99.44% pure: Useful Abstractions in Specifications

Mike Barnett [0] , David A. Naumann [1] *, Wolfram Schulte [0] , and Qi Sun [1] **

[0] Microsoft Research
{mbarnett,schulte}@microsoft.com
[1] Stevens Institute of Technology
{naumann,sunq}@cs.stevens-tech.edu

Abstract. Specification languages that use the same expression language as the implementation language must decide whether or not to permit functional abstraction, i.e., method calls in specification expressions. The difficulty is that a specification must not change the functional behavior of the associated program. There are three main current approaches: a) forbid the use of functions in specifications, b) allow only provably pure functions, or c) allow programmers free use of functions. The first approach is not scalable, the second overly restrictive and the third unsound. We propose a definition of *observational purity* for a class of benevolent functions and a sound static analysis for detecting them.

0 Introduction

An obvious truth is that a software specification is meant to be a description; it is clearly not the thing that it is describing. Software specifications which share the same expression language as the implementation programming language run the risk of blurring this distinction. When specifications contain expressions that change the state of the program, then the meaning of the program may differ depending on whether or not the specifications are present; the two are no longer independent.

Despite this, there are many reasons for using the same expression language in both an implementation and its specification. To prevent unwanted interference, specifications are usually restricted to a side-effect free (pure) subset of the expression language. An important decision to make is whether (programmer-defined) functions belong in the subset or not: there are three main current approaches.

- The simplest approach is to forbid the use of functions in specifications altogether. While easy to implement, this solution does not scale and is overly restrictive on the practical use of specifications. ESC/Java [9] uses this solution.
- From a theoretical perspective, a pleasing solution is to allow only provably pure functions. However, any realization must be overly conservative and thus overly restrictive. JML [11] uses this solution.
- An unsound solution is to request for the programmer to refrain from using side-effects in the functions they write, but to actually allow the free use of functions. While not restrictive at all (and particularly easy to implement), this means it is not possible to guarantee that a program's meaning is unchanged when including its specification. It also is impractical for library functions that are beyond the control of the programmer. Eiffel [12] uses this solution.

* Supported in part by NSF award CCR-0208984 and NJCST.
** Supported in part by NSF award CCR-0208984 and NJCST.

We are interested in a sound, practical static analysis that allows *benign* side-effects so programmers can use functions in specifications as freely as possible. We propose a definition of *observational purity* and a static analysis to determine it. The intuition behind observational purity is that a function is allowed to have side-effects only if they are not observable to callers of the function. As with programs, we restrict our attention to effects that are observable in terms of the source language (Java or C#) and ignore effects such as memory usage or power consumption. Our prototypical example of an observationally pure function is one that maintains an internal cache. Changing this internal cache is a side-effect, but it is not visible outside of the object. Other examples are methods that write to a log file that is not read by the rest of the program and methods that perform lazy initialization. Algorithms that are optimized for amortized complexity, such as a list that uses a "move to front" heuristic, also perform significant state updates that are not visible externally. Observationally pure methods often occur in library code that is highly optimized and also frequently used in specifications, e.g., the equality methods in a string library.

Our proposal uses a conservative static analysis together with a mild verification condition. It appears that for the many simple cases that occur in practice the proposal requires very little effort on the part of the programmer.

Section 1 begins by discussing the example of a function that maintains an internal cache. Then we define observational purity in semantic terms. In Section 2 we outline a static analysis that provides a conservative approximation for observational purity. We show the resulting annotations induced by the analysis in Section 3. Section 4 discusses related work and future directions for our work.

1 Observational Purity

Figure 0 shows a class C that contains a method f which is meant to compute a function, $expensive$, of type $T \rightarrow U$. This function is actually quite expensive to compute, so as an optimization the actual computation is done only the first time that f is called for each argument x. The class C maintains an internal cache to store already computed results. The cache is implemented as a hashtable, t, where it stores pairs $(x, expensive(x))$ so that future queries for x do a table lookup instead of recomputing $expensive(x)$. We assume that $expensive$ is a (strongly) pure function and so can be used in specifications. In a more complete example there would be other methods in the class. It is important to note that the class C does *not* implement the method $expensive$ in the program; clients use method f and need to be able to express conditions about $c.f(\ldots)$ for some object c of type C.

Assuming that no other methods in the class access t, this private field is effectively encapsulated in f. It should be possible to allow f to appear in specifications since $f(x) = expensive(x)$ for any x and the side effect is not observable. More formally, the reason f is observationally pure is that we can define a relation R of *indistinguishability* to describe that any side-effects of f are encapsulated. The relation is between two program heaps; for this example we define $R(h, k)$ if and only if for every allocated object o we have $h(o).F = k(o).F$ for every field F other than t. (We regard a heap as a function whose domain is the set of allocated objects.) In short, h is indistinguishable from k if they are identical except for the (contents of their) t fields.

```
class C {
  private Hashtable t := new Hashtable();
  invariant Forall{U x in t.Keys : t[x] = expensive(x)};
  [ObservationallyPure]
  public U f(T x)
    requires x ≠ null;
    ensures result = expensive(x);
  {
    if (¬t.ContainsKey(x)){
      U y = ...; // compute expensive(x)
      t.Add(x, y); }
    return (U)t[x];
  }
}
```

Fig. 0. A class C that maintains a cache t to avoid recomputing $expensive$.

If t is encapsulated to be accessed only by f, then its only influence on computations or on evaluation of assertions is by way of f. According to its postcondition, f returns a result independent from the state of t; and the stated invariant is sufficient to verify that this specification is satisfied.

We sketch a formalization ignoring termination but taking into account method preconditions. Typical examples are terminating and the ideas appear to adapt easily to take termination into account if necessary. We assume a language in which methods cannot make non-local references to mutable state except fields of objects in the heap.

Consider a method f that takes a parameter list x. If f is an instance method, then x includes the receiver object. We write

$$f(x), h \to v, k$$

to mean that f's execution on arguments x in initial heap h yields result value v and final heap k. We use similar notation for expressions and commands. If f is to be used in specifications it should be deterministic, but this does not obtrude in the sequel.

Definition 1. f is *strongly pure* for precondition P if $P(x, h)$ and $f(x), h \to v, k$ implies $h = k$.

This is slightly stronger than necessary: It is possible to allow allocation of fresh objects, disallowing only updates to preexisting objects. This is the approach taken in JML and also by Sălcianu and Rinard [17]. To focus on the main issues, we omit this refinement. To check the condition statically, it is enough to check that there are no field updates. Writes to local variables are fine.

Definition 2. Given preconditions P_g for every method g, we say f is *observationally pure* if there exists a binary relation R on heaps such that the following conditions hold. First, for f:

$$P_f(x, h) \ \wedge \ f(x), h \to v, k \Rightarrow R(h, k) \tag{0}$$

Second, for any g, if $P_g(x,h)$ and $P_g(x,h')$ then

$$R(h,h') \ \wedge \ g(x),h \rightarrow v,k \ \wedge \ g(x),h' \rightarrow v',k' \Rightarrow R(k,k') \ \wedge \ v = v' \qquad (1)$$

Condition (1) is like the *noninterference* property that formalizes secure information flow [10, 16]: it expresses that R-related states are not distinguished by any methods, not even by f itself. Condition (0) says that the state after an invocation of f cannot be distinguished from the state before.

For our running example, define the relation $R0(h,k)$ to hold just if h,k agree on fields other than t and, in both h and k, every instance of C satisfies the class invariant. Note that (1) fails for f unless we include the invariant to constrain the cache.

A consequence of condition (1), which we will call (1*), is that R is preserved by arbitrary commands and expressions.[0]

An observationally pure method may be used in preconditions and other assertions without affecting the semantics of specifications. To be more precise, we argue that

$$S \ = \ \mathbf{assert} \ Q; S$$

for any command S and any Q such that for any invocation $f(x)$ that occurs in Q we have (a) f is observationally pure and (b) the precondition of f holds.[1] The displayed equation formalizes both that Q has no effect for runtime checking and that in terms of static verification it is sound to ignore the effect of Q in reasoning about "$\mathbf{assert} \ Q; S$". Of course Q does have a semantic effect, so the semantics of the assert statement depends on evaluating Q which yields a heap. To prove the equation, consider any initial state h and any R witnessing observational purity. Suppose $Q,h \rightarrow v,h'$; then $R(h,h')$ by (a), (b), and (0). If $S,h' \rightarrow k'$ and $S,h \rightarrow k$ then by (1*) we have $R(k,k')$. Evaluating any expression e in the final states k,k' yields equal results, by (1*) for e. So the two sides of the equation are equal up to observability using expressions in the language.

Definition 2 assumes we are given a precondition P_g for every method. We are mostly interested in using the definition with $P_g = true$ for all g other than f, for two reasons. One reason is modularity: we want to check f independently of contexts in which it may be used. The other reason is soundness. We want to check programs with respect to their designated specifications and we want f to be used in specifications. If we allow some P_g to involve f, then there is a circularity —it would take a delicate argument, and additional conditions, to avoid unsoundness in this case.

Condition (1) is impractically global. The usual way to achieve such a property by modular reasoning is to encapsulate the part of state on which R depends, so that (1) always holds for g that has no dependence on the encapsulated state. Encapsulation for this purpose is studied in [2] and other disciplines for encapsulating invariants can be used as well, e.g. [5, 13]. Such disciplines typically base encapsulation boundaries on program structures such as modules and private fields. For observational purity we typically want finer-grained encapsulation. That is addressed in the next section.

[0] This holds for languages such as Java or C# ([2] or [3]). In the presence of pointer arithmetic or if an out-of-memory condition is considered observable, the consequence fails; but then even allocation of fresh objects could not be allowed for pure methods.

[1] The objective is to allow use of an observationally pure method in assertions where it is guarded in the sense that its precondition P holds, e.g., in contexts like $P \wedge f(x)$ and $P \Rightarrow f(x)$. We are agnostic as to how (b) is achieved.

To make the sketch above more precise, one would argue that the equation holds up to observability by expressions outside the encapsulation boundary; e.g., (1*) does not hold for the expression $t.ContainsKey(x)$ and the given $R0$.

Note that the definition allows the encapsulated state to be manipulated by other methods. For example, f could be a membership test on a set represented by an unordered list, with the side-effect of rearranging the list (the "move to front" heuristic). The list is accessible to methods that insert new elements, which are not observationally pure. Another (observationally pure) method might sort the list in order to compute its intersection with another set. For such examples it may require nontrivial program annotation to delimit the effect and justify observational purity.

We are particularly interested in static analysis for simple cases like our leading example where the updated state is tightly encapsulated. For this we propose a static analysis that requires only annotation for f.

2 Information Flow

To ensure that (1) holds automatically for all methods other than f, we propose to use a dependency or information flow analysis [1, 18, 16]. Information flow analyses check for complete absence of dependencies. The standard specification of the noninterference property checked by such an analysis [10, 18] uses a condition of the form (1) where the simple indistinguishability relation R expresses equality of the components of program state that are deemed to be visible.

In a class that contains a method purported to be observationally pure, we annotate fields to distinguish between *secret* and *open*. [2] Any field written by a method marked observationally pure is a *secret* field. Otherwise it is an *open* field (the unmarked default). All parameters and results are open (at least for public methods). The simple indistinguishability relation $R(h, k)$ holds just if h and k have the same domain and object states that are equal on all open fields. This makes (0) hold by construction.[3] An information flow analysis, such as a fast flow-insensitive type-based analysis can now be used to check that the secret field is encapsulated in f. It is sufficient to check the class of f. The property achieved by flow analysis is exactly (1) [18, 4].

But there is a problem. Standard analysis rules will reject f because of the manifest dependence of its (open) result on the secret field: The expression $t[x]$ involves the secret t and is thus treated as a secret. Yet, according to its specification, $f(x) = expensive(x)$ and the result from $expensive$ is not secret. Indeed, sound rules must reject f because (1) fails for the simple indistinguishabiity relation. To avoid the need for more general relations like $R0$ in the example, we resort to program annotation.

We propose the following new rule for information flow. We express it more generally in terms of assignments, as $\mathbf{return}\ e$ can be taken to abbreviate $result := e$.

If y and e_0 are open, then "$\mathbf{assert}\ e = e_0; y := e$" is allowed.

[2] We use the term "open" instead of "public" to avoid confusion with the visibility modifiers (private, protected, public) that are common in object-oriented programming.

[3] To allow for allocation of fresh objects, the definition would be refined to incorporate a bijection on visible objects [4].

(Even if the level of e is secret.) It should not be difficult to show that this is sound with respect to the noninterference property[4] (1). Information flow analysis must also take control flow into account; we return to this point later.

3 Annotated Example

To support flow analysis, class C is annotated as shown in Figure 1. Note that the

```
class C {
  [Secret]
   private Hashtable t := new Hashtable();
  invariant Forall{U x in t.Keys : t[x] = expensive(x)};
  [ObservationallyPure]
   public U f(T x)
     requires x ≠ null;
     ensures result = expensive(x);
  {
     if (¬t.ContainsKey(x)){
        U y = ...; // compute expensive(x)
        t.Add(x, y); }
     assert (U)t[x] = expensive(x);
      return (U)t[x];
  }
}
```

Fig. 1. The annotated class C. The "leak" of secret information has been guarded by an assertion.

required assertion is an immediate consequence of the class invariant that has been introduced as part of specifying the correctness of f regardless of the issue of purity. We think it is important that we avoid the need for building the invariant into the indistinguishabity relation. The annotated version of f *does* satisfy (1) for the standard indistinguishability relation.

Our approach would prevent a method such as the following from being added to class C:

```
[ObservationallyPure]
 public int leak()
{
   return t.Count;
}
```

Such a method would require the programmer to validate an assertion relating the number of items in the hashtable to some open data, which is unlikely to be possible.

[4] Unlike the more general notion of declassification that is needed to handle actual leakage of information [16, 14].

It is important to also consider how information can be revealed via control flow. For instance, suppose the programmer added this method to the example class C:

$$
\begin{array}{l}
[ObservationallyPure] \\
\textbf{public } U \ \ problem(T \ \ x) \\
\quad \textbf{requires } x \neq \textbf{null}; \\
\quad \textbf{ensures result } = expensive(x); \\
\{ \\
\quad \textbf{if } (t.ContainsKey(x)) \\
\quad\quad \textbf{throw new } Exception(\ldots); \\
\quad \textbf{else} \\
\quad\quad \textbf{return } f(x); \\
\}
\end{array}
$$

Then $problem(x)$ returns $expensive(x)$ on the first occurrence of x as an argument to f (or $problem$), but otherwise throws an exception. Most information flow analyses check that in the branches of a conditional with secret guard, there are no flows on open channels (e.g., assignments to an open variable, normal or exceptional return) [16]. For exceptional flows and unstructured code, control dependencies are tracked [8]; an open flow is not allowed if the program counter is influenced by secrets.

For our purposes, a rule like the following could be convenient:

If e_0 is open then $\textbf{assert } e = e_0; \textbf{if } (e) \textbf{ then } S_0 \textbf{ else } S_1$ is allowed.

(Even if S_0, S_1 are open and e secret.) Alternatively, the code can be rewritten to use a temporary variable for the guard condition, initialized to e using the rule in Section 2.

4 Conclusions

It is important to provide for observational purity for both theoretical and practical reasons. When specifications do not modify the observable state of a program, then specifications can be combined with programs without changing their meaning. This makes it much easier to implement both static and dynamic analysis tools. We conjecture that many library methods are observationally pure; it would be inconvenient to have them unavailable for use in contracts. Observational purity may also provide some useful concepts for dealing with object isolation and information flow.

4.0 Related Work

Runtime verification using AsmL [7] does not restrict the use of functions in specfiications. It provides an alternative data space from the implementation so that side-effects in this space are insulated from the data space of the implementation. But AsmL is unsound since it allows full interoperability with arbitrary components.

JML has decided on the conservative approach of outlawing all side-effects [11]. Library methods that cause side-effects cannot be used in specifications, instead pure replacements must be used. This complicates life for specifiers: one must always be aware of which methods one can use and which are outlawed. Also, not all of the JML tools are capable of using the replacement methods.

These issues have long been known in the Eiffel community; Meyer [12] discusses at length the desire to allow benevolent side-effects. However, Eiffel does not enforce any policy, but leaves it as a design principle.

Sălcianu and Rinard [17] have designed a purity analysis that is able to distinguish updates to pre-existing objects and newly allocated objects. The mutation of the latter is allowed in a pure method. They also are able to extract regular-expression descriptions of updates that violate purity.

4.1 Future Work

We plan to perform an analysis of the .NET base class library to see how many functions that would informally be considered as pure are actually observationally pure, but not strongly pure. We are also implementing our observational purity system in the context of the Boogie project [5, 6] within Microsoft Research. This context provides automated theorem-proving support to check assertions. For simple examples involving lazy initialization and caches, superficial syntactic heuristics might be adequate for checking the relevant assertions. The theoretical justification will adapt noninterference theory to the more general relations of representation independence [15, 2].

4.2 Acknowledements

Alexandru Sălcianu provided several helpful comments. The comments from the reviewers also improved the paper.

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *ACM Symp. on Princ. of Program. Lang.*, pages 147–160, 1999.
2. Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control. In *ACM Symp. on Princ. of Program. Lang.*, pages 166–177, 2002.
3. Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–270. IEEE Computer Society Press, 2002.
4. Anindya Banerjee and David A. Naumann. Stack-based access control for secure information flow. Submitted., 2003.
5. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. In Susan Eisenbach, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter, and Erik Poll, editors, *Formal Techniques for Java-like Programs 2003*, July 2003. Available as Technical Report 408, Department of Computer Science, ETH Zurich. A newer version of this paper is [6].
6. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. Manuscript KRML 122b, December 2003. Available from http://research.microsoft.com/~leino/papers.html.
7. Mike Barnett and Wolfram Schulte. Runtime verification of .NET contracts. *The Journal of Systems and Software*, 65(3):199–208, 2003.
8. Gilles Barthe, Amitabh Basu, and Tamara Rezk. Security types preserving compilation. In *Proceedings of Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2937 of *LNCS*, 2004.

9. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37 of *SIGPLAN Notices*, pages 234–245. ACM, May 2002.
10. J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
11. Gary Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accomodates both runtime assertion checking and formal verification. Technical Report 03-04, Department of Computer Science, Iowa State University, March 2003.
12. Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, second edition, 1997.
13. P. Müller, A. Poetzsch-Heffter, and G.T. Leavens. Modular invariants for object structures. Technical Report 424, ETH Zürich, Chair of Software Engineering, October 2003.
14. Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *IEEE Computer Security Foundations Workshop (CSFW)*, 2004. To appear.
15. John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, 1984.
16. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
17. Alexandru Sălcianu and Martin Rinard. A combined pointer and purity analysis for Java programs. Technical Report MIT-CSAIL-TR-949, Department of Computer Science, Massachusetts Institute of Technology, May 2004.
18. Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97*, number 1214 in LNCS, pages 607–621. Springer-Verlag, 1997.