Checking Observational Purity of Procedures

Himanshu Arora[1], Raghavan Komondoor[1], and G. Ramalingam[2(✉)]

[1] Indian Institute of Science, Bangalore, India
{himanshua,raghavan}@iisc.ac.in
[2] Microsoft Research, Bellevue, WA, USA
grama@microsoft.com

Abstract. Verifying whether a procedure is *observationally pure* (that is, it always returns the same result for the same input argument) is challenging when the procedure uses mutable (private) global variables, e.g., for memoization, and when the procedure is recursive.

We present a deductive verification approach for this problem. Our approach encodes the procedure's code as a logical formula, with recursive calls being modeled using a mathematical function symbol *assuming that the procedure is observationally pure*. Then, a theorem prover is invoked to check whether this logical formula agrees with the function symbol referred to above in terms of input-output behavior for all arguments. We prove the soundness of this approach.

We then present a conservative approximation of the first approach that reduces the verification problem to one of checking whether a quantifier-free formula is satisfiable and prove the soundness of the second approach.

We evaluate our approach on a set of realistic examples, using the Boogie intermediate language and theorem prover. Our evaluation shows that the invariants are easy to construct manually, and that our approach is effective at verifying observationally pure procedures.

1 Introduction

A procedure in an imperative programming language is said to be *observationally pure* (OP) if for each specific argument value it has a specific return value, across all possible sequences of calls to the procedure, irrespective of what other code runs between these calls. In other words, the input-output behavior of an OP procedure mimics a mathematical function.

A deterministic procedure that does not read any pre-existing state other than its arguments is trivially OP. However, it is common for procedures to update and read global variables, typically for performance optimization, while still being OP. In this paper, we focus on the problem of checking observational purity of procedures that read and write global variables, especially in the presence of recursion, which makes the problem harder.

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 228–243, 2019.
https://doi.org/10.1007/978-3-030-16722-6_13

```
1
2  int g := -1;
3  int lastN := 0;
4  int factCache( int n) {
5    if(n <= 1) {
6      result := 1;
7    } else if (g != -1 && n == lastN) {
8      result := g;
9    } else {
10     g = n * factCache( n - 1 );
11     lastN = n;
12     result := g;
13   }
14   return result;
15 }
```

Listing 1.1. Procedure factCache: returns n!, and memoizes most recent result.

Motivating Example. We use procedure 'factCache' in Listing 1.1 as our running example. It returns `n`! for a given argument `n`, and caches the return value of the most recent call. It uses two *private global* variables, `g` and `lastN`, to implement the caching. `g` is initialized to -1. After the first call to the procedure onwards, `g` stores the return value of the most recent call, and `lastN` stores the argument of the most recent call. Clearly this procedure is OP, and mimics the input-output behavior of a factorial procedure that does not cache any results.

Proposed Approach. Our approach is based on Floyd-Hoare logic, which typically requires a specification of the procedure to be provided. One candidate specification would be a full functional specification of the procedure. If the user specifies that `factCache` realizes `n`!, then the verifier could replace Line 10 in the code with '`g = n * (n-1)!`'. This, on paper, is sufficient to assert that Line 12 always assigns `n`! to `result`. However, to establish that Line 8 also does the same, an invariant would need to be provided that describes the possible values of `g` before an invocation to the procedure. In our example, a suitable invariant would be '$(\texttt{g} = -1) \vee (\texttt{g} = \texttt{lastN}!)$'. The verifier would also need to verify that at the procedure's exit the invariant is re-established. Lines 10–12, with the recursive call replaced by $(\texttt{n}-1)!$, suffices on paper to re-establish the invariant.

The candidate approach described above, while plausible, suffers from two weaknesses. First, a mathematical specification of the function being computed may be complex and non-trivial to write. (Note, for example, that `factCache` is defined for negative integers while factorial is not. Thus, the previous candidate specification is actually incorrect for this edge case.) Second, the underlying theorem prover would need to prove complex arithmetic properties, e.g., that `n` * $(\texttt{n}-1)!$ is equal to `n`!. Complex proofs such as this may be beyond the scope of many existing theorem provers.

Our key insight is to sidestep the challenges mentioned by introducing a function symbol, say *factCache*, and replacing the recursive call for the purposes of verification with this function symbol. (Note that we reuse the same symbol for two purposes, which may be slightly confusing here. One denotes the

procedure name, while the other denotes a function symbol for use in a logical formula. The italicized name here denotes the function symbol.) Intuitively, *factCache* represents the mathematical function that the given procedure mimics *if* the procedure is OP. In our example, Line 10 would become 'g = n * *factCache*(n − 1)'. This step needs no human involvement. The approach needs an invariant; however, in a novel manner, we allow the invariant also to refer to *factCache*. In our example, a suitable invariant would be '(g = −1) ∨ (g = lastN * *factCache*(lastN − 1))'. This sort of invariant is relatively easy to construct; e.g., a human could arrive at it just by looking at Line 2 and with a local reasoning on Lines 10 and 11. Given this invariant, (a) a theorem prover could infer that the condition in Line 7 implies that Line 8 necessarily copies the value of 'n * *factCache*(n − 1)' into 'result'. Due to the transformation to Line 10 mentioned above, (b) the theorem prover can infer that Line 12 also does the same. Note that since these two expressions are syntactically identical, a theorem prover can easily establish that they are equal in value. Finally, since Line 6 is reached under a different condition than Lines 8 and 12, the verifier has finished establishing that the procedure always returns the same expression in n for any given value of n.

Similarly, using the modified Line 10 mentioned above and from Line 11, the prover can re-establish that g is equal to 'lastN * *factCache*(lastN − 1)' when control reaches Line 12. Hence, the necessary step of proving the given invariant to be a valid invariant is also complete.

Note, the effectiveness of the approach depends on the nature of the given invariant. For instance, if the given invariant was '(g = −1) ∨ (g = lastN!)', which is also technically correct, then the theorem prover may not be able to establish that in Lines 8 and 12 the variable 'g' always stores the same expression in n. However, it is our claim that in fact it is the invariant '(g = − 1) ∨ (g = lastN * *factCache*(lastN − 1))' that is easier to infer by a human or by a potential tool, as justified by us two paragraphs above.

Salient Aspects of Our Approach. This paper makes two significant contributions. First, it tackles the circularity problem that arises due to the use of a presumed-to-be OP procedure in assertions and invariants and the use of these invariants in proving the procedure to be OP. This requires us to prove the soundness of an approach that *simultaneously* verifies observational purity as well the validity of invariants (as they cannot be decoupled).

Secondly, we show that a direct approach to this verification problem (which we call the existential approach) reduces it to a problem of verifying that a logical formula is a tautology. The structure of the generated formula, however, makes the resulting theorem prover instances hard. We show how a conservative approximation can be used to convert this hard problem into an easier problem of checking satisfiability of a quantifier-free formula, which is something within the scope of state-of-the-art theorem provers.

The most closely related previous approaches are by Barnett et al. [1,2], and by Naumann [3]. These approaches check observational purity of procedures that maintain mutable global state. However, none of these approaches use a function

```
   L ∈ Lib   ::= g := c P̄
   P ∈ Proc  ::= p (x) { S; return y }
   S ∈ Stmt  ::= x := e | x := p(y) | S ; S | if (e) then S else S
   e ∈ Expr  ::= c | x | e op e | unop e
  op ∈ Ops   ::= + | - | / | * | % | > | < | == | ∧ | ∨
unop ∈ UnOps ::= ¬
   x, y ∈ LocalId ∪ GlobalId, g ∈ GlobalId, c ∈ 𝒱, p ∈ ProcId
```

Fig. 1. Programming language syntax and meta-variables

symbol in place of recursive calls or within invariants. Therefore, it is not clear that these approaches can verify recursive procedures. Barnett et al., in fact, state "there is a circularity - it would take a delicate argument, and additional conditions, to avoid unsoundness in this case". To the best of our knowledge ours is the first paper to show that it is feasible to check observational purity of procedures that maintain mutable global state for optimization purposes and that make use of recursion.

Being able to verify that a procedure is OP has many potential applications. The most obvious one is that OP procedures can be memoized. That is, input-output pairs can be recorded in a table, and calls to the procedure can be elided whenever an argument is seen more than once. This would not change the semantics of the overall program that calls the procedure, because the procedure always returns the same value for the same argument (and mutates only private global variables). Another application is that if a loop contains a call to an OP procedure, then the loop can be parallelized (provided the procedure is modified to access and update its private global variables in a single atomic operation).

The rest of this paper is structured as follows. Section 2 introduces the core programming language that we address. Section 3 provides formal semantics for our language, as well as definitions of invariants and observational purity. Section 4 describes our approach formally. Section 5 discusses an approach for generating an invariant automatically in certain cases. Section 6 describes evaluation of our approach on a few realistic examples. Section 7 describes related work. More details about the proofs and the examples can be found in [4].

2 Language Syntax

In this paper, we assume that the input to the purity checker is a library consisting of one or more procedures, with shared state consisting of one or more variables that are private to the library. We refer to these variables as "global" variables to indicate that they retain their values across multiple invocations of the library procedures, but they cannot be accessed or modified by procedures outside the library (that is, the clients of the library).

In Fig. 1, we present the syntax of a simple programming language that we address in this paper. Given the foundational focus of this work, we keep the

programming language very simple, but the ideas we present can be generalized. A `return` statement is required in each procedure, and is permitted only as the last statement of the procedure. The language does not contain any looping construct. Loops can be modelled as recursive procedures. The formal parameters of a procedure are readonly and cannot be modified within the procedure. We omit types from the language. We permit only variables of primitive types. In particular, the language does not allow pointers or dynamic memory allocation. Note that expressions are pure (that is, they have no side effects) in this language, and a procedure call is not allowed in an expression. Each procedure call is modelled as a separate statement.

For simplicity of presentation, without loss of conceptual generality, we assume that the library consists of a single (possibly recursive) procedure, with a single formal parameter. In the sequel, we will use the symbol P (as a metavariable) to represent this library procedure, p (as a metavariable) to represent the *name* of this procedure, and will assume that the name of the formal parameter is `n`. If the procedure is of the form "p `(n) { S; return r }`", we refer to `r` as the *return* variable, and refer to "`S; return r`" as the *procedure body* and denote it as body(P). The library also contains, outside of the procedure's code, a sequence of initializing declarations of the global variables used in the procedure, of the form "`g1 := c1`; ...; `gN := cN`". These initializations are assumed to be performed once during any execution of the client application, just before the first call to the procedure P is placed by the client application.

Throughout this paper we use the word 'procedure' to refer to the library procedure P, and use the word 'function' to refer to a mathematical function.

3 A Semantic Definition of Purity

In this section, we formalize the input-output semantics of the procedure P as a relation $\rightsquigarrow_{\mathrm{P}}$, where $n \rightsquigarrow_{\mathrm{P}} r$ indicates that an invocation of P with input n may return a result of r. The procedure is defined to be observationally pure if the relation $\rightsquigarrow_{\mathrm{P}}$ is a (partial) function: that is, if $n \rightsquigarrow_{\mathrm{P}} r_1$ and $n \rightsquigarrow_{\mathrm{P}} r_2$, then $r_1 = r_2$.

The object of our analysis is a single-procedure library, not the entire (client) application. (Our approach can be generalized to handle multi-procedure libraries.) The result of our analysis is valid for any client program that uses the procedure/library. The only assumptions we make are: (a) The shared state used by the library (the global variables) are private to the library and cannot be modified by the rest of the program, and (b) The client invokes the library procedures sequentially: no concurrent or overlapping invocations of the library procedures by a concurrent client are permitted.

The following semantic formalism is motivated by the above observations. It can be seen as the semantics of the so-called "most general sequential client" of procedure P, which is the program: `while (*) x =` p `(random());`. The executions (of P) produced by this program include all possible executions (of P) produced by all sequential clients.

Let G denote the set of global variables. Let L denote the set of local variables. Let $\mathcal{V}$ denote the set of numeric values (that the variables can take). An element

$$[\text{ASSIGN-LOCAL}]\ \frac{\mathtt{x} \in L \qquad (\rho_\ell \uplus \rho_g, \mathtt{e}) \Downarrow v}{((\mathtt{x\ :=\ e;\ S}, \rho_\ell)\gamma, \rho_g) \rightarrow_{\mathrm{P}} ((\mathtt{S}, \rho_\ell[\mathtt{x} \mapsto v])\gamma, \rho_g)}$$

$$[\text{ASSIGN-GLOBAL}]\ \frac{\mathtt{x} \in G \qquad (\rho_\ell \uplus \rho_g, \mathtt{e}) \Downarrow v}{((\mathtt{x\ :=\ e;\ S}, \rho_\ell)\gamma, \rho_g) \rightarrow_{\mathrm{P}} ((\mathtt{S}, \rho_\ell)\gamma, \rho_g[\mathtt{x} \mapsto v])}$$

$$[\text{SEQ}]\ (((\mathtt{S}_1; \mathtt{S}_2); \mathtt{S}_3, \rho_\ell)\gamma, \rho_g) \rightarrow_{\mathrm{P}} ((\mathtt{S}_1; (\mathtt{S}_2; \mathtt{S}_3), \rho_\ell)\gamma, \rho_g)$$

$$[\text{IF-TRUE}]\ \frac{(\rho_\ell \uplus \rho_g, \mathtt{e}) \Downarrow \mathtt{true}}{(((\mathtt{if\ (e)\ then\ S_1 else\ S_2);\ S_3}, \rho_\ell)\gamma, \rho_g) \rightarrow_{\mathrm{P}} ((\mathtt{S_1;\ S_3}, \rho_\ell)\gamma, \rho_g)}$$

$$[\text{IF-FALSE}]\ \frac{(\rho_\ell \uplus \rho_g, \mathtt{e}) \Downarrow \mathtt{false}}{(((\mathtt{if\ (e)\ then\ S_1\ else\ S_2);\ S_3}, \rho_\ell)\gamma, \rho_g) \rightarrow_{\mathrm{P}} ((\mathtt{S_2;\ S_3}, \rho_\ell)\gamma, \rho_g)}$$

$$[\text{CALL}]\ \frac{(\rho_\ell \uplus \rho_g, \mathtt{e}) \Downarrow v \qquad \mathrm{P} = p(\mathtt{n})\ \mathtt{S}_1}{((\mathtt{y\ :=}\ p(\mathtt{e});\ \mathtt{S}_2, \rho_\ell)\gamma, \rho_g) \rightarrow_{\mathrm{P}} ((\mathtt{S}_1, [n \mapsto v])(\mathtt{y\ :=}\ p(\mathtt{e});\ \mathtt{S}_2, \rho_\ell)\gamma, \rho_g)}$$

$$[\text{RETURN}]\ \frac{(\rho_\ell \uplus \rho_g, \mathtt{r}) \Downarrow v}{((\mathtt{return\ r}, \rho_\ell)(\mathtt{y\ :=}\ p(\mathtt{e});\ \mathtt{S}, \rho'_\ell)\gamma, \rho_g) \rightarrow_{\mathrm{P}} (\mathtt{S}, \rho'_\ell[\mathtt{y} \mapsto v])\gamma, \rho_g)}$$

$$[\text{TOP-LEVEL-CALL}]\ \frac{\mathtt{B} = \mathrm{body}(\mathrm{P}) \qquad v \in \mathcal{V}}{([], \rho_g) \rightarrow_{\mathrm{P}} ([(\mathtt{B}, [n \mapsto v])], \rho_g)}$$

$$[\text{TOP-LEVEL-RETURN}]\ \frac{}{([(\mathtt{return\ r}, \rho_\ell)], \rho_g) \rightarrow_{\mathrm{P}} ([], \rho_g)}$$

Fig. 2. A small-step operational semantics for our language, represented as a relation $\sigma_1 \rightarrow_{\mathrm{P}} \sigma_2$. A state σ_i is a configuration of the form $((\mathtt{S}, \rho_\ell)\gamma, \rho_g)$ where $\mathtt{S}$ captures statements to be executed in current procedure, ρ_ℓ assigns values to local variables, γ is the call-stack (excluding current procedure), and ρ_g assigns values to global variables.

$\rho_g \in \Sigma_G = G \hookrightarrow \mathcal{V}$ maps global variables to their values. An element $\rho_\ell \in \Sigma_L = L \hookrightarrow \mathcal{V}$ maps local variables to their values. We define a *local continuation* to be a statement sequence ending with a `return` statement. We use a local continuation to represent the part of the procedure body that still remains to be executed. Let Σ_C represent the set of local continuations. The set of runtime states (or simply, *states*) is defined to be $(\Sigma_C \times \Sigma_L)^* \times \Sigma_G$, where the first component represents a runtime stack, and the second component the values of global variables. We denote individual states using symbols $\sigma, \sigma_1, \sigma_i$, etc. The runtime stack is a sequence, each element of which is a pair $(\mathtt{S}, \rho_\ell)$ consisting of the remaining procedure fragment $\mathtt{S}$ to be executed and the values of local variables ρ_ℓ. We write $(\mathtt{S}, \rho_\ell)\gamma$ to indicate a stack where the topmost entry is $(\mathtt{S}, \rho_\ell)$ and γ represents the remaining part of the stack.

We say that a state $((\mathtt{S}, \rho_\ell)\gamma, \rho_g)$ is an *entry-state* if its location is at the procedure entry point (*i.e.*, if $\mathtt{S}$ is the entire body of the procedure), and we say that it is an *exit-state* if its location is at the procedure exit point (*i.e.*, if $\mathtt{S}$ consists of just a `return` statement).

A procedure P determines a single-step execution relation $\rightarrow_{\mathrm{P}}$, where $\sigma_1 \rightarrow_{\mathrm{P}} \sigma_2$ indicates that execution proceeds from state σ_1 to state σ_2 in a single step. Figure 2 defines this semantics. The semantics of evaluation of a side-effect-free expression is captured by a relation $(\rho, \mathtt{e}) \Downarrow v$, indicating that the expression `e` evaluates to value v in an *environment* ρ (by *environment*, we mean an element of $(G \cup L) \hookrightarrow \mathcal{V}$). We omit the definition of this relation, which is straightforward. We use the notation $\rho_1 \uplus \rho_2$ to denote the union of two disjoint maps ρ_1 and ρ_2.

Note that most rules captures the usual semantics of the language constructs. The last two rules, however, capture the semantics of the most-general sequential client explained previously: when the call stack is empty, a new invocation of the procedure may be initiated (with an arbitrary parameter value).

Note that all the following definitions are parametric over a given procedure P. E.g., we will use the word "execution" as shorthand for "execution of P".

We define an *execution* (of P) to be a sequence of states $\sigma_0\sigma_1 \cdots \sigma_n$ such that $\sigma_i \rightarrow_{\mathrm{P}} \sigma_{i+1}$ for all $0 \leq i < n$. Let σ_{init} denote the *initial state* of the library; i.e., this is the element of Σ_G that is induced by the sequence of initializing declarations of the library, namely, "`g1 := c1`; ...; `gN := cN`". We say that an execution $\sigma_0\sigma_1 \cdots \sigma_n$ is a *feasible* execution if $\sigma_0 = \sigma_{\mathrm{init}}$. Note, intuitively, a feasible execution corresponds to the sequence of states visited within the library across all invocations of the library procedure over the course of a single execution of the most-general client mentioned above; also, since the most-general client supplies a random parameter value to each invocation of P, in general multiple feasible executions of the library may exist.

We define a *trace* (of P) to be a substring $\pi = \sigma_0 \cdots \sigma_n$ of a feasible execution such that: (a) σ_0 is entry-state (b) σ_n is an exit-state, and (c) σ_n corresponds to the return from the invocation represented by σ_0. In other words, a trace is a state sequence corresponding to a single invocation of the procedure. A trace may contain within it nested sub-traces due to recursive calls, which are themselves traces. Given a trace $\pi = \sigma_0 \cdots \sigma_n$, we define *initial*$(\pi)$ to be σ_0, *final*(π) to be σ_n, *input*(π) to be value of the input parameter in *initial*(π), and *output*(π) to be the value of the return variable in *final*(π).

We define the relation $\rightsquigarrow_{\mathrm{P}}$ to be $\{(\mathit{input}(\pi), \mathit{output}(\pi)) \mid \pi \text{ is a trace of P}\}$.

Definition 1 (Observational Purity). *A procedure* P *is said to be* observationally pure *if the relation* $\rightsquigarrow_{\mathrm{P}}$ *is a (partial) function: that is, if for all* n, r_1, r_2, *if* $n \rightsquigarrow_{\mathrm{P}} r_1$ *and* $n \rightsquigarrow_{\mathrm{P}} r_2$, *then* $r_1 = r_2$.

Logical Formula and Invariants. Our methodology makes use of *logical formulae* for different purposes, including to express a given *invariant*. Our logical formulae use the local and global variables in the library procedure as free variables, use the same operators as allowed in our language, and make use of universal as well as existential quantification. Given a formula φ, we write $\rho \models \varphi$ to denote that φ evaluates to true when its free variables are assigned values from the environment ρ.

As discussed in Sect. 1, one of our central ideas is to allow the names of the library procedures to be referred to in the invariant; *e.g.*, our running example becomes amenable to our analysis using an invariant such as '$(g = -1) \vee (g = lastN * factCache(lastN - 1))$'. We therefore allow the use of library procedure names (in our simplified presentation, the name p) as free variables in logical formulae. Correspondingly, we let each environment ρ map each procedure name to a mathematical function in addition to mapping variables to numeric values, and extend the semantics of $\rho \models \varphi$ by substituting the values of both variables and procedure names in φ from the environment ρ.

Given an environment ρ, a procedure name p, and a mathematical function f, we will write $\rho[p \mapsto f]$ to indicate the updated environment that maps p to the value f and maps every other variable x to its original value $\rho[x]$. We will write $(\rho, f) \models \varphi$ to denote that $\rho[p \mapsto f] \models \varphi$.

Given a state $\sigma = ((\mathtt{S}, \rho_\ell)\gamma, \rho_g)$, we define $\text{env}(\sigma)$ to be $\rho_\ell \uplus \rho_g$, and given a state $\sigma = ([], \rho_g)$, we define $\text{env}(\sigma)$ to be just ρ_g. We write $(\sigma, f) \models \varphi$ to denote that $(\text{env}(\sigma), f) \models \varphi$. For any execution or trace π, we write $(\pi, f) \models \varphi$ if for every entry-state and exit-state σ in π, $(\sigma, f) \models \varphi$. We now introduce another definition of observational purity.

Definition 2 (Observational Purity wrt an Invariant). *Given an invariant φ^{inv}, a library procedure* P *is said to satisfy* $\text{pure}(\varphi^{inv})$ *if there exists a function f such that for every trace π of* P*, $output(\pi) = f(input(\pi))$ and $(\pi, f) \models \varphi^{inv}$.*

It is easy to see that if procedure P satisfies $\text{pure}(\varphi^{inv})$ wrt any given candidate invariant φ^{inv}, then P is observationally pure as per Definition 1.

4 Checking Purity Using a Theorem Prover

In this section we provide two different approaches that, given a procedure P and a candidate invariant φ^{inv}, use a theorem prover to check conservatively whether procedure P satisfies $\text{pure}(\varphi^{inv})$.

4.1 Verification Condition Generation

We first describe an adaptation of standard verification-condition generation techniques (*e.g.*, see [5]) that we use as a common first step in both our approaches. Given a procedure P, a candidate invariant φ^{inv}, our goal is to compute a pair $(\varphi^{post}, \varphi^{vc})$ where φ^{post} is a postcondition describing the state that exists after an execution of body(P) starting from a state that satisfies φ^{inv}, and φ^{vc} is a verification-condition that must hold true for the execution to satisfy its invariants and assertions.

We first transform the procedure body as below to create an internal representation that is input to the postcondition and verification condition generator. In the internal representation, we allow the following extra forms of statements (with their usual meaning): `havoc(x)`, `assume e`, and `assert e`.

1. For any assignment statement "`x := e`" where `e` contains `x`, we introduce a new temporary variable `t` and replace the assignment statement with "`t := e; x := t`".
2. For every procedure invocation "`x := p(y)`", we first ensure that `y` is a local variable (by introducing a temporary if needed). We then replace the statement by the code fragment "`assert` φ^{inv}`; havoc(g1); ... havoc(gN); assume` $\varphi^{inv} \wedge$ `x = p(y)`", where `g1` to `gN` are the global variables.
   Note that the procedure call has been eliminated, and replaced with an "assume" expression that refers to the function symbol p. In other words, there are no procedure calls in the transformed procedure.
3. We replace the "`return x`" statement by "`assert` φ^{inv}". Note that we intentionally do *not* assert that the return value equals $p(n)$.

Let $\text{TB}(\text{P}, \varphi^{inv})$ denote the transformed body of procedure P obtained as above.

$$
\begin{array}{ll}
\text{POST}(\varphi^{pre}, \texttt{x := e}) & = (\exists \texttt{x}.\varphi^{pre}) \wedge (\texttt{x = e}) \text{ (if } \texttt{x} \notin \text{vars}(\texttt{e})) \\
\text{POST}(\varphi^{pre}, \texttt{havoc(x)}) & = \exists \texttt{x}.\varphi^{pre} \\
\text{POST}(\varphi^{pre}, \texttt{assume e}) & = \varphi^{pre} \wedge \texttt{e} \\
\text{POST}(\varphi^{pre}, \texttt{assert e}) & = \varphi^{pre} \\
\text{POST}(\varphi^{pre}, \texttt{S}_1; \texttt{S}_2) & = \text{POST}(\text{POST}(\varphi^{pre}, \texttt{S}_1), \texttt{S}_2) \\
\text{POST}(\varphi^{pre}, \texttt{if e then S}_1 \texttt{ else S}_2) & = \text{POST}(\varphi^{pre} \wedge \texttt{e}, \texttt{S}_1) \vee \text{POST}(\varphi^{pre} \wedge \neg \texttt{e}, \texttt{S}_2)
\end{array}
$$

$$
\begin{array}{ll}
\text{VC}(\varphi^{pre}, \texttt{assert e}) & = (\varphi^{pre} \Rightarrow e) \\
\text{VC}(\varphi^{pre}, \texttt{S}_1; \texttt{S}_2) & = \text{VC}(\varphi^{pre}, \texttt{S}_1) \wedge \text{VC}(\text{POST}(\varphi^{pre}, \texttt{S}_1), \texttt{S}_2) \\
\text{VC}(\varphi^{pre}, \texttt{if e then S}_1 \texttt{ else S}_2) & = \text{VC}(\varphi^{pre} \wedge \texttt{e}, \texttt{S}_1) \wedge \text{VC}(\varphi^{pre} \wedge \neg \texttt{e}, \texttt{S}_2) \\
\text{VC}(\varphi^{pre}, \texttt{S}) & = \text{true(for all other S)}
\end{array}
$$

$$
\text{POSTVC}(\text{P}, \varphi^{inv}) = (\text{POST}(\ \varphi^{inv}, \text{TB}(\text{P}, \varphi^{inv})), \text{VC}(\varphi^{inv}, \text{TB}(\text{P}, \varphi^{inv})) \wedge (\text{INIT}(\text{P}) \Rightarrow \varphi^{inv}))
$$

Fig. 3. Generation of verification-condition and postcondition.

We then compute postconditions as formally described in Fig. 3. This lets us compute for each program point ℓ in the procedure, a condition φ_ℓ that describes what we expect to hold true when execution reaches ℓ if we start executing the procedure in a state satisfying φ^{inv} and if every recursive invocation of the procedure also terminates in a state satisfying φ^{inv}. We compute this using the standard rules for the postcondition of a statement. For an assignment statement "`x := e`", we use existential quantification over `x` to represent the value of `x` prior to the execution of the statement. If we rename these existentially quantified variables with unique new names, we can lift all the existential quantifiers to the outermost level. When transformed thus, the condition φ_ℓ takes the form $\exists x_1 \cdots x_n.\varphi$, where φ is quantifier-free and $x_1, \cdots, x_n$ denote intermediate values of variables along the execution path from procedure-entry to program point ℓ.

We compute a verification condition φ^{vc} that represents the conditions we must check to ensure that an execution through the procedure satisfies its obligations: namely, that the invariant holds true at every call-site and at procedure-exit. Let ℓ denote a call-site or the procedure-exit. We need to check that

```
1  g := -1;
2  lastN := 0;
3  factCache (n) {
4    if(n <= 1) {
5      result := 1;
6    } else if (g != -1 && n == lastN) {
7      result := g;
8    } else {
9      t1 := n-1;
10     // t2 := factCache(t1);
11     assert φ^inv;
12     havoc (g); havoc (lastN);
13     assume φ^inv ∧ (t2 = factCache(t1));
14     g := n * t2;
15     lastN := n;
16     result := g;
17   }
18   // return result;
19   assert φ^inv;
20 }
```

Listing 1.2. Procedure factCache from Listing 1.1 transformed to incorporate a supplied candidate invariant φ^{inv}.

$\varphi_\ell \Rightarrow \varphi^{inv}$ holds. Thus, the generated verification condition essentially consists of the conjunction of this check over all call-sites and procedure-exit.

Finally, the function POSTVC computes the postcondition and verification condition for the entire procedure as shown in Fig. 3. (Thus, it returns a pair of formulae.) Note that this function also adds the check that the initial state must satisfy φ^{inv} to the verification condition (as the basis condition for induction). INIT(P) is basically the formula "`g1 = c1` $\wedge \ldots$ `gN = cN`" (see Sect. 2).

Example. We now illustrate the postcondition and verification condition generated from our factorial example presented in Listing 1.1. Listing 1.2 shows the example expressed in our language and transformed as described earlier (using function TB), using a supplied candidate invariant φ^{inv}.

Figure 4 illustrates the computation of postcondition and verification condition from this transformed example. In this figure, we use φ_{cs}^{pre} to denote the precondition computed to hold just before the recursive callsite, and φ_{cs}^{post} to denote the postcondition computed to hold just after the recursive callsite. The postcondition φ^{post} (at the end of the procedure body) is itself a disjunction of three path-conditions representing execution through the three different paths in the program. In this illustration, we have simplified the logical conditions by omitting useless existential quantifications (that is, any quantification of the form $\exists x.\psi$ where x does not occur in ψ). Note that the existentially quantified `g` and `lastN` in φ_{cs}^{post} denote the values of these globals before the recursive call. Similarly, the existentially quantified `g` and `lastN` in φ_3^{path} denote the values of these globals when the recursive call terminates, while the free variables `g` and `lastN` denote the final values of these globals.

$$
\begin{aligned}
\textsc{init}(\textsc{p}) &= \texttt{(g = -1)} \wedge \texttt{(lastN = 0)} \\
\varphi_1^{path} &= \varphi^{inv} \wedge \texttt{(n <= 1)} \wedge \texttt{(result = 1)} \\
\varphi_2^{path} &= \varphi^{inv} \wedge \neg\texttt{(n <= 1)} \wedge \texttt{(g != 1)} \wedge \texttt{(n = lastN)} \wedge \texttt{(result = g)} \\
\varphi_{cs}^{pre} &= \varphi^{inv} \wedge \neg\texttt{(n <= 1)} \wedge \neg\texttt{((g != 1)} \wedge \texttt{(n = lastN))} \wedge \texttt{(t1 = n-1)} \\
\varphi_{cs}^{post} &= (\exists \texttt{g} \exists \texttt{lastN}\ \varphi_{cs}^{pre}) \wedge \varphi^{inv} \wedge \texttt{(t2 = } \textit{factCache}\texttt{(t1))} \\
\varphi_3^{path} &= (\exists \texttt{g} \exists \texttt{lastN}\ \varphi_{cs}^{post}) \wedge \texttt{(g = n * t2)} \wedge \texttt{(last N = n)} \wedge \texttt{(result = g)} \\
\varphi^{post} &= \varphi_1^{path} \vee \varphi_2^{path} \vee \varphi_3^{path} \\
\varphi^{vc} &= (\varphi_{cs}^{pre} \Rightarrow \varphi^{inv}) \wedge (\varphi^{post} \Rightarrow \varphi^{inv}) \wedge (\textsc{init}(\textsc{p}) \Rightarrow \varphi^{inv})
\end{aligned}
$$

Fig. 4. The different formulae computed from the procedure in Listing 1.2 by our post-condition and verification-condition computation.

4.2 Approach 1: Existential Approach

Let P be a procedure with input parameter n and return variable r. Let $\textsc{postvc}(\textsc{p}, \varphi^{inv}) = (\varphi^{post}, \varphi^{vc})$. Let ψ^e denote the formula $\varphi^{vc} \wedge (\varphi^{post} \Rightarrow (r = p(n)))$. Let $\overline{x}$ denote the sequence of all free variables in ψ^e except for p. We define $\textsc{ea}(\textsc{p}, \varphi^{inv})$ to be the formula $\forall \overline{x}.\psi^e$.

In this approach, we use a theorem prover to check whether $\textsc{ea}(\textsc{p}, \varphi^{inv})$ is satisfiable. As shown by the following theorem, satisfiability of $\textsc{ea}(\textsc{p}, \varphi^{inv})$ establishes that P satisfies $\text{pure}(\varphi^{inv})$.

Theorem 1. *A procedure* P *satisfies* $\text{pure}(\varphi^{inv})$ *if* $\exists p.\textsc{ea}(\textsc{p}, \varphi^{inv})$ *is a tautology (which holds iff* $\textsc{ea}(\textsc{p}, \varphi^{inv})$ *is satisfiable).*

Proof. Note that p is the only free variable in $\textsc{ea}(\textsc{p}, \varphi^{inv})$. Assume that $[p \mapsto f]$ is a satisfying assignment for $\forall \overline{x}.\psi^e$. We show that for every feasible execution π: (P1) $(\pi, f) \vdash \varphi^{inv}$, and (P2) for every trace π' inside π, $\mathit{output}(\pi') = f(\mathit{input}(\pi'))$. This implies that P satisfies $\text{pure}(\varphi^{inv})$.

In particular, for any feasible execution π, we prove by induction over the execution steps in π that

1. For any entry state σ in π, $(\sigma, f) \vdash \varphi^{inv}$.
2. For any exit state σ in π, $(\sigma, f) \vdash \varphi^{inv}$.
3. For any exit state σ in π, if it is the exit state of a trace π', then $\mathit{output}(\pi') = f(\mathit{input}(\pi'))$.

If the above properties fail to hold, we can identify a trace π' corresponding to the first such failure. It can be shown that the sequence of states visited by this trace, when substituted for $\overline{x}$, are a witness that $[p \mapsto f]$ is not a satisfying assignment for $\forall \overline{x}.\psi^e$. This is a contradiction of our original assumption.

Please see [4] for more details of the proof. □

4.3 Approach 2: Impurity Witness Approach

The existential approach presented in the previous section has a drawback. Checking satisfiability of $\text{EA}(\text{P}, \varphi^{inv})$ is hard because it contains universal quantifiers and existing theorem provers do not work well enough for this approach. We now present an approximation of the existential approach that is easier to use with existing theorem provers. This new approach, which we will refer to as the impurity witness approach, reduces the problem to that of checking whether a quantifier-free formula is unsatisfiable, which is better suited to the capabilities of state-of-the-art theorem provers. This approach focuses on finding a counterexample to show that the procedure is impure or it violates the candidate invariant.

Let P be a procedure with input parameter n and return variable r. Let $\text{POSTVC}(\text{P}, \varphi^{inv}) = (\varphi^{post}, \varphi^{vc})$. Let φ^{post}_{α} denote the formula obtained by replacing every free variable x other than p in φ^{post} by a new free variable x_α. Define φ^{post}_{β} similarly. Define $\text{IW}(\text{P}, \varphi^{inv})$ to be the formula $(\neg\varphi^{vc}) \vee (\varphi^{post}_{\alpha} \wedge \varphi^{post}_{\beta} \wedge (n_\alpha = n_\beta) \wedge (r_\alpha \neq r_\beta))$.

The impurity witness approach checks whether $\text{IW}(\text{P}, \varphi^{inv})$ is satisfiable. This can be done by separately checking whether $\neg\varphi^{vc}$ is satisfiable and whether $(\varphi^{post}_{\alpha} \wedge \varphi^{post}_{\beta} \wedge (n_\alpha = n_\beta) \wedge (r_\alpha \neq r_\beta))$ is satisfiable. As formally defined, φ^{vc} and φ^{post} contain embedded existential quantifications. As explained earlier, these existential quantifiers can be moved to the outside after variable renaming and can be omitted for a satisfiability check. (A formula of the form $\exists\overline{x}.\psi$ is satisfiable iff ψ is satisfiable.) As usual, these existential quantifiers refer to intermediate values of variables along an execution path. Finding a satisfying assignment to these variables essentially identifies a possible execution path (that satisfies some other property).

Theorem 2. *A procedure* P *satisfies* $\text{pure}(\varphi^{inv})$ *if* $\text{IW}(\text{P}, \varphi^{inv})$ *is unsatisfiable.*

Proof. We say that two traces disagree if they receive the same argument value but return different values. We say that a pair of feasible executions (π_1, π_2) is an *impurity witness* if there is a trace π_a in π_1 and a trace π_b in π_2 such that π_a and π_b disagree.

A trace is said to be compatible with a function f (and vice versa) if the trace's input-output behavior matches that of the function. An execution is said to be compatible with a function (and vice versa) if every trace in the execution is compatible with the function. We say that a feasible execution π *strongly satisfies* φ^{inv} if for every function f that is compatible with π, $(\pi, f) \models \varphi^{inv}$.

We prove the theorem using the following lemmas: if $\text{IW}(\text{P}, \varphi^{inv})$ is unsatisfiable, then Lemmas 2 and 3 imply that the preconditions of Lemma 1 hold and, hence, P satisfies $\text{pure}(\varphi^{inv})$.

1. If there exists no impurity witness, and every feasible execution strongly satisfies φ^{inv}, then P satisfies $\text{pure}(\varphi^{inv})$.
2. If a feasible execution π that does not strongly satisfy φ^{inv} exists, $\text{IW}(\text{P}, \varphi^{inv})$ is satisfiable.

3. If an impurity witness exists, then $\text{IW}(\text{P}, \varphi^{inv})$ is satisfiable.

1 is straightforward.

For 2, we use a "minimal" feasible execution π that does not strongly satisfy φ^{inv} to construct a satisfying assignment to $\neg\varphi^{vc}$.

For 3, we use a "minimal" impurity witness to construct a satisfying assignment to $(\varphi_\alpha^{post} \wedge \varphi_\beta^{post} \wedge (n_\alpha = n_\beta) \wedge (r_\alpha \neq r_\beta))$.

Please see [4] for more details of the proof. □

5 Generating the Invariant

We now describe a simple but reasonably effective semi-algorithm for generating a candidate invariant automatically from the given procedure. Our approach of Sect. 4 can be used with a manually provided invariant or the candidate invariant generated by this semi-algorithm (whenever it terminates).

The invariant-generation approach is iterative and computes a sequence of progressively weaker candidate invariants $I_0, I_1, \cdots$ and terminates if and when $I_m \equiv I_{m+1}$, at which point I_m is returned as the candidate invariant. The initial candidate invariant I_0 captures the initial values of the global variable. In iteration k, we apply a procedure similar to the one described in Sect. 4 and compute the strongest conditions that hold true at every program point if the execution of the procedure starts in a state satisfying I_{k-1} and if every recursive invocation terminates in a state satisfying I_{k-1}. We then take the disjunction of the conditions computed at the points before the recursive call-sites and at the end of the procedure, and existentially quantify all local variables. We refer to the resulting formula as $\text{NEXT}(I_{k-1}, \text{TB}(\text{P}, I_{k-1}))$. We take the disjunction of this formula with I_{k-1} and simplify it to get I_k.

Figure 5 formalizes this semi-algorithm. Here, we exploit the fact that the `assert` statements are added precisely at every recursive callsite and end of procedure and these are the places where we take the conditions to be disjuncted.

In our running example, I_0 is '$\text{g} = -1 \wedge \text{lastN} = 0$'. Applying NEXT to I_0 yields I_0 itself as the pre-condition at the point just before the recursive call-site, and '$(\text{g} = -1 \wedge \text{lastN} = 0) \vee \text{g} = \text{lastN} * p(\text{lastN} - 1)$' (after certain simplifications) as the pre-condition at the end of the procedure. Therefore, I_1 is '$(\text{g} = -1 \wedge \text{lastN} = 0) \vee \text{g} = \text{lastN} * p(\text{lastN} - 1)$'. When we apply NEXT to I_1,

$I_0 = \text{INIT}(\text{P})$
$I_k = \text{SIMPLIFY}(I_{k-1} \vee \text{NEXT}(I_{k-1}, \text{TB}(\text{P}, I_{k-1})))$

$\text{NEXT}(\varphi^{pre}, \texttt{assert e}) = \exists \ell_1 \cdots \ell_m \varphi^{pre}$(where $\ell_1, \cdots, \ell_m$ are local variables in φ^{pre})
$\text{NEXT}(\varphi^{pre}, \texttt{S}_1\texttt{;}\texttt{S}_2) = \text{NEXT}(\varphi^{pre}, \texttt{S}_1) \vee \text{NEXT}(\text{POST}(\varphi^{pre}, \texttt{S}_1), \texttt{S}_2)$
$\text{NEXT}(\varphi^{pre}, \texttt{if e then S}_1 \texttt{ else S}_2) = \text{NEXT}(\varphi^{pre} \wedge \texttt{e}, \texttt{S}_1) \vee \text{NEXT}(\varphi^{pre} \wedge \neg\texttt{e}, \texttt{S}_2)$
$\text{NEXT}(\varphi^{pre}, \texttt{S}) = \text{false}$(for all other S)

Fig. 5. Iterative computation of invariant.

the computed pre-conditions are I_1 itself at both the program points mentioned above. Therefore, the approach terminates with I_1 as the candidate invariant.

6 Evaluation

We have implemented our OP checking approach as a prototype using the Boogie framework [6], and have evaluated the approach using this implementation on several examples. The objective of this evaluation was primarily a sanity check, to test how our approach does on a set of OP as well as non-OP procedures.

We tried several simple non-OP programs, and our implementation terminated with a "no" answer on all of them. We also tried the approach on several OP procedures: (1) the 'factCache' running example, (2) a version of a factorial procedure that caches all arguments seen so far and their corresponding return values in an array, (3) a version of factorial that caches only the return value for argument value 19 in a scalar variable, (4) a recursive procedure that returns the n^{th} Fibonacci number and caches all its arguments and corresponding return values seen so far in an array, and (5) a "matrix chain multiplication" (MCM) procedure. The last example is based on dynamic programming, and hence naturally uses a table to memoize results for sub-problems. Here, observational purity implies that the procedure always returns the same solution for a given sub-problem, whether a hit was found in the table or not. The appendix of a technical report associated with this paper depicts all the procedures mentioned above as created by us directly in Boogie's language, as well as the invariants that we supplied manually (in SMT2 format).

It is notable that the theorem prover was not able to handle the instances generated by the"existential approach" even for simple examples. The "impurity witness" approach, however, terminated on all the examples mentioned above with the correct answer, with the theorem prover taking less than 1 s on each example. Please see [4] for more information about the examples used in our evaluation.

7 Related Work

The previous work that is most closely related to our work is by Barnett et al. [1,2]. Their approach is based on the same notion of observational purity as our approach. Their approach is structurally similar to ours, in terms of needing an invariant, and using an inductive check for both the validity of the invariant as well as the uniqueness of return values for a given argument. However, their approach is based on a more complex notion of invariant than our approach, which relates pairs of global states, and does not use a function symbol to represent recursive calls within the procedure. Hence, their approach does not extend readily to recursive procedures; they in fact state that "there is a circularity - it would take a delicate argument, and additional conditions, to avoid unsoundness in this case". Our idea of allowing the function symbol in the invariant to

represent the recursive call allows recursive procedures to be checked, and also simplifies the specification of the invariant in many cases.

Cok et al. [7] generalize the work of Barnett et al.'s work, and suggest classifying procedures into categories "pure", "secret", and "query". The "query" procedures are observationally pure. Again, recursive procedures are not addressed.

Naumann [3] proposes a notion of observational purity that is also the same as ours. Their paper gives a rigorous but manual methodology for proving the observational purity of a given procedure. Their methodology is not similar to ours; rather, it is based finding a *weakly pure* procedure that simulates the given procedure as far as externally visible state changes and the return value are concerned. They have no notion of an invariant that uses a function symbol that represents the procedure, and they don't explicitly address the checking of recursive procedures.

There exists a significant body of work on identifying differences between two similar procedures. For instance, differential assertion checking [8] is a representative from this body, and is for checking if two procedures can ever start from the same state but end in different states such that exactly one of the ending states fails a given assertion. Their approach is based on logical reasoning, and accommodates recursive procedures. Our impurity witness approach has some similarity with their approach, because it is based on comparing the given procedure with itself. However, our comparison is stricter, because in our setting, starting with a common argument value but from different global states that are both within the invariant should not cause a difference in the return value. Furthermore, technically our approach is different because we use an invariant that refers to a function symbol that represents the procedure being checked, which is not a feature of their invariants. Partush et al. [9] solve a similar problem as differential assertion checking, but using abstract interpretation instead of logical reasoning.

There is a substantial body of work on checking if a procedure is *pure*, in the sense that it does not modify any objects that existed before the procedure was invoked, and does not modify any global variables. Sălcianu et al. [10] describe a static analysis to check purity and Madhavan et al. [11] present an abstract-interpretation based generalization of this analysis. Various tools exist, such as JML [12] and Spec# [13], that use logical techniques based on annotations to prove procedures as pure. Purity is a more restrictive notion than observational purity; procedures such as our 'factCache' example are observationally pure, but not pure because they use as well as update state that persists between calls to the procedure.

References

1. Barnett, M., Naumann, D.A., Schulte, W., Sun, Q.: 99.44% pure: useful abstractions in specifications. In: ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP) (2004)
2. Barnett, M., Naumann, D.A., Schulte, W., Sun, Q.: Allowing state changes in specifications. In: Müller, G. (ed.) ETRICS 2006. LNCS, vol. 3995, pp. 321–336. Springer, Heidelberg (2006). https://doi.org/10.1007/11766155_23

3. Naumann, D.A.: Observational purity and encapsulation. Theor. Comput. Sci. **376**(3), 205–224 (2007)
4. Arora, H., Komondoor, R., Ramalingam, G.: Checking observational purity of procedures. CoRR https://arxiv.org/abs/1902.05436 (2019)
5. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, 17–19 January 2001, pp. 193–205 (2001)
6. Leino, K.R.M.: This is Boogie 2. Manuscript KRML 178(131) (2008)
7. Cok, D.R., Leavens, G.T.: Extensions of the theory of observational purity and a practical design for JML. In: Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008). Number CS-TR-08-07 in Technical report, School of EECS, UCF, vol. 4000 (2008)
8. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 345–355. ACM (2013)
9. Partush, N., Yahav, E.: Abstract semantic differencing for numerical programs. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 238–258. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_14
10. Sălcianu, A., Rinard, M.: Purity and side effect analysis for Java programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_14
11. Madhavan, R., Ramalingam, G., Vaswani, K.: Purity analysis: an abstract interpretation formulation. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 7–24. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_6
12. Leavens, G.T., et al.: JML reference manual (2008)
13. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_3