcal{P})$: Given a set of predicates $\mathcal{P}$, does there exist a $\mathcal{R} \subseteq \mathcal{P}$ such that *Check*$(Prog, \bigvee_{p \in \mathcal{R}} p)$ is true.

These problems can also be seen as restrictions on the template abstraction problem. For example, the problem *InferMonome*$(Prog, \mathcal{P})$ can also be interpreted as *InferTempl*$(Prog, J, \mathcal{W})$ where J is restricted as $(\bigwedge_{p \in \mathcal{P}} b_p \implies p)$ and $\mathcal{W} = \{b_p | p \in \mathcal{P}\}$.

We show that for both *InferMonome*$(Prog, \mathcal{P})$ and *InferClause*$(Prog, \mathcal{P})$, the complexity of the decision problem matches the complexity of *Check*$(Prog, I)$. For instance, when the complexity of *Check*$(Prog, I)$ is **Co-NP** complete, both *InferMonome*$(Prog, \mathcal{P})$ and *InferClause*$(Prog, \mathcal{P})$ are **Co-NP** complete. We obtain these results by providing a symbolic encoding of both problems into logical formulas, such that the logical formulas are satisfiable if and only if the inference problems return false. The interesting part is that these logical formulas are polynomially bounded in *Prog* and $\mathcal{P}$.

The symbolic encoding of the inference problems also provides algorithms to answer these decision problems, in addition to establishing the complexity results. In the process, we also describe a new iterative algorithm for checking *InferClause*$(Prog, \mathcal{P})$ that can be seen as the dual of an existing algorithm `Houdini` [7] that checks *InferMonome*$(Prog, \mathcal{P})$.

2 Background

We describe some background on programs and their correctness, assertion logics, weakest liberal preconditions, and motivate the inference problems.

2.1 A simple programming language for loop-free programs

Figure 1 describes a simple programming language SimpPL for loop-free programs. The language supports scalar variables *Scalars* and mutable maps or arrays *Maps*. The language supports arithmetic operations on scalar expressions

Expr and *select-update* reasoning for array expressions *MapExpr*. The symbols **sel** and **upd** are interepreted symbols for selecting from or updating an array. The operation havoc assigns a type-consistent (scalar or map) arbitrary value to its argument. $s;t$ denotes the sequential composition of two statements s and t, $s \diamond t$ denotes a non-deterministic choice to either execute statements in s or t. This statement along with the assume models conditional statements. For example, the statement if (e) $\{s\}$ is desugared into $\{\mathsf{assume}\ e; s\} \diamond \mathsf{assume}\ \neg e$.

$$
\begin{array}{ll}
wp(\mathsf{skip}, \phi) & = \phi \\
wp(\mathsf{assert}\ \psi, \phi) & = \psi \wedge \phi \\
wp(\mathsf{assume}\ \psi, \phi) & = \psi \implies \phi \\
wp(\mathsf{x} := e, \phi) & = \phi[e/\mathsf{x}] \\
wp(\mathsf{X} := E, \phi) & = \phi[E/\mathsf{X}]
\end{array}
\qquad
\begin{array}{ll}
wp(\mathsf{havoc}\ \mathsf{x}, \phi) & = \phi[\mathsf{v}/\mathsf{x}] \\
wp(\mathsf{havoc}\ \mathsf{X}, \phi) & = \phi[\mathsf{V}/\mathsf{X}] \\
wp(s;t, \phi) & = wp(s, wp(t, \phi)) \\
wp(s \diamond t, \phi) & = wp(s, \phi) \wedge wp(t, \phi)
\end{array}
$$

Fig. 2. Weakest liberal precondition for the logic without any extensions. Here v and V represent fresh symbols.

The assertion language in *Formula* is extensible and contains the theories for equality, arithmetic, arrays, and is closed under Boolean connectives. Any formula $\phi \in$ *Formula* can be interpreted as a set of states of a program that satisfy ϕ. For any $s \in$ *Stmt*, and $\phi \in$ *Formula*, the *weakest liberal precondition* $wp(s, \phi)$ corresponds to a formula such that from any state in $wp(s, \phi)$, the statement s does not fail any assertions and any terminating execution ends in a state satisfying ϕ. For our assertion logic (without any extensions), Figure 2 shows the *wp* for statements in the programming language. For more complex extensions to the assertion logic (e.g. [15]), the rule for $wp(\mathsf{X} := E, \phi)$ is more complex. Although applying *wp* can result in an exponential blowup in the size of a program, standard methods generate a linear-sized formula that preserves validity by performing static single assignment (SSA) and introducing auxiliary variables [2].

We say that the assertion logic is *closed under wp* (for SimpPL) when for any $\phi \in$ *Formula* and for any $s \in$ *Stmt*, $wp(s, \phi) \in$ *Formula*. In the rest of the paper, we will abstract from the details of the particular assertion logic, with the following restrictions on the assertion logic:

- the assertion logic is closed under Boolean connectives (i.e. subsumes propositional logic).
- the assertion logic is closed under *wp*.
- *wp* distributes over $\wedge$, i.e., $wp(s, \phi \wedge \psi) \equiv wp(s, \phi) \wedge wp(s, \psi)$.

A *model* $\mathcal{M}$ assigns a type-consistent valuation to symbols in a formula. Any model assigns the standard values for interpreted symbols such as $=$, $+$, $-$, **sel**, **upd**, and assigns an integer value to symbols in *Scalars* and function values to symbols in *Maps*. For a given model $\mathcal{M}$, we say that the model satisfies a formula $\phi \in$ *Formula* (written as $\mathcal{M} \models \phi$) if and only if the result of evaluating

ϕ in $\mathcal{M}$ is true; in such a case, we say that ϕ is *satisfiable*. We use $\models \phi$ to denote that ϕ is *valid* when ϕ is satisfiable for any model.

Definition 1. *For any $\phi, \psi \in$ Formula and $s \in$ Stmt, the Floyd-Hoare triple $\{\phi\}\ s\ \{\psi\}$ holds if and only if the logical formula $\phi \implies wp(s, \psi)$ is valid.*

Intuitively, the Floyd-Hoare triple $\{\phi\}\ s\ \{\psi\}$ captures the specification that from a state satisfying ϕ, no execution of s fails any assertions and every terminating execution ends in a state satisfying ψ. Given our assumptions on the assertion logic *Formula*, checking the correctness of a program in SimpPL reduces to checking validity in the assertion logic.

2.2 Loops and loop invariants

Having defined the semantics of loop-free code blocks, consider the following class of programs $Prog(pre, post, body)$ below (ignore the annotation $\{\mathsf{inv}\ I\}$ initially) where $pre, post \in Formula$ and $body \in Stmt$:

$$\begin{array}{l} \{pre\} \\ \mathsf{while}\ (*)\{\mathsf{inv}\ I\}\ \mathsf{do}\ body \\ \{post\} \end{array}$$

Since this program can have unbounded computations due to the while loop, generating a verification condition requires a loop invariant. This is indicated by inv annotation. The loop invariant I is a formula in *Formula*. The Floyd-Hoare triple for *Prog* holds if and only if there exists a loop invariant I such that the three formula are valid:

$$\begin{array}{lll} \models pre & \implies & I \\ \models I & \implies & wp(body, I) \\ \models I & \implies & post \end{array}$$

For any such program $Prog(pre, post, body)$, we define $Check(Prog, I)$ to return true if and only if all the three formulas are valid. Intuitively, $Check(Prog, I)$ *checks* whether the supplied loop invariant I holds on entry to the loop, is preserved by an arbitrary loop iteration and implies the postcondition.

We now define the decision problem that corresponds to *inferring* the loop invariant I for *Prog*. For a given program $Prog(pre, post, body)$, $Infer(Prog)$ returns true if and only if there exists a formula $I \in Formula$ such that $Check(Prog, I)$ is true.

Although $Check(Prog, I)$ is efficiently decidable for our programming language for a rich class of assertions (including the one shown in Figure 1), checking $Infer(Prog)$ is undecidable even when *Prog* consists of only scalar integer variables and include arithmetic operations — one can encode the reachability problem for a two counter machine, checking which is undecidable. Therefore, most approaches search for I within some restricted space. Predicate abstraction is one such technique that searches for an I within a finite space.

3 Complexity of predicate and template abstraction

In this section, we study the complexity of two inference techniques (relative to the complexity of $Check(Prog, I)$) that search for loop invariants over a finite space:

1. The first technique is based on *predicate abstraction*, where the loop invariant is searched over Boolean combination of an input set of predicates.
2. The second technique is based on *templates*, where the loop invariant is searched over the valuations of a set of free Boolean variables in a candidate template assertion.

3.1 Predicate abstraction

Predicate abstraction [8], an instance of the more general theory of abstract interpretation [5], is a mechanism to make the $Infer(Prog)$ problem more tractable by searching for I over restricted formulas. In predicate abstraction, in addition to $Prog$, we are given a set of predicates $\mathcal{P} = \{p_1, \ldots, p_n\}$ where $p_i \in Formula$. Throughout this paper, we assume that the set $\mathcal{P}$ is closed under negation, i.e., if $p \in \mathcal{P}$ then $\neg p \in \mathcal{P}$. Instead of looking for a loop invariant I over arbitrary formulas, predicate abstraction restricts the search to those formulas that are a Boolean combination (using $\vee$ or $\wedge$) over the predicates in $\mathcal{P}$. More formally, for a program $Prog(pre, post, body)$ and a set of predicates $\mathcal{P}$, $InferPA(Prog, \mathcal{P})$ returns true if and only if there exists a formula $I \in Formula$ which is a Boolean combination over $\mathcal{P}$ such that $Check(Prog, I)$ is true.

Theorem 1. *If checking $Check(Prog, I)$ is in* **PSPACE** *in the size of $Prog$ and I, then checking $InferPA(Prog, \mathcal{P})$ is* **PSPACE** *complete in the size of $Prog$ and $\mathcal{P}$.*

Proof sketch. Showing that $InferPA(Prog, \mathcal{P})$ is **PSPACE** hard is easy. We can encode the reachability problem for a propositional transition system (which is **PSPACE** complete) into $InferPA(Prog, \mathcal{P})$ by encoding the transition system as $Prog$, and setting $\mathcal{P}$ to be the set of state variables in the transition system.

To show that $InferPA(Prog, \mathcal{P})$ is in **PSPACE**, we will provide a non-deterministic algorithm for checking if $\neg post$ can be reached by an abstract interpretation of the program $\{pre\}$ while $(*)$ do $body$ $\{post\}$ over the set of predicates in $\mathcal{P}$. Moreover, the algorithm will only use polynomial space in $Prog$ and $\mathcal{P}$.

The abstract state of the program corresponds to an evaluation of the predicates in $\mathcal{P}$ to $\{\mathsf{true}, \mathsf{false}\}$. The algorithm performs an abstract *run* of size upto $2^{|\mathcal{P}|}$ to determine if $\neg post$ can be reachable starting from pre. The non-deterministic algorithm returns false if and only if some abstract run ends up in a state satisfying $\neg post$.

We need to store two successive states in a run and the length of the run, both of which only require linear space over the inputs. Moreover, to check that an abstract state is a successor of another, one needs to make a query to check

that there is some concrete transition in *Prog* between the concretizations of the two abstract states — this can be done in **PSPACE** since the decision problem is in **PSPACE**. □

3.2 Template abstraction

In template abstraction [18, 9, 21], the user provides a *template* formula $J \in \mathit{Formula}^{\mathcal{W}}$ in addition to $\mathit{Prog}(\mathit{pre}, \mathit{post}, \mathit{body})$, where

- $\mathcal{W} = \{w_1, w_2, \ldots, w_m\}$ is a set of Boolean valued symbols, and
- $\mathit{Formula}^{\mathcal{W}}$ extends *Formula* to contain formulas whose symbols range over both state variables (*Scalars* and *Maps*) and $\mathcal{W}$.

Given J, the goal is to infer a loop invariant I by searching over the different valuations of the symbols in $\mathcal{W}$.

For a set of symbols X, let σ_X denote an *assignment* of values of appropriate types to each symbols in X. For any expression e and an assignment σ_X, $e[\sigma_X/X]$ replaces a symbol $x \in X$ in e with $\sigma_X(x)$. For a program *Prog* and a template $J \in \mathit{Formula}^{\mathcal{W}}$, $\mathit{InferTempl}(\mathit{Prog}, J, \mathcal{W})$ returns true if and only if there exists an assignment $\sigma_{\mathcal{W}}$ to the symbols in $\mathcal{W}$ such that $\mathit{Check}(\mathit{Prog}, J[\sigma_{\mathcal{W}}/\mathcal{W}])$ is true.

Example 1. Consider the simple program $\{x = 0 \wedge y = 10\}$ while $(y \neq 0)$ do $x := x+1; y := y-1;$ $\{x = 10\}$. In our language, the program would be written as $\{x = 0 \wedge y = 10\}$ while $(*)$ do assume $y \neq 0; x := x+1; y := y-1;$ $\{y = 0 \implies x = 10\}$. A potential loop invariant that proves the program is $x+y = 10$. A template J for which $\mathit{InferTempl}(\mathit{Prog}, J, \mathcal{W})$ is true is $(w_1 \implies x + y = 10) \wedge (w_2 \implies x = y)$. Clearly, for the assignment $\sigma_{w_1} = \mathsf{true}$ and $\sigma_{w_2} = \mathsf{false}$, the template is a loop invariant.

The complexity class $\mathbf{\Sigma_2^P}$ contains all problems that can be solved in **NP** using an **NP** oracle.

Theorem 2. *If checking* $\mathit{Check}(\mathit{Prog}, I)$ *is in* **Co-NP** *in the size of Prog and I, then checking* $\mathit{InferTempl}(\mathit{Prog}, J, \mathcal{W})$ *is* $\mathbf{\Sigma_2^P}$ *complete in Prog, J, and* $\mathcal{W}$.

Proof. The problem is in $\mathbf{\Sigma_2^P}$ because we can non-deterministically guess an assignment of $\mathcal{W}$ and check if the resulting J is an inductive invariant. The **NP** oracle used in this case is a checker for $\neg \mathit{Check}(\mathit{Prog}, I)$.

On the other hand, one can formulate $\mathit{InferTempl}(\mathit{Prog}, J, \mathcal{W})$ as the formula $\exists \mathcal{W}. \mathit{Check}(\mathit{Prog}, J)$. Given a quantified Boolean formula (QBF) $\exists X. \forall Y. \phi(X, Y)$ where ϕ is quantifier-free, for which checking the validity is $\mathbf{\Sigma_2^P}$ complete, we can encode it to $\mathit{InferTempl}(\mathit{Prog}, J, \mathcal{W})$, by constructing $\mathit{pre} = \mathit{post} = \mathsf{true}$, $\mathit{body} = \mathsf{skip}$, $\mathcal{W} = X$ and $J = \phi(X, Y)$. □

Having shown that the complexity of both predicate abstraction and template abstraction are considerably harder than checking an annotated program, we will focus on two restricted versions of the abstraction problem for monomials and clauses. As mentioned in the introduction, these problems can be seen as restrictions of either predicate abstraction or template abstraction.

4 Monomial abstraction

For any set $\mathcal{R} \subseteq \mathcal{P}$, a *monome* over $\mathcal{R}$ is the formula $\bigwedge_{p \in \mathcal{R}} p$. For a set of predicates $\mathcal{P}$, let us define $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ to return true if and only if there exists $\mathcal{R} \subseteq \mathcal{P}$ such that $\mathit{Check}(\mathit{Prog}, \bigwedge_{p \in \mathcal{R}} p)$ is true.

4.1 `Houdini` algorithm

$$
\begin{array}{lll}
\mathit{FailsCheck}(\mathcal{S}, \mathcal{M}) & \triangleq & \mathcal{M} \models (\bigwedge_{q \in \mathcal{S}} q) \wedge \neg \mathit{post} \\
\mathit{RemovesPredicate}(\mathcal{S}, \mathcal{M}, p) & \triangleq & \vee \mathcal{M} \models \mathit{pre} \wedge \neg p \\
 & & \vee \mathcal{M} \models (\bigwedge_{q \in \mathcal{S}} q) \wedge \neg \mathit{wp}(\mathit{body}, p)
\end{array}
$$

```
proc FindInvAux(R)
  if (exists a model M s.t. FailsCheck(R, M))
    M_guess ← M;
    return FAIL;
  if (exists q ∈ R and model M s.t. RemovesPredicate(R, M, q))
    M_guess ← M;
    Q_guess+1 ← Q_guess ∪ {q};
    guess ← guess + 1;
    return FindInvAux(R \ {q});
  return SUCCESS(R);

proc FindInv(P)
  guess ← 1;
  Q_guess ← {};
  FindInvAux(P);
```

Fig. 3. Procedure to construct either a monomial invariant or a witness to show its absence. The *shaded* lines represent extensions for computing the witness.

Figure 3 describes an algorithm *FindInv* for solving the *InferMonome* problem. Initially, ignore the shaded regions of the algorithm. The algorithm iteratively prunes the set of predicates in the invariant starting from $\mathcal{P}$, removing a predicate whenever *RemovesPredicate* holds. The algorithm terminates with a *FAIL* when *FailsCheck* holds, denoting that there is no monomial invariant that satisfies the postcondition *post*. On the other hand, the algorithm terminates with $\mathit{SUCCESS}(\mathcal{R})$ when it finds an invariant. It is easy to see that the procedure *FindInvAux* terminates within a recursion depth of $|\mathcal{P}|$ since its argument $\mathcal{R}$ monotonically decreases along any call chain.

Lemma 1. *The procedure $\mathit{FindInv}(\mathcal{P})$ satisfies the following:*

1. *$\mathit{FindInv}(\mathcal{P})$ returns FAIL if and only if $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ is* false.
2. *If $\mathit{FindInv}(\mathcal{P})$ returns $\mathit{SUCCESS}(\mathcal{R})$, then for any $\mathcal{S} \subseteq \mathcal{P}$ such that $\mathit{Check}(\mathit{Prog}, \bigwedge_{p \in \mathcal{S}} p)$, we have $S \subseteq \mathcal{R}$.*

The algorithm is a variant of the `Houdini` algorithm in `ESC/Java` [7], where *RemovesPredicate* considers each predicate p in isolation instead of the conjunction $(\bigwedge_{q \in \mathcal{S}} q)$. The `Houdini` algorithm solves the *InferMonome*$(Prog, \mathcal{P})$ problem with at most $|\mathcal{P}|$ number of theorem prover calls. However, this only provides an upper bound on the complexity of the problem. For example, making $|\mathcal{P}|$ number of queries to a **Co-NP** complete oracle (a theorem prover) does not establish that the complexity of the inference problem is **Co-NP** complete; it only establishes that the upper bound of the complexity is $\mathbf{P^{NP}}$.

In the next subsection, we provide a model-theoretic justification for the correctness of *FindInv*. Our construction will provide insight into the complexity of *InferMonome*$(Prog, \mathcal{P})$ relative to the complexity of *Check*$(Prog, I)$.

4.2 A model-theoretic proof of *FindInv*

For a $guess \in \mathbb{N}$, an indexed set of models $\{\mathcal{M}_i\}_i$, an indexed set of sets of predicates $\{\mathcal{Q}_i\}_i$, we define a predicate *NoMonomeInv*$(\mathcal{P}, guess, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$, that is true if and only if:

1. $1 \leq guess \leq |\mathcal{P}| + 1$, and
2. $\mathcal{Q}_1 = \{\}$, and
3. For $1 \leq i < guess$, $\mathcal{Q}_{i+1} \setminus \mathcal{Q}_i = \{p_i\}$ for some $p_i \in \mathcal{P}$, and
4. For each $1 \leq i < guess$, and for $p_i \in \mathcal{Q}_{i+1} \setminus \mathcal{Q}_i$, either $\mathcal{M}_i \models pre \wedge \neg p_i$ or $\mathcal{M}_i \models (\bigwedge_{p \in \mathcal{P} \setminus \mathcal{Q}_i} p) \wedge \neg wp(body, p_i)$, and
5. $\mathcal{M}_{guess} \models (\bigwedge_{p \in \mathcal{P} \setminus \mathcal{Q}_{guess}} p) \wedge \neg post$.

The following three lemmas, whose proofs are given in the appendix, establish the connection between the *InferMonome* problem, the *NoMonomeInv* predicate, and the *FindInv* algorithm.

Lemma 2. *If FindInv*$(\mathcal{P})$ *returns FAIL, then NoMonomeInv*$(\mathcal{P}, guess, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$ *holds on the values computed by the procedure.*

Lemma 3. *If NoMonomeInv*$(\mathcal{P}, guess, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$ *holds for some guess* $\in \mathbb{N}$*, a set of models* $\mathcal{M}_1, \ldots, \mathcal{M}_{guess}$*, and sets of predicates* $\mathcal{Q}_1, \ldots, \mathcal{Q}_{guess}$*, then InferMonome*$(Prog, \mathcal{P})$ *is* false*.*

Lemma 4. *If FindInv*$(\mathcal{P})$ *returns SUCCESS*$(\mathcal{R})$*, then Check*$(Prog, \bigwedge_{p \in \mathcal{R}} p)$ *is* true*, and therefore InferMonome*$(Prog, \mathcal{P})$ *is* true*.*

The proofs of these lemmas requires the use of the additional shaded lines in the Figure 3, which compute the witness to show that no monomial invariant suffices to prove the program. Together, these three lemmas allow us to conclude that *InferMonome*$(Prog, \mathcal{P})$ is false iff *NoMonomeInv*$(\mathcal{P}, guess, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$ holds for some $guess \in \mathbb{N}$, a set of models $\mathcal{M}_1, \ldots, \mathcal{M}_{guess}$, sets of predicates $\mathcal{Q}_1, \ldots, \mathcal{Q}_{guess}$. This fact is used to define the symbolic encoding of the problem *InferMonome*$(Prog, \mathcal{P})$ in the next subsection.

4.3 Symbolic encoding and complexity of $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$

In this section, we provide a symbolic encoding of $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$. That is, given a program $\mathit{Prog}(\mathit{pre}, \mathit{post}, \mathit{body})$ and a set of predicates $\mathcal{P}$, we will construct a formula $\mathit{SymbInferMonome}(\mathit{Prog}, \mathcal{P})$ which is satisfiable if and only if $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ is false. The formula can be seen as a symbolic encoding of an iterative version of the *FindInv* algorithm that symbolically captures all executions of the algorithm for any input. Finally, we will use the encoding to relate the complexity of $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ to the complexity of $\mathit{Check}(\mathit{Prog}, I)$.

The following notations are used:

- The set of predicates is $\mathcal{P}$, and $n = |\mathcal{P}|$
- For any formula ϕ, ϕ^i represents the formula with any variable x is replaced with a fresh variable x^i. We will use this for each predicate $p \in \mathcal{P}$, *pre*, *post* and $\mathit{wp}(\mathit{body}, p)$.
- The symbols b_p^j for a predicate p denotes that the predicate p was removed from consideration for the invariant in the j-th iteration.

For each $p \in \mathcal{P}$ and $i \in [1, n+1]$, we define

$$\mathit{present}_p^i \triangleq \bigwedge_{j \in [0,i)} \neg b_p^j$$

For each $i \in [1, n]$, we define

$$\begin{aligned} \mathit{iter}_i \triangleq & \vee \mathit{pre}^i \wedge \textstyle\bigwedge_{p \in \mathcal{P}} (b_p^i \implies \neg p^i) \\ & \vee \textstyle\bigwedge_{p \in \mathcal{P}} ((\mathit{present}_p^i \implies p^i) \wedge (b_p^i \implies \neg \mathit{wp}(\mathit{body}, p)^i)) \end{aligned}$$

For each $i \in [1, n+1]$, we define

$$\mathit{check}_i \triangleq \textstyle\bigwedge_{p \in \mathcal{P}} (\mathit{present}_p^i \implies p^i) \wedge \neg \mathit{post}^i$$

Finally, the desired symbolic encoding $\mathit{SymbInferMonome}(\mathit{Prog}, \mathcal{P})$ is the following formula:

$$\begin{aligned} & \wedge\ 1 \leq \mathit{guess} \leq n+1 \\ & \wedge \textstyle\bigwedge_{p \in \mathcal{P}} \neg b_p^0 \\ & \wedge (\textstyle\bigwedge_{i \in [1,n]} i < \mathit{guess} \implies \Sigma_{p \in \mathcal{P}} b_p^i = 1) \wedge (\textstyle\bigwedge_{p \in \mathcal{P}} \Sigma_{i \in [1,n]} (i \leq \mathit{guess} \wedge b_p^i) \leq 1) \\ & \wedge \textstyle\bigwedge_{i \in [1,n]} i < \mathit{guess} \implies \mathit{iter}_i \\ & \wedge \textstyle\bigwedge_{i \in [1,n+1]} i = \mathit{guess} \implies \mathit{check}_i \end{aligned}$$

To get some intuition behind the formula, observe that each of the five conjuncts in this formula resembles closely the five conjuncts in the definition of $\mathit{NoMonomeInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$. The symbols shared across the different $i \in [1, n+1]$ are the b_p^i symbols and *guess*. This is similar to the definition of $\mathit{NoMonomeInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$, where the different models in $\{\mathcal{M}_i\}_i$ only agree on the evaluation of the sets $\mathcal{Q}_i$ and *guess*. The role of the b_p^i variable is precisely to encode the sets $\mathcal{Q}_{i+1}$; $b_p^i = \mathsf{true}$ denotes that $\{p\} = \mathcal{Q}_{i+1} \setminus \mathcal{Q}_i$. The

second conjunct denotes that $\mathcal{Q}_1 = \{\}$ where no predicate has been removed. The third conjunct has two parts. The first part $i < \mathit{guess} \implies \Sigma_{p \in \mathcal{P}} b_p^i = 1$ denotes that $\mathcal{Q}_{i+1}$ and $\mathcal{Q}_i$ differ by exactly one predicate, for any $i < \mathit{guess}$. The second part $\bigwedge_{p \in \mathcal{P}} \Sigma_{i \in [1,n]}(i \leq \mathit{guess} \wedge b_p^i) \leq 1$ denotes that a predicate is removed at most once in any one of the *guess* iterations. Similarly, the fourth conjunct justifies the removal of the predicate in $\mathcal{Q}_{i+1} \setminus \mathcal{Q}_i$.

Theorem 3. *The formula* $\mathit{SymbInferMonome}(\mathit{Prog}, \mathcal{P})$ *is satisfiable if and only if* $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ *is* false.

Proof sketch. We provide a proof sketch in this paper.

"$\implies$": Let us assume that $\mathit{SymbInferMonome}(\mathit{Prog}, \mathcal{P})$ is satisfiable. Given a satisfying model $\mathcal{M}$ to $\mathit{SymbInferMonome}(\mathit{Prog}, \mathcal{P})$, one can split $\mathcal{M}$ into a set of models $\{\mathcal{M}_i\}_i$ where $\mathcal{M}_i$ assigns values to the i-th copy of the variables in ϕ^i, only agreeing on the values of b_p^i and *guess*. Also, the valuation of the b_p^i can be used to construct the sets $\mathcal{Q}_i$; $\mathcal{Q}_{i+1} \leftarrow \mathcal{Q}_i \cup \{p\}$ when b_p^i is true in $\mathcal{M}$.

"$\Longleftarrow$": Let us assume that $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ returns false. Then by Lemma 2, we can construct a model $\mathcal{M}$ by the union of the models $\{\mathcal{M}_i\}_i$, and construct an evaluation for b_p^i as follows: If $\mathcal{Q}_{i+1} \setminus \mathcal{Q}_i = \{p\}$, then assign b_p^i to true. In all other cases, assign b_p^i to be false. The model $\mathcal{M}$ satisfies $\mathit{SymbInferMonome}(\mathit{Prog}, \mathcal{P})$. □

Theorem 4. *For an assertion logic closed under wp and Boolean connectives, the complexity of* $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ *matches the complexity of* $\mathit{Check}(\mathit{Prog}, I)$.

Proof sketch. Since $\mathit{SymbInferMonome}(\mathit{Prog}, \mathcal{P})$ results in a formula which is polynomial in $\mathcal{P}$ and the size of $\mathit{wp}(\mathit{body}, p)$ for any predicate $p \in \mathcal{P}$, the complexity of checking the satisfiability of $\mathit{SymbInferMonome}(\mathit{Prog}, \mathcal{P})$ is simply the complexity of checking assertions in the assertion logic in which $\mathit{Check}(\mathit{Prog}, I)$ is expressed. □

Corollary 1. *If the decision problem for* $\mathit{Check}(\mathit{Prog}, I)$ *is* **Co-NP** *complete, then the decision problem for* $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ *is* **Co-NP** *complete.*

The encoding $\mathit{SymbInferMonome}(\mathit{Prog}, \mathcal{P})$ can also be seen as an alternative algorithm for the $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ problem. However, when the formula $\mathit{SymbInferMonome}(\mathit{Prog}, \mathcal{P})$ is unsatisfiable, it does not readily provide us with the invariant I. We believe this can be extracted from the unsatisfiable core, and we are currently working on it.

5 Clausal abstraction

For any set $\mathcal{R} \subseteq \mathcal{P}$, a *clause* over $\mathcal{R}$ is the formula $\bigvee_{p \in \mathcal{R}} p$. For a program $\mathit{Prog}(\mathit{pre}, \mathit{post}, \mathit{body})$ and a set of predicates $\mathcal{P}$, let us define $\mathit{InferClause}(\mathit{Prog}, \mathcal{P})$ to return true if and only if there exists a $\mathcal{R} \subseteq \mathcal{P}$ such that $\mathit{Check}(\mathit{Prog}, \bigvee_{p \in \mathcal{R}} p)$ is true.

5.1 Dual `Houdini` algorithm

First, let us describe an algorithm for solving the $\mathit{InferClause}(\mathit{Prog}, \mathcal{P})$ problem. Recall that the `Houdini` algorithm for solving the $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ problem starts with the conjunction of all predicates in $\mathcal{P}$ and iteratively removes predicates until a fixpoint is reached. Conversely, the dual `Houdini` algorithm starts with the disjunction of all predicates in $\mathcal{P}$ and removes predicates until a fixpoint is reached. The algorithm invokes $\mathit{FindInv}(\mathcal{P})$ (in Figure 3), only this time using the following definitions of *FailsCheck* and *RemovesPredicate* macros:

$$\begin{aligned}
&\mathit{FailsCheck}(\mathcal{S}, \mathcal{M}) && \triangleq \mathcal{M} \models (\textstyle\bigwedge_{q \in \mathcal{S}} \neg q) \wedge \mathit{pre} \\
&\mathit{RemovesPredicate}(\mathcal{S}, \mathcal{M}, p) && \triangleq \vee \mathcal{M} \models \neg \mathit{post} \wedge p \\
&&& \quad \vee \mathcal{M} \models \neg \mathit{wp}(\mathit{body}, \textstyle\bigvee_{q \in \mathcal{S}} q) \wedge p
\end{aligned}$$

In the remainder of this section, we let $\mathit{FindInv}(\mathcal{P})$ denote the algorithm with the above definitions of $\mathit{FailsCheck}(\mathcal{S}, \mathcal{M})$ and $\mathit{RemovesPredicate}(\mathcal{S}, \mathcal{M}, p)$, rather than those given in Figure 3.

Theorem 5. *The procedure $\mathit{FindInv}(\mathcal{P})$ enjoys the following properties:*

1. *$\mathit{FindInv}(\mathcal{P})$ returns FAIL if and only if $\mathit{InferClause}(\mathit{Prog}, \mathcal{P})$ is* false.
2. *If $\mathit{FindInv}(\mathcal{P})$ returns $\mathit{SUCCESS}(\mathcal{R})$, then for any $\mathcal{S} \subseteq \mathcal{P}$ such that $\mathit{Check}(\mathit{Prog}, \bigvee_{p \in \mathcal{S}} p)$, we have $S \subseteq \mathcal{R}$.*

The theorem signifies that the dual `Houdini` constructs the *weakest* clause I that satisfies $\mathit{Check}(\mathit{Prog}, I)$, as opposed to `Houdini`, which computes the *strongest* monome I that satisfies $\mathit{Check}(\mathit{Prog}, I)$. This is not surprising because `Houdini` solves the problem in the *forward* direction starting from *pre*, whereas the dual algorithm solves the problem *backwards* staring from *post*.

The structure of the rest of the section is similar to Section 4. For brevity, we mostly state the analogues of lemmas, theorems and symbolic encoding in the next two subsections, without details of the proofs.

5.2 Model-theoretic proof

For a $\mathit{guess} \in \mathbb{N}$, an indexed set of models $\{\mathcal{M}_i\}_i$, an indexed set of sets of predicates $\{\mathcal{Q}_i\}_i$, we define a predicate $\mathit{NoClauseInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$, that is true if and only if the following conditions hold:

1. $1 \leq \mathit{guess} \leq |\mathcal{P}| + 1$,
2. $\mathcal{Q}_1 = \{\}$,
3. For $1 \leq i < \mathit{guess}$, $\mathcal{Q}_{i+1} \setminus \mathcal{Q}_i = \{p_i\}$ for some $p_i \in \mathcal{P}$,
4. For each $1 \leq i < \mathit{guess}$, and for $p_i \in \mathcal{Q}_{i+1} \setminus \mathcal{Q}_i$, either $\mathcal{M}_i \models \neg \mathit{post} \wedge p_i$ or $\mathcal{M}_i \models \neg \mathit{wp}(\mathit{body}, \bigvee_{p \in \mathcal{P} \setminus \mathcal{Q}_i} p) \wedge p_i$, and
5. $\mathcal{M}_{\mathit{guess}} \models (\bigwedge_{p \in \mathcal{P} \setminus \mathcal{Q}_{\mathit{guess}}} \neg p) \wedge \mathit{pre}$.

The following three lemmas establish the connection between the *InferClause* problem, the *NoClauseInv* predicate, and the *FindInv* algorithm.

Lemma 5. *If FindInv*$(\mathcal{P})$ *returns FAIL, then NoClauseInv*$(\mathcal{P}, guess, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$ *holds on the values computed by the procedure.*

Lemma 6. *If NoClauseInv*$(\mathcal{P}, guess, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$ *holds for some guess* $\in \mathbb{N}$*, a set of models* $\mathcal{M}_1, \ldots, \mathcal{M}_{guess}$*, and sets of predicates* $\mathcal{Q}_1, \ldots, \mathcal{Q}_{guess}$*, then InferClause*$(Prog, \mathcal{P})$ *is* false*.*

Lemma 7. *If FindInv*$(\mathcal{P})$ *returns SUCCESS*$(\mathcal{R})$*, then Check*$(Prog, \bigvee_{p\in\mathcal{R}} p)$ *is* true*, and therefore InferClause*$(Prog, \mathcal{P})$ *is* true*.*

5.3 Symbolic encoding

Similar to the monomial abstraction, we define the *SymbInferClause*$(Prog, \mathcal{P})$ which is satisfiable if and only if *InferClause*$(Prog, \mathcal{P})$ returns false. The symbolic encoding for clausal abstraction retains the structure of the symbolic encoding for monomial abstraction. The only difference is that the definitions of the predicates $iter_i$ and $check_i$ change as follows:

For each $i \in [1, n]$, we define

$$\begin{aligned} iter_i \stackrel{\triangle}{=} & \vee \neg post^i \wedge \textstyle\bigwedge_{p\in\mathcal{P}}(b_p^i \implies p^i) \\ & \vee \neg wp(s, \textstyle\bigvee_{p\in\mathcal{P}} present_p^i \wedge p)^i \wedge \textstyle\bigwedge_{p\in\mathcal{P}}(b_p^i \implies p^i) \end{aligned}$$

For each $i \in [1, n+1]$, we define

$$check_i \stackrel{\triangle}{=} \textstyle\bigwedge_{p\in\mathcal{P}}(present_p^i \implies \neg p^i) \wedge pre^i$$

Finally, the analogues of Theorem 3, Theorem 4, and Corollary 1 can be shown for the clausal abstraction as well.

Theorem 6. *The formula SymbInferClause*$(Prog, \mathcal{P})$ *is satisfiable if and only if InferClause*$(Prog, \mathcal{P})$ *is* false*.*

Theorem 7. *For an assertion logic closed under wp and Boolean connectives, the complexity of InferClause*$(Prog, \mathcal{P})$ *matches the complexity of Check*$(Prog, I)$*.*

Corollary 2. *If the decision problem for Check*$(Prog, I)$ *is* **Co-NP** *complete, then the decision problem for InferClause*$(Prog, \mathcal{P})$ *is* **Co-NP** *complete.*

6 Conclusions

Formulation of predicate abstraction as a decision problem allows us to infer annotations for programs in a property guided manner, by leveraging off-the-shelf and efficient verification condition generators. In this work, we have studied the

complexity of the decision problem of predicate abstraction relative to the complexity of checking an annotated program. The monomial and clausal restrictions considered in this paper are motivated by practical applications, where most invariants are monomes and a handful of clauses [16]. We have also provided a new algorithm for solving the $\mathit{InferClause}(\mathit{Prog}, I)$ problem.

There are several questions that are still unanswered. We would like to construct an invariant from the unsatisfiable core, when the symbolic encoding of the $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ or $\mathit{InferClause}(\mathit{Prog}, \mathcal{P})$ returns unsatisfiable. It is also not clear what the complexity is for inferring invariants that are either a disjunction of up to c monomes, or a conjunction of up to c clauses, for a fixed constant c.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, pages 203–213, 2001.
2. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering (PASTE '05)*, pages 82–87, 2005.
3. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV '00)*, LNCS 1855, pages 154–169, 2000.
4. J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *Principles of Programming Languages (POPL '09)*, pages 302–314, 2009.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL '77)*, pages 238–252, 1977.
6. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL '78)*, pages 84–96, 1978.
7. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe (FME '01)*, pages 500–517, 2001.
8. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*, LNCS 1254, pages 72–83, June 1997.
9. S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*, LNCS 5403, pages 120–135, 2009.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Principles of Programming Languages (POPL)*, pages 232–244, 2004.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL '02)*, pages 58–70, 2002.
12. H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke. Word level predicate abstraction and refinement for verifying rtl verilog. In *Design Automation Conference (DAC '05)*, pages 445–450. ACM, 2005.
13. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1995.

14. S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, LNCS 2937, pages 267–281, 2004.
15. S. K. Lahiri and S. Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *Principles of Programming Languages (POPL '08)*, pages 171–182, 2008.
16. S. K. Lahiri, S. Qadeer, J. P. Galeotti, J. W. Voung, and T. Wies. Intra-module inference. In *Computer-Aided Verification (CAV '09)*, LNCS (to appear), July 2009.
17. S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):1–50, 1998.
18. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In *Static Analysis Symposium (SAS '04)*, LNCS 3148, pages 53–68, 2004.
19. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '05)*, LNCS 3385, pages 25–41, 2005.
20. Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://goedel.cs.uiowa.edu/smtlib/`.
21. A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *Programming Language Design and Implementation (PLDI '05)*, pages 281–294. ACM, 2005.

Appendix

Proof of Lemma 2

Proof. The first four conditions of $\mathit{NoMonomeInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$ are easily satisfied by construction. The fifth condition can be shown by observing that $\mathcal{Q}_{\mathit{guess}} \cup \mathcal{R} = \mathcal{P}$ is a precondition to $\mathit{FindInvAux}$. □

Proof of Lemma 3

Proof. Let us assume that there exists guess, $\{\mathcal{M}_i\}_{i\in[1,\mathit{guess}]}$, $\{\mathcal{Q}_i\}_{i\in[1,\mathit{guess}]}$ satisfying $\mathit{NoMonomeInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$. We will show that in such a case, $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ returns false.

We will prove this by contradiction. Let us assume that there is a $\mathcal{R} \subseteq \mathcal{P}$ such that $\mathit{Check}(\mathit{Prog}, \bigwedge_{p\in\mathcal{R}} p)$ holds. Let $I = \bigwedge_{p\in\mathcal{R}} p$. We claim that $\mathcal{R} \cap \mathcal{Q}_{\mathit{guess}} = \{\}$. We will prove this by induction on i for $1 \leq i \leq \mathit{guess}$, showing $\mathcal{R} \cap \mathcal{Q}_i = \{\}$. The base case for $i = 1$ holds vacuously since $\mathcal{Q}_1 = \{\}$. Let us assume that the induction hypothesis holds for all $j \leq i$. Consider the set $\{p_i\} = \mathcal{Q}_{i+1} \setminus \mathcal{Q}_i$. We show that p_i cannot be in $\mathcal{R}$. Consider the two cases how p_i gets removed.

- If $\mathcal{M}_i \models \mathit{pre} \wedge \neg p_i$, then we know that $\not\models \mathit{pre} \implies p_i$. If $p_i \in \mathcal{R}$, then $\models \mathit{pre} \implies p_i$, which is a contradiction.

– On the other hand, suppose $\mathcal{M}_i \models (\bigwedge_{p \in \mathcal{P} \setminus \mathcal{Q}_i} p) \wedge \neg wp(body, p_i)$. By induction hypothesis, we know that $\mathcal{R} \cap \mathcal{Q}_i = \{\}$, therefore $\mathcal{R} \subseteq \mathcal{P} \setminus \mathcal{Q}_i$. This implies $\mathcal{M}_i \models (\bigwedge_{p \in \mathcal{R}} p) \wedge \neg wp(body, p_i)$. If $p_i \in \mathcal{R}$, then we can conclude that $\mathcal{M}_i \models (\bigwedge_{p \in \mathcal{R}} p) \wedge \neg (\bigwedge_{p \in \mathcal{R}} wp(body, p))$. Since wp distributes over $\wedge$, we have $\bigwedge_{p \in \mathcal{R}} wp(body, p)) = wp(body, \bigwedge_{p \in \mathcal{R}} p)$ and thus $\mathcal{M}_i \models I \wedge \neg wp(body, I)$. Since I satisfies $Check(Prog, I)$, we have arrived at a contradiction.

Having shown that $\mathcal{R} \cap \mathcal{Q}_{guess} = \{\}$, we know that $\mathcal{R} \subseteq \mathcal{P} \setminus \mathcal{Q}_{guess}$. Since $\mathcal{M}_{guess} \models (\bigwedge_{p \in \mathcal{P} \setminus \mathcal{Q}_{guess}} p) \wedge \neg post$, it implies that $\mathcal{M}_{guess} \models (\bigwedge_{p \in \mathcal{R}} p) \wedge \neg post$, which in turn implies $\not\models I \implies post$, which contradicts our assumption that $Check(Prog, I)$. □

Proof of Lemma 4

Proof. Let *FindInvAux* return $SUCCESS(\mathcal{R})$ for an argument $\mathcal{R}$ to *FindInvAux*. Since both the if branches are not taken, all the following conditions hold:

1. $\models (\bigwedge_{q \in \mathcal{R}} q) \implies post$.
2. For each $q \in \mathcal{R}$, $\models pre \implies q$, and therefore $\models pre \implies (\bigwedge_{q \in \mathcal{R}} q)$.
3. For each $q \in \mathcal{R}$, $\models (\bigwedge_{p \in \mathcal{R}} p) \implies wp(body, q)$. Since wp distributes over $\wedge$, this implies $\models (\bigwedge_{p \in \mathcal{R}} p) \implies wp(body, (\bigwedge_{p \in \mathcal{R}} p))$.

These conditions mean that $Check(Prog, \bigwedge_{q \in \mathcal{R}} q)$ holds. □

Proof of Lemma 1

Proof. Part (1) is proved easily by combining Lemmas 2, 3, and 4.

To prove part (2), we establish the following precondition for *FindInvAux* procedure: For any set of predicates $\mathcal{S} \subseteq \mathcal{P}$ such that $Check(Prog, \bigwedge_{p \in \mathcal{S}} p)$ holds, $\mathcal{S} \cap \mathcal{Q}_{guess} = \{\}$. The proof follows by induction on *guess* similar to the proof of Lemma 3. Similarly, $\mathcal{R} \cup \mathcal{Q}_{guess} = \mathcal{P}$ is another precondition for *FindInvAux*. Therefore, whenever *FindInvAux* returns $SUCCESS(\mathcal{R})$, $(\bigwedge_{p \in \mathcal{R}} p)$ is the strongest monomial invariant over $\mathcal{P}$ that satisfies the program. □