Towards Imperative Modules: Reasoning about Invariants and Sharing of Mutable State

David A. Naumann [a,1]

[a] *Stevens Institute of Technology*
Castle Point on Hudson, Hoboken, NJ 07030 USA

Mike Barnett [b]

[b] *Microsoft Research*
Redmond, WA 98052 USA

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 friendship, 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. As an example the methodology is used to specify iterators, which are problematic for previous ownership systems.

Key words: program verification, data abstraction, alias control, object invariants

1 Introduction

This paper is concerned with modular reasoning about pointer-based data structures, especially in object-oriented programming languages. Modular expression, reasoning, and reuse are well supported in functional programming

[1] Partially supported by the National Science Foundation (CCR-0208984, CCR-ITR-0326540, CCF-0429894), the Office of Naval Research (N00014-01-1-0837), and Microsoft Research, Redmond, WA

languages, by constructs based on logical quantification and higher order functional abstraction. For example, lexical scope is a highly effective means of information hiding. The addition of mutable state can subvert encapsulation owing to various kinds of aliasing. The thorniest problems for higher order imperative programs —due to interaction between local variables and nested procedures (Meyer and Sieber, 1988; O'Hearn and Tennent, 1997)— are precluded in widely used imperative and object-oriented languages owing to restrictions on non-local references and/or nesting (Naumann, 2002; Banerjee and Naumann, 2005a). But these restrictions on procedural abstraction force the use of heap structure to encode higher level patterns. Performance considerations also necessitate some uses of shared, dynamically allocated mutable objects. In common practice, pointer structures are widely used both for internal representations and for interfaces between components. This is especially true in object-oriented programming; indeed, 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 (good surveys can be found in the dissertations of Clarke (2001) and Müller (2002)). 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 assertions in 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 (Morrisett et al., 1999; Appel, 2001). Ownership type systems (Clarke et al., 2001; Müller, 2002) and Separation Logic (O'Hearn et al., 2004) 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, another challenge in 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 object-oriented programs, to which we confine our attention in this paper.

The discipline formalized in this paper protects invariants using ownership, expressing ownership not in terms of types (Clarke et al., 2001; Boyapati et al., 2003) or logical connectives (Reynolds, 2002; O'Hearn et al., 2004) but rather auxiliary state. 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. The class construct is the primary unit of scope. Module constructs typically enclose multiple classes but often the most useful unit of granularity for reasoning is a small number of interconnected *instances* of one or more classes. As an example, Sect. 10 considers the iterator pattern (Gamma et al., 1995) in which each instance of the *Collection* class is associated with multiple instances of *Enumerator* with which it shares access to an encapsulated data structure.

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 Sect. 3 we show how the approach has been realized in a verification discipline developed incrementally in several papers: the "Boogie methodology" of Barnett et al. (2004) and Leino and Müller (2004) together with the "friendship system" of Barnett and Naumann (2004). The reader is encouraged to consult these papers for an expository introduction and for informal soundness arguments. The discipline does not require nonstandard 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, using per-instance invariants, which lessens the need for reachability or other inductive predicates on the heap. An example is given in Sect. 4 and admissibility conditions on invariants are explained in Sect. 5 which concludes the informal review of the discipline.

The technical contributions of this paper are to simplify and extend the discipline presented in the cited works and to give the first proof of soundness of the discipline (extended or not) in terms of a standard semantic model. Sect. 6 formalizes a class-based object-oriented language and its denotational semantics. Sect. 7 uses the semantics to formalize the notion of proper annotation. Sect. 8 proves the main result, that the program invariants of Sect. 3 are maintained in a properly annotated program. Sect. 9 addresses subtyping. Sect. 10 is devoted to the iterator example, which motivates a refinement of the discipline that is given in Sect. 11. Sect. 12 discusses related work and Sect. 13 concludes.

Our formalization treats predicates semantically, to avoid commitment to a language of formulas. In particular, we avoid commitment to a particular treatment of partial terms like field dereferences; the informal examples are written with guarded formulas like $x \neq \mathsf{null} \Rightarrow x.f > 0$. The results are useful

both for verification systems based on weakest-precondition semantics, like ESC/Java (Leino and Nelson, 2002) and the Spec# project (Barnett et al., 2005), and for those like the LOOP project (Jacobs et al., 2003) which treat program logic as derived rules for reasoning directly in terms of semantics.

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:

$$\frac{\{\mathcal{P} \wedge \mathcal{I}\}\ \mathit{body}\ \{\mathcal{Q} \wedge \mathcal{I}\}}{\{\mathcal{P}\}\ \mathbf{call}\ m\ \{\mathcal{Q}\}} \tag{1}$$

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.

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

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 made precise in terms of disjointness of identifiers. With aliasing it can be extremely difficult to express; we return to this issue later.

The benefit of rule (2), which has specifications symmetric to those in (1), is to undo the apparent unsoundness of (1). An explanation can be given in terms of proofs. 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, call it 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

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

which is allowed by an ordinary procedure call rule. Instances of the rules for sequence, iteration, and conditional are still valid when $\mathcal{I}$ is conjoined everywhere. Not so the assignment axiom —thus τ' is not quite a proof tree.

However, 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 $\mathcal{I}$ following instances of the assignment axiom (and following introduction rules for other primitives such as field assignment). So τ' can be transformed to a valid proof of the triple $\{\mathcal{P}_{main} \wedge \mathcal{I}\}\ S_{main}\ \{\mathcal{Q}_{main} \wedge \mathcal{I}\}$. Assuming that $\mathcal{I}$ holds initially, so that $\mathcal{P}_{main} \Rightarrow \mathcal{P}_{main} \wedge \mathcal{I}$, the rule of consequence can be used to obtain $\{\mathcal{P}_{main}\}\ S_{main}\ \{\mathcal{Q}_{main}\}$. Thus we have transformed a dubious proof of the latter triple into one that is clearly sound.

As an example, consider the following. The root node is an instance of the sequence rule. The left subtree is elided except for the use of rule (1). The right subtree uses the assignment axiom and consequence rule.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \{\mathcal{P} \wedge \mathcal{I}\}\ body\ \{\mathcal{R} \wedge \mathcal{I}\}\end{array}}{\{\mathcal{P}\}\ \mathbf{call}\ m\ \{\mathcal{R}\}} \qquad \dfrac{\mathcal{R} \Rightarrow \mathcal{Q}[1/x] \qquad \{\mathcal{Q}[1/x]\}\ x := 1\ \{\mathcal{Q}\}}{\{\mathcal{R}\}\ x := 1\ \{\mathcal{Q}\}}}{\{\mathcal{P}\}\ m; x := 1\ \{\mathcal{Q}\}}$$

Performing the transformation yields the following, where the elided subtree proving the body of m is unchanged.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \{\mathcal{P} \wedge \mathcal{I}\}\ body\ \{\mathcal{R} \wedge \mathcal{I}\}\end{array}}{\{\mathcal{P} \wedge \mathcal{I}\}\mathbf{call}\ m\{\mathcal{R} \wedge \mathcal{I}\}} \qquad \dfrac{\mathcal{R} \Rightarrow \mathcal{Q}[1/x] \qquad \dfrac{\{\mathcal{Q}[1/x]\}\ x := 1\ \{\mathcal{Q}\}}{\{\mathcal{Q}[1/x] \wedge \mathcal{I}\}\ x := 1\ \{\mathcal{Q} \wedge \mathcal{I}\}}}{\{\mathcal{R} \wedge \mathcal{I}\}\ x := 1\ \{\mathcal{Q} \wedge \mathcal{I}\}}}{\{\mathcal{P} \wedge \mathcal{I}\}\ \mathbf{call}\ m;\ x := 1\ \{\mathcal{Q} \wedge \mathcal{I}\}}$$

Note the instance of (2) added at the upper right. (Strictly speaking, the side condition for this use of the consequence rule is now $\mathcal{R} \wedge \mathcal{I} \Rightarrow \mathcal{Q}[1/x] \wedge \mathcal{I}$.)

This story gets subverted in programs using shared mutable objects.

The first problem is that free use of pointers makes it difficult to reason about, or even to define, the separation needed for the condition in rule (2). The solution studied in this paper involves restricting the part of the heap on which $\mathcal{I}$ depends. Heap partitioning is made explicit as a logical connective in Separation Logic (Reynolds, 2002): the separating conjunction $*$ is used in the *frame rule*

$$\frac{\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}}{\{\mathcal{P} * \mathcal{I}\}\ S\ \{\mathcal{Q} * \mathcal{I}\}} \tag{3}$$

which plays the role of (2) but needs no side condition. The antecedent is interpreted as saying that S only acts on the part of the heap supporting truth of $\mathcal{P}$ and $\mathcal{Q}$, and $\mathcal{P} * \mathcal{I}$ means that $\mathcal{I}$ is supported by a separate part of the heap. There is a second-order frame rule that embodies reasoning similar to our reasoning above to justify (1). The rule is very elegant:

$$\frac{\{\mathcal{P}\}\ \mathbf{call}\ m\ \{\mathcal{Q}\} \vdash \{\mathcal{P}'\}\ S\ \{\mathcal{Q}'\}}{\{\mathcal{P} * \mathcal{I}\}\ \mathit{body}\ \{\mathcal{Q} * \mathcal{I}\} \vdash \{\mathcal{P}' * \mathcal{I}\}\ S\ \{\mathcal{Q}' * \mathcal{I}\}} \tag{4}$$

But it has tricky interactions with the rule of conjunction (O'Hearn et al., 2004; Birkedal et al., 2005) and it relies on a non-standard assertion language. A similar effect is achieved by the solution studied here which uses a notion of ownership (Müller, 2002; Banerjee and Naumann, 2005a; Boyapati et al., 2003). Writing $\mathcal{I}(o)$ to make explicit the object for which an invariant is considered, objects on which $\mathcal{I}(o)$ depends are designated as "owned by o" and the condition in (2) becomes, roughly: "S does not update owned objects".

The second problem with our story is that calls of m are treated 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 callback*. 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. Callbacks are frequently used in object-oriented programs.

The last problem we address is the *sharing of mutable state*. Rule (2) deals with separation of state (as does (3)), i.e., the absence of relevant sharing: the part of the heap on which S acts is disjoint from the part on which $\mathcal{I}$ depends. When applicable, separation is very powerful. But what about a situation where sharing is needed? Many important design patterns involve a configuration of several interlinked objects that cooperate in a controlled way, e.g., the iterators associated with a collection share access to its underlying data structure.

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, e.g., because there is more than one dependent o. Thus $\mathcal{I}(o)$ is at risk from updates of $p.f$. For reasoning as in (2) we can use neither syntactic disjointness nor heap separation to express the absence of interference.

Suppose p cooperates by only increasing the value of f. This does not falsify $\mathcal{I}(o)$, as can be expressed by the triple $\{\mathcal{U}(p,y) \wedge \mathcal{I}\}\ p.f := y\ \{\mathcal{I}\}$ where the *update guard* $\mathcal{U}(p,y)$ is defined to be $p.f \leq y$. This suggests the following rule; it is similar to (2), using the triple to express absence of interference, but different in that precondition $\mathcal{U}$ appears in the conclusion.

$$\frac{\{\mathcal{P}\}\ E.f := E'\ \{\mathcal{Q}\} \qquad \{\mathcal{U}(E,E') \wedge \mathcal{I}\}\ E.f := E'\ \{\mathcal{I}\}}{\{\mathcal{P} \wedge \mathcal{U}(E,E') \wedge \mathcal{I}\}\ E.f := E'\ \{\mathcal{Q} \wedge \mathcal{I}\}} \tag{5}$$

The rule is sound; it is an instance of the standard rule of conjunction. Here is a reformulation of (5) to highlight the intended separation of concerns.

$$\frac{\begin{array}{c}\{\mathcal{P} \wedge \mathcal{U}(E,E')\}\ E.f := E'\ \{\mathcal{Q}\} \\ \{\mathcal{U}(x,y) \wedge \mathcal{I}\}\ x.f := y\ \{\mathcal{I}\} \quad \text{for fresh } x,y\end{array}}{\{\mathcal{P} \wedge \mathcal{U}(E,E') \wedge \mathcal{I}\}\ E.f := E'\ \{\mathcal{Q} \wedge \mathcal{I}\}} \tag{6}$$

The idea is that $\{\mathcal{P}\}\ E.f := E'\ \{\mathcal{Q}\}$ specifies the assignment in the code performing the update of $p.f$; that code is typically in the class of p, which *grants* to its *friend*, o, permission for $\mathcal{I}(o)$ to depend on $p.f$. Moreover, the class of friend o includes in its interface the triple

$$\{\mathcal{U}(x,y) \wedge \mathcal{I}\}\ x.f := y\ \{\mathcal{I}\} \tag{7}$$

where x, y are fresh variables used to express the commitment [2] that update of field f of an object x does not interfere with $\mathcal{I}$ under precondition $\mathcal{U}(x,y)$. The rules impose precondition $\mathcal{U}(E,E')$ on the update code, whereas (7) is a proof obligation on the friend class.

An obvious choice for $\mathcal{U}$ is $wp(x.f := y)(\mathcal{I})$, but this could expose internals on which $\mathcal{I}$ depends. The rules allow $\mathcal{U}$ to be expressed in some way that hides information. (As stated, rules (5–7) directly expose $\mathcal{I}$ but the idea is to hide it as in (2).)

[2] The commitment (7) can be extended with an additional postcondition that reveals to the granter something about state as viewed by the friend, as detailed in Barnett and Naumann (2004); soundness for that is a straightforward extension of the results here so we omit it.

In the sequel, we study a discipline whereby the preceding forms of reasoning can be realized for object-oriented programs using assertions, regardless of whether reasoning is formulated in terms of ordinary rules, proof outlines, weakest preconditions, or otherwise. The exposition in this section considers a single invariant $\mathcal{I}$ but for modular reasoning about object-oriented programs one normally considers object invariants, i.e., specification of a class C involves an invariant $\mathcal{I}_C(o)$ that pertains to a single instance o. This generalization is achieved using a single program invariant $\mathcal{PI}$ that quantifies over all allocated objects. The discipline ensures that $\{\mathcal{PI}\}\ S\ \{\mathcal{PI}\}$ for all S. Thus $\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$ is a consequence of $\{\mathcal{PI} \wedge \mathcal{P}\}\ S\ \{\mathcal{Q}\}$. This is our replacement for rules (1) and (2).

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

The discipline studied in this paper imposes restrictions on object invariants and requires certain preconditions for field updates, all expressed in terms of auxiliary fields that encode potential dependence and thereby circumscribe possible interference. This section is a condensed review of the discipline, which was introduced in Barnett et al. (2004) and Barnett and Naumann (2004) (see also Leino and Müller (2004)).

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 (Müller et al., 2003) and has been advocated in the literature (Liskov and Guttag, 1986; Meyer, 1997). But intermediate states are not observable in the sense of pre/post specifications. Although sound, and useful in cases where a callback is intended, this approach is too restrictive in general since in many cases no callback is intended and an outgoing call is made when the invariant does not hold.

The discipline that we study (Barnett et al., 2004) avoids exposing details about internal state by introducing a public boolean field, inv, to indicate whether an object's invariant holds. It is present in all objects (as if declared in class Object). 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:

$$(\forall o \bullet o.\mathsf{inv} \Rightarrow \mathcal{I}_{type(o)}(o)) \tag{8}$$

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 the formalization we do not include subclassing). By (8), if a method specification has $\mathsf{self}.\mathsf{inv}$ as precondition then $\mathcal{I}(\mathsf{self})$ can be assumed for verification of its implementation. (We use self to refer to the receiver object that is the implicit parameter of an instance method.)

The field inv is an auxiliary field, 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 subject to special rules. The reasoner is free to decide where inv does and does not hold, by choosing where to use the special statements. The rule for $E.f := E'$ has as precondition $\neg E.\mathsf{inv}$ which ensures that an update does not violate (8); we add further preconditions in the sequel. The special statement [3] "**pack** E" sets $E.\mathsf{inv}$ true, under precondition $\mathcal{I}(E)$; setting inv false is the purpose of **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 *program invariants* like (8) which can be exploited wherever they are needed in verification. (Subject to visibility rules for $\mathcal{I}$ —this is a key practical benefit of the discipline but we do not dwell on it in this paper.)

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. (Admissibility conditions for invariants are discussed in detail in Sect. 5 and formalized in Sect. 7.) The auxiliary field own holds a reference to an object's owner and is null if there is no owner. The auxiliary field com 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 **set-own** E **to** E' has the effect $E.\mathsf{own} := E'$. 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., $\neg E.\mathsf{inv}$, as with ordinary field update. The stipulated preconditions are summarized in Table 1 and explained incrementally in the following pages.

[3] The terms "pack" and "unpack" allude to constructs associated with existential types for data abstraction (Mitchell and Plotkin, 1988). But here the encapsulation boundary is based on assertions rather than syntactic scope.

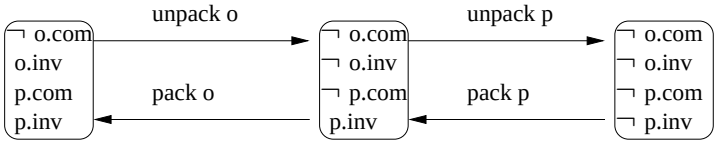
assert $E \neq \mathsf{null} \wedge \neg E.\mathsf{inv} \wedge \mathcal{I}_B(E) \wedge (\forall p \bullet p.\mathsf{own} = E \Rightarrow \neg p.\mathsf{com} \wedge p.\mathsf{inv})$; **pack** E;
assert $E \neq \mathsf{null} \wedge E.\mathsf{inv} \wedge \neg E.\mathsf{com}$; **unpack** E;
assert $\neg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

Table 1
Stipulated preconditions. Each precondition is written as an **assert** statement; its associated command is on the following line. Static types are assumed to be $E : B$, $E' : C$, $E'' : T$, and $\mathsf{self} : B$.

The statement **pack** E has the effect of setting $E.\mathsf{inv}$ true and also setting $o.\mathsf{com}$ true for all o owned by E. (Table 5 gives the formal definition.) In order to maintain program invariant (8), the stipulated precondition includes that $\mathcal{I}(E)$ holds. The other preconditions involve features explained in the sequel.

Statement **unpack** E has precondition $\neg E.\mathsf{com}$, that is, only an object's owner is allowed to unpack it. Besides setting $E.\mathsf{inv}$ false, this statement sets $o.\mathsf{com}$ false for all o with $o.\mathsf{own} = E$, thus making owned objects available for unpacking.

The pack and unpack statements effectively achieve a hierarchical notion of ownership and $\mathcal{I}(o)$ is allowed to depend on objects transitively owned by o. For o that directly owns p, this state diagram illustrates the protocol enforced by the preconditions:



The discipline yields program invariants (14) and (15), included in the summary at the end of this section, which reflect this protocol. As a consequence of

the invariants, the precondition $\neg E.\mathsf{inv}$ for field update means that an object cannot be updated unless its owner is unpacked.

It is perhaps surprising that the discipline achieves the modular reasoning offered by program invariant (8) even when all fields are considered public. But public visibility for fields is best avoided for various reasons. In particular, the precondition $\neg E.\mathsf{inv}$ must be imposed wherever a field of E is updated; to repack E, its invariant must be reestablished, which is best done in code of its class.

Friendship. The discipline of Barnett and Naumann (2004) extends Barnett et al. (2004) to handle cooperative sharing. One last auxiliary field, deps, is introduced. Just as the condition $o = p.\mathsf{own}$ licenses $\mathcal{I}(o)$ to depend on fields of p and formalizes that this dependence is safe because p is owned by o, the friendship discipline uses condition $o \in p.\mathsf{deps}$ to license $\mathcal{I}(o)$ to depend on certain fields of p in virtue of an explicit peer relationship. (The admissibility conditions for $\mathcal{I}$, involving both ownership and friendship, are discussed in detail in Sect. 5.)

The field deps is manipulated by two special statements: **attach** E adds the value of E to self.deps and **detach** E removes it. (Table 5 gives the formal semantics.)

Three special declarations are also used. First, a class G, called the *granter*, may have a sequence of friend declarations [4]

$$\textbf{friend } C \textbf{ reads } \bar{f} \tag{9}$$

where each field name f in the list $\bar{f}$ is either declared in G or is the auxiliary own (in Sect. 11 we also allow inv here). This declaration grants to C permission for its invariant to depend on fields $\bar{f}$ of a G-object (which is typically referenced from a designated *pivot field* of C, as discussed later). The declaration triggers a proof obligation in class G: Updates to any f in $\bar{f}$ are subject to a precondition, similar to $\mathcal{U}$ in rules (5) and (6). The details are discussed later.

The set of names C for which G has a friend declaration is written *friends* G. Moreover, *reads* gives the fields that a friend reads from a granter: $\mathit{reads}(C, G) = \bar{f}$ for the declaration above and $\mathit{reads}(C, G) = \varnothing$ if $C \notin \mathit{friends}\ G$.

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

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

$$\textbf{invariant } \mathcal{I}_C \tag{10}$$

where $\mathcal{I}_C$ is a predicate on $\mathsf{self} : C$ (and on the heap of course). To accomodate dynamic allocation of unboundedly many instances of C, each of which could potentially depend on a given p of type G, the auxiliary field $p.\mathsf{deps}$ holds the set of locations of friends o of type C that may depend on p. Definition 7 in the sequel requires that $\mathcal{I}_C(o)$ depends on $p.f$ only if either p is o, p is transitively owned by o, or o is in $p.\mathsf{deps}$.

The third special declaration is the *update guard*, which expresses the commitment (7). In any class C, and for any G, f with $f \in \mathit{reads}(C, G)$, there should be at least one declaration

$$\textbf{guard } \mathsf{piv}.f := \mathsf{val} \textbf{ by } \mathcal{U}_{C,G,f}(\mathsf{self}, \mathsf{piv}, \mathsf{val}) \tag{11}$$

The idea is that the invariant, $\mathcal{I}_C(o)$, of some friend o may depend on some object p of type G, and this object should only update its field $p.f$ if condition $\mathcal{U}_{C,G,f}$ holds. This condition must suffice to ensure that this update does not falsify the friend's invariant, $\mathcal{I}_C(o)$. Here the predicate [5] $\mathcal{U}_{C,G,f}(\mathsf{self}, \mathsf{piv}, \mathsf{val})$ is over $\mathsf{self} : C, \mathsf{piv} : G, \mathsf{val} : T$, where T is the type of f in G. The special variable piv is just notation by which the declaration, in the context of class C, refers to the granter p (which may or may not be expressible in terms of some pivot field). And special variable val is just notation to refer to the value assigned to $p.f$. A sensible default for $\mathcal{U}_{C,G,f}$ is *false*.

In general, the condition $\mathcal{U}_{C,G,f}$ involves the friend, the granter, and the new value. These are made explicit in the notation $\mathcal{U}_{C,G,f}(\mathsf{self}, \mathsf{piv}, \mathsf{val})$ to cater for their instantiation in the proof obligations for the granter and friend classes. Besides writing explicit parameters, as we have also done with $\mathcal{I}(o)$, we shall also need the semantic counterpart of substitution. For predicate $\mathcal{P}$, the notation $\mathcal{P}[E.f :\approx E']$ denotes the inverse image (weakest precondition) of field update $E.f := E'$. [6]

[5] The notation $\mathcal{U}_{C,G,f}$ 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 G with respect to the invariant of C. There may be more than one update guard for given C, G, f, offering alternatives for reasoning at different update sites, but we omit this complication in the notation.
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 special proof obligation is not for assignment to a but only field update $a.f := \ldots$.

[6] On formulas, this can be defined using substitution together with conditions to

Proof obligations. An update guard declaration (11) generates a proof obligation to be discharged in the context of the friend class C. This obligation corresponds to (7) in Sect. 2, but with a quantification over potential dependees.

$$(\,\forall\, o : G \bullet \mathsf{self} \in o.\mathsf{deps} \wedge \mathcal{I}_C \wedge \mathcal{U}_{C,G,f}(\mathsf{self}, o, \mathsf{val}) \Rightarrow \mathcal{I}_C[o.f :\approx \mathsf{val}]\,) \quad (12)$$

Typically, $\mathcal{I}_C$ only depends on a granter referenced by a pivot field g, and the invariant includes the condition $o \neq \mathsf{null} \wedge \mathsf{self} \in o.\mathsf{deps} \Rightarrow o = \mathsf{self}.g$ for reasons discussed later in connection with admissibility of invariants. Then (12) reduces to

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

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

Pre- and post-conditions in method specifications may mention any of the special fields $\mathsf{inv}, \mathsf{com}, \mathsf{own}, \mathsf{deps}$, as can intermediate assertions. 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. In particular, the precondition for field update includes the update guard — that is the obligation triggered by declaration (9). Generalizing from rules (5) and (6), and taking ownership into account, guard $\mathcal{U}$ is asserted for all packed friends p (i.e., with $p.\mathsf{inv}$). There may be several classes C in *friends* G that read f, and each one's guard is needed as precondition to update f.

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

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

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

Friendship is a little complicated, so let us review the two roles. The granter class G declares that friend class C may read field f, or rather that the invariant $\mathcal{I}_C$ may depend on $p.f$ for some instances p of G. For each such p, updates of $p.f$ are subject to a precondition for each o in $p.\mathsf{deps}$. That precondition is

take aliasing into account (de Boer, 1999; Apt and Olderog, 1997).

given by an update guard declaration in C which also imposes a proof obligation in C. An admissibility condition on $\mathcal{I}_C$, discussed in Sect. 5, ensures that if $\mathcal{I}_C(o)$ depends on a particular $p.f$ then o is in $p.\mathsf{deps}$.

What is achieved: a program invariant. The primary benefit of the discipline is that an object invariant $\mathcal{I}(o)$ holds at any control point in the program where $o.\mathsf{inv}$ holds, as formalized in (8) which is repeated as (13) in the following. (The informal term *control point* means place in the program text where an assertion could appear, corresponding to an intermediate state in a small-step semantics.)

Definition 1 Predicate $\mathcal{PI}$ is the conjunction of the following.

$$(\forall\, o \bullet o.\mathsf{inv} \Rightarrow \mathcal{I}_{type(o)}(o)\ ) \tag{13}$$

$$(\forall\, o \bullet o.\mathsf{inv} \Rightarrow (\forall\, p \bullet p.\mathsf{own} = o \Rightarrow p.\mathsf{com}\ )) \tag{14}$$

$$(\forall\, o \bullet o.\mathsf{com} \Rightarrow o.\mathsf{inv}\ ) \tag{15}$$

The main result of the paper is that, for any properly annotated program, $\mathcal{PI}$ is a *program invariant*, that is, it holds at every control point in the program. For a program to be properly annotated means that it includes assertions with the stipulated precondions (Table 1), it satisfies the update guard obligation (12), and the declared invariants $\mathcal{I}_C$ are admissible. Admissibility is the topic of Sect. 5 and proper annotation is formalized in Sect. 7.

4 Clock example

We give here a simple example of friendship, from Barnett and Naumann (2004); the iterator example is developed in Sect. 10. More extensive examples of friendship and ownership can be found in Barnett and Naumann (2004); Barnett et al. (2004); Leino and Müller (2004); Naumann (2005a).

In Fig. 1, the invariant of class $Master$ refers only to a field of $Master$; method $Tick$ exhibits the usual pattern for updating a field. The object invariant of class $Clock$ depends on field $time$ in an associated master clock m; many clocks may share a master and the master is owned by none. So $Clock$ declares a guard condition under which $m.time$ can be updated: a master may increase time, and this condition can be established in the context of class $Master$ without exposing field t that could well be private to $Clock$. (Throughout the paper, the prefix "self." may be omitted in field references, e.g, unqualified inv is short for self.inv. The long form is used sometimes, for clarity.)

```
class Master {
  time : int;
  invariant 0 ≤ self.time;
  friend Clock reads time;
  Master()   // constructor; initially ¬inv ∧ ¬com
    ensures inv ∧ ¬com;
  { time := 0; pack self; }
  Tick(n : int)
    requires inv ∧ ¬com ∧ 0 ≤ n;
    ensures time ≥ old(time);
  { unpack self; time := time + n; pack self; }
  Connect(c : Clock)
    requires inv;
    ensures c ∈ self.deps;
  { unpack self; attach c; pack self; }
}

class Clock {
  t : int;
  m : Master;
  invariant (self.m ≠ null ∨ self ∈ self.m.deps) ∧ 0 ≤ self.t ≤ self.m.time;
  guard piv.time := val by piv.time ≤ val;
  Clock(mast : Master)
    requires mast ≠ null ∧ mast.inv;
    ensures inv ∧ ¬com;
  { m := mast; t := 0; m.Connect(self); pack self; self.Sync(); }
  Sync()
    requires inv ∧ ¬com;
    ensures t = m.time;
  { unpack self; t := m.time; pack self; }
}
```

Fig. 1. Clocks. Synchronization with a master clock. $Inv_{Clock}(\mathsf{self})$ depends on $\mathsf{self}.m.time$ but does not own $\mathsf{self}.m$.

The invariant of *Clock* has a friend dependence only on pivot field m. The update guard obligation is

$$(\,\forall\, o : Master \bullet \mathsf{self} \in o.\mathsf{deps} \wedge \mathcal{I}_{Clock} \wedge o.time \leq \mathsf{val} \Rightarrow \mathcal{I}_{Clock}[o.time :\approx \mathsf{val}]\,)$$

This can be simplified by splitting off the case $o = m$. We get the conjunction of

$$(\,\forall\, o \bullet o \neq m \wedge \mathsf{self} \in o.\mathsf{deps} \wedge \mathcal{I}_{Clock} \wedge o.time \leq \mathsf{val} \Rightarrow \mathcal{I}_{Clock}[o.time :\approx \mathsf{val}]\,)$$

and

$$\mathsf{self} \in m.\mathsf{deps} \wedge \mathcal{I}_{Clock} \wedge m.time \leq \mathsf{val} \Rightarrow \mathcal{I}_{Clock}[m.time :\approx \mathsf{val}]$$

The first is true because $\mathcal{I}_{Clock}[o.time :\approx \mathsf{val}]$ is just $\mathcal{I}_{Clock}$ for $o \neq m$. The second formula is true because $\mathcal{I}_{Clock}$ depends monotonically on $m.time$. This case split can be avoided by strengthening the invariant to include $(\forall p \bullet \mathsf{self} \in p.\mathsf{deps} \Rightarrow p = m)$; we return to such "delimiting invariants" later.

The example is atypical in that $Master$ does not maintain explicit information about its deps. A variation in Barnett and Naumann (2004) adds a method for resetting the master clock: In order to avoid resetting when there active friends, $Connect$ increments a field $clocks$ used in invariant $clocks = size(\mathsf{deps})$.

In Barnett and Naumann (2004) the Subject/View pattern (Gamma et al., 1995) serves as another 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 $(\forall p \bullet p \in \mathsf{self}.\mathsf{deps} \Rightarrow p \in vs)$ puts it in a position to establish the precondition for updating its fields on which Views depend.

5 Admissible formulas

The discipline consists of the annotation regime (update guard obligation plus stipulated preconditions) and the requirement that object invariants satisfy an admissibility condition. The basic idea is that $\mathcal{I}_C$ is admissible provided that whenever $\mathcal{I}_C(o)$ depends on a field $p.f$ of another object p, either p is transitively owned by o or p has granted friend access to f and o is in $p.\mathsf{deps}$. The semantic Definition 7 of admissibility is slightly intricate, so this section gives sufficient but not necessary conditions for a formula to denote an admissible invariant. Useful special cases and abbreviations are also given.

A formula over $\mathsf{self} : C$ denotes an admissible invariant for class C provided that for every field reference $E.f$, one of the following holds:

(1) E is self;
(2) E is some variable x in the scope of $(\forall x \bullet \mathsf{self} = x.\mathsf{own} \Rightarrow \ldots)$, or the antecedent can be indirect ownership like $\mathsf{self} = x.\mathsf{own}.\mathsf{own}$;
(3) E has the form $\mathsf{self}.h_0.h_1 \ldots h_n$ $(n > 0)$ and the formula includes a conjunct that implies self transitively owns E;
(4) E is some variable x in the scope of $(\forall x \bullet \mathsf{self} \in x.\mathsf{deps} \Rightarrow \ldots)$ and f is declared in some class with a friend declaration that makes f readable by C;

(5) E has the form $\mathsf{self}.g$ and the formula includes a conjunct that implies $g = \mathsf{null} \vee \mathsf{self} \in g.\mathsf{deps}$ (and again f appears in a suitable friend declaration).

Also, $f \equiv \mathsf{com}$ is not allowed. (We write $\equiv$ to mean syntactic identity.) Usually $f \equiv \mathsf{inv}$ is also disallowed, but see Sect. 11. Moreover, any expression of the form $E'.\mathsf{deps}$ in the formula, with $E' \not\equiv \mathsf{self}$, must be in the context $\mathsf{self} \in E'.\mathsf{deps}$ —the only way an object's invariant may depend on another's deps field is by its own membership, as exemplified by cases (4) and (5). Finally, the formula must not be falsifiable by allocation of new objects.

Because quantification ranges over allocated objects, it can be used in ways that are falsifiable by allocation, e.g., $(\forall o : C \bullet o = \mathsf{self})$. But the quantifications in (2) and (4) are not falsifiable by allocation. A somewhat tricky use of quantification in an admissible invariant appears in class Collection2, Fig. 3.

Case (1) is exemplified by the occurrences of *time* in $\mathcal{I}_{Master}$ and t, m in $\mathcal{I}_{Clock}$ in Sect. 4. Case (5) is exemplified by the occurrence of $m.time$ in $\mathcal{I}_{Clock}$. Case (3) is actually an instance of (2). For example, a formula $\ldots \mathsf{self}.h.f \ldots$ can be rewritten as $(\forall x \bullet \mathsf{self} = x.\mathsf{own} \Rightarrow (x = \mathsf{self}.h \Rightarrow \ldots x.f \ldots))$. Similarly, (5), with $f \not\equiv \mathsf{deps}$, can be rewritten to fit (4).

To make it easier to formulate admissible invariants, Barnett et al. (2004) and Leino and Müller (2004) suggest tagging fields to abbreviate useful invariants. For a field h of class C, the tag **rep** h indicates that self owns $\mathsf{self}.h$ and its invariant can therefore depend on fields of $\mathsf{self}.h$. It is also convenient to use a tag **peer** h to indicate that h has the same owner as self. For friendship, **pivot** h could indicate that the invariant depends on fields of $\mathsf{self}.h$ in virtue of a friendship relation.

In more detail, the tag **rep** h for field h declared in C signals that the invariant of C includes the condition $h = \mathsf{null} \vee h.\mathsf{own} = \mathsf{self}$. So if $E \equiv \mathsf{self}.h_0.h_1.\ldots.h_n$ where each h_j is tagged as **rep** then self transitively owns E so case (3) applies. Similarly, a field declaration can be tagged **peer** h to signal the invariant $h = \mathsf{null} \vee \mathsf{self}.\mathsf{own} = h.\mathsf{own}$. Then $E.f$ is allowed if E has the form $\mathsf{self}.h_0.h_1.\ldots.h_i \ldots h_n$ $(i \geq 0)$ where h_0, h_1, $\ldots h_i$ are tagged **rep** and each $h_{i+1} \ldots h_n$ is tagged **peer** —thus self transitively owns $\mathsf{self}.h_0.h_1.\ldots.h_i$, and this in turn has the same owner as $\mathsf{self}.h.h_0.h_1.\ldots.h_i \ldots h_n$ owing to the peer declarations, so case (3) applies. For example, a *List* class could have field **rep** $head : Node$ and *Node* could have **peer** $next : Node$. The associated invariants imply that every node $p \in o.head.next^*$ is owned by list o —without the need to express it using such a regular path expression in the invariant of *List*.

Sometimes it is useful to maintain an invariant that delimits which objects are owned, e.g., $(\forall p \bullet \mathsf{self} = p.\mathsf{own} \Rightarrow p \in o.head.next^*)$. Note that this is admissible, by case (3).

For friendship dependency, a field g can be tagged as a **pivot**, signalling the invariant $g = \mathsf{null} \vee \mathsf{self} \in g.\mathsf{deps}$ and allowing formulas that mention $E.f$ where E is g (provided there is a declaration that C is a friend reading f). A delimiting invariant is also admissible: $(\forall p \bullet \mathsf{self} \in p.\mathsf{deps} \Rightarrow p = \mathsf{self}.g\ )$. Together, these imply $\mathsf{self}.g \neq \mathsf{null} \iff \mathsf{self} \in \mathsf{self}.g.\mathsf{deps}$. But it is neither necessary nor always feasible for deps to be so accurate.

Longer pivot chains can also be used, although it is less obvious how field tags could be used to abbreviate the requisite invariants. For example, $E.f$ is allowed in an invariant for C, with $E \equiv g.h$, in the situation

- C declares $g : D0$,
- $D0$ declares $h : D1$ and has a declaration **friend** C **reads** h, and
- $D1$ declares f and has a declaration **friend** C **reads** f.

Then C would include the invariant

$$g = \mathsf{null} \vee (\mathsf{self} \in g.\mathsf{deps} \wedge (g.h = \mathsf{null} \vee \mathsf{self} \in g.h.\mathsf{deps}))$$

A corresponding delimiting invariant could be

$$(\forall p \bullet \mathsf{self} \in p.\mathsf{deps} \Rightarrow p = \mathsf{self}.g \vee p = \mathsf{self}.g.h\ )$$

These delimiting invariants are useful in discharging the friendship guard proof obligation (12), by reducing the range of quantification to some specific pivot expressions.

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

An important generalization of admissibility that we omit in the formalization is that, if $E.f$ is allowed under the conditions above, and g is an immutable field, then $E.f.g$ can be allowed.

6 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, we use a language similar to the imperative core of Java including value parameters and results, mutable local

$C \in$ classnames	$f \in$ fieldnames $\quad m \in$ methodnames $\quad x \in$ varnames	
$CL ::=$	$\textbf{class}\ C\ \{\ \bar{f} : \bar{T};\ \bar{M}\ \}$	
$T\ ::=$	$\mathsf{bool} \mid \mathsf{unit} \mid C$	data type
$M\ ::=$	$m(\bar{x} : \bar{T}) : T\ \{S\}$	method declaration
$S\ ::=$	$\textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ S \mid S;\ S$	alternative; sequence
	$\mid\ \textbf{var}\ x : T := E\ \textbf{in}\ S \mid x := E$	local variable block; assignment
	$\mid\ x := E.m(\bar{E})$	invoke method
	$\mid\ E.f := E \mid x := \textbf{new}\ C$	assign to field; construct object
	$\mid\ \textbf{pack}\ E \mid \textbf{unpack}\ E \mid \textbf{set-own}\ E\ \textbf{to}\ E$	set and unset inv; update own
	$\mid\ \textbf{attach}\ E \mid \textbf{detach}\ E$	add and remove from $\mathsf{self.deps}$
$E\ ::=$	$x \mid \mathsf{null} \mid \textbf{true} \mid \textbf{false} \mid E.f \mid E = E$	var.; const.; field access; ref. equal.

Table 2
Grammar. 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. Type unit, often called "void", has a single value and is used for methods that return nothing useful.

variables and object fields, and dynamic instantiation of objects. Expressions have no side effects but may dereference chains of fields. The denotational semantics is adapted from Banerjee and Naumann (2005a), omitting subclasses.

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 classes (via field and method types).

To define the helping functions, suppose $CT(C)$ has the form

$$\textbf{class}\ C\ \{\ \bar{f} : \bar{T}_0;\ \bar{M}\ \}$$

For fields, define $\mathit{fields}\ C = \bar{f} : \bar{T}_0$. For use in the semantics, we extend $\mathit{fields}\ C$ to a function $\mathit{xfields}\ C$ that also assigns types to the auxiliary fields — $\mathsf{inv} : \mathsf{bool}$, $\mathsf{com} : \mathsf{bool}$, $\mathsf{own} : \mathsf{Object}$, $\mathsf{deps} : \mathit{setof}(\mathit{Loc})$. Here $\mathit{setof}(\mathit{Loc})$ means finite sets of object references; it is not a type in the programming language but notation in the metalanguage to streamline later definitions.

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

A class table is *well formed* if each class C is well formed, which simply means that each method declaration $m(\bar{x} : \bar{T}) : T\ \{S\}$ in C is well formed in the sense that $\mathsf{self} : C, \bar{x} : \bar{T}, \mathsf{result} : T \vdash S$ using the rules in Table 3.

$$\frac{C = \Gamma x \qquad x \neq \mathsf{self}}{\Gamma \vdash x := \mathbf{new}\ C} \qquad \frac{\Gamma \vdash E_0 : C \qquad (f:T) \in \mathit{fields}\ C \qquad \Gamma \vdash E_1 : T}{\Gamma \vdash E_0.f := E_1}$$

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

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

$$\frac{\Gamma \vdash E_0 : C_0 \qquad \Gamma \vdash E_1 : C_1}{\Gamma \vdash \mathbf{set\text{-}own}\ E_0\ \mathbf{to}\ E_1} \qquad \frac{\Gamma \vdash E_0 : T_0 \qquad \Gamma \vdash E_1 : T_1}{\Gamma \vdash E_0 = E_1 : \mathsf{bool}}$$

Table 3
Selected typing rules.

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

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

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. Reus (2003)). Commands denote state transformers.

We assume that a countable set *Loc* is given, along with distinguished value nil not in *Loc*. To track an object's class we assume given a function $\mathit{type} : \mathit{Loc} \to \mathit{ClassNames}$ such that for each C there are infinitely many locations o with $\mathit{type}\ o = C$. (Function application is written using juxtoposition. In Sect. 3, e.g., in equation (8), we parenthesize —$\mathit{type}(o)$— to aid the casual reader.)

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*, i.e., type respecting valuations of the variables in Γ. (An added constraint is that self is not null.) The domain

$$
\begin{array}{lcl}
[\![\Gamma \vdash E_0.f := E_1]\!]_\mu(h,s) & = & \mathsf{let}\ q = [\![\Gamma \vdash E_0 : C]\!](h,s)\ \mathsf{in} \\
& & \mathsf{if}\ q = \mathsf{nil}\ \mathsf{then}\ \bot\ \mathsf{else} \\
& & \mathsf{let}\ v = [\![\Gamma \vdash E_1 : T]\!](h,s)\ \mathsf{in}\ ([h \mid q.f \mapsto v], s) \\
[\![\Gamma \vdash x := \mathbf{new}\ C]\!]_\mu(h,s) & = & \mathsf{let}\ q = \mathit{fresh}(C,h)\ \mathsf{in} \\
& & \mathsf{let}\ h_1 = [h \mid q \mapsto [\mathit{fields}\ C \mapsto \mathit{defaults}]]\ \mathsf{in} \\
& & (h_1, [s \mid x \mapsto q]) \\
[\![\Gamma \vdash x := E.m(\bar{E})]\!]_\mu(h,s) & = & \mathsf{let}\ q = [\![\Gamma \vdash E : C]\!](h,s)\ \mathsf{in}\ \mathsf{if}\ q = \mathsf{nil}\ \mathsf{then}\ \bot\ \mathsf{else} \\
& & \mathsf{let}\ \bar{x} : \bar{T} \to T = \mathit{mtype}(m,C)\ \mathsf{in} \\
& & \mathsf{let}\ \bar{v} = [\![\Gamma \vdash \bar{E} : \bar{T}]\!](h,s)\ \mathsf{in} \\
& & \mathsf{let}\ s_1 = [\bar{x} \mapsto \bar{v}, \mathsf{self} \mapsto q]\ \mathsf{in} \\
& & \mathsf{let}\ (h_0, v_0) = \mu\, C\, m(h, s_1)\ \mathsf{in}\ (h_0, [s \mid x \mapsto v_0])
\end{array}
$$

$$
\begin{array}{l}
[\![\Gamma \vdash \mathbf{unpack}\ E]\!]_\mu(h,s) \\
\quad = \mathsf{let}\ q = [\![\Gamma \vdash E : C]\!](h,s)\ \mathsf{in}\ \mathsf{if}\ q = \mathsf{nil}\ \mathsf{then}\ \bot\ \mathsf{else} \\
\qquad \mathsf{let}\ h_1 = (\,\lambda\, p \in \mathit{dom}\ h \bullet \mathsf{if}\ h\,p.\mathsf{own} = q\ \mathsf{then}\ [h\,p \mid \mathsf{com} \mapsto \mathsf{ff}]\ \mathsf{else}\ h\,p\,)\ \mathsf{in} \\
\qquad ([h_1 \mid q.\mathsf{inv} \mapsto \mathsf{ff}],\ s) \\
[\![\Gamma \vdash \mathbf{pack}\ E]\!]_\mu(h,s) \\
\quad = \mathsf{let}\ q = [\![\Gamma \vdash E : C]\!](h,s)\ \mathsf{in}\ \mathsf{if}\ q = \mathsf{nil}\ \mathsf{then}\ \bot\ \mathsf{else} \\
\qquad \mathsf{let}\ h_1 = (\,\lambda\, p \in \mathit{dom}\ h \bullet \mathsf{if}\ h\,p.\mathsf{own} = q\ \mathsf{then}\ [h\,p \mid \mathsf{com} \mapsto \mathsf{tt}]\ \mathsf{else}\ h\,p\,)\ \mathsf{in} \\
\qquad ([h_1 \mid q.\mathsf{inv} \mapsto \mathsf{tt}],\ s)
\end{array}
$$

$$
\begin{array}{lcl}
[\![\Gamma \vdash \mathbf{attach}\ E]\!]_\mu(h,s) & = & \mathsf{let}\ q = [\![\Gamma \vdash E : C]\!](h,s)\ \mathsf{in}\ \mathsf{if}\ q = \mathsf{nil}\ \mathsf{then}\ \bot\ \mathsf{else} \\
& & \mathsf{let}\ p = s.\mathsf{self}\ \mathsf{in}\ ([h \mid p.\mathsf{deps} \mapsto h\,p.\mathsf{deps} \cup \{q\}],\ s) \\
[\![\Gamma \vdash \mathbf{detach}\ E]\!]_\mu(h,s) & = & \mathsf{let}\ q = [\![\Gamma \vdash E : C]\!](h,s)\ \mathsf{in}\ \mathsf{if}\ q = \mathsf{nil}\ \mathsf{then}\ \bot\ \mathsf{else} \\
& & \mathsf{let}\ p = s.\mathsf{self}\ \mathsf{in}\ ([h \mid p.\mathsf{deps} \mapsto h\,p.\mathsf{deps} - \{q\}],\ s) \\
[\![\Gamma \vdash \mathbf{set\text{-}own}\ E_0\ \mathbf{to}\ E_1]\!]_\mu(h,s) & = & \mathsf{let}\ q = [\![\Gamma \vdash E_0 : C_0]\!](h,s)\ \mathsf{in}\ \mathsf{if}\ q = \mathsf{nil}\ \mathsf{then}\ \bot\ \mathsf{else} \\
& & \mathsf{let}\ p = [\![\Gamma \vdash E_1 : C_1]\!](h,s)\ \mathsf{in}\ ([h \mid q.\mathsf{own} \mapsto p],\ s)
\end{array}
$$

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 nested update $[h \mid q \mapsto [h\,q \mid f \mapsto v]]$. Assume *fresh* is an arbitrary function to *Loc* such that $\mathit{type}(\mathit{fresh}(C,h)) = C$ and $\mathit{fresh}(C,h) \notin \mathit{dom}\ h$.

$[\![\mathit{state}\ C]\!]$ 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 $h\,o\ f$.

For uniformity of notation, we write $[\![\mathit{Heap}]\!]$ for the set of heaps and adopt similar suggestive notations for domains involving heaps. The domain named $[\![\mathit{Heap} \otimes \Gamma]\!]$ contains program states (h,s) consisting of a heap and a store $s \in [\![\Gamma]\!]$ with no locations that are dangling with respect to h. Similarly, $[\![\mathit{Heap} \otimes T]\!]$ contains pairs (h,v) where $v \in [\![T]\!]$ and is not a dangling location. The pre-

ceding 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 $[\![\mathit{Heap} \otimes \Gamma]\!] \to [\![T]\!]_\perp$, i.e., it returns either a value $v \in [\![T]\!]$ (such that $(h, v) \in [\![\mathit{Heap} \otimes T]\!]$) or the improper value $\perp$ which represents errors. (In this simple language the only errors are null dereferences.) The meaning of $\Gamma \vdash S$ is a function $[\![\mathit{MEnv}]\!] \to [\![\mathit{Heap} \otimes \Gamma]\!] \to [\![(\mathit{Heap} \otimes \Gamma)]\!]_\perp$ 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, with reference to Table 3 for types of constituent parts.

For conciseness, the semantic definitions are written using a metalanguage construct "$\mathsf{let}\ \alpha = \beta\ \mathsf{in}\ \gamma$" (for α, β, γ of various kinds) that denotes $\perp$ in case $\alpha = \perp$.

The domain $[\![C, \bar{x}, \bar{T} \to T]\!]$ is the set of meanings for methods of class C with result type T and parameters $\bar{x} : \bar{T}$. Ordered pointwise, this is a complete partial order with bottom. Finally, a *method environment* $\mu \in [\![\mathit{MEnv}]\!]$ sends each C and method name m declared in C to a meaning of the right type. For example, if $\mathit{mtype}(m, C)$ is $\bar{x} : \bar{T} \to T$ then $\mu\, C\, m$ is in $[\![C, \bar{x}, \bar{T} \to T]\!]$. Method environments are ordered pointwise and form a complete partial order with bottom.

Definition 2 (semantics of method declaration) Suppose M is a method declaration in class C, so M has the form $m(\bar{T}\ \bar{x}) : T\ \{S\}$. For any method environment μ, define $[\![M]\!]_\mu$ to be an element of $[\![C, \bar{x}, \bar{T} \to T]\!]$ as follows.

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

Here h ranges over heaps and s ranges over $[\![\bar{x} : \bar{T}, \mathsf{self} : C]\!]$ in accord with the definition of $[\![C, \bar{x}, \bar{T} \to T]\!]$ in Table 4.

The default values are ff for bool, it for unit, and nil for object types, in this definition and also for **new** in Table 5. Thus a new object has $\mathsf{inv} = \mathsf{ff}$, $\mathsf{com} = \mathsf{ff}$, and $\mathsf{own} = \mathsf{nil}$; also $\mathsf{deps} = \varnothing$.

Definition 3 (semantics of CT) The semantics, $[\![CT]\!]$, of a well formed class table CT is the least upper bound[7] of the ascending chain $\mu \in \mathbb{N} \to [\![\mathit{MEnv}]\!]$ defined by $\mu_0\, C\, m = (\, \lambda\, (h, s) \bullet \perp\,)$ and $\mu_{j+1}\, C\, m = [\![M]\!]_{\mu_j}$ where m is declared as M in C.

[7] The least upper bound has a simple characterization. Formal details for a similar language can be found in Banerjee and Naumann (2005b) and are machine checked in Naumann (2005b).

The idea is that μ_j approximates $[\![CT]\!]$ in a way such that, in operational terms, it gives the correct semantics for executions with method call stack bounded in depth by j.

7 Proper annotation

There are several ways the program invariants $\mathcal{PI}$ (Def. 1) could be used: as "facts", included in what is sometimes called the "background predicate" that axiomatizes the semantics of the programming language (e.g., absence of dangling locations and $\mathsf{self} \neq \mathsf{null}$); as lemmas for reasoning directly in terms of program semantics; or in rules of a logic. We want to justify that $\mathcal{PI}$ can 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 just mentioned, we use **assert** statements.

This section defines admissible invariants in terms of semantic predicates, which are also used in **assert** statements. Assertions and admissibility are then used to define proper annotation.

A *predicate* for some state type Γ is just a subset $\mathcal{P} \subseteq [\![\mathit{Heap} \otimes \Gamma]\!]$. Note that $\bot \notin \mathcal{P}$. For example, a candidate invariant for C is a predicate $\mathcal{I}_C \subseteq [\![\mathit{Heap} \otimes (\mathsf{self} : C)]\!]$. We are a little casual about coercing a predicate on one state space to a predicate on another (precise details are straightforward). We write $(h, s) \models \mathcal{P}$ to mean $(h, s) \in \mathcal{P}$, sometimes as a hint that coercion may be needed for the store s. The most common coercion is that for location o we write $\mathcal{I}_C(o)$ for $\{h \mid (h, [\mathsf{self} \mapsto o]) \in \mathcal{I}_C\}$; moreover, for any Γ we treat $\mathcal{I}_C(o)$ as a predicate for Γ that is independent from the store.

For quantification, suppose $\mathcal{P}$ is a function from locations to predicates for Γ. Then $(\,\forall\, o \bullet \mathcal{P}\, o\,)$ is the subset of $[\![\mathit{Heap} \otimes \Gamma]\!]$ defined by $(h, s) \models (\,\forall\, o \bullet \mathcal{P}\, o\,)$ iff $(h, s) \in \mathcal{P}\, o$ for all $o \in \mathit{dom}\ h$. Recall that nil is not a location. It is 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.

Assert statements were not listed in the grammar because we allow semantic predicates, to avoid commitment to a formula language. We allow $\Gamma \vdash \mathbf{assert}\ \mathcal{P}$ just when $\mathcal{P}$ is a predicate for Γ, and define the semantics by $[\![\Gamma \vdash \mathbf{assert}\ \mathcal{P}]\!]_\mu(h, s) = \mathsf{if}\ (h, s) \in \mathcal{P}\ \mathsf{then}\ (h, s)\ \mathsf{else}\ \bot$. This is independent from μ, and $[\![\mathit{Heap} \otimes \Gamma]\!]$ is flat, so there is no problem with continuity.

In terms of formulas, a predicate depends on $E.f$ if updating $E.f$ can falsify the predicate. The following semantic formulations are convenient.

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 with $[h \mid o.f \mapsto v] \in [\![\mathit{Heap}]\!]$.

Definition 5 (new-closed) We say $\mathcal{P}$ is *new-closed* iff $(h, s) \in \mathcal{P}$ implies $([h \mid o \mapsto \mathit{defaults}], s) \in \mathcal{P}$ for all $o \notin \mathit{dom}\, h$.

Definition 6 (transitive ownership) For any heap h, the transitive ownership relation $\succ^h$ on $\mathit{dom}\ h$ is defined inductively by the conditions

- $o = h\, p.\mathsf{own} \Rightarrow o \succ^h p$; and
- $o \succ^h q \wedge q = h\, p.\mathsf{own} \Rightarrow o \succ^h p$.

Definition 7 (admissible invariant) A predicate $\mathcal{P} \subseteq [\![\mathit{Heap} \otimes (\mathsf{self} : C)]\!]$ is *admissible as an invariant for* C provided that it is new-closed and for every (h, s) and p, f such that $\mathcal{P}$ depends on $p.f$ in (h, s), field f is neither inv nor com, and one of the following conditions holds:

local: $p = s(\mathsf{self})$
owner: $s(\mathsf{self}) \succ^h p$ and $f \not\equiv \mathsf{deps}$
friend: $s(\mathsf{self}) \in h\, p.\mathsf{deps}$ and either $f \in \mathit{reads}(C, \mathit{type}\, p)$ or $f \equiv \mathsf{deps}$. Moreover, in case $f \equiv \mathsf{deps}$ we also require that for any X with $(h, s) \in \mathcal{P}$ and $([h \mid p.\mathsf{deps} \mapsto X], s) \notin \mathcal{P}$ we have either $([h \mid p.\mathsf{deps} \mapsto X \cup \{s(\mathsf{self})\}], s) \in \mathcal{P}$ or $([h \mid p.\mathsf{deps} \mapsto X - \{s(\mathsf{self})\}], s) \in \mathcal{P}$.

The complicated condition for $f \equiv \mathsf{deps}$ in friend dependencies ensures that the only way for an object's invariant to depend on $h\, p.\mathsf{deps}$ for another object p is via the condition $s(\mathsf{self}) \in h\, p.\mathsf{deps}$. In Sect. 11 we allow $f \equiv \mathsf{inv}$ as well, for friend dependencies.

It is instructive to check that formulas satisfying the constraints in Sect. 5 are admissible. The constraints are not necessary, however. For example, with a friend dependency on $E.g.f$ where g is a pivot, the presence of a top-level conjunct $g = \mathsf{null} \vee \mathsf{self} \in g.\mathsf{deps}$ is not necessary; it suffices that this is implied in the states where the formula actually depends on $E.g.f$. Moreover, depending on the logic's treatment of partiality and quantification, additional care is needed in formalizing those constraints. Because formulas are not included in this paper's technical content, there is no technical result stating that the syntactic constraints of Sect. 5 ensure admissibility.

The following result captures a key feature of the $\mathsf{inv}/\mathsf{own}$ discipline. It is the reason that the precondition for field update need not quantify over all

transitive owners, by contrast with the situation for friend dependents.

Lemma 7.1 (transitive ownership) Suppose $h \models \mathcal{PI}$, $o \succ^h p$, and $h\, o.\mathsf{inv} = \mathsf{tt}$. Then $h\, p.\mathsf{com} = \mathsf{tt}$.

Proof: By induction on $\succ^h$. In the base case, $o \succ^h p$ means $h\, p.\mathsf{own} = o$, and the result follows by (14). In the induction step, $o \succ^h p$ means there is q with $h\, p.\mathsf{own} = q$ and $o \succ^h q$. By induction, $h\, q.\mathsf{com} = \mathsf{tt}$. Hence by (15) we have $h\, q.\mathsf{inv} = \mathsf{tt}$. By $h\, o.\mathsf{own} = q$ and $h\, q.\mathsf{inv} = \mathsf{tt}$ we get $h\, q.\mathsf{com} = \mathsf{tt}$ by (14). □

Aside 1 If initially both $\neg o.\mathsf{inv}$ and $\neg p.\mathsf{inv}$ then **set-own** can establish $o \succ^h p$ and $p \succ^h o$, But it is then not possible to establish $o.\mathsf{inv}$ or $p.\mathsf{inv}$ because to set $o.\mathsf{inv}$ requires $p.\mathsf{com}$ and this cannot be set unless $p.\mathsf{inv}$. The same is true of longer cycles and also the case of $o.\mathsf{own} = o$. □

Definition 8 (properly annotated class table) A properly annotated class table is one such that

- there are declarations as defined in Sect. 3;
- each object invariant $\mathcal{I}_C$ is admissible;
- each field update and special statement is preceded by an **assert** that implies the stipulated precondition (Table 1); and
- each update guard satisfies its obligation (12).

The assertions are not required to be correct; this is not required for our main theorem, which is expressed in terms of partial correctness. The reason is that in an initial state where a predicate $\mathcal{P}$ is false, the outcome from **assert** $\mathcal{P}$ is $\perp$ (which represents divergence). Thus if a program is properly annotated but some of the assertions are incorrect, it will vacuously preserve invariants.

8 Soundness

Our soundness result shows, essentially, that for *any* constituent command S of a program properly annotated with assertions (Table 1), we have $\{\mathcal{PI}\}\ S\ \{\mathcal{PI}\}$ in the sense of partial correctness. [8]

[8] To avoid unilluminating complications, we use an error-insensitive notion of partial correctness; that is, the semantics identifies null-dereference errors with divergence, i.e., the outcome $\perp$. 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. For the main statements of interest in this paper it is straightforward to formulate preconditions for the absence of such errors and they are included in Table 1.

Soundness for commands is formulated as follows: If $[\![\Gamma \vdash S]\!]_\mu(h,s) \neq \bot$ and $h \models \mathcal{PI}$ then $h_0 \models \mathcal{PI}$ 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 $\mathcal{PI}$. To show that the assumption is discharged, Theorem 8.1 says that $\mathcal{PI}$ is maintained by every method in the environment μ denoted by a properly annotated class table.

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

Theorem 8.1 If class table CT is properly annotated then $[\![CT]\!]$ maintains $\mathcal{PI}$.

Proof: Let (μ_j) be the approximation chain of which $[\![CT]\!]$ is the least upper bound (Definition 3). Claim: μ_j maintains $\mathcal{PI}$, for all j. The proof is by induction on j. The base case is that μ_0 maintains $\mathcal{PI}$, which holds because $\mu_0\, C\, m$ is the everywhere-$\bot$ function and "maintains" is defined in the sense of partial correctness. For the induction step, if μ_j maintains $\mathcal{PI}$ then we can unfold the definition of μ_{j+1} and apply Lemma 8.2.

The least upper bound can be characterized pointwise[9] (for each C, m the least upper bound is essentially the union of compatible partial functions $\mu_j\, C\, m$), so the claim implies that $[\![CT]\!]$ maintains $\mathcal{PI}$. $\square$

To formulate the main lemma it is convenient to decompose commands in such a way that each interesting primitive command, like field update and **pack**, is combined with the assertion that precedes it. The *annotated commands* are given by the following grammar.

$$
\begin{aligned}
S ::= {} & \textbf{assert}\ \mathcal{P};\ E.f := E \mid \textbf{assert}\ \mathcal{P};\ \textbf{pack}\ E \mid \textbf{assert}\ \mathcal{P};\ \textbf{unpack}\ E \\
& \mid \textbf{assert}\ \mathcal{P};\ \textbf{attach}\ E \mid \textbf{assert}\ \mathcal{P};\ \textbf{detach}\ E \mid \textbf{assert}\ \mathcal{P};\ \textbf{set-own}\ E\ \textbf{to}\ E \\
& \mid \textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ S \mid S;\ S \mid \textbf{var}\ x\!:\!T := E\ \textbf{in}\ S \mid x := E \mid x := E.m(\bar{E}) \\
& \mid x := \textbf{new}\ C \mid \textbf{assert}\ \mathcal{P}
\end{aligned}
$$

Every method body in a properly annotated program can be parsed as an annotated command. Note that there may be additional **assert** statements besides those that are required.

Lemma 8.2 (main lemma) Suppose that CT is properly annotated and μ maintains $\mathcal{PI}$. If S is an annotated command that is a constituent of a

[9] Similar arguments are formalized precisely in Banerjee and Naumann (2005b) and are machine checked in Naumann (2005b).

method in CT then S maintains $\mathcal{PI}$. That is, for all (h, s), if $h \models \mathcal{PI}$ and $(h_0, s_0) = [\![\Gamma \vdash S]\!]_\mu(h, s)$ then $h_0 \models \mathcal{PI}$.

Proof: By induction on the structure of annotated command S. The interesting cases are the primitive commands that can falsify $\mathcal{PI}$, by extending the range of quantifications (**new**) or updating fields. Separate Lemmas 8.4–8.10 are given below for these cases. As an example, in the case of **assert** $\mathcal{P}$; $E.f := E$ the predicate $\mathcal{P}$ implies the stipulated precondition for $E.f := E$, by proper annotation. So, by semantics of ";" and **assert**, the precondition holds before $E.f := E$; now Lemma 8.5 completes the argument.

For the case of method call $x := E.m(\bar{E})$, by semantics (h_0, s_0) is obtained by forming argument store s_1 and applying $\mu\, C\, m(h, s_1)$ which yields the result heap h_0 and a result value that is assigned to x to obtain s_0. Then $h_0 \models \mathcal{PI}$ by hypothesis that μ maintains $\mathcal{PI}$. Hence $(h_0, s_0) \models \mathcal{PI}$ because $\mathcal{PI}$ is independent from the store.

For the case $x := E$, $\mathcal{PI}$ is maintained because it depends only on the heap. Similarly, **assert** $\mathcal{P}$ yields $\perp$ or else an unchanged state. For sequence, conditional, and local variable block the the result goes by induction on S. □

Section 2 concludes with the claim that the discipline allows $\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$ to be deduced from $\{\mathcal{PI} \wedge \mathcal{P}\}\ S\ \{\mathcal{Q}\}$, for any $S, \mathcal{P}, \mathcal{Q}$. Put differently, $\mathcal{PI}$ can be asserted "at any control point". Since $\mathcal{PI}$ quantifies over allocated objects, it is easy to see that it holds in an initial state where no objects have been allocated. So the claim is a consequence of the following.

Corollary 8.3 (soundness) If annotated command S is a constituent of a method body in a properly annotated class table CT then it maintains $\mathcal{PI}$, i.e., $\{\mathcal{PI}\}\ S\ \{\mathcal{PI}\}$.

Proof: By Lemma 8.2, taking μ to be $[\![CT]\!]$ and using Theorem 8.1. □

This section concludes with the results used to prove the Main Lemma 8.2.

Lemma 8.4 (new) If $h \models \mathcal{PI}$ and $(h_0, s_0) = [\![\Gamma \vdash x := \textbf{new}\ C]\!]_\mu(h, s)$ then $h_0 \models \mathcal{PI}$.

Note that $(h_0, s_0) = [\![\Gamma \vdash x := \textbf{new}\ C]\!]_\mu(h, s)$ implies that the outcome is non-$\perp$. The outcome from **new** is never $\perp$, and under the stipulated preconditions the other commands of interest have an outcome of $\perp$ only when subexpressions do. The lemmas only consider the interesting case, non-$\perp$.

Proof: Suppose q is the fresh object, so that $h_0 = [h \mid q \mapsto \mathit{defaults}]$. We consider each of the conditions in $\mathcal{PI}$ in turn. For (13): $h_0\ q.\mathsf{inv} = \mathsf{ff}$ by

definition. Because admissible $\mathcal{I}_C$ is new-closed, adding q to the heap does not falsify (13) for existing objects. For (14): similarly to the preceding case, noting that $h_0\ q.\mathsf{own} = \mathsf{null}$. For (15): similarly to the preceding case, noting that $h_0\ q.\mathsf{com} = \mathsf{ff}$. $\square$

Lemma 8.5 (field update) Suppose $(h_0, s_0) = [\![\Gamma \vdash E.f := E']\!]_\mu(h, s)$ and $h \models \mathcal{PI}$. Then $h_0 \models \mathcal{PI}$ provided that the stipulated preconditions are satisfied, i.e.,

- $q \neq \mathsf{null}$ where $q = [\![\Gamma \vdash E : B]\!]_\mu(h, s)$
- $h\, q.\mathsf{inv} = \mathsf{ff}$
- for all $p \in h\, q.\mathsf{deps}$, if $f \in \mathit{reads}(\mathit{type}(p), B)$ then either $h\, p.\mathsf{inv} = \mathsf{ff}$ or $h \models \mathcal{U}_{\mathit{type}(p),B,f}(p, q, v)$, where $v = [\![\Gamma \vdash E' : T]\!]$

Proof: By semantics, $h_0 = [h \mid q.f \mapsto v]$. For (13): Suppose, for some o, D that $\mathcal{I}_D(o)$ depends on $q.f$ in (h, s). We must show that either $h_0\ o.\mathsf{inv} = \mathsf{ff}$ or $h_0 \models \mathcal{I}_D(o)$. By admissibility of $\mathcal{I}_D$ it suffices to consider these cases:

- $q = o$ —Then $h\, o.\mathsf{inv} = \mathsf{ff}$ by precondition.
- $o \succ^h q$ —Then precondition $h\, q.\mathsf{inv} = \mathsf{ff}$ implies $h\, q.\mathsf{com} = \mathsf{ff}$ by $h \models \mathcal{PI}(15)$ and then $h\, o.\mathsf{inv} = \mathsf{ff}$ by transitive ownership Lemma 7.1. So $h_0\ o.\mathsf{inv} = \mathsf{ff}$.
- $o \in h\, q.\mathsf{deps}$ and $f \in \mathit{reads}(D, B)$. (As we are considering ordinary field update, $f \not\equiv \mathsf{deps}$.) Now by precondition we have either $h\, o.\mathsf{inv} = \mathsf{ff}$, whence $h_0\, o.\mathsf{inv} = \mathsf{ff}$ by definition of h_0, or else $h \models \mathcal{U}_{D,B,f}(o, q, v)$. In the latter case, $h \models \mathcal{I}_D(o)$; instantiating the update guard obligation (12) with q for o we obtain $h \models \mathcal{I}_D(o)[q.f :\approx v]$, whence $h_0 \models \mathcal{I}_D(o)$.

For (14) and (15): the relevant fields are not updated. $\square$

Lemma 8.6 (pack) Suppose $h \models \mathcal{PI}$ and $(h_0, s_0) = [\![\Gamma \vdash \mathbf{pack}\ E]\!]_\mu(h, s)$. Then $h_0 \models \mathcal{PI}$ provided that the stipulated preconditions (Table 1) are satisfied, i.e., letting $q = [\![\Gamma \vdash E : B]\!](h, s)$ we have

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

Proof: For (13): For any o, if $o \neq q$ then $o.\mathsf{inv} \Rightarrow \mathcal{I}(o)$ by $h \models \mathcal{PI}$, because **pack** only changes inv and com on which admissible invariants do not depend. For the case of $o = q$ we have $\mathcal{I}_B(q)$ by precondition. For (14): We have $h_0 \models (\,\forall p \bullet p.\mathsf{own} = q \Rightarrow p.\mathsf{com}\,)$ by semantics of **pack**. For $o \neq q$, no owner fields are changed in h_0 nor is any com changed to false in h_0. For (15): If $h\, o.\mathsf{com} = \mathsf{ff}$ but $h_0\ o.\mathsf{com} = \mathsf{tt}$ then $h\, o.\mathsf{own} = q$ by definition of h_0; and $h_0\ o.\mathsf{inv} = \mathsf{tt}$ by the last of the preconditions listed above. $\square$

Lemma 8.7 (unpack) Suppose $(h_0, s_0) = [\![\Gamma \vdash \textbf{unpack}\ E]\!]_\mu(h, s)$ and $(h, s) \in \mathcal{PI}$. Then $(h_0, s_0) \in \mathcal{PI}$ provided that the stipulated preconditions hold, i.e.,

- $q \neq \mathsf{null}$, where $q = [\![\Gamma \vdash E : B]\!]_\mu(h, s)$
- $h\, q.\mathsf{inv} = \mathsf{tt}$
- $h\, q.\mathsf{com} = \mathsf{ff}$

Proof: For (13): An admissible $\mathcal{I}_C$ does not depend on inv or com, which are the only fields updated from h to h_0, so no $\mathcal{I}_C(o)$ is falsified. Setting $h_0\ q.\mathsf{inv}$ falsifies the antecedent in (13). For (14): The only p for which $p.\mathsf{com}$ gets falsified are those for which $h\, p.\mathsf{own} = q$ and then $h_0\ p.\mathsf{inv} = \mathsf{ff}$ by semantics of unpack. For (15): No com field gets truthified. For inv, only $q.\mathsf{inv}$ gets falsified; and $h\, q.\mathsf{com} = \mathsf{ff}$ by precondition so $h_0\ q.\mathsf{com} = \mathsf{ff}$ by semantics of unpack. □

Lemma 8.8 (set-own) Suppose $(h_0, s_0) = [\![\Gamma \vdash \textbf{set-own}\ E\ \textbf{to}\ E']\!]_\mu(h, s)$ and $(h, s) \models \mathcal{PI}$. Then $(h_0, s_0) \models \mathcal{PI}$ provided that the stipulated preconditions are satisfied, i.e.,

- $q \neq \mathsf{null}$ and $h\, q.\mathsf{inv} = \mathsf{ff}$, where $q = [\![\Gamma \vdash E : B]\!](h, s)$
- $q' \neq \mathsf{null}$ or $h\, q'.\mathsf{inv} = \mathsf{ff}$, where $q' = [\![\Gamma \vdash E' : C]\!](h, s)$
- for all $p \in h\, q.\mathsf{deps}$, if $\mathsf{own} \in \mathit{reads}(\mathit{type}(p), B)$ then either $h\, p.\mathsf{inv} = \mathsf{ff}$ or $h \models \mathcal{U}_{\mathit{type}(p),B,\mathsf{own}}(p, q, v)$

Proof: Similar to the proof for ordinary field update. □

Lemma 8.9 (detach) Suppose $(h_0, s_0) = [\![\Gamma, \mathsf{self} : B \vdash \textbf{detach}\ E']\!]_\mu(h, s)$ and $(h, s) \models \mathcal{PI}$. Then $(h_0, s_0) \models \mathcal{PI}$ provided that the stipulated preconditions are satisfied, i.e.,

- $h\, p.\mathsf{inv} = \mathsf{ff}$, where $p = s(\mathsf{self})$
- $q \neq \mathsf{null}$, where $q = [\![\Gamma \vdash E' : C]\!]_\mu(h, s)$
- $h\, q.\mathsf{inv} = \mathsf{ff}$

Proof: For (14) and (15): the relevant fields are not updated. For (13): We consider cases on how $\mathcal{I}_D(o)$ could depend on $h\, p.\mathsf{deps}$ for some D and o.

- $o = p$ —but then $h\, o.\mathsf{inv} = \mathsf{ff}$ by precondition
- $o \succ^h p$ —but then $f \not\equiv \mathsf{deps}$ by admissibility of $\mathcal{I}_D$, i.e., ownership dependencies are not on the deps field
- $o \in h\, p.\mathsf{deps}$ —then for the invariant to be falsified we would have $h\, o.\mathsf{inv} = \mathsf{tt}$ and $h \models \mathcal{I}_D(o)$ but $h_0 \not\models \mathcal{I}_D(o)$. In case $o = q$, it could well be that removing q from $p.\mathsf{deps}$ falsifies $\mathcal{I}_D(q)$ but $h\, q.\mathsf{inv} = \mathsf{ff}$ by precondition. It remains to consider $o \neq q$. By semantics, $h_0 = [h \mid p.\mathsf{deps} \mapsto p.\mathsf{deps} \cup \{q\}]$. Since $\mathcal{I}_D(o)$ depends on $p.\mathsf{deps}$ in h, we can instantiate X in Def. 7 as $X := h\, p.\mathsf{deps} \cup \{q\}$ and then, by admissibility, either $[h \mid p.\mathsf{deps} \mapsto X \cup$

$\{o\}] \models \mathcal{I}_D(o)$ or $[h \mid p.\mathsf{deps} \mapsto X - \{o\}] \models \mathcal{I}_D(o)$. Now either o is in $h\,p.\mathsf{deps}$ —whence by $o \neq q$ we have $X = X \cup \{o\}$— or o is not in $h\,p.\mathsf{deps}$ and then $X = X - \{o\}$; in either case we get $[h \mid p.\mathsf{deps} \mapsto X] \models \mathcal{I}_D(o)$ which contradicts the hypothesis $h_0 \not\models \mathcal{I}_D(o)$. □

Lemma 8.10 (attach) Suppose $(h_0, s_0) = [\![\Gamma, \mathsf{self} : B \vdash \textbf{attach}\ E']\!]_\mu(h, s)$ and $(h, s) \models \mathcal{PI}$. Then $(h_0, s_0) \models \mathcal{PI}$ provided that the stipulated preconditions are satisfied, i.e.,

- $h\,p.\mathsf{inv} = \mathsf{ff}$, where $p = s(\mathsf{self})$
- $q \neq \mathsf{null}$, where $q = [\![\Gamma \vdash E' : C]\!]_\mu(h, s)$

Proof: Similar but slightly simpler than the proof for detach. □

9 Subtypes

The inv/own discipline of Barnett et al. (2004) encompasses subclassing and inheritance in a way we briefly review below. The friendship discipline presented in Barnett and Naumann (2004) does not take subtyping into account although it is compatible with the treatment of subtyping for inv/own. In this section we sketch the interaction between friendship and subtyping, which does not lead to an interesting extension. Soundness for the inv/own discipline with subtyping is a straightforward extension of our results so we do not formalize it.

In a language with subclassing, a given object is an instance not only of its class but of all its superclasses, each of which may have invariants. The methodology takes this into account as follows. Instead of inv being a boolean, it ranges over class names C such that C is a superclass of the object's allocated type. That is, it is an invariant (enforced by typing rules) that $o.\mathsf{inv} \geq \mathit{type}\,o$. Program invariant (13) is changed to $(\forall C, o \bullet o.\mathsf{inv} \leq C \Rightarrow \mathcal{I}^C(o))$. That is, if o is packed at least to class C then the invariant $\mathcal{I}^C$ for C holds. Statement **unpack** E is augmented with syntax to express the class C from which E is being unpacked and the precondition $E.\mathsf{inv}$ is changed to $E.\mathsf{inv} = C$. The effect is to set $E.\mathsf{inv}$ to the direct superclass of C. Similarly, the precondition for **pack** E includes that $E.\mathsf{inv}$ equals the superclass of C.

The own field is changed to range over pairs (o, C), so that if $p.\mathsf{own} = (o, C)$ then o directly owns p and an admissible invariant $\mathcal{I}^D(o)$ may depend on p for types D with $\mathit{type}\,o \leq D \leq C$. Transitive ownership is formulated so that q is transitively owned by o at C if $q.\mathsf{own} = (o, C)$ or q is transitively owned *at some class* by some p such that $p.\mathsf{own} = (o, C)$.

These changes are compatible with friendship dependencies as formalized in Sect. 7 and soundness for the revised discipline can be proved by mild adaptations of the proofs in Sect. 8. But friendship appears to be best treated as a relationship between two classes, so that subtypes do not inherit the relationship and thus are not able to change its terms.

It would not be modular for a subtype to declare a friendship relation for a field declared in a supertype. Suppose G declares a field x. If G' (a subtype of G) uses x in a friend declaration for the class F, then the code of G must be available so that it can be verified against the update guard(s) imposed by F. This is because the updates to x must satisfy the update guards: it cannot wait until the object is being repacked to the G' level.

However, the following situation can be allowed. Suppose a class F declares a field g (of type G) and G declares F' (a subtype of F) as a friend. F' is allowed to use its inherited field g as a pivot field since if some method in F updates the field g, then it must be unpacked. When F is unpacked, then the protocol ensures F' is unpacked, so it doesn't matter if its invariant is falsified. Of course, at the point that it is re-packed to the F' level, its invariant will be checked which includes the (possibly new) value reached through the updated pivot field.

If G and F are friends (G the granter and F the friend), then F' (F' a subtype of F) is allowed to depend upon G in its invariant, but it is not allowed to impose any further update guards on G (for the same modularity concerns). If it does depend on G, then it has the same obligation to prove that any updates made by G to the friend fields do not falsify its invariant. If in the same situation, a subtype G' of G updates a friend field x, then it is subject to the update guard F defined on x.

While it appears to be unlikely, it is possible for both F and F' (where F' is a subtype of F) to both be declared to be friends of G. In that case, the values kept in each object's *deps* field become pairs (o, T) of an object and the class at which the friendship exists.

10 Iterators

The specification of a collection and its associated enumerators (iterators) has long been a challenging problem. A collection and its enumerators do not fit with standard notions of ownership since enumerators are dependent on the collection from which they were spawned, and can directly access its internal representation, but cannot own it or be owned by it. A recent discussion of the difficulties can be found in Aldrich and Chambers (2004). They propose an

ownership mechanism that admits the desired pattern of sharing but they leave open the question of specification. We give three specifications to illustrate the expressiveness and effectiveness of the discipline and to motivate an extension: allowing a friendship based invariant to depend on inv. The following section spells out the technical changes needed to allow such invariants.

Briefly, a collection is a mutable ordered list. One of its frequently used operations is to retrieve an enumerator from it: a read-only view that permits traversing the collection. Since it is a read-only view, any modifications made to its underlying collection invalidate the enumerator.

For efficiency reasons, collections and the enumerators often share program state, e.g., the collection may own the linked list nodes that it is using to store the collection elements and the enumerator has direct access to the nodes. A standard means of implementing them is to use version numbers so that a collection does not need to keep references to all of the enumerators that it hands out. Instead, it is each enumerator's responsibility to keep track of its collection's version number and invalidate itself when the collection has changed. Note that this scheme requires cooperation from both parties: a collection increments its version number monotonically and enumerators must never change theirs.

In the three versions we use the single method *Add* as an example of a method in the collection that updates the underlying collection.

The first version, Fig. 2, illustrates the simplest relationship: a one-way friendship in which the collection is bound by the update guards expressed by the enumerator. It cannot be sure that incrementing its version number in *Add* suffices to maintain the friend's update guard unless it knows that none of its dependent enumerators have their version number greater than its own. Therefore it must express this as a precondition on the *Add* method. This specification for the *Add* method imposes a proof obligation on all clients of the collection. Using modifies specifications (not shown), a client should be able to maintain an invariant about the extant enumerators, sufficient to establish the precondition of *Add*.

Since the enumerator always has a field referencing the collection, we prefer to write the pivot field's name, *coll*, in the update guards instead of the keyword piv.

Maintaining the proper relationship is an internal affair between the collection and the enumerator so the second version, Fig. 3, uses two-way friendship: each class is a friend of the other. For the collection to be a friend of the enumerator means that we can convert the precondition from *Add* into (part of) the object invariant in the collection. And we can directly impose constraints on the enumerator, using update guards. We could restrict the enumerator from

```
class Collection1 {
  ver : int;
  state : State;
  friend Enumerator1 reads ver, state;
  GetEnumerator() : Enumerator1
    ensures result ∈ self.deps;
  { result := new Enumerator1(self); }
  Add(i : int)
    requires inv ∧ ¬com;
    requires (∀ o : Enumerator1 • o ∈ self.deps ⇒ o.vsn ≤ ver );
  { unpack self; ver := ver + 1; state := ...; pack self; }
  Attach(x : Enumerator1) { unpack self; attach x; pack self; }
}
class Enumerator1 {
  vsn : int;
  coll : Collection1; // pivot
  model elmnts : State
  invariant vsn ≤ coll.ver ∧ self ∈ coll.deps∧
    (vsn = coll.ver ⇒ elmnts.Equals(coll.state));
  guard coll.ver := val by coll.ver ≤ val;
  guard coll.state := val by coll.ver ≠ vsn;
  Enumerator1(c : Collection1) // constructor
    requires c.inv ∧ ¬c.com;
    ensures self ∈ c.deps ∧ coll = c;
  { coll := c; vsn := c.ver; c.Attach(self); pack self; }
  MoveNext() : bool requires vsn = coll.ver { ... }
}
```

Fig. 2. Enumerator Version 1

changing its version number at all, but in terms of fulfilling the mutual contract, it suffices that a collection can invalidate all of its extant enumerators by simply incrementing its version number.

While the second example nicely expresses the protocol by which they operate, the update guards of the enumerator impose an ordering restriction on the collection: it must first increment its version number before it can update its internal state. This restriction is because after every field update, the friend's invariant must hold. An update guard is a constraint over two adjacent states: the state just before the field update and the state after. When the friend is dependent upon several fields, it may be more natural for the granter to update the fields in any order as long as there is a single point at which the friend's invariant holds again. The unpack-pack boundary provides just such granularity. (One can imagine extending the idea to atomic blocks.)

```
class Collection2 {
  ver : int;
  state : State;
  friend Enumerator2 reads ver, state;
  invariant ( ∀ o : Enumerator2 • o ∈ self.deps ⇒
                    (o.vsn ≤ ver ∧ o.coll = self ∧ self ∈ o.deps) );
  guard piv.vsn := val by val ≤ ver;
  guard piv.coll := val by val = self;
  GetEnumerator() : Enumerator2
    ensures result ∈ self.deps;
  { result := new Enumerator2(self); }
  Add(i : int)
    requires inv ∧ ¬com;
  { unpack self; ver := ver + 1; state := ...; pack self; }
  Attach(x : Enumerator2)
    requires x.vsn ≤ ver ∧ x.coll = self ∧ self ∈ x.deps ∧ inv ∧ ¬com;
  { unpack self; attach x; pack self; }
}
class Enumerator2 {
  vsn : int;
  coll : Collection2; // pivot
  model elmnts : State
  friend Collection2 reads vsn, coll;
  invariant vsn ≤ coll.ver ∧ self ∈ coll.deps ∧
    (vsn = coll.ver ⇒ elmnts.Equals(coll.state));
  guard coll.ver := val by coll.ver ≤ val;
  guard coll.state := val by coll.ver ≠ vsn;
  Enumerator2(c : Collection2)
    requires c.inv ∧ ¬c.com;
    ensures self ∈ c.deps ∧ coll = c ∧ c ∈ self.deps;
  { vsn := c.ver; coll := c; attach c; c.Attach(self); elmnts := c.state; pack self; }
  MoveNext() : bool requires vsn = coll.ver { ... }
}
```

Fig. 3. Enumerator Version 2

So in version three (Fig. 4), the enumerator is now dependent on the inv field of the collection. This frees all of the other update guards to require only that the collection is unpacked. The collection can perform its updates in any order. The update guard for the inv field means that at the point of the **pack** statement in the collection the states of the two objects are compared. As shown, the simplest way for the collection to satisfy the update guard for inv is to increment its version number. Alternatively, a collection could maintain an invariant connecting version numbers with elements, but then

```
class Collection3 {
  ver : int;
  state : State;
  friend Enumerator3 reads ver, state, inv;
  invariant ( ∀ o : Enumerator3 • o ∈ self.deps ⇒
                    (o.vsn ≤ ver ∧ o.coll = self ∧ self ∈ o.deps) );
  guard piv.vsn := val by val ≤ ver;
  guard piv.coll := val by val = self;
  GetEnumerator() : Enumerator3
    ensures result ∈ self.deps;
  { result := new Enumerator3(self); }
  Add(i : int)
    requires inv ∧ ¬com;
  { unpack self; state := ...; ver := ver + 1; pack self; }
  Attach(x : Enumerator3)
    requires x.vsn ≤ ver ∧ x.coll = self ∧ self ∈ x.deps ∧ inv ∧ ¬com;
  { unpack self; attach x; pack self; }
}
class Enumerator3 {
  vsn : int;
  coll : Collection3; // pivot
  model elmnts : State
  friend Collection3 reads vsn, coll;
  invariant self ∈ coll.deps ∧
      (coll.inv ⇒
        vsn ≤ coll.ver ∧ (vsn = coll.ver ⇒ elmnts.Equals(coll.state)));
  guard coll.ver := val by ¬coll.inv;
  guard coll.state := val by ¬coll.inv;
  guard coll.inv := val by val ⇒ (vsn = coll.ver ⇒ elmnts.Equals(coll.state));
  Enumerator3(c : Collection3)
    requires c.inv ∧ ¬c.com;
    ensures self ∈ c.deps ∧ coll = c ∧ c ∈ self.deps;
  { vsn := c.ver; coll := c; attach c; c.Attach(self); elmnts := c.state; pack self; }
  MoveNext() : bool requires vsn = coll.ver { ... }
}
```

Fig. 4. Enumerator Version 3

the enumerator would have to add the *elmnts* field to its friend declaration and the collection could express an update guard for it.

Because the enumerator's invariant now depends on the inv field of the collection, in the conjunct ($coll.\mathsf{inv} \Rightarrow vsn \leq coll.ver \wedge \ldots$), the enumerater can deduce little from its own invariant unless it also knows the collection is

packed. This may seem overly restrictive, but the enumerator is in fact dependent upon the state of the collection, so it requires that the collection be in a consistent state at the points that it needs to retrieve an element from the collection and that is exactly what the inv field represents.

It is tempting to write $(\forall o : \mathit{Enumerator} \bullet o.\mathit{coll} = \mathsf{self} \Rightarrow o.\mathit{vsn} \leq \mathit{ver})$ as invariant for *Collection* in the second or third versions. But this is not admissible, as it depends on the *coll* field of all *Enumerator*s.

The specifications given in the figures are chosen to illuminate the points under discussion and are not quite complete enough to verify the programs. In particular, look again at the condition $\mathsf{self} \in o.\mathsf{deps}$ in the invariant of *Collection*2 and *Collection*3. This is established by method *Attach*, but what prevents an enumerator o from later removing its collection from $o.\mathsf{deps}$? This can be done, but owing to the stipulated precondition of **detach** it can only be done when the collection is unpacked. So the difficulty is actually for the collection to establish its invariant as precondition for packing itself. One way to do so would be for *Enumerator*2 to maintain an invariant $\mathit{coll} \in \mathsf{self.deps}$ (which is admissible and is maintained by the code in *Enumerator*2 and *Enumerator*3). An alternative is to add an update guard in collection for deps; but it is left to future work to check the details. (Note that we do not require the deps field to be listed in the **friend** declaration of the enumerator, because whenever a friend depends on some field of a granter, the discipline forces the friend to also depend on the granter's deps field.)

11 Invariants can depend on inv

The last example in Sect. 10 shows that it can be useful for a friendship based invariant to depend on the granter's inv field. (There appears to be no need to allow dependence on inv for ownership based invariants so we do not allow that.) This section spells out this extension of the discipline.

We allow inv to be in $\mathit{reads}(C, G)$. A predicate $\mathcal{P} \subseteq [\![\mathit{Heap} \otimes (\mathsf{self} : C)]\!]$ is *admissible as an invariant for C in the extended sense* provided that it is new-closed and for every (h, s) and o, f such that $\mathcal{P}$ depends on $o.f$ in (h, s), field f is not com, and one of the following conditions holds:

local: $o = s(\mathsf{self})$ and $f \not\equiv \mathsf{inv}$
owner: $s(\mathsf{self}) \succ^h o$ and $f \not\equiv \mathsf{deps}$, $f \not\equiv \mathsf{inv}$
friend: $s(\mathsf{self}) \in h\, o.\mathsf{deps}$ and $f \in \mathit{reads}(C, \mathit{type}(o))$ or $f \equiv \mathsf{deps}$. Moreover, in case $f \equiv \mathsf{deps}$ we also require that for any X with $(h, s) \in \mathcal{P}$ and $([h \mid o.\mathsf{deps} \mapsto X], s) \notin \mathcal{P}$ we have either $([h \mid o.\mathsf{deps} \mapsto X \cup \{s(\mathsf{self})\}], s) \in \mathcal{P}$ or $([h \mid o.\mathsf{deps} \mapsto X - \{s(\mathsf{self})\}], s) \in \mathcal{P}$.

In case inv is in $reads(C, G)$, the friend C must provide an update guard in C for inv; this is subject to the normal proof obligation (12) in C.

Only pack and unpack update inv. For both **pack** E and **unpack** E (with $E : B$), the stipulated precondition of Table 1 is extended by conjoining the following:

$$(\,\forall\, p \in E.\mathsf{deps} \bullet \mathsf{inv} \in reads(type(p), B) \;\Rightarrow\; \neg p.\mathsf{inv} \vee \mathcal{U}_{type(p),B,\mathsf{inv}}(p, E, v)\,);$$

where v is the constant tt for pack and ff for unpack.

Theorem 11.1 Suppose class table CT is properly annotated, in the revised sense that inv can occur in friendship-based invariants and the condition displayed just above is included in the preconditions for pack and unpack. Then $[\![CT]\!]$ maintains $\mathcal{PI}$.

The proof is very similar to the proof of Theorem 8.1. The only commands that update inv are **pack** and **unpack** . It suffices to show that these cannot falsify an invariant $\mathcal{I}_D(o)$ that depends on inv, so that program invariant (13) is maintained. But such an $\mathcal{I}_D(o)$ has an associated update guard that is stipulated as precondition for both **pack** and **unpack** . Then the argument is the same as for field update in Lemma 8.5.

12 Related work

This paper is a revised and extended version of Naumann and Barnett (2004). It includes more proofs and examples, consideration of subclassing and friendship, and extension of the discipline to allow invariants to depend on the inv field. An unnecessary program invariant was removed, admissibility has been slightly generalized, and the role of pivot fields has been downplayed.

The ownership discipline or "Boogie methodology" is introduced in Barnett et al. (2004) using syntactic rules based on **rep** fields (see Sect. 3). Leino and Müller (2004) encode the owner as a ghost field, allowing ownership transfer and invariants that quantify over owned objects. Barnett and Naumann (2004) add the friendship discipline. Informal arguments for soundness appear in Barnett et al. (2004), in terms of axiomatic semantics. The first proof of soundness using a formal semantics is in Naumann and Barnett (2004). A proof sketch for soundness of the ownership discipline in terms of small step semantics is given by Leino and Müller (2004) (although the definition of admissibility is not entirely clear).

Pierik et al. (2005) weaken the notion of admissibility to allow invariants

that are falsifiable by allocation of new objects; they adapt our notion of update guards to "creation guards" to protect such invariants. This work is in the context of a sound and complete proof outline logic for object oriented programs (Pierik, 2006). Leino and Müller (2005) extend the discipline to invariants that depend on static fields (class-scoped global variables) using a static field sinv that tracks exposure of a class in the same way that inv tracks exposure of an instance. Jacobs et al. (2005) extend the discipline to multithreaded programs. Banerjee and Naumann (2005c) adapt the discipline to simulation relations.

Various techniques have been proposed to hide information about an invariant while expressing that it is in force. One alternative is to introduce a type-state (DeLine and Fähndrich, 2001) to stand for "the invariant is in force". Müller (2002) uses a model field for this purpose. Another approach is to treat the invariant's name as opaque with respect to its definition (Bierman and Parkinson, 2005), as may be done in higher order logic using existential quantification (Biering et al., 2005). Another way to treat the invariant as an opaque predicate, which to the authors' knowledge has not been explored, is to use a pure method (Leavens et al., 2003) to represent the invariant; this could be of practical use in runtime verification and hiding of internals could be achieved using visibility rules of the programming language.

Some ownership systems prevent harmful updates by preventing the existence of references from client to rep (the dominator property that all paths to a rep go through its owner). It is easy to violate the dominator property: a method could return a rep pointer, or pass one as an argument to a client method. The dominator property can be enforced using a type system such as the Universe system (Müller, 2002) and variations on Ownership Types (Clarke et al., 2001; Boyapati et al., 2003, 2002; Aldrich and Chambers, 2004). These systems do not directly enforce the dominator property, which is expressed in terms of paths. Rather, they constrain references, disallowing any object outside an ownership domain from having a pointer to inside the domain. The inv/own discipline prevents harmful updates by restricting uses of references rather than their existence; one benefit is that ownership transfer can be handled. Skalka and Smith (2005) also study use-based object confinement, for different purposes.

We noted in Sect. 1 that Separation Logic (Reynolds, 2002) provides a way to express that a predicate depends on only some objects in the heap. The "tight interpretation" of triples lets them express on what part of the heap a command acts. The logic has been used for encapsulating invariants in simple imperative programs (O'Hearn et al., 2004) and some steps have been taken to adapt the logic to object-oriented programs Parkinson (2005). This is an exciting line of research, but adoption of a nonstandard logic for specification and verification has significant cost.

By contrast, the ingredients of the inv/own discipline are just assertions and ghost fields,[10] including updates to ghost fields of an unbounded number of objects (in **pack** E, for example, the com field of every object owned by E is updated). This is quite limited machinery and thus the discipline is suitable for use in a variety of settings. It could be formalized within an ordinary program logic, most attractively a proof outline logic (Pierik and de Boer, 2005). It is being explored in the context of Spec#, a tool based directly on a system of verification conditions, and in a tool developed by de Boer and Pierik (2002). In both cases the assertion language is (roughly) first order plus reachability but that is not essential.

13 Conclusions

We have formalized and shown soundness for the programming discipline of Barnett and Naumann (2004), built on Barnett et al. (2004), in which auxiliary fields in annotations express intended atomicity and encapsulation. The Main Lemma 8.2 and Theorem 8.1 justify appealing to program invariant $\mathcal{PI}$ where needed. Then $\mathcal{PI}(13)$ 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" (O'Hearn et al., 2004).

In O'Hearn et al. (2004), 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 logical 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, in 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?

[10] With inv, own ranging over values that include class names, i.e., slightly beyond ordinary program data types. Similar use of class names is available in the JML specification via the *Type* operator (Leavens et al., 2003).

Acknowledgments Thanks to DeLine, Fähndrich, Leino, and Schulte whose work on ownership with the second author laid the foundation under which we have built. This phrase we borrow from a paper by Tony Hoare a quarter century ago. Tony, Wolfram Schulte, Gilles Barthe, and the organizers of FMCO 2004 provided the first author with stimulating environments in which to carry out and get feedback on the work. Cees Pierik pointed out a bug in an earlier proof of Lemma 8.4. The papers by Müller et al. (2003) and O'Hearn et al. (2004) were instrumental in illuminating the perspective we have attempted to articulate in Sect. 1. Conversations with these authors, Anindya Banerjee, and Frank de Boer have been very helpful. A number of corrections and improvements were provided by thorough and thoughtful anonymous reviewers.

References

Aldrich, J., Chambers, C., 2004. Ownership domains: Separating aliasing policy from mechanism. In: European Conference on Object-Oriented Programming (ECOOP). pp. 1–25.

Appel, A. W., 2001. Foundational proof-carrying code. In: Proceedings of LICS. pp. 247–258.

Apt, K. R., Olderog, E.-R., 1997. Verification of Sequential and Concurrent Programs, 2nd Edition. Springer.

Banerjee, A., Naumann, D. A., 2002. Representation independence, confinement and access control. In: ACM Symp. on Princ. of Program. Lang. (POPL). pp. 166–177.

Banerjee, A., Naumann, D. A., Nov. 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., 2004. 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 LNCS. pp. 49–69.

Barnett, M., Naumann, D. A., 2004. Friends need a bit more: Maintaining

invariants over shared state. In: Kozen, D., Shankland, C. (Eds.), Mathematics of Program Construction. Vol. 3125 of LNCS. pp. 54–84.

Biering, B., Birkedal, L., Torp-Smith, N., 2005. BI-hyperdoctrines, higher-order separation logic, and abstra ction. Tech. Rep. ITU-TR-2005-69, IT University of Copenhagen.

Bierman, G., Parkinson, M., 2005. Separation logic and abstraction. In: ACM Symp. on Princ. of Program. Lang. (POPL). pp. 247–258.

Birkedal, L., Torp-Smith, N., Yang, H., 2005. Semantics of separation-logic typing and higher-order frame rules. In: IEEE Symp. on Logic in Computer Science (LICS). pp. 260–269.

Boyapati, C., Lee, R., Rinard, M., 2002. Ownership types for safe programming: Preventing data races and deadlocks. In: Object-oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 211–230.

Boyapati, C., Liskov, B., Shrira, L., 2003. Ownership types for object encapsulation. In: ACM Symp. on Princ. of Program. Lang. (POPL). pp. 213–223.

Clarke, D., 2001. Object ownership and containment, dissertation, Computer Science and Engineering, University of New South Wales, Australia.

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.

de Boer, F., Pierik, C., 2002. Computer-aided specification and verification of annotated object-oriented programs. In: Jacobs, B., Rensink, A. (Eds.), Formal Methods for Open Object-Based Distributed Systems. pp. 163–177.

de Boer, F. S., 1999. A WP-calculus for OO. In: Proceedings of Foundations of Software Science and Computation Structure (FoSSaCS). Vol. 1578 of LNCS. pp. 135–149.

DeLine, R., Fähndrich, M., 2001. Enforcing high-level protocols in low-level software. In: ACM Conf. on Program. Lang. Design and Implementation (PLDI). pp. 59–69.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Jacobs, B., Kiniry, J., Warnier, M., 2003. Java program verification challenges. In: de Boer, F., Bonsangue, M., Graf, S., de Roever, W.-P. (Eds.), Formal Methods for Components and Objects (FMCO 2002). Vol. 2852 of LNCS. pp. 202–219.

Jacobs, B., Piessens, F., Leino, K. R. M., Schulte, W., 2005. Safe concurrency for aggregate objects with invariants. In: Aichernig, B. K., Beckert, B. (Eds.), Software Engineering and Formal Methods (SEFM). pp. 137–147.

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 LNCS. Springer, pp. 262–284.

Leino, K. R. M., Müller, P., 2004. Object invariants in dynamic contexts. In: European Conference on Object-Oriented Programming (ECOOP). pp.

491–516.

Leino, K. R. M., Müller, P., 2005. Modular verification of static class invariants. In: Proceedings, Formal Methods. Vol. 3582 of LNCS. pp. 26–42.

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, A. R., Sieber, K., 1988. Towards fully abstract semantics for local variables: Preliminary report. In: Proceedings, Fifteenth POPL. pp. 191–203.

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

Mitchell, J. C., Plotkin, G. D., 1988. Abstract types have existential type. ACM Trans. Prog. Lang. Syst. 10 (3), 470–502.

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

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

Müller, P., Poetzsch-Heffter, A., Leavens, G., Oct. 2003. Modular invariants for object structures. Tech. Rep. 424, ETH Zürich, Chair of Software Engineering.

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

Naumann, D. A., 2005a. Assertion-based encapsulation, object invariants and simulations. In: de Boer, F. S., Bonsangue, M. M., Graf, S., de Roever, W. P. (Eds.), Post-proceedings, Formal Methods for Components and Objects (FMCO 2004). Vol. 3657 of LNCS. pp. 251–273.

Naumann, D. A., 2005b. Verifying a secure information flow analyzer. In: Hurd, J., Melham, T. (Eds.), 18th International Conference on Theorem Proving in Higher Order Logics TPHOLS. Vol. 3603 of Lecture Notes in Computer Science. Springer, pp. 211–226.

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

O'Hearn, P., Yang, H., Reynolds, J., 2004. Separation and information hiding. In: ACM Symp. on Princ. of Program. Lang. (POPL). pp. 268–280.

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

Parkinson, M. J., Nov. 2005. Local reasoning for Java. Tech. Rep. 654, University of Cambridge Computer Laboratory, dissertation.

Pierik, C., 2006. Validation techniques for object-oriented proof outlines, dissertation, Universiteit Utrecht.

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

Pierik, C., de Boer, F. S., 2005. A proof outline logic for object-oriented programming. Theoretical Comput. Sci.To appear.

Reus, B., 2003. Modular semantics and logics of classes. In: Baaz, M., Makowsky, J. A. (Eds.), Computer Science Logic (CSL). Vol. 2803 of LNCS. pp. 456–469.

Reynolds, J. C., 2002. Separation logic: a logic for shared mutable data structures. In: LICS. pp. 55–74.

Skalka, C., Smith, S., 2005. Static use-based object confinement. Springer International Journal of Information Security 4 (1-2).