Observational Purity and Encapsulation

David A. Naumann [1]

Stevens Institute of Technology
Castle Point on Hudson, Hoboken, NJ 07030 USA

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 weakly 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 a notion of observational purity, justifies the use of weakly and observationally pure methods in specifications, and shows that a method is observationally pure if it simulates a weakly pure method.

Key words: specification and verification, information hiding, benevolent side effects

1 Introduction

Consider the correctness statement $\{y+1 > 0\}\ x := y+1\ \{x > 0\}$. The expression $y+1$ plays two different roles here: In the precondition $y+1 > 0$ it is a term in a formula, which has a standard meaning in mathematical logic. In the command $x := y+1$ it is an expression to be evaluated during program execution. The axiom of assignment successfully blurs the distinction, substituting a program expression in a formula. For reasoning about + in the formula, soundness demands rules that are consistent with + as an executable

[1] Partially supported by the National Science Foundation (CCR-0208984, CCF-0429894) and Microsoft Research.

operation —for this reason, the paper in which Hoare (1969) introduced his logic begins with a discussion of computer arithmetic. The logic also demands that expressions in assignments and branch conditions have no side effects —surely side effects have no place in logical formulae?

In the many theoretical and practical works that followed Hoare's paper the prohibition against side effects has typically been enforced by distinguishing between mathematical primitives like + and programmed procedures. The latter are considered impure; they are not allowed in specifications or other assertions. The influential book by Liskov and Guttag (1986) and the Larch projects (Guttag and Horning, 1993), for example, go to considerable lengths to maintain the distinction. Recently, however, in the context of object oriented languages there has been a trend to loosen the distinction, allowing assertions that invoke procedures, subject to purity conditions. For practical reasons, purity is relaxed to allow allocation of new objects (Leavens et al., 2003) and even benevolent side effects (Barnett et al., 2004b).

The purpose of this paper is to provide a theoretical foundation for these relaxed notions of purity and to elucidate ways to check purity. Notions of purity have other applications, e.g., in program transformation, but we confine attention to program specification and verification. This problem is particularly pressing for object-oriented implementation languages where effects are ubiquitous and practical sound reasoning is difficult to achieve due to several problematic language features.

Eiffel (Meyer, 1997), JML (Leavens et al., 2003), Spec# (Barnett et al., 2005) and perhaps other specification systems for Java-like languages attempt to make verification tools more practical by using relatively little specialized notation. Mathematical functions are provided in the form of library procedures.[2] Owing to the limited type structure of the programming language, a mathematical entity —even as simple as a pair of integers— usually needs to be encoded as a heap-allocated object. For this reason it is necessary to allow procedures that are *weakly pure*, meaning that they may allocate new objects though not mutate existing ones. For example, JML allows a procedure to be labelled as pure and prescribes a static check that there are no field updates. The idea is that freshly allocated objects are only used in the course of evaluating the assertion and can be ignored afterwards.

Within the evaluation of an assertion, it is not at all clear that allocation effects can be ignored. The key example is the equality test

$$\textbf{new}\ C == \textbf{new}\ C$$

[2] One should say "methods", but dynamic dispatch and inheritance are not relevant in this paper.

```
class $Cell$ {
   public $val$ : int;
   proc $pos(c : Cell)$ : bool { return $c.val > 0$; }    }
class $D0$ {
   private $f, arg, farg$ : int;
   proc $pureProd(s : D0, n :$ int$) : Cell$ {
      $x : Cell :=$ new $Cell$;  $x.val := s.f * n$;   return $x$; }
   proc $memoProd(s : D0, n :$ int$) : Cell$ {
      $x : Cell :=$ new $Cell$;
      if $n = 0$ then $x.val := 0$; return $x$;
      elseif $s.arg \neq n$ then $s.arg := n$;  $s.farg := s.f * n$;   end;
      $x.val := s.farg$;   return $x$; }
   proc $get(s : D0)$ : int { return $s.f$}
   proc $set(s : D0, v :$ 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 $D0$-objects o.

which is always false in executed code since two fresh objects are distinct; it could be true if the effect of the first **new** were to be discarded. Several solutions are possible, e.g., Darvas and Müller (2006) introduce an explicit parameter for the heap and thread its updates through the formula.

In this paper we focus on the problem of effects at the boundary between assertions and programs, not within assertions. But we go beyond weak purity. Many software libraries include procedures that one would expect to be pure, but which in fact mutate preexisting objects for purposes such as lazy initialization, memoization, and other optimizations. A standard optimization example is the move-to-front heuristic for a set represented by an unordered list. An example of lazy initialization is found in the implementation of `String.hashCode` in Java. The solution adopted in JML is to duplicate such library procedures with pure ones to be used in specifications. But this results in the proliferation of redundant libraries and it undercuts the goal of making specifications seem more familiar to ordinary programmers. An alternative is to liberalize the notion of purity even further, to *observational purity* which allows updates of preexisting objects so long as these effects are encapsulated in a suitable sense.

In the sequel a simple but representative example is used; see Figure 1. Instances of class $D0$ hold an integer in field f. Procedures *pureProd* and *memoProd* multiply f by parameter n. The product is returned in the *val* field of a *Cell* object, to illustrate weak purity. To illustrate observational purity, the program memoizes a product $f * arg$ in field *farg*. In the context of some class B and, say, a procedure with parameters $d : D0$ and $i :$ **int**, one might find expression $pos(pureProd(d, i))$ in a specification or intermediate assertion. An 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 in 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.

Hoare (1972) noted that a procedure specified to have no effect may be allowed to have *benevolent* side effects, i.e., effects on the internal representation and which do not affect the abstract value represented. The justification relies on encapsulation: behavior of a client of an abstraction is supposed to be independent from the representation of the abstraction. Benevolent side effects within a program need no special consideration: they are allowed simply in virtue of the use of abstractions in specification (and suitable interpretation of modifies specifications (Liskov and Guttag, 1986)). Our contribution is to treat effects in specifications and other forms of assertions. This also justifies use of the naive axiom of assignment where the assigned expression includes procedure calls. To this end we adapt the notion of simulation, the standard technique for proving equivalence of implementations that differ in their data representation. In this way we obtain a sound and general theory without the need to prove the hardest of the results from scratch. Unlike the work cited above, we address encapsulation in the presence of shared mutable objects.

In order to treat weak purity, we develop a notion of program equivalence that is insensitive to garbage. This is used to give a rigorous justification of weakly pure expressions in specifications, which had not appeared in the literature. We also make precise the necessary restrictions on specifications; for example, to disallow $(\exists o \bullet \mathit{allocated}(o) \wedge o.\mathsf{type} = C)$ which is sensitive to allocation. Equivalence then serves as basis for a suitable notion of simulation with which we justify observationally pure expressions in specifications. Our formulation is compatible with extant encapsulation systems for specific object-oriented languages, such as Ownership Types (Clarke et al., 2001; Clarke and Drossopoulou, 2002) and assertion-based ownership (Barnett et al., 2004a).

A conservative static analysis for weak purity is easy: check for complete absence of assignments and field updates (except initializers). To admit cases in which new objects are repeatedly updated, e.g., in a temporary data structure, pointer analysis can be used (Sălcianu and Rinard, 2004).

For checking observational purity, our theory supports two techniques. One technique is to check the property directly, for a special case that can be reduced to "visible indistinguishability" plus an object invariant. Then reasoning about invariants can be combined with a static analysis developed for secure information flow. This is discussed in Sections 5.3 and 6. The other technique exploits the fact that simulation can be used to prove equivalence. It can be considered to be the main result of the paper: a procedure is observationally pure if it simulates one that is weakly pure in the sense of allowing allocation of new objects but no mutation of preexisting ones (Section 5.2). This result means one can dispense with the notion of observational purity per se.

Consider now the hazards of effects in assertions. For runtime assertion checking, a straightforward implementation executes the asserted expression and, say, throws a special exception if the assertion does not evaluate to true. Thus execution of the fragment **assert** $Q; S$ from some initial state results in execution of S in a state different from the initial one, if Q has effects. But runtime assertion checking is typically used as a means for testing; in production runs of the program, assertion checking may well be omitted for reasons of performance. So we would like to restrict effects so that "**assert** Q" is equivalent to "**skip**".

Static verification of a partial correctness assertion $\{P\}S\{Q\}$ connects executions of S with predicates P, Q on initial and final states. How is P to be interpreted as a predicate, if the semantics of a procedure it calls has an effect? A simple answer is to ignore the effect: a state satisfies P if execution of P returns true, regardless of what else it does. Clearly this does not match runtime checking. Moreover, automated verifiers sometimes encode the partial correctness condition in the form $\{\mathit{true}\}S'\{\mathit{true}\}$ where S' is **assume** P; S; **assert** Q, thus embedding the specification in the program itself. Further rewriting may be done, e.g., transformation to single assignment form in order to improve performance of a particular theorem prover. Such transformations could make it difficult to ignore effects in P or Q.

In each scenario, inconsistencies are avoided so long as "**assert** Q" is equivalent to "**skip**". (We omit discussion of **assume** since it is very similar to **assert**.) As a simple and general way to justify the use of procedure calls in specifications, we investigate conditions under which the equivalence holds. The notion of equivalence must be compositional, i.e., a congruence, and correctness-preserving. Our account addresses partial correctness which suffices to illustrate the core ideas and difficulties. In the conclusion we describe how the approach can be adapted to total correctness.

Weak 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 $D0$, meaning that if it occurs in Q then **assert** Q is equivalent to **skip** only in the context of a class other than $D0$. However, our results also account for using Q in the specification of procedures of $D0$, e.g., if the example class $D0$ included another procedure that mentioned *memoProd* in its specification. This is discussed at the start of Section 5.2.

Overview. Section 2 formalizes a simple language sufficient to illustrate the ideas. Section 3 defines weak purity. A notion of equivalence is defined and justified, such that **assert** Q is equivalent to **skip** for weakly pure Q. Section 4 adds visibility to the language in order to formalize observational purity. It is shown that **assert** Q is equivalent to **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. Building on the results of Sections 3 and 4, it shows that if Q simulates some weakly pure term then **assert** Q is equivalent to **skip** in any context. Section 6 concludes.

This paper is revised and extended from the conference version (Naumann, 2005). It includes full proofs and more thorough discussion of related work, as well as expository changes including adoption of the term "weak purity" for what had been called strong purity.

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 $v \ntriangleleft f$ 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

To formalize our results with minimum fuss and maximum perspecuity, we consider a simple procedural language with dynamically allocated mutable objects. The syntax is given in Table 1. Note that there is no syntactic distinction between expressions and commands. The short word *term* is used for both.[3]

[3] Language theorists should note that a reverse terminology is sometimes used, where "terms" are the effect-free "expressions".

$C, D \in \mathit{ClassName}$		$x, y \in \mathit{VariableName}$
$p \in \mathit{ProcedureName}$		$f \in \mathit{FieldName}$
$M \ ::=$	**assert** M	
$\mid$	$x \mid x := M$	read, write local variable
$\mid$	$M.f \mid x.f := M$	read, write field of heap object
$\mid$	**new** C	reference to freshly allocated object of class C
$\mid$	$p(M)$	invoke procedure p on argument M
$\mid$	$M = M \mid$ **null**	pointer equality test, the null pointer
$\mid$	**var** x **in** M	local block
$\mid$	**skip** $\mid M; M \mid$ **if** M **then** M **else** M $\mid$ **while** M **do** M	

Table 1
Grammar of effectful terms, omitting arithmetic primitives etc.

The language is designed to streamline the formalism as much as possible and of course to encompass side effects of expressions in assertions. Thus it includes some very odd programs and some which are idiomatic in the C language but nonetheless unattractive from a reasoning perspective. For practical purposes it is the effects of **new** and procedure invocation that matter. We see little interest in expressions that have side effects in the form of assignments.

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 on which procedure calls would be dynamically dispatched. 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, body p, should be given. In Section 4 we add visibility control for fields, by associating procedures with classes as in the concrete syntax of 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 as well as specification constructs such as quantifiers and regular path expressions. What we need is that the language satisfies Proposition 2.1 and Assumption 5.1 in the sequel; for example, sequential fragments of Java and C# (Banerjee and Naumann, 2005a).

The details of typing, although important to preclude pointer arithmetic, are ignored in the formalism for readability. In particular, the semantics is only intended to be applied to type-correct programs, e.g., to execute a field dereference $x.f$ we assume that, if not null, the value of x is an object of the class in which f is a field. Figure 1 uses the syntax **proc** $p(x : T) : T'\{M\}$ for procedure declarations, but we refrain from including that in the gramar since it involves types.

Semantics. The language happens to be deterministic; in particular an arbitrary but deterministic memory allocator is used. But 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 of which 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. Note that **null** is a value but not a location and therefore not in the domain of a heap. Integer and boolean values are also distinct from locations. A *global 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. The term *state*, without qualification, means global state.

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. A bit of care has been taken in the semantics so that desirable program equivalences are not falsified, e.g., the semantics of $x := M$ restores res after evaluation of M. But only the treatment of res in the semantics of **assert** has any bearing on the main results.

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

The semantics of a procedure invocation $p(N)$ is defined using an auxiliary relation. Suppose $\mathsf{body}\, p$ is the term M. The relation $\dashv p \vdash\!\!\to$ is defined by

$$h, s \dashv p \vdash\!\!\to k, v \iff \text{there is } t \text{ such that } M, h, s \to k, t \text{ and } v = t\,\mathsf{res}$$

This gives the meaning of a p in terms of its local state, where s stores the parameter and res for M. The semantics of an invocation $p(N)$ provides initial values for the parameter and for res and it uses v as the result from $p(N)$.

The semantics makes **assert** N yield a final state only if N yields a final state (k, u) in which $u\,\mathsf{res}$ is true. The final state of the **assert** reflects the effect of N on the heap and on the store, except that res retains its initial value just as it does in the semantics of assignment —otherwise an **assert** could never be equivalent to **skip**.

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 $t = [u \mid x \mapsto u\,\mathsf{res} \mid \mathsf{res} \mapsto s\,\mathsf{res}]$ 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 $g, \mathsf{arg}(r\,\mathsf{res}) \dashv\!p\!\mapsto k, v$ 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
$N; N'$	$N, h, s \to g, u$ and $N', g, u \to k, t$ for some g, u
var x **in** N	$N, h, [s \mid x \mapsto \mathit{default}] \to k, u$ for some u and if $x \in \mathsf{dom}\,s$ then $t = [u \mid x \mapsto s\ x]$ else $t = x \lessdot u$
while Q **do** N	$Q, h, s \to g, r$ for some g, r, and either $r\,\mathsf{res} = \mathit{false}$ and $k, t = g, r$ or else $r\,\mathsf{res} = \mathit{true}$ and $(N; \mathbf{while}\ Q\ \mathbf{do}\ N), g, r \to k, t$

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 of object o by $[h \mid o.f \mapsto v]$. The *default_C_state* has, in particular, value C for the immutable field type.

In the case that M is a loop **while** Q **do** N, the semantics of M is defined in terms of M itself in an unfolding of the loop. Thus Table 2 must be viewed as an inductive definition, i.e., $\to$ is the least relation satisfying the conditions. In the cases omitted from the Table, suitable semantics are straightforward.

Example 2.1 For readers interested in details of the semantics, here is an illustrative but otherwise useless example. Consider execution of the term "$x.f;$ **assert** $(y := ((z := 1); 3)) = 2$" from initial state (h, s). Evaluation of $x.f$ changes the store to $[s \mid \mathsf{res} \mapsto v]$ where v is the value in h 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, $((z := 1); 3)$ is evaluated, with the effect of setting z to 1 and res to 3. Then $y := \ldots$ can be completed, updating y but restoring res to v. Then the equality is evaluated, comparing v with 2. If they are equal, the final store is $[s \mid \mathsf{res}, z, y \mapsto v, 1, 3]$ because the semantics of **assert**, like $:=$, discards the intermediate res values. If $v \neq 2$ there is no outcome. There is also no outcome if f is not in the fields of class $h(s\,x).\mathsf{type}$, but that cannot happen in a program that is typable. □

In the rest of the paper we confine attention to *closed states*, i.e., (h, s) 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 restriction to closed states could be dropped, at the cost of some complications starting with Definition 3.2. There is little motivation to drop the restriction because the following is easily proved for the language in Table 1.

Proposition 2.1 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 Weak purity

A *strongly pure* term is one with no effect whatsoever. In our semantics that means its final state differs from its initial state only in the special variable res. A weakly 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 3.1 Term M is *weakly pure* iff $M, h, s \to k, t$ implies $\mathsf{dom}\, h \lhd k = h$ and $\mathsf{res} \ntriangleleft t = \mathsf{res} \ntriangleleft s$. Procedure p is *weakly pure* iff $h, s \dashv p \mapsto k, v$ implies $\mathsf{dom}\, h \lhd k = h$. $\square$

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* in Figure 1 is weakly pure, but *memoProd* is not. Weak 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 weakly pure but the following block is: $\{\textbf{var}\ x\ \textbf{in}\ x := \textbf{new}\ C;\ x.f := y\}$.

For a procedure p, a sufficient condition for p to be weakly pure is that $\mathsf{body}\, p$ is a weakly pure term. This is not necessary because $\mathsf{body}\, p$ could assign to the parameters but only the final value of res is used. What matters most is the following.

Fact 3.1 If p and M are weakly pure then so is the invocation $p(M)$.

Proof: Suppose $p(M), h, s \to k, t$. We must show that $\mathsf{dom}\, h \lhd k = h$ and $\mathsf{res} \ntriangleleft t = \mathsf{res} \ntriangleleft s$. By semantics, we have some g, r, v with $M, h, s \to g, r$ and $g, \mathsf{arg}(r\, \mathsf{res}) \dashv p \mapsto k, v$ and $t = [r \mid \mathsf{res} \mapsto v]$. To show $\mathsf{res} \ntriangleleft t = \mathsf{res} \ntriangleleft s$, observe that $\mathsf{res} \ntriangleleft t = \mathsf{res} \ntriangleleft [r \mid \mathsf{res} \mapsto v] = \mathsf{res} \ntriangleleft r$ and by weak purity of M we have $\mathsf{res} \ntriangleleft r = \mathsf{res} \ntriangleleft s$. Now we show $\mathsf{dom}\, h \lhd k = h$. By weak purity of p we have

$\mathsf{dom}\, g \lhd k = g$. By Proposition 2.1 we have $\mathsf{dom}\, h \subseteq \mathsf{dom}\, g$, hence we have $\mathsf{dom}\, h \lhd k = \mathsf{dom}\, h \lhd g$. By weak purity of M we have $\mathsf{dom}\, h \lhd g = h$. $\square$

It is easy to show that weak purity is preserved by the other term constructs —except for variable assignment and field update of course.

3.1 Equivalence modulo renaming

Our objective is to justify invocations of pure procedures in assertions by showing that such an assertion is the same as **skip**. For this purpose we need a suitable notion of equivalence. For example, **skip** is not semantically equal to **assert** $pos(pureProd(a, i))$ because the latter 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 **skip**. 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 relevant object states. We use the term *location bijection* for a partial bijective relation on locations.

Definition 3.2 (state equivalence $\sim_\beta$) 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 $s \sim_\beta s'$. $\square$

Note that every variable in a store must be related. Hence if a pair of locations o, o' are related by β, and $h \sim_\beta h'$, then locations in all fields of $h\, o$ and $h'\, o'$ must be related. In particular, $h\, o.\mathsf{type} = h'\, o'.\mathsf{type}$, since 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).

A kind of transitivity holds, via composing bijections; what we need is in Lemma 4.1 in the sequel. A kind of reflexivity holds: $(h, s) \sim_{\delta\, h} (h, s)$ where $\delta\, h$ denotes the identity relation on $\mathsf{dom}\, h$. Also, $\sim_{\delta\, h}$ is symmetric. The notation $\delta\, h$ is used extensively in the sequel. It lets us characterize weak purity as follows.

Lemma 3.2 (weak purity) *M is weakly pure iff $M, h, s \to k, t$ implies $k \sim_{\delta\, h} h$ and $\mathsf{res} \lhd\!\!\!\!- t \sim_{\delta\, h} \mathsf{res} \lhd\!\!\!\!- s$.*

Proof: Because h is closed and $\mathsf{dom}\,h \subseteq \mathsf{dom}\,k$ (from Proposition 2.1), the object states in $\mathsf{dom}\,h \lhd k$ have no locations outside $\mathsf{dom}\,h$. Thus, by the definitions, $k \sim_{\delta h} h$ is equivalent to $\mathsf{dom}\,h \lhd k = h$. Similarly, since the stores in (h, s) and (k, t) are also closed, we get the result for states. $\square$

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

Definition 3.3 (term equivalence $\approx$) For terms M, M' to be equivalent, written $M \approx M'$, means that if $(h, s) \sim_\beta (h', s')$ and $M, h, s \to k, t$ and $M', h', s' \to k', t'$ then there is $\gamma \supseteq \beta$ such that $(k, t) \sim_\gamma (k', t')$. $\square$

Here the implicitly universally quantified β, γ range over location bijections, so γ is the same as β for preexisting locations. The longwinded condition can be depicted as follows.

$$\begin{tikzcd}
(h,s) \arrow[r, phantom, "\sim_\beta"] \arrow[d, "M"'] & (h',s') \arrow[d, "M'"] \\
(k,t) \arrow[r, phantom, "\sim_\gamma"] & (k',t')
\end{tikzcd}$$

The need for the inclusion $\gamma \supseteq \beta$ is discussed later, following Proposition 3.4.

As an example, **new** C is not equivalent to **skip** because **new** updates res. On the other hand, **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 an 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 $x := x$ would not be equivalent to **skip** if we used a semantics for assignment that had an effect on res.

Finally we can begin to justify weak purity.

Theorem 3.3 If Q is weakly pure then **assert** $Q \approx$ **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 **assert** we have $Q, h, s \to k, u$ for some u. By weak purity of Q we have $\mathsf{dom}\,h \lhd k = h$. By Definition 3.2 we have $\mathsf{dom}\,\beta \subseteq \mathsf{dom}\,h$, whence, using $\mathsf{dom}\,h \lhd k = h$ and $h \sim_\beta h'$, we obtain $k \sim_\beta h'$. Hence $(k, s) \sim_\beta (h', s')$. By weak purity of Q we have $\mathsf{res}{\lhd}u = \mathsf{res}{\lhd}s$ and by semantics of **assert** we have $t = [u \mid \mathsf{res}{\mapsto}s\,\mathsf{res}]$, hence $t = s$. By semantics of **skip** we have $(h', s') = (k', t')$, so we conclude that $(k, t) \sim_\beta (k', t')$. $\square$

What remains is to justify that this equivalence is respected by any context and to justify that the equivalence relation is not too coarse.

A *context* $\mathcal{C}[-]$ is a term that may have a missing subterm, called the *hole* and indicated by $-$. As usual, $\mathcal{C}[M]$ denotes substitution of M for the hole, allowing free variables of M to be captured. For example, if $\mathcal{C}$ is $x := -; y := N$ then $\mathcal{C}[M]$ is $x := M; y := N$ and if $\mathcal{C}[-]$ is **var** x **in** $-$ then $\mathcal{C}[x := N]$ is **var** x **in** $x := N$.

Proposition 3.4 (congruence) If $M \approx N$ then $\mathcal{C}[M] \approx \mathcal{C}[N]$ for all contexts $\mathcal{C}[-]$.

This is straightforward but not trivial to prove for the language in Table 1. We sketch here just a couple of highlights; a detailed proof is in appendix A. One proves the result together with the fact that the auxiliary relation $-\!|p|\!\!\rightarrow$ preserves $\sim$. The proof is by induction on the structure of $\mathcal{C}[-]$ and on the calling graph, which is acyclic by an assumption in Section 2 (otherwise the proof would use fixpoint induction). It is instructive to prove the case for $p(M)$ because it fails for the relation $\approx^C$ in the sequel. Moreover, it shows why we need $\gamma \supseteq \beta$ in Definition 3.3.

So consider the context $p(-)$ for procedure p. Suppose $M \approx N$. To show $p(M) \approx p(N)$, suppose $(h, s) \sim_\beta (h', s')$. Let $M, h, s \to g, r$ and $M, h', s' \to g', r'$, so by $M \approx N$ there is some $\alpha \supseteq \beta$ with $(g, r) \sim_\alpha (g', r')$. Now apply the semantics of p, i.e., suppose $g, \mathsf{arg}(r\,\mathsf{res}) -\!|p|\!\!\rightarrow k, v$ and $g', \mathsf{arg}(r'\,\mathsf{res}) -\!|p|\!\!\rightarrow k', v'$. By the induction hypothesis about $-\!|p|\!\!\rightarrow$ we have some $\gamma \supseteq \alpha$ such that $k \sim_\gamma k'$ and $v \sim_\gamma v'$. It follows from $r \sim_\alpha r'$ and $\gamma \supseteq \alpha$ that $(k, [r \mid \mathsf{res} \mapsto v]) \sim_\gamma (k', [r' \mid \mathsf{res}{\mapsto}v'])$ which completes the proof of $p(M) \approx p(N)$.

A direct consequence of Proposition 3.4 and Theorem 3.3 is the following.

Corollary 3.5 If Q is weakly pure then $\mathcal{C}[\textbf{assert}\ Q] \approx \mathcal{C}[\textbf{skip}]$ for all $\mathcal{C}[-]$.

It remains to show that equivalence is correctness preserving.

3.2 Observation and specification

Unreachable objects cannot be detected by ordinary source program constructs. But consider the predicate $(\exists o \bullet o.\mathsf{type} = C)$ where o ranges over allocated objects and C is some class. Two implementations that are related by $\approx$

might be distinguished by a specification with postcondition $(\exists o \bullet o.\mathsf{type} = C)$. They could also be distinguished by a postcondition involving address arithmetic. Congruence would also be broken if such a predicate could be expressed by a term in an assertion.

The decision in languages like JML (Leavens et al., 2003) to allow weakly pure procedure 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 address this issue in terms of semantic conditions. That is, we consider *predicates* to be sets of states —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 rules out the example $(\exists o \bullet o.\mathsf{type} = C)$.

To formalize the conditions just mentioned, first define $reach(h, s)$ be the set of locations reached transitively from s. The garbage collection function on states is defined by $gc(h, s) = ((reach(h, s) \lhd h)\ ,\ s)$. For set ψ of states, we say that ψ is *healthy* iff $(h, s) \in \psi$ implies $(k, t) \in \psi$ whenever $gc(h, s) \sim_\beta gc(k, t)$ for some β.

Terms in our illustrative language denote healthy predicates. To make this claim precise, consider any term Q. Define the set of states $[Q]$ by $(h, s) \in [Q]$ iff there is some k, t with $Q, h, s \rightarrow k, t$ and $t\,\mathsf{res} = true$. To show that $[Q]$ is healthy, suppose $(h, s) \sim_\beta (h', s')$ and $Q, h, s \rightarrow k, t$ and $Q, h', s' \rightarrow k', t'$. Now $Q \approx Q$ by Proposition 3.4, so we get $(h, t) \sim_\gamma (h', t')$ for some γ, whence $(t\,\mathsf{res}) \sim_\gamma (t'\,\mathsf{res})$. Thus $t\,\mathsf{res} = true$ iff $t'\,\mathsf{res} = true$, by definition of $\sim_\gamma$.

Finally, equivalent terms are not distinguished by healthy specifications. Here a specification is a pair $pre, post$ of predicates. We refrain from spelling out the usual notion of satisfaction for partial correctness.

Proposition 3.6 Suppose $M \approx N$. Then for any $pre, post$ specification where $post$ is healthy, M satisfies the specification iff N does.

This is a straightforward consequence of part (b) in the following.

Lemma 3.7 (a) If $(h, s) \sim_\beta (h', s')$ then $gc(h, s) \sim_\gamma gc(h', s')$ where γ is obtained by restricting β; that is, $\gamma = \beta \cap (reach(h, s) \times reach(h', s'))$.
(b) Suppose $M \approx N$ and $M, h, s \rightarrow k, t$ and $N, h, s \rightarrow k', t'$. If ψ is healthy then $(k, t) \in \psi$ iff $(k', t') \in \psi$.

Proof: For (a): γ agrees with β on everything still reachable in $gc(h, s)$ (respectively, $gc(h', s')$). For (b): We have $(h, s) \sim_{\delta\, h} (h, s)$ by definitions. By $M \approx N$ there is some $\beta \supseteq \delta\, h$ with $(k, t) \sim_\beta (k', t')$. Then by part (a) there is γ with $gc(k, t) \sim_\gamma gc(k', t')$ so the result holds by healthiness of ψ. □

With Proposition 3.6 we have completed the justification of calls to weakly pure procedures in assertions. If procedure calls in Q are weakly pure then Q is too, using Fact 3.1. So by Corollary 3.5 the **assert** can be replaced by **skip**. This replacement is correctness-preserving, by Proposition 3.6.

4 Observational purity

Our objective is to find a notion of purity that validates a result like Corollary 3.5 but which 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 **assert** $Q;\ y := x.f \ \not\approx$ **skip**; $y := x.f$. The example in Figure 1 suggests that some updates can be allowed provided the modified state is suitable encapsulated.

4.1 Visibility

A simple notion of encapsulation suffices for our purposes. (As discussed in Section 6, it can be specialized to extant encapsulation systems for languages like Java.) A field f of class C may or may not be visible in procedures 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 restriction of 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 (Hogg et al., 1992; Leino and Nelson, 2002; Banerjee and Naumann, 2005a). For our purposes we just need a semantic notion of visibility that takes locations into account. This can be expressed using the location bijection; a location not visible in a particular context is not in the bijection.

We refrain from formalizing concrete syntax like the public/private field modifiers in Figure 1 or modules and module scoped fields. Instead we directly formalize the visibility relation for a program. We assume that distinct classes have disjoint field names. We assume that each procedure p is declared in some class, denoted $\mathsf{class}\,p$. Furthermore, for each class C we are given a set $\mathsf{vis}\,C$ of fields visible in C. Note that, since $\mathsf{vis}\,C$ can contain fields of any and all classes, this encompasses fields with private, public (global), and module-scoped visibility. (And even "protected" visibility for subclasses, though we do not model subclasses or inheritance.)

For a location $o \in \mathsf{dom}\,h$, we can write $\mathsf{vis}\,C \lhd h\ o$ for the part of the object state $h\,o$ that is visible in code of class C. Thus, using $\sim_\beta$ for stores from Definition 3.2, we can write $\mathsf{vis}\,C \lhd h\,o \sim_\beta \mathsf{vis}\,C \lhd h'\,o'$ to say that the two object states, $h\,o$ and $h'\,o'$, have related visible fields

If $\mathsf{class}\,p = C$ then the only fields that may be read or written in $\mathsf{body}\,p$ are those in $\mathsf{vis}\,C$. Henceforth we confine attention to programs that respect this visibility condition. To make clear when we are considering a term M that would be allowed in a procedure of class C, we revise the semantic notation, writing $M, h, s \xrightarrow{C} k, t$.

The visibility-based equivalences are straightforward.

Definition 4.1 **($\sim^C_\beta$, $\approx^C$)** For heaps, define $h \sim^C_\beta h'$ 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 $(h, s) \sim^C_\beta (h', s')$ iff $s \sim_\beta s'$ and $h \sim^C_\beta h'$.
For terms, define $M \approx^C M'$ iff $(h, s) \sim^C_\beta (h', s')$ and $M, h, s \xrightarrow{C} k, t$ and $M', h', s' \xrightarrow{C} k', t'$ implies there is $\gamma \supseteq \beta$ such that $(k, t) \sim^C_\gamma (k', t')$. $\square$

Note that $h \sim_\beta h'$ implies $h \sim^C_\beta h'$, for any β, C, because $\sim_\beta$ is like $\sim^C_\beta$ 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 and we only use $\sim^C_\beta$ to relate states for executions where the locals and parameters are visible.

The following technical results follow easily from the definitions.

Lemma 4.1 (a) If $h \sim_\alpha g$ and $g \sim^C_\beta k$ then $h \sim^C_{\alpha\cdot\beta} k$.
(b) If $\delta\,h \subseteq \beta$, $h \sim_{\delta\,h} g$, and $g \sim^C_\beta k$ then $h \sim^C_{\delta\,h} k$.
(c) If $h \sim^C_\beta k$ and $\gamma \supseteq \beta$ then $h \sim^C_\gamma k$ provided that $\mathsf{dom}\,\gamma \subseteq \mathsf{dom}\,h$ and $\mathsf{rng}\,\gamma \subseteq \mathsf{dom}\,k$. Similarly for stores.

4.2 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 3.2 we adapt the definition of weak purity to one using the visible relations. (Whereas metavariable C is used to range over arbitrary classes, D is used throughout the paper for some designated class with respect to which purity is considered.)

Definition 4.2 Term M is *observationally pure outside* D provided that the following holds for all $C \neq D$. If $M, h, s \xrightarrow{C} k, t$ then $k \sim^C_{\delta\,h} h$ and $\mathsf{res} \ntriangleleft t \sim_{\delta\,h} \mathsf{res} \ntriangleleft s$. Procedure p is *observationally pure outside* D iff $h, s \dashv\!|p|\!\mapsto k, v$ implies

$k \sim^{C}_{\delta h} h$ for all $C \neq D$. $\square$

Weak purity implies observational purity (outside any D) because $k \sim_{\beta} h$ implies $k \sim^{C}_{\beta} h$ for any C.

Procedure *memoProd* of class $D0$ in Figure 1 is observationally pure outside $D0$. It updates fields of preexisting objects but those fields are not visible outside $D0$ 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 weak 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 some D then so is $p(M)$. Another similarity with weak purity is the following.

Fact 4.2 If Q is observationally pure outside D then **assert** $Q \approx^{C}$ **skip** for all $C \neq D$.

Proof: Suppose $(\textbf{assert}\ Q), h, s \xrightarrow{C} k, t$ and $(h, s) \sim^{C}_{\beta} (h', s')$. By semantics we have $\textbf{skip}, h', s' \xrightarrow{C} h', s'$. It suffices to show $(k, t) \sim^{C}_{\beta} (h', s')$. By semantics of **assert** we have $Q, h, s \xrightarrow{C} k, u$ for some u. By observational purity of Q we have $k \sim^{C}_{\delta h} h$. By Lemma 4.1(b) we obtain $k \sim^{C}_{\delta h} h'$, hence $(k, s) \sim^{C}_{\beta} (h', s')$ using Lemma 4.1(c). By observational purity of Q we have $\mathsf{res} \lhd u \sim_{\delta h} \mathsf{res} \lhd s$ so by semantics of **assert** we have $t = s$. $\square$

This result is not satisfactory, however, because unlike the situation for weak 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 3.4).

Example 4.1 Consider the term $pos(memoProd(y, i))$, evaluation of which may well update $y.arg$ and $y.farg$. By Fact 4.2, **assert** $pos(memoProd(y, i)) \approx^{C}$ **skip**. Moreover the procedures of $D0$ 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 $D0$ declared procedure $leak(self : D0) : \textbf{int}\{\ \textbf{return}\ self.arg\}$. 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. $\Box$

In fact the failure of congruence is more fundamental. We claim $memoProd \not\approx^C memoProd$. Now congruence for terms fails, since if p is a procedure such that $p \not\approx^C p$ then $M \approx^C N$ does not imply $p(M) \approx^C p(N)$. To justify the claim, note that even if $h \sim^C_\beta h'$ for all $C \neq D0$, it is possible for there to be $(o, o') \in \beta$ with $h\, o.\mathsf{type} = D0$ and moreover $h\, o\, arg = h'\, o'\, arg$ but $h\, o\, farg \neq h'\, o'\, farg$ because these fields are not visible outside $D0$. 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. The problem is solved in Section 5 and Definition 4.2 is retained.

An apparent shortcoming of Definition 4.2 is that checking the property appears to be a nontrivial and nonstandard problem. In fact, the check can be reduced to equivalence as follows. We say N *terminates when* M *does* provided that $M, h, s \Downarrow$ implies $N, h, s \Downarrow$, where $M, h, s \Downarrow$ means there exists k, t with $M, h, s \to k, t$.

Fact 4.3 Suppose $M \approx^C N$ for all $C \neq D$, and suppose N is weakly pure. If N terminates when M does then M is observationally pure outside D.

Proof: Suppose $C \neq D$ and $M, h, s \xrightarrow{C} k, t$, to show that M satisfies Definition 4.2. By termination hypothesis, there is (k', t') such that $N, h, s \xrightarrow{C} k', t'$. Note that $(h, s) \sim^C_{\delta h} (h, s)$, so by $M \approx^C N$ there is some $\beta \supseteq \delta h$ with $(k, t) \sim^C_\beta (k', t')$. By weak purity of N and Lemma 3.2 we have $k' \sim_{\delta h} h$ and $\mathsf{res} \lhd t' \sim_{\delta h} \mathsf{res} \lhd s$; hence $h \sim_{\delta h} k'$ by symmetry of $\sim_{\delta h}$. From $h \sim_{\delta h} k'$ and $k' \sim^C_\beta k$ we get $h \sim^C_{\delta h} k$ by chaining Lemma 4.1(b) and similarly for $\mathsf{res} \lhd t \sim_{\delta h} \mathsf{res} \lhd s$. $\Box$

As an example, procedure *memoProd* is equivalent to *pureProd* which is weakly pure and terminates when *memoProd* does.

The termination antecedent is necessary. As an extreme case, if N never terminates then it is weakly pure and $M \approx^C N$ for any M and any C 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 (Mitchell, 1996; de Roever and Engelhardt, 1998). Using a simulation to establish the antecedent of Fact 4.3 has the added benefit of congruence.

5 Observational purity via simulation

This Section gives the main result, equivalence of $\mathcal{C}[\textbf{assert } Q]$ and $\mathcal{C}[\textbf{skip}]$ for observationally pure Q and any context $\mathcal{C}[-]$. To this end, we generalize from specific equivalences on states to an arbitrary relation subject to some conditions. The relation is provided by the reasoner who wishes to treat a procedure as observationally pure outside some designated class D.

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 thing, a family, indexed by bijections, of binary relations $\asymp_\beta$ on heaps.

Example 5.1 In the context of Figure 1, define $\dot{\asymp}$ by $h \mathrel{\dot{\asymp}_\beta} h'$ iff

- $h \sim^C_\beta h'$ for every $C \neq D0$, and
- for all $(o, o') \in \beta$, if $h\, o.\mathsf{type} = D0$ then $h\, o.f = h'\, o'.f$ and both $h\, o$ and $h'\, o'$ satisfy the invariant mentioned in the caption of Figure 1 —that is, $h\, o.arg \neq 0$ implies $h\, o.farg = h\, o.f * h\, o.arg$ and *mutatis mutandis* for h', o'.

From two states related by $\dot{\asymp}_\beta$, *memoProd* gives the same results, indeed that is true for all the procedures of $D0$. □

If the cache involved other objects, an encapsulation condition would be imposed on them as well, e.g., via ownership (Clarke et al., 2001; Clarke and Drossopoulou, 2002; Barnett et al., 2004a). That prevents problems like Example 4.1. In our formulation, encapsulation at the level of classes is sufficient; it need not be instance-based. Thus our notion of coupling applies to those of Banerjee and Naumann (2005a,c) but is more general.

To express healthiness conditions on a given coupling $\asymp$ we extend it to states, terms, and procedures following the usual pattern.

Definition 5.1 Given 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, define $M \asymp M'$ iff $(h, s) \asymp_\beta (h', s')$ and $M, h, s \xrightarrow{C} k, t$ and $M', h', s' \xrightarrow{C} k', t'$ implies that there is $\gamma \supseteq \beta$ such that $(k, t) \asymp_\gamma (k', t')$ (for any C). Finally, $p \asymp p'$ iff $(h, s) \asymp_\beta (h', s')$ and $h, s \dashv\! p \!\mapsto k, v$ 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 5.2 (coupling, simulation) A *D-coupling* is a bijection indexed family, $\asymp$, of relations on heaps, such that

(1) if $h \asymp_\beta k$ then $\mathsf{dom}\,\beta \subseteq \mathsf{dom}\,h$ and $\mathsf{rng}\,\beta \subseteq \mathsf{dom}\,k$
(2) $h \asymp_\alpha g$ and $g \sim_\beta k$ implies $h \asymp_{\alpha\cdot\beta} k$

(3) $h \asymp_\beta k$ implies $h \sim_\beta^C k$ for all $C \neq D$

A *D-simulation* is a D-coupling such that the following hold.

(4) there is a term *Init* such that for any C, β, h, s, h', s', if $(h, s) \sim_\beta^C (h', s')$ then there is some k, t, k', t' with $\mathit{Init}, h, s \xrightarrow{C} k, s$ and $\mathit{Init}, h', s' \xrightarrow{C} k', s'$ and there is some $\gamma \supseteq \beta$ with $(k, s) \asymp_\gamma (k', s')$
(5) $p \asymp p$ for every procedure p in every class

Items (1) and (2) are simple healthiness conditions (compare Definition 3.2 and healthy predicates in Section 3). Item (3) says that the relation reduces to equality modulo renaming, for classes other than D.

Initialization is often needed to establish a coupling $\asymp$; it typically does not simply follow from $(h, s) \sim_\beta^C (h', s')$ because $\sim_\beta^C$ allows arbitrary difference in non-visible fields. Item (4) is a simple formalization of initialization that follows the pattern used in the literature for single-instance modules (de Roever and Engelhardt, 1998). For dynamic allocation, it is the object constructor (or default values) that establishes the relation (Banerjee and Naumann, 2005a; Cavalcanti and Naumann, 2002). To cater for this in our simple setup, one can take *Init* to be an assertion of a predicate like "all existing $D0$-objects have $\mathit{arg} = 0 = f$", or even "no $D0$-objects exist".[4] Note also that (4) says *Init* must be legal in the context of any class C, e.g., it could be the invocation of a procedure of class D.

Item (5) requires all procedures to preserve $\asymp$. It precludes *leak* in Example 4.1. All procedures in Figure 1 preserve the relation $\dot{\asymp}$ of Example 5.1. Item (5) may seem 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 an instance of a more general result, the *abstraction theorem* of the theory of representation independence (which encompasses relations between two different implementations of a class). The abstraction theorem says that only procedures of class D, which have privileged access to encapsulated state, must be shown to preserve the coupling. (See Section 6.)

For practical application, one also wants the simulation to be defined "locally", i.e., in terms of a single instance of the class D in question. To this end, additional conditions may be imposed on the coupling as well as on programs. For our purposes, a sufficient condition on couplings is expressed by item (5). About programs we make the following.

Assumption 5.1 Suppose $\asymp$ is a D-simulation. If $M \asymp N$ then $\mathcal{C}[M] \asymp \mathcal{C}[N]$ for any context $\mathcal{C}[-]$.

[4] The latter is not a healthy predicate as defined in section 3, but there is no problem because the healthiness condition is not needed for preconditions.

This congruence property does not hold in interesting languages without restricting "any context" to mean those that respect encapsulation in some sense. A sufficient restriction can be obtained by combining strong typing, class based visibility, and some form of ownership for alias confinement; associated conditions must be imposed on $\asymp$. Section 6 cites results of this form, proved for richer languages than the illustrative one in this paper.

5.1 Using D-simulations for purity

The straightforward generalization of Definition 4.2 is as follows.

Definition 5.3 Let $\asymp$ be a D-coupling. Then M is *observationally pure for* $\asymp$ iff for all $C \neq D$, if $M, h, s \xrightarrow{C} k, t$ then $k \asymp_{\delta\, h} h$ and $\mathsf{res} \lhd t \sim_{\delta\, h} \mathsf{res} \lhd s$. Procedure p is *observationally pure for* $\asymp$ iff $h, s \dashv p \mapsto k, v$ implies $k \asymp_{\delta\, h} h$. $\square$

Instantiating $\asymp$ with $\sim$ gives the condition in Definition 4.2. Moreover, for any D it is easy to show that $\sim$ satisfies conditions (1–3) of Definition 5.2 of D-coupling. Indeed, simulation condition (4) holds, taking *Init* to be **skip**, and condition (5) follows from congruence, Proposition 3.4. Thus for a term or procedure to be observationally pure for $\sim$ is the same as its being observationally pure outside D according to Definition 4.2 (for any D). As a special case, if M is weakly pure then it is observationally pure for $\sim$ (again, outside any D). A sort of converse is given by the following.

Fact 5.2 Suppose M is observationally pure for some D-coupling $\asymp$. Then it is observationally pure outside D.

Proof: Suppose $C \neq D$ and $M, h, s \xrightarrow{C} k, t$. By observational purity for $\asymp$ we have $k \asymp_{\delta\, h} h$ and $\mathsf{res} \lhd t \sim_{\delta\, h} \mathsf{res} \lhd s$. The latter condition is the same as the condition for stores in Definition 4.2 of observational purity. For the heap, we get $k \sim^{C}_{\delta\, h} h$ using Definition 5.2(3). $\square$

This Fact, together with Fact 4.2, implies **assert** $M \approx^{C}$ **skip** for $C \neq D$, if M is observationally pure for some $\asymp$. But Fact 4.3 suggests that for interchangeability of an **assert** with **skip**, it should be enough to formulate observational purity as in Definition 4.2. The role of a coupling is then to prove an antecedent equivalence and in addition to enjoy a congruence property. This is worked out in our main result to follow.

Analogous to Fact 4.3, one might expect the following: If $M \asymp N$ for some weakly pure N, 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 coupled states and the coupling relation need not be reflexive, so the proof of Fact 4.3 does not directly generalize. However, we can prove the following Fact. It uses a termination condition that would be imposed everywhere for simulations in a total-correctness setting.

Definition 5.4 *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 5.3 If $M \asymp N$ and N is weakly pure then **assert** $M \asymp$ **skip** provided that $\asymp$ is a D-coupling and N terminates when M does, modulo $\asymp$.

Proof: Suppose (**assert** $M), h, s \xrightarrow{C} k, t$ and $(h, s) \asymp_\beta (h', s')$. By semantics, **skip**$, h', s' \xrightarrow{C} h', s'$, so we need to show $(k, t) \asymp_\gamma (h', s')$ for some $\gamma \supseteq \beta$. We leave the store part to the reader and show $k \asymp_\gamma h'$. By $M \asymp N$ and termination hypothesis for N, we have $M, h, s \xrightarrow{C} k, r$ and $N, h', s' \xrightarrow{C} g', r'$ and $(k, r) \asymp_\gamma (g', r')$ for some r, g', r' and some $\gamma \supseteq \beta$. By weak purity of N and Lemma 3.2, we have $g' \sim_{\delta\, h'} h'$. Thus $k \asymp_\gamma g'$ and $g' \sim_{\delta\, h'} h'$, hence $k \asymp_\gamma h'$ using Definition 5.2(2). □

5.2 Main result

In this section we show that if Q is observationally pure for some D-coupling $\asymp$ then $\mathcal{C}$[**assert** Q] and $\mathcal{C}$[**skip**] are equivalent in any context outside class D. For reasoning within D, condition (5) of Definition 5.2 justifies the use of Q in preconditions: procedures of D do not distinguish between the state before and after Q. Condition (5) together with Assumption 5.1 justifies the use of Q in postconditions in D. Free use of Q in assertions within procedures of D cannot be justified.

To prove the main result, two more ingredients are needed. The first is the notion of 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 3.2 but taking into account visibility restrictions on specifications; this we leave to the reader.

Definition 5.5 (initialized equivalence, $\dot{\approx}^C$) Suppose *Init* is given as in Definition 5.2. Define $M \dot{\approx}^C M'$ iff *Init*$; M \approx^C$ *Init*$; M'$. □

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

Lemma 5.4 If $M \asymp N$ and $\asymp$ is a D-simulation then $M \mathrel{\dot{\approx}}^C N$ for any $C \neq D$.

Proof: To show $\mathit{Init}; M \approx^C \mathit{Init}; N$, suppose $(h, s) \sim_\beta^C (h', s')$. By Definition 5.2(4) there is g, g', α such that $\mathit{Init}, h, s \xrightarrow{C} g, s$ and $\mathit{Init}, h', s' \xrightarrow{C} g', s'$ and $(g, s) \asymp_\alpha (g', s')$ and $\alpha \supseteq \beta$. If $M, g, s \xrightarrow{C} k, t$ and $M', g', s' \xrightarrow{C} k', t'$ then by $M \asymp N$ we have $(k, t) \asymp_\gamma (k', t')$ for some $\gamma \supseteq \alpha$. Then $(k, t) \sim_\gamma^C (k', t')$ by Definition 5.2(3). $\square$

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 5.5 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 $M \mathrel{\dot{\approx}}^C Q$.

Proof: To show $M \mathrel{\dot{\approx}}^C Q$, suppose $(h, s) \sim_\beta^C (h', s')$ and $(\mathit{Init}; M), h, s \xrightarrow{C} k, t$ and $(\mathit{Init}; Q), h', s' \xrightarrow{C} k', t'$. We must show $(k, t) \sim_\gamma^C (k', t')$ for some $\gamma \supseteq \beta$, i.e., $k \sim_\gamma^C k'$ and $t \sim_\gamma t'$. Here is a diagram to illustrate the following argument.

$$\begin{tikzcd}
(h, s) \arrow[r, phantom, "\sim_\beta^C"] \arrow[d, "\mathit{Init}"'] & (h', s') \arrow[d, "\mathit{Init}"] & \\
(g, s) \arrow[r, phantom, "\asymp_\alpha"] \arrow[d, "M"'] & (g', s') \arrow[d, "N"'] \arrow[dr, "Q"] & \\
(k, t) \arrow[r, phantom, "\sim_\gamma^C"] & (j, r) \arrow[r, phantom, "\sim_\phi"] & (k', t')
\end{tikzcd}$$

From Definition 5.2(5) for Init we have $\mathit{Init}, h, s \xrightarrow{C} g, s$ and $\mathit{Init}, h', s' \xrightarrow{C} g', s'$ with $(g, s) \asymp_\alpha (g', s')$ for some $\alpha \supseteq \beta$ and some g, s, g', s' (using that Init is deterministic). From semantics of ";" we have $M, g, s \xrightarrow{C} k, t$ and $Q, g', s' \xrightarrow{C} k', t'$. By termination hypothesis for N and $(g, s) \asymp_\alpha (g', s')$ we have $N, g', s' \xrightarrow{C} j, r$ for some j, r. Then by $M \mathrel{\dot{\approx}}^C N$ we get $(k, t) \sim_\gamma^C (j, r)$ for some $\gamma \supseteq \beta$. By $N \approx Q$ using $(g', s') \sim_{\delta g'} (g', s')$ we have $(j, r) \sim_\phi (k', t')$ for some $\phi \supseteq \delta g'$. So using Lemma 4.1 for $(k, t) \sim_\gamma^C (j, r)$ and $(j, r) \sim_\phi (k', t')$ we get $(k, t) \sim_{\gamma \cdot \phi}^C (k', t')$. Finally, $\gamma \cdot \phi \supseteq \beta$ follows from $\gamma \supseteq \beta$ and $\phi \supseteq \delta g'$ using $\mathsf{rng}\, \beta \subseteq \mathsf{dom}\, h' \subseteq \mathsf{dom}\, g'$. $\square$

Finally, here is the main result of the paper.

Theorem 5.6 Suppose $\asymp$ is a D-simulation and N terminates when Q does, modulo $\asymp$. If $Q \asymp N$ and N is weakly pure then $\mathcal{C}[\textbf{assert } Q] \mathrel{\dot{\approx}}^C \mathcal{C}[\textbf{skip}]$ for all contexts $\mathcal{C}$ and classes $C \neq D$.

Proof: From $Q \asymp N$ we get $\mathcal{C}[\textbf{assert } Q] \asymp \mathcal{C}[\textbf{assert } N]$ by congruence Assumption 5.1. Thus $\mathcal{C}[\textbf{assert } Q] \dot{\approx}^C \mathcal{C}[\textbf{assert } N]$ by Lemma 5.4. By weak purity of N and Corollary 3.5 we have $\mathcal{C}[\textbf{assert } N] \approx \mathcal{C}[\textbf{skip}]$. Because all constructs of the language are monotonic with respect to termination, we have that $\mathcal{C}[\textbf{assert } N]$ terminates when $\mathcal{C}[\textbf{assert } Q]$ does, modulo $\asymp$. Thus Lemma 5.5 applies to yield the result. □

5.3 An alternative

Theorem 5.6 avoids the need to explicitly use any notion of observational purity. It says that to use Q in a specification, the reasoner must find some weakly pure N and some simulation $\asymp$ such that $Q \asymp N$. The alternative is to make direct use of observational purity for $\asymp$, following the pattern of Theorem 3.3 and Corollary 3.5. This avoids the need to exhibit a weakly pure N and prove $Q \asymp N$, though of course the simulation property of $\asymp$ must still be proved. The alternative requires a kind of transitivity condition on couplings. This condition is satisfied in all the observational purity examples encountered by the author, but it is not included in Definition 5.2 because none of the other results depend on it. Also, 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 5.7 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 $\textbf{assert } Q \asymp \textbf{skip}$.

Proof: Suppose $(h, s) \asymp_\beta (h', s')$, $(\textbf{assert } Q), h, s \longrightarrow k, t$, and $\textbf{skip}, h', s' \longrightarrow k', t'$. We shall show $(k, t) \asymp_\beta (k', t')$. The argument for stores t, t' is similar to that in the proof of Theorem 3.3, so we consider just the heap. By semantics of **skip** we have $h' = k'$ so it remains to show $k \asymp_\beta h'$. By semantics of **assert** we have $Q, h, s \longrightarrow k, u$ for some u. By observational purity of Q for $\asymp$ we have $k \asymp_{\delta\, h} h$. So we have $k \asymp_{\delta\, h} h \asymp_\beta h'$ and thus $k \asymp_{\delta\, h\cdot\beta} h'$ by the hypothesis about transitive composition. Because $\mathsf{dom}\,\beta \subseteq \mathsf{dom}\, h$ we have $\delta\, h \cdot \beta = \beta$ so we are done. □

Corollary 5.8 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 $\mathcal{C}[\textbf{assert } Q] \dot{\approx}^C \mathcal{C}[\textbf{skip}]$.

Proof: By Theorem 5.7 we have $\textbf{assert } Q \asymp \textbf{skip}$, hence $\mathcal{C}[\textbf{assert } Q] \asymp \mathcal{C}[\textbf{skip}]$ by congruence Assumption 5.1. Thus $\mathcal{C}[\textbf{assert } Q] \dot{\approx}^C \mathcal{C}[\textbf{skip}]$ by Lemma 5.4. □

6 Conclusion

To avoid logical anomalies and misleading results from runtime assertion checking, practical verification systems impose various purity requirements for specifications and annotations. ESC/Java allowed no procedure invocations (Flanagan et al., 2002), JML prescribes weak purity checking (Leavens et al., 2003), and Eiffel merely advises programmers to avoid effects (Meyer, 1997). But for verification to scale to large systems it is important to consider as pure some 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 **assert** Q with **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 (Theorem 5.6) 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, some weakly pure term N. 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 weak purity rather than a more specialized analysis.

To apply our results one needs a technique for defining D-simulations. In particular, it is essential that condition (5) in Definition 5.2 only needs to be checked for procedures of D; for procedures of $C \neq D$ it should follow by a preservation/congruence theorem that is proved once and for all for the programming language. Suitable results —analogs of our Assumption 5.1 and Lemma 5.4— are a special case of the theory of representation independence and have been developed for many sorts of languages (Mitchell, 1996; de Roever and Engelhardt, 1998). For Java-like languages, Banerjee and Naumann (2005a) give such a theory under the assumption of suitable alias control which can be achieved using static analysis (Banerjee and Naumann, 2005a; Müller, 2002; Clarke and Drossopoulou, 2002). In these works, alias control is based on the idea that an object "owns" some objects that comprise its encapsulated representations; the static analysis uses annotated types to check that the representation objects are not exposed to clients.

An alternative to type-checking for ownership is to reason about it using assertions (Barnett et al., 2004a; Naumann and Barnett, 2006). This state based approach to encapsulation affords some flexibility, e.g., ownership may be transferred. A representation independence result has been given using state based encapsulation (Banerjee and Naumann, 2005c).

Such results are difficult to prove for complex languages so it is fortunate that we could treat observational purity using a notion of simulation compatible with extant results on encapsulation.

In justifying the choice of program equivalence we uncovered an issue for weak 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 weak purity is not sound. Quantifications over all allocated objects have been used in some settings, e.g., the program invariants of the Boogie discipline (Barnett et al., 2004a; Naumann and Barnett, 2006), but in that context programmer-defined predicates are in fact restricted to reachability in terms of auxiliary fields. Pierik et al. (2005) advocate global invariants such as "there is at most one C-object" which are apparently incompatible with weak purity. Program equivalence modulo garbage collection has been studied by Calcagno et al. (2003) and others (Banerjee and Naumann, 2005a).

Using the results. To illustrate the practical application our results we turn once more to the example in Figure 1. Suppose we want to use the term $\mathit{memoProd}(x, y)$ in an assertion in some context outside class $D0$, where x and y are variables. Example 5.1 defines a suitable coupling $\dot{\asymp}$. To apply Theorem 5.6, this coupling must be a $D0$-simulation and some weakly pure term N must be found such that $\mathit{memoProd}(x, y) \dot{\asymp} N$. The coupling $\dot{\asymp}$ in Example 5.1 is induced by a "local coupling" relation on a single pair o, o' of $D0$-objects which can be expressed by a formula:

$$\begin{aligned} o.f = o'.f \;\wedge\; & (o.\mathit{arg} \neq 0 \Rightarrow o.\mathit{farg} = o.f * o.\mathit{arg}) \\ \wedge\; & (o'.\mathit{arg} \neq 0 \Rightarrow o'.\mathit{farg} = o'.f * o'.\mathit{arg}) \end{aligned} \tag{1}$$

The relation only depends on the fields of o and o', not on any other objects. For couplings of this sort, it is a corollary of Banerjee and Naumann (2005a) that Assumption 5.1 holds, in a Java-like language, for all contexts that respect private visibility. That is, the fact that these fields have private visibility is enough to encapsulate them within the class. To show that this coupling is a $D0$-simulation is a matter of checking that it is preserved by each procedure of class $D0$ (i.e., $\mathit{get} \dot{\asymp} \mathit{get}$, $\mathit{set} \dot{\asymp} \mathit{set}$, $\mathit{pureProd} \dot{\asymp} \mathit{pureProd}$, and $\mathit{memoProd} \dot{\asymp} \mathit{memoProd}$) which in this case is not difficult to show. It remains to find a weakly pure term N and show $\mathit{memoProd}(x, y) \dot{\asymp} N$. A suitable choice for N is $\mathit{pureProd}(x, y)$. A simple syntactic condition is enough to show this is weakly pure: the only potential impurity is procedure $\mathit{pureProd}$ and the only field update in its body is of a newly allocated Cell. Finally, $\mathit{memoProd}(x, y) \dot{\asymp} \mathit{pureProd}(x, y)$ follows from $\mathit{memoProd} \dot{\asymp} \mathit{pureProd}$ which can be proved directly in terms of the definition of $\asymp$ and the semantics. Practical ways to prove particular simulations like $\mathit{memoProd} \dot{\asymp} \mathit{pureProd}$ and $\mathit{set} \dot{\asymp} \mathit{set}$, without direct recourse to the semantics, are discussed later under related work.

Procedure *leak* in Example 4.1 does not preserve $\dot{\asymp}$, because the final stores are not related by $\sim$ as stipulated in Definition 5.1. Thus if this procedure is added to class $D0$ then $\dot{\asymp}$ fails to be a $D0$-simulation and therefore cannot be used to justify use of $memoProd(x, y)$ in an assertion.

Consider a variation of the example, where *arg* and *farg* are not integers but rather references to *Cell* objects used to hold the corresponding integers. The coupling relation in Example 5.1 would be adapted *mutatis mutandis.* These *Cell* objects would be designated as owned by their referencing $D0$-instance, in order for the corresponding version of $\dot{\asymp}$ to be considered admissible by the ownership rules used in the representation independence theories of Banerjee and Naumann (2005a,c). The *val* field in *Cell* is public and could be updated by a client program that obtained a reference to one of these *Cell* objects. Such a reference could be obtained by the analogous version of procedure *leak* from Example 4.1; for just this reason, the *leak* procedure would be rejected by the static analysis in Banerjee and Naumann (2005a) and by ownership type systems (Müller, 2002; Clarke and Drossopoulou, 2002). The representation independence theories say that Assumption 5.1 holds for all contexts that conform to the restrictions of the ownership system. Moreover, the theories require the reasoner to supply only the definition of a local coupling, like Equation 1; the relation $\asymp$ on global states is given by a general construction.

For practical purposes it seems advisable to disallow explicit assignments in assertions and specifications. Then the only possible effects are by way of procedure invocations and these could be restricted to procedures explicitly marked as observationally pure. The theory would then be used to justify the purity of those procedures, as in the example of the preceding paragraphs which reduced the proof obligation to $memoProd \mathrel{\dot{\asymp}} pureProd$.

In summary, to apply our theory to justify that a procedure p is observationally pure outside some D the following steps are taken: define a local coupling that depends only on private fields and owned objects; show that it is preserved by all procedures of D (whence by general theory it is a D-simulation); give a weakly pure procedure and show that it is simulated by p, or give a weakly pure term and show that it is simulated by the body of p.

An alternative way to justify use of some term Q in a specification, which avoids the need to define weakly pure N that simulates Q, is to give a simulation $\asymp$ and show directly that Q is observationally pure for $\asymp$. For Q to be observationally pure for $\asymp$ requires that its initial and final states are always related by $\asymp$. For the latter property there is a general proof technique which applies whenever $h \asymp_\beta h'$ can be expressed as a conjunction of the form

$$I(h) \text{ and } I(h') \text{ and } h \sim^C_\beta h' \text{ for all } C \text{ with } C \neq D$$

for some predicate I. This is the case with Example 5.1 and also the variation using *Cell* objects. In this situation, Corollary 5.8 applies since such a relation has the requisite transitivity property. To prove that a coupling of this form is a simulation one can separately prove that I is an invariant preserved by the procedures of class D and that $\sim^C$ is preserved by the procedures of class D. Moreover observational purity of Q for $\asymp$ (Definition 5.3) is decomposed to two properties: I is preserved by Q and $k \sim^C_{\delta\, h} h$ where k is the heap that Q yields from initial heap h. In practice, the global invariant I is the ownership invariant together with the conjunction, over all instances, of a local object invariant like the one in the caption of Figure 1 (also used in Equation (1)). The condition that $\sim^C$ is preserved by procedures is also known as "noninterference" and can be checked by static analyses developed for secure information flow (Sabelfeld and Myers, 2003; Banerjee and Naumann, 2005b). Such analyses also check the condition $k \sim^C_{\delta\, h} h$ where k is the final state of the candidate pure term Q —the property is sometimes called *write confinement.* This alternative is the focus of Barnett et al. (2006). It is under discussion for the JML tools,[5] and is being implemented in the Spec#/Boogie project (Barnett et al., 2004a).

Our formulation of observational purity is based on the class as a unit of encapsulation, for specificity, but the idea can be adapted to other units of encapsulation. In particular, the information flow analysis technique allows to designate some unit smaller than a class, by labeling the relevant fields as "secret".

Related and future work. The most closely related work is that of Barnett et al. (2004b, 2006), where a seemingly ad hoc condition combining Definitions 4.2 and 5.3 was proposed. The original workshop presentation provoked heated discussion, perhaps in part because the connection with established ideas about benevolent effects was not at all clear so the theory seemed unmotivated. Rather than drawing on the general theory of encapsulation and simulation, the work focused on the noninterference property from information security. In retrospect, our theory appears to be a natural adaptation of old ideas about benevolent side effects together with a more modern treatment of abstraction (He et al., 1986) and in particular the work of Banerjee and Naumann (2005a) for heap encapsulation. Nonetheless it was tricky to get the technical details to work: e.g., Lemma 5.5 does not seem obvious and care is needed in orienting some definitions so that $\asymp_\beta$ does not have to be symmetric.

Leavens et al. (2003) discuss the rationale and static analysis for weak purity in JML. Sălcianu and Rinard (2004) give a more precise static analysis for

[5] David Cok and Gary Leavens, personal communication July 2005.

the weak purity condition. Verification conditions for weakly pure methods in assertions have been investigated by Cok (2005) and by Darvas and Müller (2006).

For proving the simulation property in specific cases like $set \mathrel{\dot{\asymp}} set$ and $memoProd \mathrel{\dot{\asymp}} pureProd$, one alternative is to use a specialized logic for relational properties, such as those of Benton (2004) and Yang (2004). The more established alternative is the method of Reynolds (de Roever and Engelhardt, 1998) in which a relational property like $M \asymp N$ is reduced to an ordinary correctness property. A renamed copy of the program variables is used so that the state relation $\asymp$ can be treated as a predicate R on a "doubled state". The renamed copy N' of N acts on the renamed part of the state, so that $M \asymp N$ is equivalent to the partial correctness property $\{R\}\ M; N'\ \{R\}$ which can be proved using standard techniques. This method of proving relational properties has recently been rediscovered in the special case where $M = N$ (Barthe et al., 2004; Terauchi and Aiken, 2005). The author has extended the technique to apply to programs acting on heap objects, for which renaming is inadequate to encode two states or two computations as one (Naumann, 2006). Ghost fields are used to encode location bijections and by this technique simple cases like $memoProd \mathrel{\dot{\asymp}} pureProd$ can be proved automatically by verification tools like ESC/Java2.

There has been considerable work refining and extending ownership encapsulation to encompass a wide variety of design patterns in object oriented programs (Müller, 2002; Clarke and Drossopoulou, 2002; Boyapati et al., 2003; Barnett et al., 2004a; Naumann and Barnett, 2006). Much of this work focuses on object invariants but, for the most part, what works for invariants works for simulations as demonstrated by Banerjee and Naumann (2005a,c).

To extend observational purity to total correctness, equivalence is replaced by refinement of **assert** Q by **skip**. It should be possible to obtain a suitable simulation theory for Java-like languages by adapting existing work (Cavalcanti and Naumann, 2002; Banerjee and Naumann, 2005a,c). We conjecture that the extension to concurrency is also straightforward, given suitable control of atomicity. Procedures called in assertions need to be deterministic in order to apply logical reasoning, but our theory relies on determinacy only of *Init* in the proof of Lemma 5.5 and perhaps even that is not necessary.

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 weakly 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.

Acknowledgements Wolfram Schulte posed and named the problem of observational purity and suggested the connection with secure information flow. Wolfram and Mike Barnett helped in preliminary attempts at devising a theory and, together with Sun Qi, explored examples in the C# and Java libraries. Gary Leavens, David Cok, Rustan Leino, and anonymous reviewers provided helpful feedback and information.

References

Banerjee, A., Naumann, D. A., 2002. Representation independence, confinement and access control. In: ACM Symposium on Principles of Programming Languages (POPL). pp. 166–177.

Banerjee, A., Naumann, D. A., 2005a. Ownership confinement ensures representation independence for object-oriented programs. Journal of the ACM 52 (6), 894–960, extended version of Banerjee and Naumann (2002).

Banerjee, A., Naumann, D. A., 2005b. Stack-based access control for secure information flow. Journal of Functional Programming 15 (2), 131–177, special issue on Language Based Security.

Banerjee, A., Naumann, D. A., 2005c. State based ownership, reentrance, and encapsulation. In: European Conference on Object-Oriented Programming (ECOOP). pp. 387–411.

Barnett, M., DeLine, R., Fähndrich, M., Leino, K. R. M., Schulte, W., 2004a. Verification of object-oriented programs with invariants. Journal of Object Technology 3 (6), 27–56, special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.

Barnett, M., Leino, K. R. M., Schulte, W., 2005. The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (Eds.), Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop (CASSIS 2004), Revised Selected Papers. Vol. 3362 of Springer LNCS. pp. 49–69.

Barnett, M., Naumann, D. A., Schulte, W., Sun, Q., 2004b. 99.44% pure: Useful abstractions in specifications. In: ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP). Technical Report NIII-R0426, University of Nijmegen.

Barnett, M., Naumann, D. A., Schulte, W., Sun, Q., 2006. Allowing state changes in specifications. In: International Conference on Emerging Trends in Information and Communication Security (ETRICS). To appear. Extended version of Barnett et al. (2004b).

Barthe, G., D'Argenio, P. R., Rezk, T., 2004. Secure information flow by self-composition. In: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04). pp. 100–114

Benton, N., 2004. Simple relational correctness proofs for static analyses and program transformations. In: ACM Symposium on Principles of Programming Languages (POPL). pp. 14–25.

Boyapati, C., Liskov, B., Shrira, L., 2003. Ownership types for object encapsulation. In: ACM Symposium on Principles of Programming Languages (POPL). pp. 213–223.

Calcagno, C., O'Hearn, P., Bornat, R., 2003. Program logic and equivalence in the presence of garbage collection. Theoretical Computer Science 298 (3), 557–581.

Cavalcanti, A. L. C., Naumann, D. A., 2002. Forward simulation for data refinement of classes. In: Eriksson, L., Lindsay, P. A. (Eds.), Formal Methods Europe. Vol. 2391 of Springer LNCS. pp. 471–490.

Clarke, D., Drossopoulou, S., Nov. 2002. Ownership, encapsulation and the disjointness of type and effect. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 292–310.

Clarke, D. G., Noble, J., Potter, J. M., 2001. Simple ownership types for object containment. In: Knudsen, J. L. (Ed.), ECOOP 2001 - Object Oriented Programming. pp. 53–76.

Cok, D. R., 2005. Reasoning with specifications containing method calls and model fields. Journal of Object Technology 4 (8), 77-103, special issue for ECOOP 2004 Workshop FTfJP.

Darvas, A., Müller, P., 2006. Reasoning about method calls in interface specifications. To appear in Journal of Object Technology 2006, special issue for ECOOP 2005 Workshop FTfJP.

de Roever, W.-P., Engelhardt, K., 1998. Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge University Press.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., Stata, R., 2002. Extended static checking for Java. In: ACM Conference on Programming Language Design and Implementation (PLDI). pp. 234–245.

Guttag, J. V., Horning, J. J. (Eds.), 1993. Larch: Languages and Tools for Formal Specification. Texts and Monographs in Computer Science. Springer-Verlag, with Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

He, J., Hoare, C. A. R., Sanders, J., 1986. Data refinement refined (resumé). In: European Symposium on Programming. Vol. 213 of Springer LNCS.

Hoare, C. A. R., 1969. An axiomatic basis for computer programming. Communications of the ACM 12, 576–80, 583.

Hoare, C. A. R., 1972. Proofs of correctness of data representations. Acta Inf. 1, 271–281.

Hogg, J., Lea, D., Wills, A., deChampeaux, D., Holt, R., 1992. The Geneva Convention on the treatment of object aliasing. OOPS Messenger 3 (2), 11–16.

Leavens, G. T., Cheon, Y., Clifton, C., Ruby, C., Cok, D. R., 2003. How the design of JML accommodates both runtime assertion checking and formal verification. In: de Boer, F. S., Bonsangue, M. M., Graf, S., de Roever, W.-P. (Eds.), Formal Methods for Components and Objects (FMCO 2002). Vol. 2852 of Springer LNCS. pp. 262–284.

Leino, K. R. M., Nelson, G., 2002. Data abstraction and information hiding. ACM Trans. Prog. Lang. Syst. 24 (5), 491–553.

Liskov, B., Guttag, J., 1986. Abstraction and Specification in Program Development. MIT Press.

Meyer, B., 1997. Object-oriented Software Construction, 2nd Edition. Prentice Hall, New York.

Mitchell, J. C., 1996. Foundations for Programming Languages. MIT Press.

Müller, P., 2002. Modular Specification and Verification of Object-Oriented Programs. Vol. 2262 of Springer LNCS.

Naumann, D. A., 2005. Observational purity and encapsulation. In: M. Cerioli (Ed.), Fundamental Aspects of Software Engineering (FASE). Vol. 3442 of Springer LNCS. pp. 190–204.

Naumann, D. A., 2006. From coupling relations to mated invariants for secure information flow and data abstraction. In: European Symposium on Research in Computer Security (ESORICS), to appear.

Naumann, D. A., Barnett, M., 2004. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In: IEEE Symposium on Logic in Computer Science (LICS). pp. 313–323.

Naumann, D. A., Barnett, M., 2006. Towards imperative modules: Reasoning about invariants and sharing of mutable state. Theoretical Computer Science. Extended version of Naumann and Barnett (2004), to appear.

Pierik, C., Clarke, D., de Boer, F. S., 2005. Controlling object allocation using creation guards. In: Proceedings, Formal Methods. Vol. 3582 of Springer LNCS. pp. 59–74.

Sabelfeld, A., Myers, A. C., Jan. 2003. Language-based information-flow security. IEEE J. Selected Areas in Communications 21 (1), 5–19.

Sălcianu, A., Rinard, M., May 2004. A combined pointer and purity analysis for Java programs. Tech. Rep. MIT-CSAIL-TR-949, Department of Computer Science, Massachusetts Institute of Technology.

Terauchi, T., Aiken, A., 2005. Secure information flow as a safety problem. In: 12th International Static Analysis Symposium (SAS). Vol. 3672 of Springer LNCS. pp 352–367.

Yang, H., 2004. Relational separation logic. Theoretical Computer Science. To appear.

A Proof of Proposition 3.4

For a precise proof some additional notions are needed. To formalize that a procedure meaning, say $\dashv p \mapsto$, preserves $\sim$ we define $p \approx p'$ in a way analogous to Definition 3.3. Specifically, $p \approx p'$ just if $(h, s) \sim_\beta (h', s')$ and $h, s \dashv p \mapsto k, v$ and $h', s' \dashv p' \mapsto k', v'$ implies there is $\gamma \supseteq \beta$ such that $k \sim_\gamma k'$ and $v \sim_\gamma v'$. Clearly $p \approx p'$ follows from the relation $(\mathsf{body}\, p) \approx (\mathsf{body}\, p')$ on terms.

For any term M, define *callees* M to be the set of procedures (i.e., procedure names) that occur in M. Similary for the callees of a term $\mathcal{C}[-]$ that may have a hole. Define *callees* p to be *callees*($\mathsf{body}\, p$). By the assumption that the calling graph is acyclic, there is an enumeration $p_0, \ldots, p_n$ of all procedures in the program, such that *callees* $p_i \subseteq \{p_0, \ldots, p_{i-1}\}$ for all i in $0 \ldots n$.

Proposition 3.4 is consequence of the following which strengthens it to provide an induction hypothesis.

Lemma A.1 For all i in $0 \ldots n$:

(1) $p_i \approx p_i$ and
(2) if $M \approx N$ then $\mathcal{C}[M] \approx \mathcal{C}[N]$ for all contexts $\mathcal{C}[-]$ such that $\mathit{callees}(\mathcal{C}[-]) \cup \mathit{callees}(\mathcal{C}[-]) \subseteq \{p_0, \ldots, p_i\}$

To prove Lemma A.1 we need a "preservation lemma" for each term construct other than procedure invocation. For constructs such as **skip** and **new** C that do not have subterms, the preservation lemma simply equates the term with itself: **skip** $\approx$ **skip** and **new** $C \approx$ **new** C. For a construct with a subterm, such as $x := -$, the preservation lemma is an implication like $M \approx N \Rightarrow x := M \approx x := N$. In each case the preservation lemma can be proved easily by unfolding the definition of $\approx$ and the semantic definition. We consider one case as an example. Suppose $M \approx N$. To show $x := M \approx x := N$, one considers related initial states; the semantics of $x := -$ first evaluates M and N, which by hypothesis yields related states, including related values for the res variable. The final states are obtained by assigning these values to x.

The proof of Lemma A.1 is by induction on i.

For the base case $i = 0$, the argument for item (1) of the Lemma, i.e., $p_0 \approx p_0$, is by induction on the structure of $\mathsf{body}\, p_0$, using the preservation lemmas. For item (2), the argument is by induction on the structure of $\mathcal{C}[-]$. For the case of procedure invocation, (1) is used together with the semantics of invocation (only p_0 can be called); all other cases are handled using the preservation lemmas.

For the induction step, the strong induction hypothesis is needed, in particular $p_j \approx p_j$ for all j in $0 \ldots i-1$. For $(1)_i$, observe that $p_i \approx p_i$ follows from $\mathsf{body}\, p_i \approx \mathsf{body}\, p_i$ which holds by induction hypothesis $(2)_{i-1}$. This appeal to the induction hypothesis uses that *callees* $p_i \subseteq \{p_0, \ldots, p_{i-1}\}$ and hence we can consider $\mathsf{body}\, p_i$ to be a context (which does not happen to have a hole) with callees in $\{p_0, \ldots, p_{i-1}\}$. The argument for $(2)_i$ in the induction step is the same as in the base case. In particular, for procedure invocation we use that $(2)_i$ has $\mathit{callees}(\mathcal{C}[-]) \cup \mathit{callees}(\mathcal{C}[-]) \subseteq \{p_0, \ldots, p_i\}$ and $(1)_i$ gives $p_j \asymp p_j$

for all $j \in 0 \ldots i$. This corresponds to the informal appeal to the "induction hypothesis about $-\!|p|\!\!\rightarrow$" in the proof sketch following Proposition 3.4 in the main body of the paper.