Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract)

David A. Naumann[*]
Stevens Institute of Technology
naumann@cs.stevens-tech.edu

Mike Barnett
Microsoft Research
mbarnett@microsoft.com

February 2, 2004

Abstract

Imperative and object-oriented programs make ubiquitous use of shared mutable objects. Updating a shared object can and often does transgress a boundary that was supposed to be established using static constructs such as a class with private fields. This paper shows how auxiliary fields can be used to express two state-dependent encapsulation disciplines: ownership, a kind of separation, and local co-dependence, a kind of sharing. A methodology is given for specification and modular verification of encapsulated object invariants and shown sound for a class-based language.

1. Introduction

The use of pointer-based data structures makes formal reasoning about programs quite difficult. In common practice, pointer structures are widely used both for internal representations and for interfaces between components. Functional languages offer strong encapsulation to protect internal invariants from outside interference, but in higher order imperative programs various kinds of aliasing subvert encapsulation. The thorniest problems, due to interaction between local variables and nested procedures [14, 20], are precluded [18, 3] in widely used imperative and object-oriented languages owing to restrictions on non-local references and/or nesting. But these restrictions on procedure abstraction force the use of heap structure to encode higher level patterns. Performance considerations also necessitate the use of heap sharing and programmers are often taught to exploit "object identity" in specification and design. Compositional reasoning depends on control of aliasing but straightforward ways to control aliasing in the heap have been found too restrictive for general use (see surveys in [10, 16]). The contribution of this paper is to formalize and prove sound a discipline that supports modular reasoning about object invariants, caters for common patterns of sharing, and is compatible with standard first and higher order logics.

There has been a resurgence of work on encapsulated invariants in stateful programs. State-dependent types are needed to enforce simple data-type invariants in low-level code where local variables (registers) are re-used and cannot be given a single fixed type [15, 1]. Ownership type systems [11, 16] and Separation Logic [19] focus on partitioning the heap so an internal data structure can be described as a pool of objects separated from outside clients.

In addition to aliasing, one of the challenges when dealing with object invariants is object reentrancy. Common object-oriented design patterns —some explicitly intended to express higher order functional style— involve invoking an operation on an encapsulated abstraction while one is already in progress. This is a problem even in sequential programs, to which we confine our attention in this paper.

The discipline formalized in this paper protects invariants using ownership, but expressing ownership not in terms of types [11, 9] or logical connectives [23, 19] but rather auxiliary state. As explained in the sequel, this addresses the problem of reentrancy in a flexible way. Moreover, it offers a conceptually attractive way to limit the part of heap on which an object invariant depends, achieving encapsulation in a way that offers a glimpse of what an *imperative notion of module* might be.

Finally, the discipline goes beyond separation to deal with cooperation between objects without total dissolution of their individual encapsulation. The object is not the only useful unit of granularity for reasoning, but it is the unit of addressability in object-oriented languages. Moreover, the class construct is the primary unit of scope.

In Section 2 we explicate the problems in terms of program logic, leading to an approach using both ownership and pre/post commitments that describe cooperative interference between objects. In Section 3 we show how the approach has been realized in a verification discipline called Boogie [5, 7]. The reader is encouraged to consult the cited papers for an expository introduction. Informal soundness arguments are sketched in [5, 7]. The technical contribution of this paper is to formalize and prove soundness in terms of a standard semantic model.

[*]Partially supported by NSF award CCR-0208984; SRI Visiting Fellowship; Microsoft Research

Predicates are treated semantically so the results are useful both for verification systems based on weakest-precondition semantics, like ESC/Java [13] and the Boogie project, and for those like the LOOP project [12] which treat program logic as derived rules for reasoning directly in terms of semantics. Moreover there is no need for non-standard logical connectives, type systems, or any particular language for expressing properties of pointer structure. One of the benefits of the discipline is that properties such as double-linking in a list can be expressed in a decentralized way that lessens the need for reachability or other inductive predicates on the heap.

2. How encapsulation and atomicity justify modular reasoning about object invariants

Suppose that $\mathcal{I}$ is a predicate intended to be an invariant for an encapsulated data structure on which method m acts and $\mathcal{P}, \mathcal{Q}$ are predicates on data visible to callers of m. The aim of this paper is to justify reasoning of this form:

$$\frac{\{\mathcal{P} \wedge \mathcal{I}\}\ \mathit{body}\ \{\mathcal{Q} \wedge \mathcal{I}\}}{\{\mathcal{P}\}\ m\ \{\mathcal{Q}\}} \qquad (1)$$

In the rule, m is an invocation of the method and body is its implementation. The rule is very attractive. It allows the implementer of m to exploit the invariant while not exposing it to the client: $\mathcal{I}$ can involve identifiers that are in scope for body but not for call sites.

On the face of it, the rule is unsound: for $\mathcal{Q}$ to be established may well depend on precondition $\mathcal{I}$ which is not given in the conclusion. The idea is that $\mathcal{I}$ should *depend only on encapsulated state* so that it cannot be falsified by client code. To explain, we consider this rule.

$$\frac{\{\mathcal{P}\}\ S\ \{\mathcal{Q}\} \qquad S \text{ does not interfere with } \mathcal{I}}{\{\mathcal{P} \wedge \mathcal{I}\}\ S\ \{\mathcal{Q} \wedge \mathcal{I}\}} \qquad (2)$$

The condition "S does not interfere with $\mathcal{I}$" is intended to apply when S is outside the scope of encapsulation. In simple settings the condition can be expressed in terms of disjointness of identifiers. With aliasing it can be extremely difficult to express.

The benefit of rule (2), which has specifications symmetric to those in (1), is to undo the apparent unsoundness of (1). The argument is instructive. Consider a proof tree τ for some triple $\{\mathcal{P}_{main}\}\ S_{main}\ \{\mathcal{Q}_{main}\}$, that uses rule (1) and various other rules. That is, each node is an instance of a rule as usual. Now consider the tree τ' obtained from τ by changing some of the pre- and post-conditions, as follows. For every node n for rule (1) we conjoin $\mathcal{I}$ to the pre- and post-conditions in the conclusion, leaving the antecedent unchanged. Every node in the subtree at n (i.e., verifying body) is left unchanged. For the remaining nodes, $\mathcal{I}$ is conjoined everywhere. The result, τ', concludes with $\{\mathcal{P}_{main} \wedge \mathcal{I}\}\ S_{main}\ \{\mathcal{Q}_{main} \wedge \mathcal{I}\}$. Each use of the dubious rule (1) has become

$$\frac{\{\mathcal{P} \wedge \mathcal{I}\}\ \mathit{body}\ \{\mathcal{Q} \wedge \mathcal{I}\}}{\{\mathcal{P} \wedge \mathcal{I}\}\ m\ \{\mathcal{Q} \wedge \mathcal{I}\}}$$

which is allowed by an ordinary procedure call rule. But τ' is not a proof tree, because some other nodes no longer match rules. Suppose that the program exhibits proper encapsulation, in the sense that the state on which $\mathcal{I}$ depends is only manipulated inside body. Then rule (2) may be used to justify the introduction of $\mathcal{I}$ so τ' can be transformed to a proof tree. The end result is a proof of

$$\{\mathcal{P}_{main}\}\ \mathit{init}\ \{\mathcal{P}_{main} \wedge \mathcal{I}\}\ S_{main}\ \{\mathcal{Q}_{main} \wedge \mathcal{I}\} \Rightarrow \{\mathcal{Q}_{main}\}$$

and the original task has been accomplished.[1]

Object-oriented programming subverts this story in two ways. First, free use of pointers makes it difficult to ensure or even define and reason about the separation needed for the condition in rule (2). Second, our argument treats calls of m as *atomic* in a sense: Within the subtree for a call node, we did not and cannot conjoin $\mathcal{I}$ throughout; invariants are violated during updates to data structures. But if body invokes an operation on some object outside the encapsulation boundary, there is the possibility of a reentrant call. When that call occurs, $\mathcal{I}$ might not hold, but the point of rule (1) is to insist that, unbeknownst to the client, $\mathcal{I}$ is established before every invocation of m. The discipline presented in Section 3 deals with both of these problems.

The last problem we address is the *sharing of mutable state*. Rule (2) deals with separation of state. This is made beautifully explicit in Separation Logic, where $\wedge$ is replaced by the separating conjunction $*$ which obviates the need for any side condition: From $\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$ we can infer $\{\mathcal{P} * \mathcal{I}\}\ S\ \{\mathcal{Q} * \mathcal{I}\}$. This expresses the absence of relevant sharing: S and $\mathcal{I}$ depend on disjoint parts of the heap. When applicable this is very powerful. But what about the situation where sharing is needed? Common design patterns abound with situations where a configuration of several interlinked objects cooperate in a controlled way.

We write $\mathcal{I}(o)$ to make explicit the object for which an invariant is considered. Suppose $\mathcal{I}(o)$ depends on field f of another object p, say because there is a field g with $o.g = p$ and $\mathcal{I}(o)$ requires $o.g.f \geq 1$. Moreover, for some reason o does not own p. Though $\mathcal{I}(o)$ is at risk from updates of $p.f$, suppose p cooperates by only increasing

[1]O'Hearn *et al.* [19] express similar reasoning in a rule of this shape:

$$\frac{\{\mathcal{P}\}\ m\ \{\mathcal{Q}\} \vdash \{\mathcal{P}'\}\ S\ \{\mathcal{Q}'\}}{\{\mathcal{P} \wedge \mathcal{I}\}\ \mathit{body}\ \{\mathcal{Q} \wedge \mathcal{I}\} \vdash \{\mathcal{P}' \wedge \mathcal{I}\}\ S\ \{\mathcal{Q}' \wedge \mathcal{I}\}}$$

the value of f. Consider the following rule.

$$\frac{\{\mathcal{P}\}\, E.f \,:=\, E'\, \{\mathcal{Q}\} \qquad \{\mathcal{U} \wedge \mathcal{I}\}\, E.f \,:=\, E'\, \{\mathcal{I}\}}{\{\mathcal{P} \wedge \mathcal{U} \wedge \mathcal{I}\}\, E.f \,:=\, E'\, \{\mathcal{Q} \wedge \mathcal{I}\}}$$

The idea is that $\{\mathcal{P}\}\, E.f \,:=\, E'\, \{\mathcal{Q}\}$ specifies the assignment in the class performing the update of $p.f$. The triple $\{\mathcal{U} \wedge \mathcal{I}\}\, E.f \,:=\, E'\, \{\mathcal{I}\}$ is a commitment, advertised by the class of o, expressing that under condition $\mathcal{U}$ its invariant is not falsified. The conclusion is used in a way similar to rule (2) but imposes proof obligation $\mathcal{U}$.

The above rule is sound, being an instance of the standard rule of conjunction. But its intended use is for $\mathcal{I}$ declared in one class and $E.f \,:=\, E'$ occuring in code of a different class. The rule must be rejected for being incompatible with scope-based encapsulation.

The rule above is intended to be used in the context of the class of object p, which "grants" to o permission to depend on $p.f$, and the left-hand antecedent would be discharged in that context. But the antecedent $\{\mathcal{U} \wedge \mathcal{I}\}\, E.f \,:=\, E'\, \{\mathcal{I}\}$ should be discharged in the "friend" class to which access is granted. In that context $\mathcal{I}$ is visible but E, E' are not. The solution studied in this paper is for the friend class, in which $\mathcal{I}$ is declared, to include in its interface a commitment

$$\{\mathcal{U}(x,y) \wedge \mathcal{I}\}\, x.f \,:=\, y\, \{\mathcal{I}\} \tag{3}$$

where x, y are fresh variables used only to specify the commitment [7]. The actual formulation hides $\mathcal{I}$, so the commitment is that "under precondition $\mathcal{U}(x,y)$, an assignment $x.f \,:=\, y$ does not falsify my encapsulated invariant" and $\mathcal{U}$ is chosen to expose appropriate state of x. The commitment can be extended to yield a postcondition as well [7] but soundness for that is straightforward and omitted here.

3. Recovering encapsulation and atomicity in the presence of sharing and reentrancy

Atomicity. Atomicity poses a difficult problem for invariants in object-oriented programs. A sound approach which has seen considerable use is for a *caller* to establish its own invariant before it makes any outgoing method call, just in case it leads to a reentrant call back. In terms of the above proof tree transformation, this means $\mathcal{I}$ must hold as a precondition at nodes for each outgoing call in $body$ and then it is conjoined to predicates in the subtree for that call, to ensure that it holds for any nested calls back to the object for which we are maintaining $\mathcal{I}$. This approach has been called visible state semantics of invariants [17], but intermediate states are not observable in the sense of pre/post specifications. Although sound, this proposal is too restrictive for many practical purposes.

The discipline that we study [5] avoids exposing details about internal state by introducing a public boolean field inv to indicate whether an object's invariant holds. Being a boolean, it poses no difficulty with aliasing. Instead of struggling to decide in which states the rules should require $\mathcal{I}$ to hold, we require that the following holds in *all* states:

$$(\,\forall\, o \,\bullet\, o.inv \Rightarrow \mathcal{I}_{type(o)}(o)\,) \tag{4}$$

Our quantifications range over locations allocated in the current heap. We write $type\ o$ for the type of the object (its so-called dynamic class, though in this extended abstract we omit subclassing). If a method specification has $\mathsf{self}.inv$ as precondition then $\mathcal{I}(\mathsf{self})$ can be assumed for verification of its implementation.

The field inv is an auxiliary, meaning that it may be used in specifications but not in ordinary code. To update this and other auxiliaries, we do not use ordinary field assignments but rather special statements that can be distinguished from ordinary assignments and are subject to special rules. The reasoner is free to decide where inv does and does not hold. The rule for $E.f \,:=\, E'$ has as precondition $\neg E.inv$ which ensures that an update does not violate (4); we add further preconditions in the sequel. The special statement "$\mathbf{pack}\ E$" sets $E.inv$ true; setting it false is the purpose of $\mathbf{unpack}$. These are defined later because they involve the next topic.

Ownership. Like atomicity, ownership and cooperative friendship are treated using auxiliary fields which express state-dependent encapsulation. Encapsulation is realized in *system invariants* like (4) which can be exploited wherever they are needed in verification.

Ownership is a state-dependent form of encapsulation: an invariant $\mathcal{I}(o)$ is allowed to depend on fields of o and fields of objects owned by o. The auxiliary field own holds a reference to an object's owner and is $\mathbf{null}$ if there is no owner. The auxiliary field $comm$ is a boolean that represents whether an object is *committed* to its owner, in which case only its owner is allowed to unpack it. The special statement $\mathbf{set\text{-}owner}\ E\ \mathbf{to}\ E'$ has the effect $E.own \,:=\, E'$. Making it a special statement indicates that it has no observable effect on the program semantics, although it is subject to stipulated preconditions, e.g., $\neg E.inv$ just as with any field update. The stipulated preconditions are summarized in Table 1.

The statement $\mathbf{pack}\ E$ has the effect of setting $E.inv$ true and also setting $o.comm$ true for all o owned by E. (Table 5 gives the formal definition.) In order to maintain system invariant (4), the stipulated precondition includes that $\mathcal{I}(E)$ holds. The other preconditions involve features explained in the sequel; see [5] or [7] for a more leisurely introduction to the ownership discipline.

Statement $\mathbf{unpack}\ E$ has precondition $\neg E.comm$, that is, only an object's owner is allowed to unpack it. Besides setting $E.inv$ false, this statement sets $o.comm$ false

$E \neq \mathbf{null} \wedge \neg E.inv \wedge \mathcal{I}_B(E) \wedge (\forall p \bullet p.own = E \Rightarrow \neg p.comm \wedge p.inv\ );$	$\mathbf{pack}\ E$
$E \neq \mathbf{null} \wedge E.inv \wedge \neg E.comm;$	$\mathbf{unpack}\ E$
$\neg \mathsf{self}.inv \wedge E' \neq \mathbf{null};$	$\mathbf{attach}\ E'$
$\neg \mathsf{self}.inv \wedge E' \neq \mathbf{null} \wedge (\neg E'.inv \vee (\forall g : B \in \mathit{pivots}\ C \bullet E'.g \neq \mathsf{self}\ ));$	$\mathsf{self} : B \vdash \mathbf{detach}\ E'$
$E \neq \mathbf{null} \wedge \neg E.inv$ $\wedge (\forall p \bullet p \in E.deps \wedge f \in \mathit{reads}(type(p), B) \Rightarrow \neg p.inv \vee \mathcal{U}_{type(p),B,f}(p, E, E')\ );$	$E.f := E'$
$E \neq \mathbf{null} \wedge \neg E.inv \wedge (E' = \mathbf{null} \vee \neg E'.inv)$ $\wedge (\forall p \bullet p \in E.deps \wedge own \in \mathit{reads}(type(p), B) \Rightarrow \neg p.inv \vee \mathcal{U}_{type(p),B,own}(p, E, E')\ );$	$\mathbf{set\text{-}owner}\ E\ \mathbf{to}\ E'$

Table 1. Stipulated preconditions. Static types are assumed to be $E : B$ and $E' : C$.

for all o with $o.own = E$, thus making owned objects available for unpacking.

The pack and unpack statements effectively achieve a hierarchical notion of ownership, so $\mathcal{I}(o)$ is allowed to depend on objects that it transitively owns. The discipline yields system invariants (10) and (11), included in the summary at the end of this section. As a consequence of the invariants, the precondition $\neg E.inv$ for field update means that an object cannot be updated unless its owner is unpacked. Surprisingly, this is sound even with all fields considered public —not to say that is advisable in practice.

Friendship. In the discipline of [7], which extends [5] to handle cooperative sharing, three special declarations may appear in a class B. First, there may be a sequence of friend declarations[2]

$$\mathbf{friend}\ C\ \mathbf{reads}\ \bar{f} \tag{5}$$

where each field name f in $\bar{f}$ is either declared in B or is the auxiliary own. The set of names C for which B has a friend declaration is written $friends\ B$. Moreover, *reads* gives the fields that a friend reads from a granter: $\mathit{reads}(C, B) = \bar{f}$ for the declaration above, and $\mathit{reads}(C, B) = \varnothing$ if $C \notin friends\ B$.

Second, in each class C there should be exactly one declaration

$$\mathbf{invariant}\ \mathcal{I}_C\ \mathbf{pivots}\ pivs \tag{6}$$

where $\mathcal{I}_C$ is a predicate on $\mathsf{self} : C$ (and the heap of course) and $pivs$ is a sequence of (fieldname,classname) pairs, written $g : B$. Define $\mathit{pivots}\ C = pivs$. For each $(g : B)$ in $\mathit{pivots}\ C$, field g must be declared in C and C must be in $friends\ B$. Definition 6 in the sequel requires that $\mathcal{I}_C(o)$ depends on $p.f$ only if either $p = o$, p is transitively owned by o, or $p = o.g$ for some pivot field. Thus we limit friend dependencies to a single step of indirection in this extended abstract.[3]

[2]The term "reads" is slightly misleading in that (in this paper) all fields are public and thus subject to update from code in any class.

[3]Dependence on $o.g.h$ for immutable field h of $o.g$ is clearly allowable (e.g. $length$ if g is an array). And friendship localizes invariants, lessening the need for farther reaching expressions.

The third special declaration is the update guard. In any class C there should be at least one declaration

$$\mathbf{guard}\ \mathsf{piv}.f := \mathsf{val}\ \mathbf{by}\ \mathcal{U}_{C,B,f}(\mathsf{self}, \mathsf{piv}, \mathsf{val}) \tag{7}$$

for each B, f with $f \in \mathit{reads}(C, B)$. It can be the predicate *false* by default. Here the predicate $\mathcal{U}_{C,B,f}(\mathsf{self}, \mathsf{piv}, \mathsf{val})$ is over $\mathsf{self} : C, \mathsf{piv} : B, \mathsf{val} : T$, where T is the type of f in B and $\mathsf{piv}, \mathsf{val}$ are special variables used only for this purpose.[4] Note that the pivot name g need not be visible to B. Generalizing from (3) in Section 2, there may be several classes C in $friends\ B$ that read f, and each one's guard is needed as precondition to update f.

To accomodate dynamic allocation of unboundedly many instances of C, each of which could potentially depend on a given p of type B, we introduce one last auxiliary field, $deps$, so that $p.deps$ is a set of locations of friends o that may depend on p in that $o.g = p$ for some pivot g. The system invariant associated with $deps$ is that $o.inv \Rightarrow o \in p.deps$ if o has a pivot pointing at p, see Definition 1(12). To maintain this invariant, an admissible object invariant $\mathcal{I}(o)$ is required to imply $o \in p.deps$ (see Definition 6 in Section 6).

Besides writing $\mathcal{I}(o)$ to denote instantiation with location o, we use a squiggly notation $\mathcal{P}[E.f :\approx E']$ for the semantic counterpart of substitution (i.e., the inverse image of field update). Also, unqualified g is short for $\mathsf{self}.g$.

Declaration (7) generates a proof obligation. For each $(g : B) \in \mathit{pivots}\ C$ and each $f \in \mathit{reads}(C, B)$, the following must be valid (where $\mathsf{self} : C$).

$$\mathcal{I}_C \wedge g \neq \mathbf{null} \wedge \mathcal{U}_{C,B,f}(\mathsf{self}, g, \mathsf{val}) \Rightarrow \mathcal{I}_C[g.f :\approx \mathsf{val}] \tag{8}$$

That is, if $\mathcal{I}_C$ depends on $g.f$ then it is maintained by an assignment of val to $g.f$ under precondition $\mathcal{U}_{C,B,f}$.

[4]The notation $\mathcal{U}_{C,B,f}$ is just a way for our formalism to keep track (in Table 1) that this is the predicate used to guard any update of f in an object of type B with respect to the invariant of C. It is important to allow more than one update guard for given C, B, f, offering a choice for use at different update sites.

Note that in program logic it is common to treat a field (or array) update $a.f := E$ as a simple assignment $a := [a \mid f := E]$ but our formulations are in terms of the syntax $a.f := E$.

The field $deps$ is manipulated by two special statements: $\mathbf{attach}\ E$ adds the value of E to $\mathsf{self}.deps$ and $\mathbf{detach}\ E$ removes it. See Table 5.

Field update revisited. Pre- and post-conditions in method specifications may mention any of the special fields $inv, comm, own, deps$, as can intermediate assertions used in reasoning. There is no restriction on method specifications or on where special statements are used. But these statements and field updates are subject to the preconditions stipulated in Table 1.

The primary benefit of the discipline is that object invariants hold at any control point in the program where inv holds, as formalized in (9) of the following.

Definition 1 (system invariant) *The system invariant $\mathcal{SI}$ is the conjunction of the following.*

$$(\forall\, o \bullet o.inv \Rightarrow \mathcal{I}_{type(o)}(o)\,) \tag{9}$$

$$(\forall\, o \bullet o.inv \Rightarrow (\forall\, p \bullet p.own = o \Rightarrow p.comm\,)) \tag{10}$$

$$(\forall\, o \bullet o.comm \Rightarrow o.inv\,) \tag{11}$$

$$\text{For every } C \text{ and } (g:B) \in pivots\ C\text{:} \quad (\forall\, p:C, o:B \bullet p.g = o \wedge p.inv \Rightarrow p \in o.deps\,) \tag{12}$$

Because quantifiers range over locations allocated in the current heap, the antecedent in (12) holds when $p.g = \mathsf{nil}$.

The main result of the paper is that $\mathcal{SI}$ is invariant for any properly annotated program.

The precondition for field update may appear daunting. But the program text gives finitely many C such that $f \in$ *reads*(C, B). So the condition can be expressed as a finite conjunction, indexed by the classes C in $friends\ B$ such that $f \in$ *reads*(C, B). Each conjunct takes the form

$$(\forall\, p:C \bullet p \in E.deps \Rightarrow \neg p.inv \vee \mathcal{U}_{C,B,f}(p, E, E')\,)$$

That is, the displayed condition must be established for each of the friends C declared in B that read f. Typically there are few or none.

Admissible formulas. Because the semantic Definition 6 of admissibility is slightly intricate, we mention here sufficient but not necessary conditions for a formula to denote an admissible invariant. A formula over $\mathsf{self}:C$ will denote an admissible invariant provided that for every field reference $E.f$, f is neither inv nor $comm$ and moreover one of the following hold:

- E is self
- E is some variable x in the scope of $(\forall\, x \bullet x.own = \mathsf{self} \Rightarrow \dots)$
- E is $\mathsf{self}.g$ for declared pivot $g:B$ with $C \in friends\ B$ and moreover a top level conjunct of the formula is or implies $\mathsf{self}.g = \mathbf{null} \vee \mathsf{self} \in \mathsf{self}.g.deps$

In some examples the invariant $\mathsf{self}.g \neq \mathbf{null} \iff \mathsf{self} \in \mathsf{self}.g.deps$ is useful, but in others there is no need for $deps$ to be so accurate.

A useful idea that we omit in the formal treatment is to tag field declarations as $\mathbf{rep}\ h$ to say that field h satisfies invariant $h = \mathbf{null} \vee h.own = \mathsf{self}$; then we can also allow E to be a sequence of $\mathbf{rep}$ fields, as that implies transitive ownership. Similarly, declaration $\mathbf{peer}\ h$ can be introduced to indicate that $h = \mathbf{null} \vee \mathsf{self}.own = h.own$ (this is admissible, given suitable friend and pivot declarations). Then $E.f$ is allowed if $E \equiv h.h_0.h_1.\dots.h_n$ where h is tagged $\mathbf{rep}$ and each h_i is tagged $\mathbf{peer}$. For example, a $List$ class could have field $\mathbf{rep}\ head:Node$ and $Node$ could have $\mathbf{peer}\ next:Node$. These local invariants imply that every node $p \in o.head.next^*$ is owned by list o, without the need to express it using such a path expression in the invariant of $List$.

In [7] the Subject/View pattern serves as an example of reasoning about $deps$. The Subject notifies its views when its state changes, so it maintains a data structure that represents the set of its current views, vs. Its invariant $p \in \mathsf{self}.deps \Rightarrow p \in vs$ puts it in a position to establish the precondition for updating its fields on which Views depend.

The discipline has been formulated in a way that admits aliasing among pivots. An interesting exercise is to consider class B with field $f:\mathbf{int}$ and $\mathbf{friend}\ C\ \mathbf{reads}\ f$, where $\mathcal{I}_C$ is $g.f = 0 \Rightarrow g'.f = 1$ for pivots $g, g':B$.

4. An illustrative language

The key features of the discipline involve only field update and the five primitive statements that manipulate auxiliary fields. To demonstrate that the discipline scales to practical languages including general recursion and object-oriented constructs, and to lay the groundwork for the refinements needed to cope with subclassing and inheritance [5], we use a language similar to the imperative core of Java (as in, e.g., [4, 8]) including value parameters and results, mutable local variables and object fields, and dynamic instantiation of objects. Expressions have no side effects but may dereference chains of fields.

Syntax. The grammar is in Table 2. A complete program is given as a *class table*, CT, that associates each declared class name with its declaration. The typing rules make use of some helping functions that are defined in terms of CT, so the typing relation $\vdash$ depends on CT but this is elided in the notation. Because typing of each class is done in the context of the full table, methods can be recursive (mutually) and so can field types.

To define the helping functions, suppose $CT(C) = \mathbf{class}\ C\ \{\ \bar{f}:\bar{T}_0;\ \ \bar{M}\ \}$. For fields, define *fields* $C =$

CL	$::=$	$\mathbf{class}\ C\ \{\ \bar{f}:\bar{T};\ \bar{M}\ \}$	class named C with public fields $\bar{f}$, public methods $\bar{M}$
T	$::=$	$\mathbf{bool} \mid \mathbf{unit} \mid C$	data type, where C ranges over class names
M	$::=$	$m(\bar{x}:\bar{T}):T\ \{S\}$	method m with result type T, parameters $\bar{x}$ of types $\bar{T}$
S	$::=$	$\mathbf{if}\ E\ \mathbf{then}\ S\ \mathbf{else}\ S \mid S;\ S$	alternative; sequence
	$\mid$	$x := E \mid \mathbf{var}\ x:T := E\ \mathbf{in}\ S$	assign to variable; initialized local variable block
	$\mid$	$x := E.m(\bar{E}) \mid E.f := E \mid x := \mathbf{new}\ C$	invoke method; assign to field; construct object
	$\mid$	$\mathbf{pack}\ E \mid \mathbf{unpack}\ E \mid \mathbf{set\text{-}owner}\ E\ \mathbf{to}\ E$	set and unset inv; update own
	$\mid$	$\mathbf{attach}\ E \mid \mathbf{detach}\ E$	add and remove from $\mathsf{self}.deps$
E	$::=$	$x \mid \mathbf{null} \mid \mathbf{true} \mid \mathbf{false} \mid E.f \mid E = E$	variable; constants; field access; reference equality

Table 2. Programming language. A distinguished variable name, self, is used for the target parameter and another, result, is used for the return value of a method. Identifiers like $\bar{T}$ with bars on top indicate finite lists (the bar has no meaning). Type **unit**, often called "void", has a single value and is used for methods that return nothing useful.

$\bar{f}:\bar{T}_0$. For use in the semantics, we extend *fields* C to a function *xfields* C that also assigns types to the auxiliary fields: $inv:\mathbf{bool}$, $comm:\mathbf{bool}$, $own:(Loc\cup\{\mathsf{nil}\})$, $deps:setof(Loc)$. Here $(Loc\cup\{\mathsf{nil}\})$ and $setof(Loc)$ are not types in the programming language, but notation in the metalanguage to streamline later definitions.

For M in the list $\bar{M}$ of method declarations, with $M = m(\bar{x}:\bar{T}_1):T\ \{S\}$, we define *mtype*$(m,C) = \bar{x}:\bar{T}_1 \to T$. In the semantics it is convenient for the input to a method to be a store, mapping self and $\bar{x}$ to their values.

$$\frac{C = \Gamma x \qquad x \neq \mathsf{self}}{\Gamma \vdash x := \mathbf{new}\ C}$$

$$\frac{\Gamma \vdash E_0 : C \qquad (f:T) \in \mathit{fields}\ C \qquad \Gamma \vdash E_1 : T}{\Gamma \vdash E_0.f := E_1}$$

$$\frac{\Gamma \vdash E : C}{\Gamma \vdash \mathbf{pack}\ E \qquad \Gamma \vdash \mathbf{unpack}\ E}$$

$$\frac{\Gamma \vdash E : C \qquad C \in friends(\Gamma\ \mathsf{self})}{\Gamma \vdash \mathbf{attach}\ E \qquad \Gamma \vdash \mathbf{detach}\ E}$$

$$\frac{\Gamma \vdash E : C \qquad \Gamma \vdash E' : C'}{\Gamma \vdash \mathbf{set\text{-}owner}\ E\ \mathbf{to}\ E'}$$

Table 3. Selected typing rules.

A class table is well formed if each class is well formed, which simply means that each method declaration is well formed according to the following rule (see also Table 3).

$$\frac{\mathsf{self}:C, \bar{x}:\bar{T}, \mathsf{result}:T \vdash S}{C \vdash m(\bar{x}:\bar{T}):T\ \{S\}}$$

Semantics. Methods are associated with classes, in a *method environment*, rather than with object states. For this reason the semantic domains are relatively simple; there are no recursive domain equations to be solved (cf. [22, 21]).

Table 4 defines the semantic domains. For data type T, the domain $[\![T]\!]$ is a set of values. This induces the domain $[\![\Gamma]\!]$ of *stores* as usual. The domain $[\![state\ C]\!]$ of states for an object of type C is just stores for *xfields* C (that is, including the auxiliary fields). A *heap* is a finite map from locations to object states, such that every location in any field is in the domain of the heap. Function application associates to the left, so $h\ o\ f$ looks up f in the object state $h\ o$. We also use a dot for emphasis with fields, writing $h\ o.f$ for $h\ o\ f$.

The domain $[\![Heap \otimes \Gamma]\!]$ contains program states (h,s) consisting of a heap and a store $s \in [\![\Gamma]\!]$ with no dangling locations. Similarly, $[\![Heap \otimes T]\!]$ contains pairs (h,v) where $v \in [\![T]\!]$ and is not a dangling location. The preceding domains are all complete partial orders, ordered by equality. The next domains are function spaces into lifted domains. The meaning of expression $\Gamma \vdash E:T$ is a function $[\![Heap \otimes \Gamma]\!] \to [\![T_\perp]\!]$. The meaning of $\Gamma \vdash S$ is a function $[\![MEnv]\!] \to [\![Heap \otimes \Gamma]\!] \to [\![(Heap \otimes \Gamma)_\perp]\!]$ that takes a method environment μ (see below) and a state (h,s) and returns a state or $\perp$ for divergence or error. Table 5 gives the semantics for commands.

The domain $[\![C, \bar{x}, \bar{T} \to T]\!]$ is the set of meanings for methods of class C with result T and parameters $\bar{x}:\bar{T}$. Ordered pointwise, this is a complete partial order with bottom. Finally, a *method environment* $\mu \in [\![MEnv]\!]$ sends each C and method name m declared in C to a meaning

$$[\![C]\!] = \{\mathsf{nil}\} \cup \{o \mid o \in \mathit{Loc} \wedge \mathit{type}\ o = C\} \qquad [\![\mathbf{bool}]\!] = \{\mathsf{tt},\mathsf{ff}\} \qquad [\![\mathbf{unit}]\!] = \{\mathsf{it}\}$$

$$\begin{array}{lcl}
[\![\Gamma]\!] & = & \{s \mid \mathit{dom}\ s = \mathit{dom}\,\Gamma \wedge s\ \mathsf{self} \neq \mathsf{nil} \wedge (\,\forall x \in \mathit{dom}\ s \bullet s\ x \in [\![\Gamma\ x]\!]\,)\} \\
[\![\mathit{state}\,C]\!] & = & \{s \mid \mathit{dom}\ s = \mathit{dom}(\mathit{xfields}\ C) \wedge (\,\forall (f:T) \in \mathit{xfields}\ C \bullet s\ f \in [\![T]\!]\,)\} \\
[\![\mathit{Heap}]\!] & = & \{h \mid \mathit{dom}\ h \subseteq_{\mathit{fin}} \mathit{Loc} \wedge \mathit{noDanglingRef}\ h \wedge (\,\forall o \in \mathit{dom}\ h \bullet h\ o \in [\![\mathit{state}\,(\mathit{type}\ o)]\!]\,)\} \\
 & & \text{where } \mathit{noDanglingRef}\ h \text{ iff } \mathit{rng}\ s \cap \mathit{Loc} \subseteq \mathit{dom}\ h \text{ for all } s \in \mathit{rng}\ h \\
[\![\mathit{Heap} \otimes \Gamma]\!] & = & \{(h,s) \mid h \in [\![\mathit{Heap}]\!] \wedge s \in [\![\Gamma]\!] \wedge \mathit{rng}\ s \cap \mathit{Loc} \subseteq \mathit{dom}\ h\} \\
[\![\mathit{Heap} \otimes T]\!] & = & \{(h,v) \mid h \in [\![\mathit{Heap}]\!] \wedge v \in [\![T]\!] \wedge (v \in \mathit{Loc} \Rightarrow v \in \mathit{dom}\ h)\} \\
[\![C,\, \bar{x},\, \bar{T} \to T]\!] & = & [\![\mathit{Heap} \otimes (\bar{x}:\bar{T}, \mathsf{self}:C)]\!] \to [\![(\mathit{Heap} \otimes T)_\perp]\!] \\
[\![\mathit{MEnv}]\!] & = & \{\mu \mid \ (\,\forall C, m \bullet \mu\ C\ m \text{ is defined iff } \mathit{mtype}(m,C) \text{ is defined,} \\
 & & \qquad \text{and then } \mu\ C\ m \in [\![C,\bar{x},\bar{T}\to T]\!] \text{ where } \mathit{mtype}(m,C) = \bar{x}:\bar{T}\to T\ )\ \}
\end{array}$$

Table 4. Semantic domains. For $s \in \mathit{state}\,C$ we take $s\ \mathit{own} \in \mathit{Loc} \cup \{\mathsf{nil}\}$ and $s\ \mathit{deps} \in \mathbb{P}_{\mathit{fin}}(\mathit{Loc})$.

of the right type. This too is a complete partial order with bottom.

Definition 2 (semantics of method declaration) *For a declaration $M = m(\bar{T}\ \bar{x}):T\ \{S\}$ in class C, and any method environment μ, define $[\![M]\!]\mu$ by*

$$\begin{array}{rl}
[\![M]\!]\mu(h,s) = & \mathsf{let}\ s_1 = [s + \mathsf{result} \mapsto \mathit{default}]\ \mathsf{in} \\
 & \mathsf{let}\ \Gamma = \bar{x}:\bar{T}, \mathsf{self}:C, \mathsf{result}:T\ \mathsf{in} \\
 & \mathsf{let}\ (h_0, s_0) = [\![\Gamma \vdash S]\!]\mu(h,s)\ \mathsf{in} \\
 & (h_0,\ s_0\ \mathsf{result})
\end{array}$$

The default values are it for $\mathbf{unit}$, ff for $\mathbf{bool}$, and nil for object types, here and for $\mathbf{new}$. Thus a new object has $\mathit{inv} = \mathsf{ff}$, $\mathit{comm} = \mathsf{ff}$, and $\mathit{own} = \mathsf{nil}$; also $\mathit{deps} = \varnothing$.

Definition 3 (semantics of complete program) *The semantics $[\![CT]\!]$ of a well formed class table CT is the least upper bound of the ascending chain $\mu \in \mathbb{N} \to [\![\mathit{MEnv}]\!]$ defined by $\mu_0\ C\ m = (\,\lambda\,(h,s) \bullet \perp\,)$ and $\mu_{j+1}\ C\ m = [\![M]\!]\mu_j$ for M the declaration of m in C.*

5. Dependence and assertions

A *predicate* for some state type Γ is just a subset $\mathcal{P} \subseteq [\![\mathit{Heap} \otimes \Gamma]\!]$. Note that $\perp \notin \mathcal{P}$. An invariant for C is a predicate $\mathcal{I}_C \subseteq [\![\mathit{Heap} \otimes (\mathsf{self}:C)]\!]$. Care is needed in properly converting a predicate on one state space to a predicate on another but the details are straightforward. A few details are in the Appendix. We write $(h,s) \models \mathcal{P}$ to mean $(h,s) \in \mathcal{P}$, sometimes as a hint that coercion may be needed, e.g., to take an instance $\mathcal{I}_C(o)$ to be in $\mathbb{P}[\![\mathit{Heap}]\!]$ so $\mathcal{SI}$ depends only on heap.

If $\mathcal{P}_o$ is in $\mathbb{P}[\![\mathit{Heap} \otimes \Gamma]\!]$ for every location o then $(\,\forall o \bullet \mathcal{P}\,)$ is the subset of $[\![\mathit{Heap} \otimes \Gamma]\!]$ defined by

$$(h,s) \models (\,\forall o \bullet \mathcal{P}_o\,) \text{ iff for all } o \in \mathit{dom}\ h,\ (h,s) \in \mathcal{P}_o$$

Recall that nil is not a location. It is also convenient to restrict the range of quantification to a particular type: define $(\,\forall o:C \bullet \mathcal{P}\,)$ to abbreviate $(\,\forall o \bullet \mathit{type}\ o = C \Rightarrow \mathcal{P}\,)$. Note that quantification is over all allocated objects, and in the semantics there is neither explicit deallocation nor garbage collection; the range of quantification includes unreachable objects but this does not obtrude in the sequel.

In terms of formulas, a predicate depends on $E.f$ if updating $E.f$ can falsify the predicate. The following semantic formulation is convenient for our purposes.

Definition 4 (depends) *Predicate $\mathcal{P}$* depends on *$o.f$ iff there is some (h,s) such that $\mathcal{P}$ depends on $o.f$ in (h,s). Moreover, $\mathcal{P}$* depends on *$o.f$* in *(h,s) iff $(h,s) \in \mathcal{P}$, $o \in \mathit{dom}\ h$, and $([h \mid o.f \mapsto v], s) \notin \mathcal{P}$ for some v.*

There are several ways the system invariants $\mathcal{SI}$ could be used: as a "fact", included in what is sometimes called the "background predicate" that axiomatizes the semantics of the programming language (e.g., absence of dangling locations, $\mathsf{self} \neq \mathbf{null}$); as lemmas for reasoning directly in terms of program semantics; or in rules of a logic. We want to justify that $\mathcal{SI}$ to be asserted at any control point, and this is sound only if the stipulated preconditions are imposed on field updates and special statements. Aiming for a formulation that is perspicuous and lends itself to various uses like those listed, we use $\mathbf{assert}$ statements. Our main results show, essentially, that for *any* constituent command S of a program properly annotated with assertions (Table 1), we have $\{\mathcal{SI}\}\ S\ \{\mathcal{SI}\}$ in the sense of partial correctness.

The notion of partial correctness we choose is error-ignoring, for which reason our semantics identifies null-dereference errors with divergence. For practical purposes, it is more useful to use a correctness notion that implies the absence of runtime errors, especially for verification systems intended for use on development code which rarely has full functional specifications. But for the main statements of interest in this paper it is straightforward to formulate

$$\begin{aligned}
[\![\Gamma \vdash E_0.f := E_1]\!]\mu(h,s) &= \text{let } q = [\![\Gamma \vdash E_0 : C]\!](h,s) \text{ in}\\
&\quad\ \text{if } q = \text{nil then } \bot \text{ else let } v = [\![\Gamma \vdash E_1 : T]\!](h,s) \text{ in } ([h \mid q.f \mapsto v], s)\\
[\![\Gamma \vdash x := \mathbf{new}\ C]\!]\mu(h,s) &= \text{let } q = \mathit{fresh}(C,h) \text{ in let } h_1 = [h + q \mapsto [\mathit{fields}\ C \mapsto \mathit{defaults}]] \text{ in } (h_1, [s \mid x \mapsto q])\\
[\![\Gamma \vdash \mathbf{pack}\ E]\!]\mu(h,s) &= \text{let } q = [\![\Gamma \vdash E : C]\!](h,s) \text{ in if } q = \text{nil then } \bot \text{ else}\\
&\quad\ \text{let } h_1 = (\,\lambda p \in \mathit{dom}\ h \bullet \text{if } h\,p.\mathit{own} = q \text{ then } [h\,p \mid \mathit{comm} \mapsto \text{tt}] \text{ else } h\,p\,) \text{ in}\\
&\quad\ ([h_1 \mid q.\mathit{inv} \mapsto \text{tt}],\ s)\\
[\![\Gamma \vdash \mathbf{unpack}\ E]\!]\mu(h,s) &= \text{let } q = [\![\Gamma \vdash E : C]\!](h,s) \text{ in if } q = \text{nil then } \bot \text{ else}\\
&\quad\ \text{let } h_1 = (\,\lambda p \in \mathit{dom}\ h \bullet \text{if } h\,p.\mathit{own} = q \text{ then } [h\,p \mid \mathit{comm} \mapsto \text{ff}] \text{ else } h\,p\,) \text{ in}\\
&\quad\ ([h_1 \mid q.\mathit{inv} \mapsto \text{ff}],\ s)\\
[\![\Gamma \vdash \mathbf{attach}\ E]\!]\mu(h,s) &= \text{let } q = [\![\Gamma \vdash E : C]\!](h,s) \text{ in if } q = \text{nil then } \bot \text{ else}\\
&\quad\ \text{let } p = s \text{ self in } ([h \mid p.\mathit{deps} \mapsto h\,p.\mathit{deps} \cup \{q\}],\ s)\\
[\![\Gamma \vdash \mathbf{detach}\ E]\!]\mu(h,s) &= \text{let } q = [\![\Gamma \vdash E : C]\!](h,s) \text{ in if } q = \text{nil then } \bot \text{ else}\\
&\quad\ \text{let } p = s \text{ self in } ([h \mid p.\mathit{deps} \mapsto h\,p.\mathit{deps} - \{q\}],\ s)\\
[\![\Gamma \vdash \mathbf{set\text{-}owner}\ E_0\ \mathbf{to}\ E_1]\!]\mu(h,s) &= \text{let } q = [\![\Gamma \vdash E : C]\!](h,s) \text{ in if } q = \text{nil then } \bot \text{ else}\\
&\quad\ \text{let } p = [\![\Gamma \vdash E' : C']\!](h,s) \text{ in } ([h \mid q.\mathit{own} \mapsto p],\ s)
\end{aligned}$$

Table 5. Semantics of selected commands. We let v range over values of various types, and write q or p where the value is either a location or nil. (N.B. elsewhere in the paper these identifiers usually range over locations only.) The function update expression $[h \mid q.f \mapsto v]$ abbreviates the update $[h \mid q \mapsto [h\,q \mid f \mapsto v]]$. We write $[h + q \mapsto \ldots]$ for function extension. The metalanguage construct "let $v = \alpha$ in $\ldots$" is $\bot$ if $\alpha = \bot$. Assume fresh is an arbitrary function to Loc such that $\mathit{type}(\mathit{fresh}(C,h)) = C$ and $\mathit{fresh}(C,h) \notin \mathit{dom}\ h$.

preconditions for the absence of such errors and they are included in Table 1.

Soundness for commands is formulated as follows: If $h \models \mathcal{SI}$ and $[\![\Gamma \vdash S]\!]\mu(h,s) \neq \bot$ then $h_0 \models \mathcal{SI}$ where $(h_0, s_0) = [\![\Gamma \vdash S]\!]\mu(h,s)$. In the case that S is a method call, this depends on the assumption that each method meaning $\mu\ C\ m$ maintains $\mathcal{SI}$. To show that the assumption is discharged requires more commitment to a particular program semantics and we have chosen a denotational one in which method meanings are given as the lub of a chain of approximations. Thus the main theorem states that $\mathcal{SI}$ is maintained by every method in the environment μ denoted by a properly annotated class table.

Assert statements were not listed in the grammar because we allow semantic predicates not necessarily expressible in a particular language. Define $[\![\Gamma \vdash \mathbf{assert}\ \mathcal{P}]\!]\mu(h,s) =$ if $(h,s) \in \mathcal{P}$ then (h,s) else $\bot$. This is independent from μ, and $[\![\mathit{Heap} \otimes \Gamma]\!]$ is flat, so there is no problem with continuity.

6. Soundness

Definition 5 (transitive ownership) *For any heap h, the transitive ownership relation $\preceq^h$ on $\mathit{dom}\ h$ is defined inductively by the conditions $o = h\,p.\mathit{own} \Rightarrow o \preceq^h p$ and $o \preceq^h q \wedge q = h\,p.\mathit{own} \Rightarrow o \preceq^h p$.*

Definition 6 (admissible invariant) *A predicate $\mathcal{P} \subseteq [\![\mathit{Heap} \otimes (\mathsf{self} : C)]\!]$ is admissible as an invariant for C provided that for every (h,s) and every o, f such that $\mathcal{P}$ depends on $o.f$ in (h,s), field f is neither inv nor comm, and one of the following named conditions holds:*

local: $o = s(\mathsf{self})$

owner: $s(\mathsf{self}) \preceq^h o$ *and* $f \not\equiv \mathit{deps}$

friend: $o = h(s(\mathsf{self})).g$ *and* $s(\mathsf{self}) \in h\,o.\mathit{deps}$, *for some* $(g : B) \in$ *pivots* C. *And either* $f \in$ *reads*(C,B) *or* $f \equiv \mathit{deps}$*; in the latter case, for any* X *with* $([h \mid o.\mathit{deps} \mapsto X], s) \notin \mathcal{P}$ *we have* $([h \mid o.\mathit{deps} \mapsto X \cup \{s(\mathsf{self})\}], s) \in \mathcal{P}$.

The only auxiliary field allowed in *reads*(C,B) is own, so the condition for $f \equiv \mathit{deps}$ in friend dependencies essentially ensures that the only way for an object to depend on some other object's deps is by $s(\mathsf{self}) \in h\,o.\mathit{deps}$.

Note that $o \preceq^h o$ implies $\neg h\ o.\mathit{inv}$.

To clarify the definitions, suppose $\mathcal{I}_C$ is admissible for C and $h \models \mathcal{SI}$. If $\mathcal{I}_C(q)$ depends on $o.f$ in (h,s) then $f \not\equiv \mathit{inv}$, $f \not\equiv \mathit{comm}$, and one of the following holds:
(local) $o = q$;
(owner) $q \preceq^h o$ and $f \not\equiv \mathit{deps}$; or
(friend) $o = h\,q.g$ and $q \in h\,o.\mathit{deps}$, for some $(g : B) \in$ *pivots* C such that $f \in$ *reads*(C,B) or $f \equiv \mathit{deps}$. In

case $f \equiv deps$, we have that $[h \mid o.deps \mapsto X] \not\models \mathcal{I}_C(q)$ implies $[h \mid o.deps \mapsto X \cup \{q\}] \models \mathcal{I}_C(q)$ for any X.

Lemma 6.1 (co-dependence) *If $o \in dom\ h$ and $(h,s) \models \mathcal{I}_C(o)$ for admissible $\mathcal{I}_C$ then for any $g \in pivots\ C$ we have $h\,o.g \neq \mathsf{nil} \Rightarrow o \in h(h\,o.g).deps$.*

Lemma 6.2 (transitive ownership) *Suppose $h \models \mathcal{SI}$, $o \preceq^h p$, and $h\,o.inv = \mathtt{tt}$. Then $h\,p.comm = \mathtt{tt}$.*

Definition 7 (properly annotated class table) *A properly annotated class table is one such that*

- *there are declarations as defined in Section 3;*
- *each object invariant $\mathcal{I}_C$ is admissible;*
- *each field update and special statement is preceded by an $\mathbf{assert}$ that implies the stipulated precondition (Table 1); and*
- *each update guard satisfies its obligation (8).*

We say method environment μ *maintains* $\mathcal{SI}$ provided for any C, m, h, s, if $h \models \mathcal{SI}$ and $\mu\ C\ m(h,s) = (h_0, v)$ (and thus $\mu\ C\ m(h,s) \neq \bot$) then $h_0 \models \mathcal{SI}$.

Theorem 6.3 *If CT is a properly annotated class table then $[\![CT]\!]$ maintains $\mathcal{SI}$.*

The proof is by fixpoint induction,[5] using the following.

Lemma 6.4 (main lemma) *If CT is properly annotated and μ maintains $\mathcal{SI}$ then any constituent command S of a method in CT maintains $\mathcal{SI}$. That is, for all (h,s), if $h \models \mathcal{SI}$ and $(h_0, s_0) = [\![\Gamma \vdash S]\!]\mu(h,s)$ then $h_0 \models \mathcal{SI}$.*

The proof is by structural induction on S. The interesting cases are the primitive commands that can falsify $\mathcal{SI}$, by extending the range of quantifications ($\mathbf{new}$) or updating fields. Method call is easy because $\mathcal{SI}$ only involves heap. In this extended abstract we consider some key cases.

Lemma 6.5 ($\mathbf{new}$) *If $h \models \mathcal{SI}$ and $(h_0, s_0) = [\![\Gamma \vdash x := \mathbf{new}\ C]\!]\mu(h,s)$ then $h_0 \models \mathcal{SI}$.*

Suppose q is the fresh object, so that $h_0 = [h \mid\!\!- q \mapsto defaults]$. For (9): $h_0\ q.inv = \mathsf{ff}$ by definition. By $h \models \mathcal{SI}$, freshness, and admissibility, no $\mathcal{I}_C(p)$ can depend on q in (h,s) so adding q to the heap does not falsify (9) for existing objects. For (12): The new object q extends the range for o in (12), but the antecedent is false as q is fresh. It extends the range for p as well, but then $p.g = \mathbf{null}$ and o ranges over allocated, non-null locations. □

[5]It suffices that (a) the requisite property holds for every method environment in the approximation chain and (b) the property is preserved by lub (admissible for fixpoint induction). Both are straightforward.

Lemma 6.6 (pack) *Suppose $h \models \mathcal{SI}$ and $(h_0, s_0) = [\![\Gamma \vdash \mathbf{pack}\ E]\!]\mu(h,s)$. Then $h_0 \models \mathcal{SI}$ provided that the stipulated preconditions (Table 1) are satisfied, i.e.,*

- $q \neq \mathbf{null}$
- $h\,q.inv = \mathsf{ff}$
- $h \models \mathcal{I}_B(q)$
- $h \models (\forall p \bullet p.own = q \Rightarrow \neg p.comm \wedge p.inv\ )$

where $q = [\![\Gamma \vdash E : B]\!](h,s)$.

For (9): For any o, if $o \neq q$ then $\mathcal{I}(o)$ by $h \models \mathcal{SI}$, because $\mathbf{pack}$ only changes inv and $comm$ on which admissible invariants do not depend. If $o = q$ we have $\mathcal{I}_B(q)$ by precondition. For (10): We have $h_0 \models (\forall p \bullet p.own = q \Rightarrow p.comm\ )$ by semantics of $\mathbf{pack}$. For $o \neq q$, no owner fields are changed in h_0 nor is any $comm$ changed to false in h_0. For (11): If $h\,o.comm = \mathsf{ff}$ but $h_0\ o.comm = \mathtt{tt}$ then $h\,o.own = q$ by definition of h_0; and $h_0\ o.inv = \mathtt{tt}$ by hypothesis. For (12): This can be falsified by changing a pivot, but neither inv nor $comm$ can be pivots (as they have type $\mathbf{bool}$). It can also be falsified by updating $deps$ but $\mathbf{pack}$ does not modify $deps$. Finally, it can be falsified by setting inv, which is done here for q. So suppose $o\!:\!C$ is in $dom\ h$ and $(g\!:\!C) \in pivots\ B$ with $h_0\ q.g = o$; then we must show $q \in h_0\ o.deps$, and this follows by admissibility from $h \models \mathcal{I}(q)$ because $h_0\ o.deps = h\,o.deps$. □

Lemma 6.7 (field update) *Suppose $h \models \mathcal{SI}$ and $(h_0, s_0) = [\![\Gamma \vdash E.f := E']\!]\mu(h,s)$. Then $h_0 \models \mathcal{SI}$ provided that the stipulated preconditions are satisfied, i.e.,*

- *$q \neq \mathbf{null}$ and $h\,q.inv = \mathsf{ff}$,*
- *for all $p \in h\,q.deps$, if $f \in reads(type(p), B)$ then either $h\,p.inv = \mathsf{ff}$ or $h \models \mathcal{U}_{type(p),B,f}(p,q,v)$*

where $q = [\![\Gamma \vdash E : B]\!]\mu(h,s)$ and $v = [\![\Gamma \vdash E' : B]\!]$.

By semantics, $h_0 = [h \mid q.f \mapsto v]$. For (9): Suppose, for some o, D that $\mathcal{I}_D(o)$ depends on $q.f$ in (h,s). We must show that either $h_0\ o.inv = \mathsf{ff}$ or $h_0 \models \mathcal{I}_D(o)$. By admissibility of $\mathcal{I}_D$ it suffices to consider these cases:

- $q = o$ —Then $h\,o.inv = \mathsf{ff}$ by precondition.
- $o \preceq^h q$ —Then precondition $h\,q.inv = \mathsf{ff}$ implies $h\,q.comm = \mathsf{ff}$ by $h \models \mathcal{SI}$ (11) and then $h\,o.inv = \mathsf{ff}$ by transitive ownership Lemma 6.2. So $h_0\ o.inv = \mathsf{ff}$.
- $q = h\,o.g$ and $o \in h\,q.deps$ for some $g\!:\!B \in pivots\ D$ such that $f \in reads(D,B)$. (As we are considering ordinary field update, $f \not\equiv deps$.) Now by precondition we have either $h\,o.inv = \mathsf{ff}$, whence $h_0\,o.inv = \mathsf{ff}$ by definition of h_0, or else $h \models \mathcal{U}_{D,B,f}(o,q,v)$. In the latter case, by $h\,o.g = q \neq \mathbf{null}$ and $h \models \mathcal{I}_D(o)$ we can use the update guard obligation (8) to obtain $h_0 \models \mathcal{I}_D(o)$.

For (10) and (11): the relevant fields are not updated. For (12): This can be falsified in the update from h to h_0 only by f being a pivot, i.e., at instance $p, g := q, f$ of (12). But then precondition $h\, q.inv = \mathsf{ff}$ falsifies the antecedent.

7. Conclusions

We have formalized and shown soundness for the programming discipline of [7], built on [5], in which auxiliary fields in annotations express intended atomicity and encapsulation. The Main Lemma 6.4 and Theorem 6.3 justify appealing to system invariant $\mathcal{SI}$ where needed. Then $\mathcal{SI}(9)$ licenses asserting an object invariant $\mathcal{I}(o)$ where $o.inv$ holds and $\mathcal{I}$ is visible. As in Separation Logic, concepts like ownership are "in the eye of the asserter" [19].

In [19], which deals with ownership for a single-instance class and without reentrancy, a major result is that certain predicates in specifications need to be restricted to be "precise" in the sense that they uniquely determine a satisfying region of heap. Otherwise there is a problem akin to the problem of adaptation rules when auxiliary variables can have more than one satisfying instantiation. We plan to explore precision in connection with what is achieved by our use of auxiliaries own and $deps$. We also plan to check our soundness proof using an existing deep embedding of the semantics in the PVS prover. Finally, the discipline seems well suited for extension to concurrency, both in its use of auxiliary state and in the update guards which can be seen as a simple rely-guarantee interface.

During a presentation by Peter O'Hearn on a rule for monitors [August 2002], the first author was struck by the realization that such a rule was no less than a way to pick up a thread dropped in the early '70s —What are the structural constructs that correspond to commands the way modules correspond to lambda abstractions? The technical achievements and inspiring ideas of Reynolds, O'Hearn *et al.* are gratefully acknowledged.

References

[1] A. W. Appel. Foundational proof-carrying code. In *Proceedings of LICS*, 2001.

[2] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 2 edition, 1997.

[3] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. Journal version of [4], submitted. Available from http://www.cs.stevens-tech.edu/~naumann/oceri.ps, 2002.

[4] A. Banerjee and D. A. Naumann. Representation independence, confinement and access control. In *POPL*, pages 166–177, 2002.

[5] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. In S. Eisenbach, G. T. Leavens, P. Müller, A. Poetzsch-Heffter, and E. Poll, editors, *Formal Techniques for Java-like Programs 2003*, July 2003. Available as Technical Report 408, Department of Computer Science, ETH Zurich. A newer version of this paper is [6].

[6] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. Manuscript KRML 122b, Dec. 2003. Available from http://research.microsoft.com/~leino/papers.html.

[7] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. Submitted; available from http://www.cs.stevens-tech.edu/~naumann/friends.pdf, 2004.

[8] G. Bierman, M. Parkinson, and A. Pitts. An imperative core calculus for java and java with effects. Technical Report 563, University of Cambridge Computer Laboratory, Apr. 2003.

[9] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, 2003.

[10] D. Clarke. Object ownership and containment. Dissertation, Computer Science and Engineering, University of New South Wales, Australia, 2001.

[11] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In J. L. Knudsen, editor, *ECOOP 2001 - Object Oriented Programming*, 2001.

[12] B. Jacobs, J. Kiniry, and M. Warnier. Java program verification challenges. In F. de Boer, M. Bonsangue, and S. G. W.-P. de Roever, editors, *Formal Methods for Components and Objects (FMCO 2002)*, LNCS 2852, pages 202–219, 2003.

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

[14] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Proceedings, Fifteenth POPL*, pages 191–203, 1988.

[15] G. Morrisett, K. Crary, N. Glew, and D. Walker. From system F to typed assembly language. *ACM Trans. Prog. Lang. Syst.*, 21(3):528–569, 1999.

[16] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. Number 2262 in LNCS. Springer, 2002.

[17] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for object structures. Technical Report 424, ETH Zürich, Chair of Software Engineering, Oct. 2003.

[18] D. A. Naumann. Soundness of data refinement for a higher order imperative language. *Theoretical Comput. Sci.*, 278(1–2):271–301, 2002.

[19] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *POPL*, pages 268–280, 2004.

[20] P. W. O'Hearn and R. D. Tennent. *Algol-like Languages (Two volumes)*. Birkhäuser, Boston, 1997.

[21] B. Reus. Modular semantics and logics of classes. In *CSL*, 2003.

[22] B. Reus and T. Streicher. Semantics and logics of objects. In *LICS*, 2002.

[23] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.

8. Appendix

The appendix gives one more lemma and a bit more detail about predicates. Tables 6, 7, and 8 give typing and semantics cases omitted in the body of the paper.

Lemma 8.1 (detach) *Suppose* $h \models \mathcal{SI}$ *and* $(h_0, s_0) = [\![\Gamma, \mathsf{self} : B \vdash \mathbf{detach}\ E']\!]\mu(h, s)$. *Then* $h_0 \models \mathcal{SI}$ *provided that the stipulated preconditions are satisfied, i.e.,*

- $h\,p.inv = \mathsf{ff}$
- $q \neq \mathbf{null}$
- *either* $h\,q.inv = \mathsf{ff}$ *or* $h\,q.g \neq p$ *for all* $(g : B) \in$ *pivots* C

where $p = s(\mathsf{self})$ *and* $q = [\![\Gamma \vdash E' : C]\!]\mu(h, s)$.

Proof: By semantics, $h_0 = [h \mid p.deps \mapsto p.deps - \{q\}]$.

For (9): We consider cases on how $\mathcal{I}_D(o)$ could depend on $h\,p.deps$ for some D and $o \in dom\ h$.

- $o = p$ —but then $h\,o.inv = \mathsf{ff}$ by precondition.
- $o \preceq^h p$ —but then, by admissibility, $\mathcal{I}_D$ does not depend on $h\,p.deps$.
- $D \in friends\ B$, in which case $o \in h\,p.deps$ by $\mathcal{SI}$ (12). If $h_0 \not\models \mathcal{I}_D(o)$ then by admissibility of $\mathcal{I}_D$ and definition of h_0 we have $[h \mid p.deps \mapsto p.deps - \{q\} \cup \{o\}] \models \mathcal{I}_D(o)$. Thus $o = q$. Now by precondition, either $h\,q.inv = \mathsf{ff}$, falsifying the antecedent of (9), or $h\,q.g \neq p$ for all pivots g, in which case by admissibility $\mathcal{I}_D(o)$ does not depend on $h\,p.deps$.

For (10) and (11): the relevant fields are not updated.

For (12): This can only be falsified in h_0 for the instance $[o, p := p, q]$ of (12), that is: $h\,q.g = p \wedge h\,q.inv \Rightarrow q \in h\,p.deps$. But we have the precondition that either $h\,q.inv = \mathsf{ff}$ or $h\,q.g \neq p$ for all pivots g. $\square$

Details about predicates. Just as a formula over some variable x may be taken to over x, y, we need to adapt (semantic) predicates to different state spaces. In this paper our primary concern is with object invariants, that is, predicates $\mathcal{P} \subseteq [\![Heap \otimes (\mathsf{self} : C)]\!]$ for self, and their instantiations $\mathcal{P}(o)$ for particular locations o. Now $\mathcal{P}(o)$ projects to a predicate in $\mathbb{P}[\![Heap]\!]$ and lifts to other contexts. Negation is complement with respect to the appropriate state set and this determines the meaning of implication. To streamline the notation in this extended abstract we do not belabor the point, as there are no technical difficulties, but rather interpret the type of various $\mathcal{P}$ in a loose way.

For variables, we only need substitution in specific cases, e.g., to substitute a specific location self in an invariant. For this we use a streamlined notation. If $\mathcal{P} \subseteq [\![Heap \otimes (\mathsf{self} : C)]\!]$ and o is a location of type C then $\mathcal{P}(o)$ is the predicate (over $[\![Heap]\!]$ or $[\![Heap \otimes \Gamma]\!]$ for whatever Γ you like, as remarked above) defined as follows:

$$(h, s) \in \mathcal{P}(o) \iff o \in dom\ h \wedge (h, [s \mid \mathsf{self} \mapsto o]) \in \mathcal{P}$$

The condition $o \in dom\ h$ ensures that $[s \mid \mathsf{self} \mapsto o]$ is a closed store. For $\mathcal{P}$ that depends on a larger store, say $\mathsf{self} : C, x : T$, the definition of $\mathcal{P}(o, v)$ is similar.

The weakest precondition for a field assignment can be expressed as a kind of "substitution", though as an operation on formulas care must be taken due to sharing (e.g., [2]). We use a wiggly assignment symbol to remind that this is a semantic function —simply the inverse image of field update. For field updates, if v is value and o an object, we define $\mathcal{P}[o.f :\approx v]$ by

$$(h, s) \in \mathcal{P}[o.f :\approx v] \iff o \in dom\ h \wedge ([h \mid o \mapsto f]v, s) \in \mathcal{P}$$

$$\Gamma \vdash x : \Gamma x \qquad \Gamma \vdash \mathbf{true} : \mathbf{bool} \qquad \Gamma \vdash \mathbf{null} : C \qquad \frac{\Gamma \vdash E_0 : T_0 \qquad \Gamma \vdash E_1 : T_1}{\Gamma \vdash E_0 = E_1 : \mathbf{bool}}$$

$$\frac{\Gamma \vdash E : C \qquad (f : T) \in \mathit{fields}\, C}{\Gamma \vdash E.f : T} \qquad \frac{\Gamma \vdash E : T \qquad T = \Gamma\, x \qquad x \neq \mathsf{self}}{\Gamma \vdash x := E}$$

$$\frac{\Gamma \vdash E : C \qquad \mathit{mtype}(m, C) = \bar{x} : \bar{T} \to T \qquad T = \Gamma\, x \qquad \Gamma \vdash \bar{E} : \bar{T} \qquad x \neq \mathsf{self}}{\Gamma \vdash x := E.m(\bar{E})}$$

$$\frac{\Gamma \vdash E : \mathbf{bool} \qquad \Gamma \vdash S_0 \qquad \Gamma \vdash S_1}{\Gamma \vdash \mathbf{if}\ E\ \mathbf{then}\ S_0\ \mathbf{else}\ S_1} \qquad \frac{\Gamma \vdash E : T \qquad x \neq \mathsf{self} \qquad (\Gamma, x : T) \vdash S}{\Gamma \vdash \mathbf{var}\ x : T := E\ \mathbf{in}\ S}$$

$$\frac{\Gamma \vdash S_0 \qquad \Gamma \vdash S_1}{\Gamma \vdash S_0;\ S_1}$$

Table 6. Additional typing rules for program expressions and commands.

$$\begin{array}{lcl}
[\![\Gamma \vdash x : T]\!](h, s) & = & s\, x \\
[\![\Gamma \vdash \mathbf{null} : C]\!](h, s) & = & \mathsf{nil} \\
[\![\Gamma \vdash E_0 = E_1 : \mathbf{bool}]\!](h, s) & = & \mathsf{let}\ v_0 = [\![\Gamma \vdash E_0 : T_0]\!](h, s)\ \mathsf{in} \\
 & & \mathsf{let}\ v_1 = [\![\Gamma \vdash E_1 : T_1]\!](h, s)\ \mathsf{in\ if}\ v_0 = v_1\ \mathsf{then\ tt\ else\ ff} \\
[\![\Gamma \vdash E.f : T]\!](h, s) & = & \mathsf{let}\ o = [\![\Gamma \vdash E : C]\!](h, s)\ \mathsf{in\ if}\ o = \mathsf{nil}\ \mathsf{then}\ \bot\ \mathsf{else}\ h\, o.f
\end{array}$$

Table 7. Semantics of expressions.

$$\begin{array}{lcl}
[\![\Gamma \vdash x := E]\!]\mu(h, s) & = & \mathsf{let}\ v = [\![\Gamma \vdash E : T]\!](h, s)\ \mathsf{in}\ (h, [s \mid x \mapsto v]) \\
[\![\Gamma \vdash S_0;\ S_1]\!]\mu(h, s) & = & \mathsf{let}\ (h_1, s_1) = [\![\Gamma \vdash S_0]\!]\mu(h, s)\ \mathsf{in}\ [\![\Gamma \vdash S_1]\!]\mu(h_1, s_1) \\
[\![\Gamma \vdash \mathbf{if}\ E\ \mathbf{then}\ S_0\ \mathbf{else}\ S_1]\!]\mu(h, s) & = & \mathsf{let}\ v = [\![\Gamma \vdash E : \mathbf{bool}]\!](h, s)\ \mathsf{in} \\
 & & \mathsf{if}\ v\ \mathsf{then}\ [\![\Gamma \vdash S_0]\!]\mu(h, s)\ \mathsf{else}\ [\![\Gamma \vdash S_1]\!]\mu(h, s) \\
[\![\Gamma \vdash \mathbf{var}\ x : T := E\ \mathbf{in}\ S]\!]\mu(h, s) & = & \mathsf{let}\ v = [\![\Gamma \vdash E : T]\!](h, s)\ \mathsf{in} \\
 & & \mathsf{let}\ s_1 = [s + x \mapsto v]\ \mathsf{in\ let}\ (h_1, s_2) = [\![(\Gamma, x : T) \vdash S]\!]\mu(h, s_1)\ \mathsf{in}\ (h_1, (s_2 \lfloor x)) \\
[\![\Gamma \vdash x := E.m(\bar{E})]\!]\mu(h, s) & = & \mathsf{let}\ q = [\![\Gamma \vdash E : C]\!](h, s)\ \mathsf{in} \\
 & & \mathsf{if}\ q = \mathsf{nil}\ \mathsf{then}\ \bot\ \mathsf{else\ let}\ \bar{x} : \bar{T} \to T = \mathit{mtype}(m, C)\ \mathsf{in} \\
 & & \mathsf{let}\ \bar{v} = [\![\Gamma \vdash \bar{E} : \bar{T}]\!](h, s)\ \mathsf{in\ let}\ s_1 = [\bar{x} \mapsto \bar{v}, \mathsf{self} \mapsto q]\ \mathsf{in} \\
 & & \mathsf{let}\ (h_0, v_0) = \mu\ C\ m(h, s_1)\ \mathsf{in}\ (h_0, [s \mid x \mapsto v_0])
\end{array}$$

Table 8. Semantics of additional commands.