Observational Purity and Encapsulation

David A. Naumann[⋆]

Department of Computer Science, Stevens Institute of Technology
naumann@cs.stevens.edu

Abstract. Practical specification languages for imperative and object-oriented programs, such as JML, Eiffel, and Spec#, allow the use of program expressions including method calls in specification formulas. For coherent semantics of specifications, and to avoid anomalies with runtime assertion checking, expressions in specifications and assertions are typically required to be strongly pure in the sense that their evaluation has no effect on the state of preexisting objects. For specification of large systems using standard libraries this restriction is impractical: it disallows many standard methods that mutate state for purposes such as caching or lazy initialization. Calls of such methods can sensibly be used for specifications and annotations in contexts where their effects cannot be observed. This paper formalizes and extends a recently proposed notion of observational purity, reducing the proof obligation to a familiar one for equivalence of two class implementations.

1 Introduction

There are a number of uses for identifying pure expressions, i.e., those without side effects. For example, they admit transformations such as re-ordering and they may be used without difficulty in program specifications. For verification of programs in object oriented languages such as Java, it is important to allow annotations (including specifications and intermediate assertions) to invoke methods whose calls are pure in a more liberal sense: allowing construction of fresh objects. For example, to return a pair of values, a pair objects may be created. This notion of purity is used in the JML behavioral interface specification language [12].

Many software libraries include methods that one would expect to be pure, such as `String.equals` in Java, but which in fact mutate preexisting objects for purposes such as memoization, caching, or lazy initialization. The solution adopted in JML is to duplicate such library methods with pure ones to be used in specifications, but this is awkward at best. It has recently been proposed to liberalize the notion further, to allow methods that have "benign" side effects, i.e., mutation of preexisting objects so long as these effects are not visible in the context where the method is treated as pure.

Allowing benign side effects is important for specification and program transformation to scale up to large systems, but it poses challenges: How do such effects interact

[⋆] Partially supported by the National Science Foundation under grants CCR-0208984 and CCF-0429894 and by Microsoft Research.

```
class Cell {
  public val : int;
  proc pos(c : Cell) : bool { return c.val > 0; }    }
class D {
  private f, arg, farg : int;
  proc pureProd(s : D, n : int) : Cell {
    x : Cell := new Cell; x.val := s.f * n; return x; }
  proc memoProd(s : D, n : int) : Cell {
    x : Cell := new Cell;
    if n = 0 then x.val := 0; return x;
    elseif s.arg ≠ n then s.arg := n; s.farg := s.f * n; end;
    x.val := s.farg; return x; }
  proc get(s : D) : int { return s.f}
  proc set(s : D, v : int) {s.f := v; s.arg := 0; }    }
```

Fig. 1. Example program in simple language with class-bound procedures. It maintains an invariant: $o.arg \neq 0$ implies $o.farg = o.f * o.arg$ for all D-objects o

with "modifies" specifications? What is the meaning of an effectful predicate in a precondition? How do effects interact with runtime assertion checking?

A definition of observational purity is proposed by Barnett et al. [6] along with a static analysis based on secure information flow [18] combined with verified program assertions. But the definition has been criticized as ad hoc and obscure and the checking technique seems rather specialized. In this paper we disentangle and extend the ideas, showing how observational purity can be formulated in terms of established notions of abstraction and encapsulation. This opens the way to using existing methods to verify observational purity. Moreover, we obtain a sound and general theory without the need to prove the hardest of the results from scratch.

The key idea is that an observationally pure method is equivalent to one that is strongly pure in the sense of allowing allocation of new objects but no mutation of preexisting ones. This requires an account of strong purity, which we have not seen in the literature. Our account is set in the context of partial correctness; in the conclusion we describe how the approach can be adapted to total correctness using refinement. The core difficulties and ideas are already present in the partial correctness setting.

As a simple and general way to justify the use of pure method calls in specifications and annotations we seek conditions under which "**assert** Q" is equivalent to "**skip**", where Q is a boolean expression that may include method invocations as well as specification constructs such as quantifiers. The notion of equivalence must be compositional, i.e., a congruence, and correctness-preserving. We base our theory on simulation, the standard technique for proving equivalence of implementations that differ in their data representation.

In the sequel a simple but representative example is used; see Figure 1. This program memoizes a product $f * arg$ in a field $farg$. In the context of some class B with access to $d : D$ and $i : \mathbf{int}$ one might find expression $pos(pureProd(d, i))$ in a specification. The argument for allowing this is that, although it has an effect on the heap, it changes no preexisting objects and thus cannot interfere with the meaning of other terms of the asserted formula. Another argument for allowing it is that one could turn runtime assertion checking on or off without affecting the outcome from the program: the fresh object returned by $pureProd$ is examined in evaluating the asserted formula but then discarded. This could have an effect, e.g., via out-of-memory condition; or via pointer arithmetic because it affects where the next allocation takes place. But for many purposes none of these sorts of observation are of interest. It is under such idealization that our results are of interest.

Strong purity is a property of a procedure in isolation. Observational purity is a property of a class (or module) in which the effects of the observationally pure procedure are encapsulated. Procedure $memoProd$ in Figure 1 is observationally pure, but this depends on cooperation by the other procedures, which neither interfere with the cache nor expose it. Moveover, $memoProd$ is observationally pure *outside* its declaring class D, meaning that if it occurs in Q then $\mathbf{assert}\ Q$ is equivalent to $\mathbf{skip}$ only in the context of a class other than D.

Outline. Section 2 formalizes a simple language sufficient to illustrate the ideas. Section 3 defines strong purity which admits $pureProd$ in Figure 1. A notion of equivalence is defined and justified, such that $\mathbf{assert}\ Q$ is equivalent to $\mathbf{skip}$ for strongly pure Q. Section 4 adds visibility to the language in order to formalize a notion of observational purity. It is shown that $\mathbf{assert}\ Q$ is equivalent to $\mathbf{skip}$ for observationally pure Q, but for a notion of visible equivalence that is not a congruence. Section 5 generalizes equivalence to simulations, which are congruences. Section 6 concludes. Most proofs are omitted for lack of space.

Notation. We write $f\ v$ for application of function f to v. Application associates to the left and binds more tightly than other binary operators. For subset X of the domain of f, we write $X \lhd f$ for the restriction of f to X. And $v \mathbin{\lhd\!\!\!-} f$ denotes f with v removed from its domain. We write $[f \mid v \mapsto u]$ for overriding or extending f to map v to u. Relational operators like $\sim$ bind less tightly than others such as $\lhd$, e.g., $\mathsf{dom}\, h \lhd k \sim h$ is parsed as $((\mathsf{dom}\, h) \lhd k) \sim h$. The product of relations α, β is written $\alpha \cdot \beta$.

2 Illustrative Language

We consider a procedural language with dynamically allocated mutable objects, as this suffices to expose the main ideas. The syntax is given in Table 1. A program consists of a collection of class and procedure declarations The declaration of a class named C gives its fields. A distinguished field, type, gives the class name of an object; it is not allowed to be the target of assignment. Because we do not consider subclassing, we need not distinguish a "self" parameter for special treatment. In fact for simplicity in the formalism we consider only procedures that return a value and have exactly one parameter (passed by value). For each procedure p a term, $\mathsf{body}\, p$, should be given. In Section 4 we add visibility control for fields and thus associate procedures with classes as in Figure 1. Non-local variables and static fields are omitted. In order to avoid unilluminating complications in the proofs, we assume there are no recursive procedures. It should be straightforward to extend the results to these and other program constructs

Table 1. Grammar of effectful terms

$C, D \in \mathit{ClassName} \qquad p \in \mathit{ProcedureName} \qquad x, y \in \mathit{VarName} \qquad f \in \mathit{FieldName}$

$M, N, Q ::=$	$\mathbf{assert}\ M$	
$\mid$	$M = M$	equality test, for values
$\mid$	x	read local variable
$\mid$	$x := M$	write local variable
$\mid$	$M.f$	read field of heap object
$\mid$	$x.f := M$	write field of heap object
$\mid$	$\mathbf{new}\ C$	reference to freshly allocated object of class C
$\mid$	$p(M)$	invoke procedure p on argument M
$\mid$	$\mathbf{null} \mid \mathbf{skip} \mid M; M \mid \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M \mid \mathbf{while}\ M\ \mathbf{do}\ M \mid \mathbf{var}\ x\ \mathbf{in}\ M$	

as well as specification constructs such as quantifiers and regular path expressions. What we need is that the language satisfies Propositions 1 and 20 in the sequel.

The details of typing, although important to preclude pointer arithmetic, are ignored in the formalism due to space limitation.

Because we focus on side effects of expressions, we refrain from distinguishing between expressions and commands; the short word *term* is used for both.

Semantics. The language is deterministic; in particular an arbitrary but deterministic memory allocator is used. Purity, which is about effects, does not depend on determinacy. Of course determinacy for assertions is important to facilitate reasoning.

A *store* is a finite mapping from identifiers to primitive values (booleans, integers, locations). An *object state* is just a store; the domain is the object's field names including the distinguished name, type, that records the class of the object. A *heap* is a finite mapping from locations to object states. A *state* is a pair (h, s) where h is a heap and s is a store. The idea is that the domain of s has local variables and parameters for a particular procedure.

A special variable, res, is present in the store part of every state, but is not allowed to occur in the program text. It is used in the semantics like a temporary register, to record the value of a term. This formalization helps streamline subsequent definitions, e.g., a single definition for equivalence of stores serves for both the value and effect of a term.

For partial correctness it suffices to use a relational (evaluation) semantics; Table 2 gives representative cases. For term M, the relation $M, \cdot \rightarrow \cdot$ on states is written $M, h, s \rightarrow k, t$ and interpreted to mean that in initial state (h, s) execution of M can yield outcome (k, t). To model that M diverges from (h, s), there is no (k, t) such that $M, h, s \rightarrow k, t$.

For invocation of a procedure named p, execution of the body of p affects the variables in scope for the body, namely res and the parameter, but only the value of res is needed for semantics of the invocation. The auxiliary relation $\dashv p \mapsto$ is defined by

$$h, s \dashv p \mapsto k, v \iff M, h, s \rightarrow k, t \text{ and } v = t(\mathsf{res}) \text{ for some } t\text{, where } M = \mathsf{body}\, p.$$

This gives the meaning of a procedure in terms of its local state.

The semantics makes $\mathbf{assert}\ M$ yield a final state only if M yields a final state (k, u) in which $u(\mathsf{res})$ is true. The final state of the $\mathbf{assert}$ retains the effect of M on

Table 2. Semantics for selected terms. We assume that *fresh* is a total function from heaps to locations such that $\mathit{fresh}\ h \notin \mathsf{dom}\ h$. We abbreviate a nested update to field f by $[h \mid o.f \mapsto v]$

If M is …	then $M, h, s \to k, t$ iff …
null	$k = h$ and $t = [s \mid \mathsf{res} \mapsto \mathbf{null}]$
skip	$k = h$ and $t = s$
x	$k = h$ and $t = [s \mid \mathsf{res} \mapsto s\, x]$
$x := N$	$N, h, s \to k, u$ and $t = [u \mid x \mapsto u(\mathsf{res}), \mathsf{res} \mapsto s(\mathsf{res})]$ for some u
$N.f$	$N, h, s \to k, u$ and $u(\mathsf{res}) \neq \mathbf{null}$ and $t = [u \mid \mathsf{res} \mapsto k(u(\mathsf{res})).f]$ for some u
$x.f := N$	$s\, x \neq \mathbf{null}$ and $N, h, s \to g, u$ and $t = [u \mid \mathsf{res} \mapsto s(\mathsf{res})]$ and $k = [g \mid s\, x.f \mapsto u(\mathsf{res})]$ for some g, u
assert N	$N, h, s \to k, u$ for some u with $u(\mathsf{res}) = \mathit{true}$, and $t = [u \mid \mathsf{res} \mapsto s(\mathsf{res})]$
new C	$k = [h \mid o \mapsto \mathit{default\_C\_state}]$ and $t = [s \mid \mathsf{res} \mapsto o]$ where $o = \mathit{fresh}\ h$
$p(N)$	$N, h, s \to g, r$ and $g, \mathsf{arg}(r(\mathsf{res})) \dashv p \models\!\!\to k, v$ and $t = [r \mid \mathsf{res} \mapsto v]$ for some g, r, v, where $\mathsf{arg}(y) \hat{=} [x \mapsto y, \mathsf{res} \mapsto \mathit{default}]$ and x is the parameter of p

the heap and on the store, except that res has its initial value —otherwise an **assert** could never be equivalent to **skip**.

As an illustrative but otherwise useless example, consider execution of the term "$x.f;$ **assert** $((y := 1) = 2)$" from initial state (h, s). Evaluation of $x.f$ changes the store to $[s \mid \mathsf{res} \mapsto v]$ where v is the value of field f of object $s\, x$ (i.e., $v = h(s\, x).f$) and there is no outcome if $s\, x$ is null. Next, $y := 1$ is executed, updating y but restoring res to v. Then the equality is evaluated, comparing v with 2. If they are equal, the final store is $[s \mid \mathsf{res}, y \mapsto v, 1]$ because the semantics of **assert**, like $:=$, discards the intermediate res values. If $v \neq 2$ there is no outcome.

The semantic definitions do not explicitly impose the obvious condition that terms are evaluated in the context of a suitable initial store (that includes all free variables of the term) or that the final store has the same domain. Like typing, the precise conditions can easily be provided by the interested reader. To prove the results in the sequel, it is important to confine attention to *closed states* (h, s), i.e., those such that every location that occurs in s or in an object field in h is in $\mathsf{dom}\ h$. (More precisely, if o is in $\mathsf{rng}\ s$ or in $\mathsf{rng}\ r$ for some object state r in $\mathsf{rng}\ h$ then o is in $\mathsf{dom}\ h$.) The following is easily proved for the language in Table 1.

Proposition 1. If $M, h, s \to k, t$ and (h, s) is closed then (k, t) is closed, $\mathsf{dom}\ s = \mathsf{dom}\ t$, and $\mathsf{dom}\ h \subseteq \mathsf{dom}\ k$.

3 Strong Purity

A strongly pure term is one that does not write fields of any initially existing objects. Nor does it write any local variables except possibly res.

Definition 2. Term M is *strongly pure* iff $M, h, s \to k, t$ implies $\mathsf{dom}\ h \lhd k = h$ and $\mathsf{res} \mathrel{\triangleleft\!\!\!-} t = \mathsf{res} \mathrel{\triangleleft\!\!\!-} s$. Procedure p is *strongly pure* iff $h, s \dashv p \models\!\!\to k, v$ implies $\mathsf{dom}\ h \lhd k = h$.

In this and subsequent definitions we abuse notation for brevity, omitting universal quantifiers (e.g., for h, s, k, t after the first "iff").

As an example, *pureProd* from Figure 1 is strongly pure, but *memoProd* is not. In general, strong purity allows that in the final store res may point to a new object from which other new objects are reachable, and these may point to preexisting objects —but preexisting objects are not mutated and in particular do not point to the new ones. The update $x.f := y$ is not strongly pure but $\{\mathbf{var}\ x\ \mathbf{in}\ x := \mathbf{new}\ C;\ x.f := y\}$ is. A conservative static analysis for strong purity is easy: check for complete absence of assignments and field updates (except initializers). To admit cases in which new objects are repeatedly updated, pointer analysis can be used [19].

For a procedure p, a sufficient condition for p to be strongly pure is that $\mathsf{body}\, p$ is a strongly pure term. This is not necessary because $\mathsf{body}\, p$ could assign to the parameters but only the final value of res is used. The important fact is that if p and M are strongly pure then so is the invocation $p(M)$.

Equivalence Modulo Renaming. Our objective is to justify invocations of pure methods in assertions by showing that such an assertion is the same as $\mathbf{skip}$. For this purpose we need a suitable notion of equivalence. For example, $\mathbf{assert}\ \mathit{pos}(\mathit{pureProd}(a, i))$ is not semantically equal to $\mathbf{skip}$, because it allocates a new *Cell* object. This object is only used in evaluation of the asserted formula; afterward it is unreachable, but nonetheless the final state is not identical to the final state after $\mathbf{skip}$.

To formalize a suitable notion of equivalence we adopt a standard technique: state (h, s) is equivalent to (h', s') if there is a bijective renaming from $\mathsf{dom}\, h$ to $\mathsf{dom}\, h'$ by which s, s' correspond and so do all object states. We use the term *location bijection* for a partial bijective relation on locations.

Definition 3. *($\sim_\beta$)* Let β be a location bijection. Define relation $\sim_\beta$ on values by $v \sim_\beta v'$ iff either v, v' have primitive type and $v = v'$, or $v = \mathbf{null} = v'$, or $(v, v') \in \beta$. For stores with the same domain, define $s \sim_\beta s'$ iff $s\,x \sim_\beta s'\,x$ for all $x \in \mathsf{dom}\, s$. For heaps, $h \sim_\beta h'$ iff $\mathsf{dom}\,\beta \subseteq \mathsf{dom}\, h$, $\mathsf{rng}\,\beta \subseteq \mathsf{dom}\, h'$, and $h\ o \sim_\beta h'\ o'$ for all $(o, o') \in \beta$. For states, $(h, s) \sim_\beta (h', s')$ iff $h \sim_\beta h'$ and $s \sim_\beta s'$.

Note that every variable in a store must be related. Hence if a pair of locations o, o' are related by β then locations in all fields of $h\ o$ and $h'\ o'$ must be related. In particular, $h\ o\,\mathsf{type} = h'\ o'\,\mathsf{type}$, as we treat the classname-valued field type like a primitive type. But there may be locations in $\mathsf{dom}\, h$ and in object fields in h that are not in the domain of β (and in $\mathsf{dom}\, h'$ but outside the range of β).

These relations are easily shown to be symmetric and we use this without remark in the sequel. A kind of transitivity holds, via composing bijections; what we need is in Lemma 12 in the sequel. A kind of reflexivity holds: $(h, s) \sim_{\delta\, h} (h, s)$ where $\delta\ h$ denotes the identity relation on $\mathsf{dom}\, h$. The notation $\delta\, h$ is used extensively in the sequel. For example, it lets us reformulate strong purity as follows.

Lemma 4. M is strongly pure iff $M, h, s \to k, t$ implies $k \sim_{\delta\, h} h$ and $\mathsf{res} \lhd t \sim_{\delta\, h} \mathsf{res} \lhd s$.

Equivalence for states is lifted to terms in a straightforward way, suited to partial correctness and dynamic allocation.

Definition 5. *($\approx$)* For terms M, M' to be equivalent, written $M \approx M'$, means that if $(h,s) \sim_\beta (h',s')$, $M,h,s \to k,t$, and $M',h',s' \to k',t'$ then there is $\gamma \supseteq \beta$ such that $(k,t) \sim_\gamma (k',t')$.

Here the implicitly universally quantified β, γ range over location bijections, so γ is the same as β for preexisting locations.

As an example, $\mathbf{new}\ C$ is not equivalent to $\mathbf{skip}$ because $\mathbf{new}$ updates res. On the other hand, $\mathbf{skip}$ is equivalent to the block $\{\mathbf{var}\ x\ \mathbf{in}\ x := \mathbf{new}\ C;\}$ which allocates an object that is unreachable in the final state. From initial bijection β the witnessing γ is also β, which does not have the fresh object in its domain. As another example, $x := \mathbf{new}\ C; x1 := \mathbf{new}\ D \approx x1 := \mathbf{new}\ D; x := \mathbf{new}\ C$. This can be shown by taking $\gamma = \beta \cup \{(a,d),(b,c)\}$ if the left side allocates objects a,b and the right allocates c,d (in that order). Note that $\{\mathbf{var}\ x\ \mathbf{in}\ x := \mathbf{new}\ C\}$ would not be equivalent to $\mathbf{skip}$ if we used a semantics for assignment that had an effect on res.

Theorem 6. If Q is strongly pure then $\mathbf{assert}\ Q \approx \mathbf{skip}$.

Proof. Suppose $(h,s) \sim_\beta (h',s')$, $(\mathbf{assert}\ Q),h,s \to k,t$, and $\mathbf{skip},h',s' \to k',t'$. We must choose $\gamma \supseteq \beta$ and show $(k,t) \sim_\gamma (k',t')$; we choose $\gamma = \beta$. By semantics of $\mathbf{assert}$ we have $Q,h,s \to k,u$ for some u. By strong purity of Q we have $\mathit{dom}\ h \lhd k = h$. By Definition 3 we have $\mathsf{dom}\,\beta \subseteq \mathsf{dom}\,h$, whence, using $\mathit{dom}\ h \lhd k = h$ and $h \sim_\beta h'$, we obtain $k \sim_\beta h'$. Hence $(k,s) \sim_\beta (h',s')$. By strong purity of Q we have $\mathsf{res} \lessdot u = \mathsf{res} \lessdot s$ and by semantics of $\mathbf{assert}$ we have $t = [u \mid \mathsf{res} \mapsto s(\mathsf{res})]$, hence $t = s$. By semantics of $\mathbf{skip}$ we have $(h',s') = (k',t')$, so we conclude that $(k,t) \sim_\beta (k',t')$. □

What remains is to justify that this equivalence is respected by any context and to justify that the equivalence relation is not too coarse. Regarding contexts, we have the following which is straightforward to prove for the language in Table 1. (It is instructive to prove the case for $p(M)$ because it fails for the relation $\approx^C$ in the sequel.)

Proposition 7. *(congruence)* If $M \approx N$ then $\mathcal{C}[M] \approx \mathcal{C}[N]$ for all contexts $\mathcal{C}[-]$.

Observation and Specification. Unreachable objects cannot be detected by ordinary source program constructs, but what about the predicate $(\exists o \bullet o.\mathsf{type} = C)$? Two implementations that are related by $\approx$ might be distinguished by a specification with postcondition $(\exists o \bullet o.\mathsf{type} = C)$. The could also be distinguished by a postcondition involving address arithmetic. (Congruence would also be broken.)

The decision in languages like JML to allow strongly pure method calls in specifications is only sound if predicates are restricted so they cannot make undesired distinctions. We aim for results that are generally applicable so we aim for minimal semantic conditions rather than considering syntax for formulas. This is important because verification systems often use a shallow embedding of formulas in the language of a theorem prover. One condition is that predicates should not depend on particular locations, i.e., they should respect bijective renaming. Another condition is garbage-insensitivity, which would disallow the example above.

Let $\mathit{reach}(h,s)$ be the set of locations reached transitively from s. We define $\mathit{gc}(h,s) = (\mathit{reach}(h,s) \lhd h\, ,\ s)$. For set ψ of states, we say that ψ is *healthy* iff $(h,s) \in \psi$ implies $(k,t) \in \psi$ whenever $\mathit{gc}(h,s) \sim_\beta \mathit{gc}(k,t)$.

Lemma 8. (a) If $(h, s) \sim_\beta (h', s')$ then $gc(h, s) \sim_\gamma gc(h', s')$ where γ is obtained by restricting β, to wit $\gamma = \beta \cap (reach(h, s) \times reach(h', s'))$.
(b) Suppose $M \approx N$, $M, h, s \rightarrow k, t$, and $N, h, s \rightarrow k', t'$. If ψ is healthy then $(k, t) \in \psi$ iff $(k', t') \in \psi$.

A straightforward consequence of Lemma 8(b) is the following. We refrain from spelling out the straightforward notion of satisfaction for partial correctness.

Corollary 9. Suppose $M \approx N$. Then for any $pre, post$ specification where $post$ is healthy, M satisfies the specification iff N does.

Strongly Pure Terms in Context. A direct consequence of Proposition 7 and Theorem 6 is the following.

Corollary 10. If Q is strongly pure then $\mathcal{C}[\mathbf{assert}\ Q] \approx \mathcal{C}[\mathbf{skip}]$ for all $\mathcal{C}[-]$.

With this we have justified the use of calls to strongly pure procedures in assertions. If method calls in Q are strongly pure then Q is so; and then by Corollary 10 the **assert** can be replaced by **skip**. This replacement is correctness-preserving, by Corollary 9.

4 Observational Purity

Our objective is to find a notion of purity that validates a result like Corollary 10 but allows updates of preexisting fields. Clearly not all updates can be allowed. For example, suppose Q is an invocation $p(x)$ where boolean-valued p checks whether $x.f$ is positive but also sets $x.f$ to 0. For the context $-;\ y := x.f$ we then have $\mathbf{assert}\ Q;\ y := x.f \not\approx \mathbf{skip};\ y := x.f$ Sensible updates are to encapsulated state as in Figure 1.

Visibility. A familiar notion of encapsulation suffices for our purposes. A field f of class C may or may not be visible in methods of class D. Two heaps are equivalent, as viewed in code of class C, if corresponding objects have corresponding values for all visible fields. It is well known that field access is inadequate to achieve encapsulation; additional restrictions on heap sharing are needed to prevent interference with *objects* that are intended to be private. For our purposes we need not formalize a discipline such as ownership types [9, 1] to control aliasing. The requisite assumptions can be expressed using the location bijection; a location not visible in a particular context is not in the bijection.

To impose visibility restrictions, we assume that each procedure p is declared in some class, denoted $\mathsf{class}\, p$. Furthermore, for each class C there is a set $\mathsf{vis}\ C$ of fields visible in C. (We assume that distinct classes have disjoint field names and we are not modelling subclasses or inheritance). This encoding can represent private, global, and module-scoped visibility. For an object $o \in \mathsf{dom}\, h$, $\mathsf{vis}\ C \lhd h\ o$ is the part of the object state $h\ o$ that is visible in code of class C. If $\mathsf{class}\, p = C$ then the only fields that may be read or written in $\mathsf{body}\, p$ are those in $\mathsf{vis}\, C$. The semantics is revised in a straightforward way, writing $C, M, h, s \rightarrow k, t$ to make explicit that M is executed as a constituent of a procedure of class C.

Definition 11. *($\sim_\beta^C$, $\approx^C$)* For heaps, define $h \sim_\beta^C h'$ iff $\mathsf{dom}\,\beta \subseteq \mathsf{dom}\,h$, $\mathsf{rng}\,\beta \subseteq \mathsf{dom}\,h'$, and $\mathsf{vis}\; C \lhd h\; o \sim_\beta \mathsf{vis}\; C \lhd h'\; o'$ for all $(o,o') \in \beta$. For states, define $(h,s) \sim_\beta^C (h',s')$ iff $s \sim_\beta s'$ and $h \sim_\beta^C h'$. For terms, $M \approx^C M'$ iff $(h,s) \sim_\beta^C (h',s')$, $C,M,h,s \to k,t$, and $C,M',h',s' \to k',t'$ implies there is $\gamma \supseteq \beta$ such that $(k,t) \sim_\gamma^C (k',t')$.

Note that $\sim_\beta \subseteq \sim_\beta^C$, because $\sim_\beta^C$ is $\sim_\beta$ with no fields hidden. Note also that for the store component of a state it suffices to use relation $s \sim_\beta s'$ because the store models local variables and parameters. (We omit global variables and static fields.) The following technical results are needed for later proofs.

Lemma 12. If $h \sim_\alpha g$ and $g \sim_\beta^C k$ then $h \sim_{\alpha\cdot\beta}^C k$. If $\delta\, h \subseteq \beta$, $h \sim_{\delta\, h} g$, and $g \sim_\beta^C k$ then $h \sim_{\delta\, h}^C k$. If $h \sim_\beta^C k$ and $\gamma \supseteq \beta$ then $h \sim_\gamma^C k$ provided that $\mathsf{dom}\,\gamma \subseteq \mathsf{dom}\,h$ and $\mathsf{rng}\,\gamma \subseteq \mathsf{dom}\,k$. Similarly for stores.

Observational Purity. Our goal is for **assert** $Q \approx^C$ **skip** to hold provided that Q has no effect observable in class C —e.g., Q is a call $p(x)$ that changes fields of x but only fields private to D with $D \neq C$. Following the pattern of Lemma 4 we adapt the definition of strong purity to one using the visible relations.

Definition 13. Term M is *observationally pure outside* D provided that the following holds for all $C \neq D$. If $C,M,h,s \to k,t$ then $k \sim_{\delta\, h}^C h$ and $\mathsf{res} \lessdot t \sim_{\delta\, h} \mathsf{res} \lessdot s$.

Procedure p is *observationally pure outside* D iff $h,s \dashv\! p \!\mapsto k,v$ implies $k \sim_{\delta\, h}^C h$ for all $C \neq D$.

Procedure *memoProd* of class D in Figure 1 is observationally pure outside D. It updates fields of preexisting objects but those fields are not visible outside D and the updates do not make it possible to reach the newly allocated object (return value). For initial heap h, the new object is not in the range of $\delta\, h$.

As in the case of strong purity, a sufficient but not necessary condition for a procedure to be observationally pure is that its body is. Moreover, if p and M are observationally pure outside D then so is $p(M)$.

Fact 14. If Q is observationally pure outside D then **assert** $Q \approx^C$ **skip** for all $C \neq D$.

This result is not yet satisfactory, however, because unlike the case of strong purity we do not get congruence in general. That is, $M \approx^C M'$ does not imply $\mathcal{C}[M] \approx^C \mathcal{C}[M']$ (compare Proposition 7).

Example 15. Consider the term $pos(memoProd(y,i))$, evaluation of which may well update $y.arg$ and $y.farg$. By Fact 14, **assert** $pos(memoProd(y,i)) \approx^C$ **skip** Moreover the procedures of D do not leak information about fields updated by $memoProd$, so for example we have $(\textbf{assert}\; pos(memoProd(y,i)));\; get(y) \approx^C \textbf{skip};\; get(y)$. But suppose D declared procedure $leak(self : D) : \textbf{int}\{\, \textbf{return}\; self.arg\}$. Then

$$\textbf{assert}\; pos(memoProd(y,i)); leak(y) \;\not\approx^C\; \textbf{skip}; leak(y)$$

because the result of $leak(y)$ after $memoProd(y,i)$ is i whereas after **skip** it is the initial value of $y.arg$. The problem is that $leak$ violates encapsulation and makes the cache indirectly visible. □

A related problem is that even if $h \sim_\beta^C h'$ for all $C \neq D$, it is possible for there to be $(o,o') \in \beta$ with $h\, o\, \mathsf{type} = D$ and moreover $h\, o\, arg = h'\, o'\, arg$ but $h\, o\, farg \neq h'\, o'\, farg$ because these fields are not visible outside D. From such a pair of states, the corresponding pair of results from $memoProd$ are $Cell$-objects with different val field; thus $\not\sim^C$ for the final state. Thus $memoProd \not\approx^C memoProd$.

In general the problem with congruence is that if $p \not\approx^C p$ then $M \approx^C N$ does not imply $p(M) \approx^C p(N)$. The problem is solved in section 5.

A shortcoming of Definition 13 is that checking it appears to be a nontrivial and nonstandard problem. In fact, the check can be reduced to equivalence.

Theorem 16. Suppose $M \approx^C N$ for all $C \neq D$, and suppose N is strongly pure. If N terminates when M does[1] then M is observationally pure outside D.

As an example, procedure $memoProd$ is equivalent to $pureProd$ which is strongly pure and terminates when $memoProd$ does. The termination antecedent is necessary. As an extreme case, if N never terminates then it is strongly pure and $M \approx^D N$ for any M whatsoever.

A standard technique for proving program equivalence in the presence of encapsulated state is to use simulation relations —unlike mere visible equivalence, a simulation can track correspondence of internals and impose invariants [14, 10]. Using a simulation to establish the antecedent of Theorem 16 has the added benefit of congruence.

5 Observational Purity via Simulation

This section gives the main result, equivalence of $\mathcal{C}[\mathbf{assert}\ Q]$ and $\mathcal{C}[\mathbf{skip}]$ for observationally pure Q. To this end, we generalize from specific equivalences on states to an arbitrary relation subject to some conditions. As before, the relation involves renaming of locations. So what we consider is a ternary relation, written $\asymp$ and read "couples", on two heaps and a bijection —or what amounts to the same, a family, indexed by bijections, of binary relations $\asymp_\beta$ on heaps.

Example 17. In the context of Figure 1, define $\asymp$ by $h \asymp_\beta h'$ iff (a) $h \sim_\beta^C h'$ for any $C \neq D$, and (b) for all $(o,o') \in \beta$, if $h\, o\, \mathsf{type} = D$ then $h\, o.f = h'\, o'.f$ and both $h\, o$ and $h'\, o'$ satisfy the invariant mentioned in the caption of Figure 1. From two states related by $\asymp_\beta$, $memoProd$ gives the same results, indeed that is true for all the procedures of D. □

If the cache involved other objects, an encapsulation condition would be imposed on them as well, e.g., via ownership [9, 5]. In our formulation, encapsulation at the level of classes is sufficient; it need not be instance-based.

To express healthiness conditions on $\asymp$ we use the following routine extensions.

[1] $M, h, s \Downarrow$ implies $N, h, s \Downarrow$, where $M, h, s \Downarrow$ means there exists k, t with $M, h, s \rightarrow k, t$.

Definition 18. Gived a bijection-indexed family of relations $\asymp_\beta$ on heaps, define $\asymp_\beta$ on states by $(h,s) \asymp_\beta (h',s')$ iff $h \asymp_\beta h'$ and $s \sim_\beta s'$. For terms, $M \asymp M'$ iff $(h,s) \asymp_\beta (h',s')$, $C,M,h,s \to k,t$, and $C,M',h',s' \to k',t'$ implies there is $\gamma \supseteq \beta$ such that $(k,t) \asymp_\gamma (k',t')$. Finally, $p \asymp p'$ iff $(h,s) \asymp_\beta (h',s')$, $h,s \dashv\!\! p \!\!\mapsto k,v$, and $h',s' \dashv\!\! p' \!\!\mapsto k',v'$ implies there is $\gamma \supseteq \beta$ such that $k \asymp_\gamma k'$ and $v \sim_\gamma v'$.

Definition 19 (coupling, simulation). A D*-coupling* is $\asymp$ such that

(a) if $h \asymp_\beta k$ then $\mathsf{dom}\,\beta \subseteq \mathsf{dom}\,h$ and $\mathsf{rng}\,\beta \subseteq \mathsf{dom}\,k$
(b) $h \asymp_\alpha g$ and $g \sim_\beta k$ implies $h \asymp_{\alpha\cdot\beta} k$
(c) $h \asymp_\beta k$ implies $h \sim^C_\beta k$ for all $C \neq D$

A D*-simulation* is a D-coupling such that

(d) there is a term *Init* such that for any C,β,h,s,h',s', if $(h,s) \sim^C_\beta (h',s')$ then there is some k,t,k',t',γ with $C,\mathit{Init},h,s \to k,s$ and $C,\mathit{Init},h',s' \to k',s'$ and $\gamma \supseteq \beta$ and $(k,s) \asymp_\gamma (k',s')$.
(e) $p \asymp p$ for every procedure p (in every class)

Items (a) and (b) are simple healthiness conditions (compare Definition 3 and healthy predicates in Section 3). Item (c) says that the relation reduces to equality modulo renaming, for classes other than D. A consequence is that $(h,s) \asymp_\beta (h',s')$ implies $(h,s) \sim^C_\beta (h',s')$ for all $C \neq D$.

As usual, the role of initialization is to establish a relation which does not simply follow from $(h,s) \sim^C_\beta (h',s')$ because $\sim^C_\beta$ allows arbitrary difference in non-visible fields. Item (d) is a simple formalization of initialization that follows the pattern used in the literature for single-instance modules [10]. For dynamic allocation, it is the object constructor (or default values) that established the relation [2, 8]. To cater for this in our simple setup, one can take *Init* to be an assertion of a predicate like "all existing D-objects have $\mathit{arg} = 0 = f$", or even "no D-objects exist".[2]

Item (e) requires all procedures to preserve $\asymp$. It precludes *leak* in Example 15. All procedures in Figure 1 preserve the relation in Example 17. Item (e) appears alarmingly strong. But for programs using suitable encapsulation, $p \asymp p$ holds for all p provided that it holds for all p of class D. This is the core of the theory of representation independence which has been well studied; see Section 6. The preservation result in such a theory yields the following.

Proposition 20. Suppose $\asymp$ is a D-simulation. If $M \asymp N$ then $\mathcal{C}[M] \asymp \mathcal{C}[N]$.

We shall take this as an assumption. Such a result depends on several things: conditions on the relation; conformance of the program with rules to ensure encapsulation (e.g., absence of pointer arithmetic, alias confinement); and preservation by the methods of D, which have privileged access to encapsulated state.

[2] This is not a healthy predicate as defined in Section 3, but there is no problem because the healthiness condition is not needed for preconditions.

5.1 Using D-Simulations for Purity

Definition 21. Let $\asymp$ be a D-coupling. Then M is *observationally pure for* $\asymp$ iff for all $C \neq D$, if $C, M, h, s \rightarrow k, t$ then $k \asymp_{\delta\, h} h$ and $\mathsf{res} \lhd t \sim_{\delta\, h} \mathsf{res} \lhd s$.

p is *observationally pure for* $\asymp$ if $h, s \dashv\!p\!\mapsto k, v$ implies that $h \asymp_{\delta\, h} k$.

Fact 22. Suppose M is observationally pure for some D-coupling $\asymp$. Then it is observationally pure outside D.

This Fact, together with Fact 14, implies **assert** $M \approx^{C}$ **skip** for $C \neq D$. But Theorem 16 suggests that for interchangeability of an **assert** with **skip**, it is enough to formulate observational purity as in Definition 13. The role of a coupling is then to prove the antecedent equivalence of the Theorem and in addition to enjoy a congruence property. This is worked out in our main result to follow.

Analogous to Theorem 16, one might expect the following: If $M \asymp N$ for N is strongly pure, and N terminates when M does, then M is observationally pure outside D. But the property $M \asymp N$ is only applicable to a pair of initially related states and the relation need not be reflexive, so the proof of Theorem 16 does not directly generalize. However, we can prove the following Fact. It uses the termination condition that would be imposed everywhere for simulations in a total-correctness setting.

Definition 23. *N terminates when M does, modulo* $\asymp$, iff $(h, s) \asymp_{\beta} (h', s')$ and $C, M, h, s \Downarrow$ implies $C, N, h', s' \Downarrow$.

Fact 24. If $M \asymp N$ and N is strongly pure then **assert** $M \asymp$ **skip** provided that $\asymp$ is a D-coupling and N terminates when M does, modulo $\asymp$.

5.2 Main Result

Two more ingredients are needed. The first is equivalence for properly initialized programs. The step from simulation to program equivalence requires that the programs proved equivalent are properly initialized, so that from equivalence of initial states one gets the coupled states needed to exploit the simulation. In the setting of our formalization, the following is suitable. It can be justified by an analysis of specifications as in Section 3 but taking into account visibility restrictions on specifications. For lack of space we omit the details of the justification.

Definition 25 **($\dot{\approx}^{C}$).** Suppose *Init* is given as in Definition 19. Define $M \dot{\approx}^{C} M'$ iff $\mathit{Init}; M \approx^{C} \mathit{Init}; M'$.

The point of using simulations is to get both congruence and the following which expresses how simulation implies equivalence.

Theorem 26. If $M \asymp N$ and $\asymp$ is a D-simulation then $M \dot{\approx}^{C} N$ for any $C \neq D$.

The last ingredient needed for the main result is a way to compose the main relations. We have defined several relations on terms and they enjoy various composition properties, most of which turn out not to help. What we need is the following.

Lemma 27. Suppose $\asymp$ is a D-simulation and N terminates when M does, modulo $\asymp$. If $M \mathrel{\dot{\approx}^C} N$ and $N \approx Q$ then $M \mathrel{\dot{\approx}^C} Q$.

Theorem 28. Suppose $\asymp$ is a D-simulation and N terminates when Q does, modulo $\asymp$. If $Q \asymp N$ and N is strongly pure then $\mathcal{C}[\mathbf{assert}\ Q] \mathrel{\dot{\approx}^C} \mathcal{C}[\mathbf{skip}]$ for all contexts $\mathcal{C}$ and classes $C \neq D$.

Proof. From $Q \asymp N$ we get $\mathcal{C}[\mathbf{assert}\ Q] \asymp \mathcal{C}[\mathbf{assert}\ N]$ by congruence Proposition 20. Thus $\mathcal{C}[\mathbf{assert}\ Q] \mathrel{\dot{\approx}^C} \mathcal{C}[\mathbf{assert}\ N]$ by Theorem 26. By strong purity of N and Corollary 10 we have $\mathcal{C}[\mathbf{assert}\ N] \approx \mathcal{C}[\mathbf{skip}]$. Because all constructs of the language are monotonic with respect to termination, we have that $\mathcal{C}[\mathbf{assert}\ N]$ terminates when $\mathcal{C}[\mathbf{assert}\ Q]$ does, modulo $\asymp$. Thus Lemma 27 applies to yield the result. □

Our main Theorem 28 avoids the need to use the notions of observational purity or observational purity for $\asymp$ but it comes at the cost of proving simulation with a strongly pure term. The alternative is to use observational purity following the pattern of Corollary 10. This depends on a transitivity condition on simulations that is satisfied in all the observational purity examples we have considered. It is not included in Definition 19 because no other results depend on it. Transitivity does not make sense for simulations used for changes of data representation, where the source and target of the relation are different state spaces.

Theorem 29. Suppose $\asymp$ is a D-simulation such that $\asymp_\alpha \cdot \asymp_\beta = \asymp_{\alpha\cdot\beta}$ for all α, β. If Q is observationally pure for $\asymp$ then $\mathbf{assert}\ Q \asymp \mathbf{skip}$.

Corollary 30. *Suppose $\asymp$ is a D-simulation such that $\asymp_\alpha \cdot \asymp_\beta = \asymp_{\alpha\cdot\beta}$. If Q is observationally pure for $\asymp$ then for any context $\mathcal{C}[-]$ and any class $C \neq D$ we have $\mathcal{C}[\mathbf{assert}\ Q] \mathrel{\dot{\approx}^C} \mathcal{C}[\mathbf{skip}]$.*

Proof. By Theorem 29, Proposition 20, and Theorem 26. □

6 Conclusion

To avoid logical anomalies and misleading results from runtime assertion checking, practical verification systems impose various purity requirements for specifications and annotations: no invocations allowed (ESC/Java [11]), strong purity checking (JML [12]), or unchecked advice to programmers (Eiffel [13]). But for verification to scale to large systems it is important to consider as pure even procedures which, for reasons such as caching, update preexisting objects, provided that the updates are unobservable. Absence of anomalies for formula Q can be made precise by equating $\mathbf{assert}\ Q$ with $\mathbf{skip}$ — the presence of Q has no effect on the properties of following code— using a notion of equivalence that is a congruence and correctness-preserving.

Our main result shows that Q satisfies the equivalence, in the context of some class C, provided that it simulates, in the context of a different class D, a strongly pure term. The main application is where Q invokes procedures of D and is used to reason about procedures of C. The result reduces admissibility of Q to a proof obligation (simulation) together with static analysis for strong purity rather than a more specialized analysis. To apply our results one needs a method for defining D-simulations. In particular, it is essential that condition (e) in Definition 19 need only be checked for procedures of D; for procedures of $C \neq D$ it should follow by a preservation/congruence theorem. Such theories (analogs of our Proposition 20 and Theorem 26) have been developed for many sorts of languages [14, 10]). For Java-like languages, Banerjee and Naumann [2] give such a theory under the assumption of suitable alias control which can be achieved using static analysis [2, 15, 9]; a similar result has recently been given [4] using state-based enforcement of encapsulation [5, 16]. Such results are difficult to prove for complex languages so it is fortunate that we could treat observational purity in terms of existing formulations.

In justifying the choice of program equivalence we have uncovered an issue for strong purity. If, in postconditions, it is allowed to use quantification over all allocated objects, even unreachable ones, then pre/post specifications can "observe" allocation and even strong purity is not sound. Quantifications over all allocated objects have been used in some settings, e.g., the program invariants of the Boogie discipline [5, 16], but in that context programmer-defined predicates are in fact restricted to reachability in terms of auxiliary fields. Pierik et al. [17] advocate global invariants such as "there is at most one C-object" which are apparently incompatible with strong purity.

The most closely related work is that of Barnett et al. [6], where a seemingly ad hoc condition combining Definitions 13 and 21 is proposed. Rather than drawing on the general theory of encapsulation and simulation, the work uses the noninterference property from information security. There may be some advantage to that approach in avoiding the full generality of simulation theory. It is being explored as part of the Spec#/Boogie project [5]. The full version of [6] will include examples of observationally pure procedurs from the .NET libraries. Leavens et al. [12] discuss the rationale and static analysis for strong purity in JML. Sălcianu and Rinard [19] give a more precise static analysis for the strong purity condition. Program equivalence modulo garbage collection has been studied by Calcagno et al. [7] and others [2].

To extend observational purity to total correctness, equivalence is replaced by refinement of **assert** Q by **skip**. Suitable simulation theory for a Java-like languages can be adapted from existing work [8, 2, 4]. We conjecture that the extension to concurrency is also straightforward. Procedures called in assertions need to be deterministic in order to apply logical reasoning, but our theory depends in no way on determinacy.

We leave open the question of completeness: if M is observationally pure outside D then is it simulated by some stongly pure N? Given such M, it is straightforward to define a relation R such that R is strongly pure (semantically) and suitably coupled with M. But the coupling needs to be a simulation for all procedures of D and R needs to be denoted by a term in the language.

References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *European Conference on Object-Oriented Programming*, pages 1–25, 2004.

2. A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 2002. Accepted, revision pending. Extended version of [3].
3. A. Banerjee and D. A. Naumann. Representation independence, confinement and access control. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 166–177, 2002.
4. A. Banerjee and D. A. Naumann. State based ownership, reentrance, and encapsulation. Submitted, Dec. 2004.
5. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
6. M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2004. Technical Report NIII-R0426, University of Nijmegen.
7. C. Calcagno, P. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Comput. Sci.*, 298(3):557–581, 2003.
8. A. L. C. Cavalcanti and D. A. Naumann. Forward simulation for data refinement of classes. In *Formal Methods Europe*, volume 2391 of *LNCS*, pages 471–490, 2002.
9. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, Nov. 2002.
10. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
11. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conf. on Program. Lang. Design and Implementation (PLDI)*, pages 234–245, 2002.
12. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In *Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS*, pages 262–284, 2003.
13. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, second edition, 1997.
14. J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
15. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. Number 2262 in *LNCS*. Springer, 2002.
16. D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *IEEE Symp. on Logic in Computer Science (LICS)*, pages 313–323, 2004.
17. C. Pierik, D. Clarke, and F. S. de Boer. Creational invariants. In *Proceedings of ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2004. Technical Report NIII-R0426, University of Nijmegen.
18. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
19. A. Sălcianu and M. 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.