State Based Encapsulation and Generics

Anindya Banerjee
Computing and Information Science
Kansas State University

David A. Naumann
Department of Computer Science
Stevens Institute of Technology

Technical Report CS-2004-11
(also appears as Kansas State University CIS TR 2004-5)

State based encapsulation and generics

December 20, 2004

Anindya Banerjee[⋆1] and David A. Naumann[⋆⋆2]

[1] Department of Computing and Information Sciences
Kansas State University, Manhattan KS 66506 USA
ab@cis.ksu.edu
[2] Department of Computer Science
Stevens Institute of Technology, Hoboken NJ 07030 USA
naumann@cs.stevens.edu

Abstract. A properly encapsulated data representation can be revised without affecting the correctness of client programs and extensions but encapsulation is difficult to achieve for heap based structures and object-oriented (OO) programs with reentrant callbacks. Building on a discipline that uses assertions and auxiliary fields to manage invariants and transferrable ownership, we give a rule for modular reasoning based on simulations. This representation independence result is proved for a sequential OO language with recursive, generic classes.

1 Introduction

You are responsible for a library consisting of many Java classes. A bug is found and you revise the code (but not the syntactic interface) of a certain class, perhaps also taking the opportunity to revise an internal data structure in order to improve its performance. You are in no position to check, or even be aware of, the many applications that use the class via its instances or by subclassing it. In principle, the class could have a full functional specification. It would then suffice to prove that the new version meets the specification. In practice, full specifications are rare. Nor is there a well established logic and method for modular reasoning about the code of a class in terms of the specifications of the classes it uses, without regard to their implementations or the users of the class in question [25]. The problem is that encapsulation, crucial for modular reasoning about invariants, is difficult to achieve in programs that involve shared mutable objects and reentrant callbacks which violate simple layering of abstractions. Yet complicated heap structure and calling patterns are used, in well designed object-oriented (OO) programs, precisely for orderly composition of abstractions in terms of other abstractions.

There is an alternative to verification with respect to a specification. One can attempt to prove that the revised version is behaviorally equivalent to the original. Of course their behavior is not identical, but at the level of abstraction of source code (e.g., modulo specific memory addresses) and perhaps under a specified precondition, it may be possible to show equivalence of behavior. The technique is standard [22, 33, 16, 42]: Define a *coupling relation* to connect the states of the two versions and prove that it, or an extension of it to other types, has the *simulation property*,

⋆ Supported in part by NSF grants CCR-0209205 and NSF Career Award CCR-0296182.
⋆⋆ Supported in part by NSF grants CCR-0208984, CCF-0429894 and by New Jersey Commission on Science and Technology.

i.e., it holds initially and is preserved by paired invocation of the two versions of each method. In most cases, one would want to define the relation for a single pair of instances of the class, as methods act primarily on a target object (self). A language with good encapsulation should enjoy a *representation independence* theorem that says a simulation for the revised class induces a simulation for any program built using the class. Suitable couplings are the identity except inside the abstraction boundary and an *identity extension lemma* says simulation implies behavioral equivalence of two programs that differ only by revision of a class. Again, such reasoning can be invalidated by heap sharing, which violates encapsulation of data, and by callbacks, which violate hierarchical control structure.

There is a close connection between the equivalence problem and verification: verification of OO code involves object invariants that constrain the internal state of an instance. Encapsulation involves defining the invariant in a way that protects it from outside interference so it holds globally provided it is preserved by the methods of the class. Simulations are like invariants over two copies of the state space, and again modular reasoning requires that the coupling for a class be independent from outside interference. *The main contribution of this paper is a representation independence theorem using a state-based discipline for heap encapsulation and control of callbacks.*

For simple imperative code and single-instance modules, O'Hearn *et al.* [38] prove a strong rule for local reasoning about object invariants using separation logic [45] which transparently expresses interdependence in the heap (and its absence). The authors give a representation independence result for OO code[2], using static rules expressed in terms of ordinary program types to achieve *ownership confinement*, the absence of boundary-crossing pointers through which interference can occur. Other confinement disciplines based on static analysis have been given with the objective of encapsulation for modular reasoning, though mostly without formal results on modular reasoning [13, 9, 14]. Müller proves soundness for a discipline of invariants in OO code based on an ownership type system [35]. Work using types makes confinement a *program invariant*, i.e., a property required to hold in every reachable state. This makes it difficult to transfer ownership, due to temporary sharing at intermediate states. Most disciplines preclude transfer; where it is allowed, it is achieved using nonstandard constructs such as destructive reads and restrictive linearity constraints.

As challenging as heap encapsulation is the problem of reentrant callbacks. Extant theories of data abstraction [22, 32, 16, 47] assume, in one way or another, a hierarchy of abstractions such that control does not reenter an encapsulation boundary while already executing inside it. In many programming languages it is impossible to write code that fails to satisfy the assumption.[1] But it is commonplace in OO programs for a method acting on some object o to invoke a method on some other object which in turn leads to invocation of another method on o. This makes it difficult to reason about when an object's invariant holds [46]. There is an analogous problem for reasoning with simulations. Although the results in [2] are sound for programs with reentrant callbacks, no help is given on how to show preservation by methods that make calls that can lead to reentrant callbacks. The theorem allows the programmer to assume that all methods preserve the relation when proving it for the method bodies —but this assumption is of no use if a call is made in an uncoupled intermediate state.

[1] This is trivial to prove for simple imperative programs etc and quite difficult to prove for Algol-like languages [39].

In a recent advance, [5, 30] reentrancy is managed using an explicit auxiliary field to designate states in which an object invariant is to hold. Ownership is also represented by an auxiliary mutable field. Ownership is managed more flexibly than with type-based static analyses because the ownership invariant need only hold in certain states. Heap encapsulation is achieved not by disallowing boundary-crossing pointers but by limiting, in a state-dependent way, their use.

The focus of this paper is to adapt the discipline[2] of [5, 30] to proving equivalence by simulation. In large part the discipline is unchanged, as one would hope in keeping with the idea that a coupling is just an invariant over two parallel states. But we have to adapt some features, in ways that make sense in terms of informal considerations of information hiding. The discipline imposes no control on field reads, only writes. For representation independence we need a stronger form of encapsulation. The discipline also allows ownership transfer quite freely, though it is not trivial to design code that correctly performs transfers. For representation independence, the transfer of previously-encapsulated data to clients (an unusual form of controlled "rep exposure" [17]) is allowed but must occur only in the code of the encapsulating class in order for its coupling to be preserved; even then, it poses a difficult technical challenge. The significance of our adaptations are discussed Section 2 and further in Section 8.

Our results are given for a language with generics. The type-based confinement regime in [2] precludes use of type Object to encode type parameterization and thus is best used with generics, but they are handled neither in [2] nor in other confinement work cited above. Given that much beautiful theory on simulations is related to relational parametricity of generics [44], one might expect difficulties in an OO language: generics fail to enjoy parametricity due to the presence of type tests and casts (and effects due to dynamic dispatch). (Generics do not eliminate the need for casts, as these languages lack disjoint sums, and libraries vary in coding style.) But we are concerned with relations between two implementations of a class, not two types at which a generic may be applied; it turns out that this form of representation independence does not need relational parametricity for type variables. Moreover, generics have little impact on the invariant discipline.

Contributions. We extend the invariant/ownership discipline [5] to generics and give a semantic proof of soundness (Section 4). In passing, we get type soundness for generics using a denotational model in which a generic denotes its ground instantiations (Section 3). The main contributions (Sections 5 and 6) are (a) formulation of a suitable notion of instance-based coupling analogous to invariants in the invariant/ownership discipline and manipulated according to the same discipline; (b) proof of a representation independence result for a language with inheritance and dynamic dispatch, recursive methods and callbacks; mutable objects, type casts, generics, and recursive types; and (c) results on identity extension and program equivalence. These results let the programmer concentrate on proving that methods of the specific class being revised preserve the coupling. We have also proved rules for establishing the basic coupling invariant for those methods and for maintaining the coupling in the presence of transfers of ownership across encapsulation boundaries, but they are omitted for lack of space. But the representation independence result encompasses such transfers: from client to abstraction [17], between abstractions, and even from abstraction to client [38, 4].

[2] Sometimes called "Boogie", which is the name of a current project at Microsoft Research which implements the discipline as part of a comprehensive verification system inspired by the ESC projects [18].

```
class Task {
  unit run(){...} }
class Qnode<X ◁ Task> {
  private X tsk;
  private Qnode<X> nxt;
  private int count, limit;
  //invar: tsk ≠ null ∧ 0≤count≤limit;
  ...
  unit run() { tsk.run(); count := count+1; }
  unit setTsk(X t, int lim) {
    tsk := t; limit := lim; count := 0;
    pack self as Qnode; } }
```

Fig. 1. Classes Task and Qnode.

Our results are based on a conventional program semantics. Predicates used in assertions required by the invariant/ownership discipline are treated as sets of states and our proofs identify necessary properties of invariants and coupling relations. There is a broad range of possible applications, e.g., in proof carrying code [37], where limited properties are verified using higher order logic, and in functional verification where specification languages restrict predicates to respect program visibility rules and to enforce behavioral subclassing [31, 29]. Further related work is mentioned in the discussion (Section 8).

2 Background and overview

The inv/own discipline. The challenge of reentrant callbacks is illustrated in the example code in Figures 1 and 2. Class Task serves to define an interface consisting of a single method run. In our illustrative language, all methods are dynamically dispatched and have public visibility; class types are references. The return type unit (also known as "void") indicates that the method is only of use for its effect on the heap. Class Qnode is parameterized on a type X that must be a subtype of Task [24, 21]. Field nxt is used to form singly linked lists of nodes. The special command **pack** in setTsk is discussed later.

In Qnode, method run can falsify the invariant 0≤count≤limit mentioned in a comment. Also, if the invariant tsk≠null is initially false, i.e., isn't actually invariant, then the invocation tsk.run() crashes. To reason about an invariant in a class, it is desirable to confine attention to the code of the class. Moreover, the public interface should not expose internal design decisions (which are subject to revision), as would happen, e.g., if we gave an explicit precondition tsk≠null for public method Qnode.run.

Table 2 declares a class intended to maintain a linked list of Qnode rooted at its field tsks. The invariant that all tasks have the same type is expressed using a type parameter. Another intended invariant is that each count field is an accurate count of the number of times the task has been run. Method Queue.runAll invokes run on each task for which the limit is not exceeded. Suppose type X is instantiated as RTask declared as follows.

```
class RTask {
```

```
class Queue<X ◁ Task> {
  private Qnode<X> tsks;
  private int runs := 0;
  . . .
  int getRuns() { result := runs; }
  unit runAll() {
    unpack self from Queue;
    Qnode<X> p := tsks;
    while p ≠ null do {
      if (p.getCount() < p.getLimit()) then runs := runs+1; p.run(); endif;
      p := p.getNxt(); }
    pack self as Queue; }
  unit add(X t, int lim){
    unpack self from Queue;
    Qnode<X> n := new Qnode<X>;
    setowner n to (self,Queue);
    n.setTsk(t,lim); n.setNxt(tsks); tsks := n;
    pack self as Queue; }}
  protected Qnode<X> getTsks() { //for subclass only
    result := tsks;}
  Enumeration<X> allTasks() {
    result := "an enum the contents of tsks" }}
```

Fig. 2. Class Queue.

```
Queue<RTask> q;
unit run(){q.runAll();} . . . }
```

Consider a state in which o points to an instance of Queue<RTask> and the first node in the list has count=0 and lim=1. Moreover, suppose field q of that node's task has value o. Invocation of o.runAll diverges: before count is incremented to reflect the first invocation, the task makes a *reentrant call* on o.runAll which again invokes run on the task.

The toy example illustrates a reentrant call to the same method, runAll, but it could as well be a different method, e.g., getRuns. The issue is that an invocation commences while one is in progress on the same object — the second invocation may find the object in an inconsistent state. Here, one inconsistency is that count is not an accurate count of the number of runs.

One expects to temporarily suspend invariants during execution of a procedure and to assume that an object's invariant holds initially for every invocation. There are practical situations in which it makes sense for a particular method *not* to assume the invariant as a precondition, e.g., a method intended to be used in reentrant callbacks —for example, getRuns in our example could be intended to give lower bound on the number of times runAll has run to completion, in which case a reentrant call is fine because this behavior of runAll does not depend on the invariant.[3] By

[3] To prove that getRuns has this behavior, we would have to add to the invariant. But we are not concerned with proving full specifications. The point here is just that there is some declared invariant that we want to preserve, and that invariant is needed as a precondition for some but not all methods.

far the most common case is where reentrant callbacks are neither intended nor foreseen. This poses the question: when does an object invariant hold?

An often-proposed answer is to require an object to establish its own invariant before making any method call, lest that call lead to a reentrant callback. Most calls do not, e.g., p.getNxt, so this is draconian and impractical. For example, after n.setTsk and before n.setNxt in method add, see Fig. 2.

Another non-solution is to abandon the idea that the object invariant is a precondition for any method. A more practical solution is to write object invariants in the form $o.inv \Rightarrow \mathcal{I}(o)$ where $\mathcal{I}$ is the invariant predicate of interest and *inv* is a field, publicly readable or at least visible in specifications, that explicitly represents whether the invariant is in force. To disallow reentrant callbacks, a method can have precondition $inv = true$ and set *inv* false before invoking other methods. The condition $inv \Rightarrow \mathcal{I}$ is maintained as a so-called *program invariant*, i.e., true in every state. In virtue of precondition and program invariant, the implementer can exploit $\mathcal{I}$ in the method body. It is the responsibility of a caller to establish the precondition. The code in RTask.run has no way of establishing q.inv so it is not justified to invoke q.runAll.

This idea has been explained in greater depth in [5, 30], where it is made into a methodology using special commands and rules to manipulate *inv*. In runAll, the command unpack self from Queue has, to a first approximation, the effect of setting *inv* false to prevent reentrant callbacks. It is set to true by the command pack self as Queue. To justify the assumption that $inv \Rightarrow \mathcal{I}$ holds in every reachable state, the command pack is subject to the precondition that $\mathcal{I}$ holds. Further preconditions are discussed below (see also Table 1).

A given object is an instance not only of its class but of all its superclasses, each of which may have invariants. The discipline of [5, 30] takes this into account as follows. The value of *inv* is not boolean but rather the name, C, of some superclass of the object's dynamic type. Instead of $inv \Rightarrow \mathcal{I}$, the *program invariant* is that

$$o.inv \leq C \Rightarrow \mathcal{I}^C(o) \tag{1}$$

for all C and all allocated objects o. That is, if o is "packed" at least to class C then the invariant $\mathcal{I}^C$ for C holds.

For the example above, Queue.runAll would be specified to have precondition self.inv $\leq$ Queue and RTask.run would be specified to have precondition self.inv $\leq$ Rtask. In the absence of formal specifications, the intended effect can be achieved by using an **assert** at the beginning of the bodies of these methods. Thinking in terms of semantics —runtime checking if you will—, in the situation where an RTask is in a Queue and has a link to that Queue, when runAll is invoked the Queue gets unpacked and then the invokes run which in turn invokes runAll in a state where the assertion will fail, aborting the computation. In terms of static checking, the call to runAll in RTask.run cannot be verified because the precondition cannot be established.

Typically, unpacking and packing exhibits a lexical structure as in runAll, akin to operations on existential types (from which the terminology is adopted). But this is not required. For new objects *inv* is initialized to Object and setTsk exhibits the pattern often found in constructors (which we omit for brevity).

Modularity demands that $\mathcal{I}(o)$ depends only (or largely) on encapsulated state. For fields of o, this can be achieved via private visibility —though note that class based visibility would allow the code to read private fields of other instances. But $\mathcal{I}(o)$ may depend on other objects, e.g., if Queue has an invariant that count<limit for each node in its list. The nodes can be encapsulated

$e \neq \textbf{null} \ \wedge \ e.\mathit{inv} = \mathit{super}\, C \ \wedge \ \mathcal{I}^C(e) \ \wedge \ \forall p \mid p.\mathit{own} = (e, C) \ \Rightarrow \ \neg p.\mathit{com} \ \wedge \ p.\mathit{inv} = \mathit{name}(\mathit{type}\, p)$ **pack** e **as** C	
$e \neq \textbf{null} \ \wedge \ e.\mathit{inv} = C \ \wedge \ \neg e.\mathit{com}$ **unpack** e **from** C	
$e_1 \neq \textbf{null} \ \wedge \ e_1.\mathit{inv} > C$	where C is the class that declares f; i.e., $f \in \mathit{dom}(\mathit{dfields}\, C)$
$e_1.f := e_2$	
$e_1 \neq \textbf{null} \ \wedge \ e_1.\mathit{inv} = \mathsf{Object} \ \wedge \ (e_2 = \textbf{null} \vee e_2.\mathit{inv} > C)$ **setown** e_1 **to** (e_2, C)	

Table 1. Stipulated preconditions of field update and the special commands. (We include the conjuncts $e \neq \textbf{null}$ in order to avoid worry about null dereferences in the rest of the formula; in fact $e \neq \mathit{error}$ is also needed. But these details might be worked out differently depending on the verification system in which our methodology is being used.)

using the notion that they are owned by o. Ownership is embodied in an auxiliary field *own*. The idea is that if $p.\mathit{own} = (o, C)$ then p is owned by o and moreover it is the invariants of o at types $\geq C$ that can depend on p. An object invariant is allowed to depend on transitively owned objects as well —collectively called an *island*.

Following [5, 30], we formalize the discipline in terms of fields with public visibility, to illuminate the conditions necessary for sound reasoning about updates and invariants. In practice of course, private and protected visibility are often used. In many cases, the conditions required by the Boogie discipline are consequences of visibility restrictions.

An invariant $\mathcal{I}^C(o)$ for object o is *admissible* if it can only be falsified by update of fields of o and of objects owned by o. For example, the invariant for Qnode depends only on its own fields and the count field in owned nodes. The discipline requires, for an update of the form $e_1.f := e_2$, that any object o with an invariant $\mathcal{I}^C(o)$ dependent on e_1 is sufficiently unpacked, i.e., $o.\mathit{inv} > C$. But the objects dependent on e_1 are e_1 itself and the objects that transitively own e_1. If e_1 is unpacked, i.e., not in a consistent state, then its owners should not be considered consistent, so it is enough to check $e_1.\mathit{inv}$. This idea is made precise using the third and last of the auxiliary fields, *com*, of boolean type. In addition to (1), two additional conditions are imposed as program invariants, i.e., to hold in all reachable states of all objects. The first may be read "an object is committed to its owner if its owner is packed":

$$o.\mathit{inv} \leq C \wedge p.\mathit{own} = (o, C) \ \Rightarrow \ p.\mathit{com} \tag{2}$$

The second says that a committed object is fully packed:

$$o.\mathit{com} \ \Rightarrow \ o.\mathit{inv} = \mathit{name}(\mathit{type}(o)) \tag{3}$$

Conditions (1–3) are program invariants provided that field updates and pack/unpack are subject to preconditions as stipulated in Table 1.

The effect of **pack** is not only to set *inv* but also to commit all currently-owned objects. More precisely, "**pack** e **as** C" sets $e.\mathit{inv} := C$ and sets $p.\mathit{com} := \mathit{true}$ for all p with $p.\mathit{own} = (e, C)$. Its precondition requires that $p.\mathit{inv}$ equals the dynamic type, $\mathit{type}\, p$, for each owned p, whence by (1) each is fully packed.

The effect of **unpack** includes un-committing all owned objects, making them susceptible to unpacking and also to ownership transfer. In detail, "**unpack** e **from** C" sets $e.inv := super\ C$ (where *super* gives the direct superclass) and sets $p.com := false$ for all p with $p.own = (e, C)$. Both **pack** and **unpack** update unboundedly many objects, but all affected fields are auxiliaries for reasoning, not part of the runtime state. The effect of "**setown** e_1 **to** (e_2, C)" is simply $e_1.own := (e_2, C)$. The default value of *com* is false; of *own* is (**null**, Object); of *inv* is Object.

References [5, 30] contain more leisurely introductions and extensive examples. Although it is not relevant in the sequel, let us mention one strength of the discipline: It facilitates the interpretation of the "modifies clause" used to specify frame conditions —if a method explicitly modifies o then it may also modify objects transitively owned by o and committed.

Another strength of the approach is that it is not tied to a particular logic for state predicates and correctness assertions. In our formalization we adopt a semantic formulation using **assert** statements which facilitates compositional formulation of the theory.

Generics. Among the various type-based approaches for ownership [15, 35, 2], none has been integrated with generics (Potanin *et al.* [43] report preliminary findings on one aspect). Generics have been treated in work on package confinement [51] but no reasoning principles have been developed exploiting package confinement, which is rather coarse grained. (We point out a potential use in the sequel.) Variations on Ownership Types [13, 9, 8] have been extensively developed and applied. The syntax resembles generic classes but the systems are closer to dependent types than to generics, in that an instance of a class is parameterized on an owning instance, not on a type. The *inv*/*own* discipline uses type names in auxiliary state: the value of *inv* is a class name and *own* pairs a pointer (or **null**) with a class name. Previous work on representation independence by the authors [2] uses class names to enforce a form of ownership confinement. Generics are not only useful but, for the latter work, necessary in order for the results to be practical: the root type Object allows values of all types so it is unusable for alias control, yet without generics its use is ubiquitous (to encode generics).

We adapt the *inv*/*own* discipline to generics. We choose a C#-like language [26] in which casts and type tests involve full types, not just class names as in GJ [10, 24]. Class names are used in the *inv*/*own* discipline to track levels in the hierarchy of class declarations, so it turns out that type arguments are not needed to make the discipline work with generics. Our semantics is a straightforward denotational one in which type variables give rise to indexing over all ground instances. This works smoothly even though the language is quite expressive owing to F-bounded subclassing and methods with type parameters [27].

Representation independence. Consider the subclass of Queue declared in Figure 3. For any subclass X of Task, AQueue<X> is a subclass of Queue<X>. It maintains an array, actsks, of tasks which is used in an overriding declaration of runAll intended as an optimization. We've dropped runs. Method add exhibits a typical pattern: unpack to establish the condition in which a super call can be made (as the superclass unpacks from its own level); after that call, reestablish the current class invariant. (This imposes proof obligations on inheritance, see [5].)

Consider this revision of AQueue: instead of using one array, with null replacing tasks past their limit, use an array artsk of X together with a parallel array brtsk of booleans to indicate whether a task is past its limit. Either way, the arrays can be suitably compressed at appropriate times, and the choice of times is an internal design decision that may be made differently in the two versions.

```
class AQueue< X ◁ Task> extends Queue<X>{
  private Qnode<X>[ ] actsks;
  private int alen;
  . . .
  unit add(X t, int lim) {
    unpack self from AQueue;
    super.add(t,lim);
    actsks[alen] := self.getTsks(); alen := alen+1;
    pack self as AQueue; }
  unit runAll() {
    unpack self from AQueue;
    int i := alen-1;
    while i ≥ 0 do {
      X at := actsks[i];
      if (at ≠ null)
      then if (at.getCount() < at.getLimit())
           then at.run();
           else actsks[i] := null;
      i := i-1; }
    pack self as AQueue;}}
```

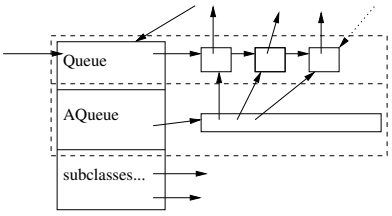
Fig. 3. Class AQueue. An invariant: actsks[0..alen) contains any n in tsks with n.count $<$ n.limit, in reverse order. (There may also be nulls and some n with n.count $=$ n.limit).

Clearly the fields of AQueue should be private if we are to revise its implementation but retain compatibility of syntactic interfaces, i.e., if we wish to reason modularly about the class without regard to all its actual uses. To justify that the new version behaves the same as the old one, we argue as follows. Let $filt1(o.\mathsf{actsks})$ be the sequence of non-null elements of $o.$actsks with count $<$ limit. Let $filt2(ts, bs)$ take an array ts of tasks and a same-length array bs of booleans and return the subsequence of n in ts where bs is true and $n.\mathsf{count} < n.\mathsf{limit}$. Consider the following relation that connects a state for an instance o of the original implementation (Table 3) with an instance o' for the alternative:

$$filt1(o.\mathsf{actsks}) = filt2(o'.\mathsf{artsk}, o'.\mathsf{brtsk})$$

The idea is that methods of the new version behave the same as the old version, modulo this change of representation. Formally, from a pair of states related this way, execution of the corresponding method bodies leads to a related pair. In fact this is not likely to be true unless we also conjoin the invariants associated with the two versions, e.g., the second version requires artsks.length=brtsks.length.

Some theories treat simulations as relations on the global state space, but this is impractical for our situation: the programmer thinks in terms of a single instance and the objects on which it depends —what we call its *island*. Below is a picture of an instance of some subclass of AQueue, sliced into the fields of Queue, AQueue, and subclasses; on the right are objects encapsulated at the two levels relevant to reasoning about AQueue.



A relation between two such islands, called a *basic coupling*, should determine a relation on global states, just as (1) gives a global predicate based on predicates $\mathcal{I}^C(o)$ that depend only on fields of o and objects owned by it, at class C and superclasses.

(Aside: Note that the dotted pointer in the picture is allowed by the discipline but disallowed by ownership confinement disciplines.)

Suppose we are convinced that the two versions are correct in the sense of preserving the basic coupling (we explore this in detail later). We would like to conclude that, no matter how many instantiations are made of AQueue, and no matter what subclasses are declared, no client program can distinguish between one implementation and the other. Such a conclusion depends on encapsulation, as otherwise leaked references to internals could be used to spoil the connection. Moreover it is a nontrivial property of the language's constructs. For example, we do not allow pointer arithmetic, for if we did then clients could observe changes in memory layout (e.g., the addition of a private field). Our main result is that preservation of a basic coupling by methods of AQueue is indeed sufficient. The absence of pointer arithmetic is embodied in the theory by a bijection on locations as part of the notion of coupling. This bijection is also used to help express the encapsulation properties needed.

Adapting the discipline. As for proving correspondence between two versions of a method body, it is possible to reason directly in terms of the semantics [16, 19] or a special logic of relation-preservation [7, 49]. Another alternative is to rename things to create disjoint pieces that can be combined and subject to ordinary program logic: This can be seen as two transformations, first to introduce additional variables, then to revise some parts to use the new variables (treating the coupling as an ordinary invariant), and finally eliminating the old variables which are now superfluous [40, 20, 34]. We are unaware of the transform approach having been worked out in detail for the complications of OO languages. In any approach, we confront the problem of reentrant callbacks. The theorem of [2] says that the proof obligation is to show the methods of AQueue preserve the coupling under the assumption that any methods called do so. But in the middle of a method body, the coupling may not hold, in which case the hypothesis about calls is inapplicable.

The *inv*/*own* discipline works beautifully here. Because *inv* is part of the behavior observable at the level of reasoning, we can assume both versions follow the same pattern of packing (though not necessarily of control structure). The basic coupling serves to describe a corresponding pair of objects when they are in steady state. The induced coupling relation on global states requires corresponding pairs o, o' of objects to satisfy not only $o.inv = o'.inv$ but also $o.inv \leq AQueue \Rightarrow BC(o, o')$ where BC is the basic coupling (a relation on not only o, o' but also their transitively owned objects, i.e., their respective islands.).

Several features of the discipline need to be adapted, in ways which also make sense in terms of informal considerations of information hiding. The discipline does not constrain field access, as reading cannot falsify an invariant predicate. But simulation breaks if a client can read fields

of encapsulated objects, so we impose a precondition on field access: outside of the code of the class Abs under revision, we do not allow access to fields of objects transitively owned by objects of class Abs. Moreover, fields of Abs are private (though others are public in this paper). Because this precondition is expressed in terms of transitive ownership, it is less amenable for direct verification than the conditions in Table 1, but a lightweight confinement discipline such as package confinement [48] can be used to establish the ownership transfer preconditions automatically in sensible code. Yet there remains the possibility to verify tricky code that is beyond the reach of conservative automated analysis.

Another problematic feature is that "**pack** e **as** C" can occur in any class, so long as its preconditions are established. But for "**pack** e **as** Abs", to establish the precondition for a simulation means establishing the basic coupling; such a command only makes sense in code of Abs.

Although the discipline supports hierarchical ownership, our technical treatment benefits from heap partitioning ideas from separation logic (we highlight the connections where possible, e.g., in Proposition 2). For this reason, it is convenient to prevent an instance of Abs from transitively owning another instance of Abs (lest their "islands" be nested). This can be achieved by a simple syntactic restriction. It does not preclude that, say, class AQueue can be instantiated with the element type X itself being (or containing) type AQueue, because an instance of AQueue owns its representation objects (the Qnodes), not the tasks they contain.

The most challenging feature is ownership transfer. Transfer between instances of an abstraction occurs, for example, in load balancing for task queues [4, 28]. The hardest case is where a hitherto-encapsulated object is released to a client, e.g., when a memory manager allocates nodes from a free list [38, 4]. This can be seen as a deliberate exposure of representation and thus is observable behavior that must be retained in a revised version. We have solved the technical problems; as with **pack**, we use a precondition so that outward transfer of objects encapsulated for Abs can occur only in code of Abs (but we lack other compelling examples).

3 Language

3.1 Syntax

This section formalizes the language, adapting notations and typing rules from Featherweight Generic Java [24], adding imperative features and the special commands. Barred identifiers like $\bar{T}$ indicate finite lists, e.g., $\bar{T}\ \bar{f}$ stands for a list $\bar{f}$ of field names with corresponding types $\bar{T}$.

In most respects self and result are like any other variables but self cannot be the target of assignment; the final value of result serves as the result returned by a method.

A class type has the form $C\langle\bar{T}\rangle$ where $\bar{T}$ is a list of types which may include variables. We assume that class Object$\langle\rangle$ has no parameters but distinguish it from the bare class name Object. A program consists of a class table CT that maps a class name C to a declaration $CT(C)$ of the form

$$\textbf{class}\ C\langle\bar{X} \lhd \bar{N}\rangle \lhd N\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \} \qquad (*)$$

The categories N, T, M are given by this grammar:

$C, D \in \mathit{ClassName}$	$X, Y \in \mathit{TypeVar}$	$m \in \mathit{MethName}$
$f \in \mathit{FieldName}$	$x, \mathsf{self}, \mathsf{result} \in \mathit{VarName}$	
$T ::=$ **bool** $\mid$ **unit** $\mid X \mid C\langle\bar{T}\rangle$		types (also U, V)
$N ::= C\langle\bar{T}\rangle$		nonvariable class type
$M ::= \langle\bar{X} \lhd \bar{N}\rangle\ T\ m(\bar{T}\ \bar{x})\ \{S\}$		method declaration
$S ::= x := e \mid e.f := e$		assign to var., to field
$\mid\ x :=$ **new** N		object construction
$\mid\ x := e.m\langle\bar{T}\rangle(\bar{e})$		method call
$\mid\ T\ x := e$ **in** $S \mid S;\ S$		local variable, sequence
$\mid$ **if** e **then** S **else** S **fi**		conditional
$\mid$ **pack** e **as** C		set *inv* to C
$\mid$ **unpack** e **from** C		set *inv* to *super* C
$\mid$ **setown** e **to** (e', C)		set $e.own$ to (e', C)
$\mid$ **assert** $\mathcal{P}$		assert (see text for $\mathcal{P}$)
$e ::= x \mid$ **null** $\mid$ **true** $\mid$ **false**		variable, constant
$\mid\ e.f \mid e = e$		field access, ptr. equality
$\mid\ e$ **is** $N \mid (N)\ e$		type test, cast

A declaration of method named m with body S takes the form $\langle\bar{X} \lhd \bar{N}\rangle\ T\ m(\bar{T}\ \bar{x})\ \{S\}$ which binds additional type parameters $\bar{X}$ in the parameter types $\bar{T}$ and return type T. In the formal language, expressions do not have side effects. Method call is not an expression but rather a command that assigns the result value to a variable. Object construction occurs only as a command $x :=$ **new** C. Fields are initialized to "0 equivalent" default values: false for **bool**, null for object types, and the unique value it of type **unit** (called `void` in some languages; mathematically this is odd though perhaps the word "unit" has to many meanings).

A class declaration, see $(*)$ above, is parameterized by a list $\bar{X}$ of type parameters; the bounds $\bar{N}$ are nonvariable types which may contain variables in $\bar{X}$. Typing judgements carry a context Δ which is a finite mapping of type names to nonvariable types. We write $X \lhd N$ for the mapping $X \mapsto N$.[4] As is enforced in the typing rules, the parameters of a class are bound throughout the class declaration.

The type of an actual object instance is variable-free, and the meaning of a parameterized class is defined in terms of its ground instances. Our language follows $C\#$ in allowing runtime cast and test for exact types rather than just the class name as in GJ. Useful discussion and results on the interaction between type parameters and subtyping can be found in [24, 50, 27]

Definition 1 (subtyping). For any Δ, the subtyping relation $\Delta \vdash - \leq -$ on types is defined by

- $\Delta \vdash X \leq \Delta\, X$ for all $X \in dom\Delta$
- $\Delta \vdash C\langle\bar{T}\rangle \leq [\bar{T}/\bar{X}]N$ for any $\bar{T}$ and declaration **class** $C\langle\bar{X} \lhd \bar{N}\rangle \lhd N\ \{\ \ldots\ \}$
- $\Delta \vdash T \leq T$
- $\Delta \vdash T \leq V$ if $\Delta \vdash T \leq U$ and $\Delta \vdash U \leq V$

Here $[\bar{T}/\bar{X}]$ denotes simultaneous substitution of $\bar{T}$ for $\bar{X}$. Note that $C\langle\bar{T}\rangle \leq [\bar{T}/\bar{X}]N$ holds regardless of whether $\bar{T}$ satisfies the declared constraint $\bar{X} \lhd \bar{N}$. Note that $\Delta \vdash C\langle T\rangle \leq C\langle U\rangle$

[4] The symbol $\leq$ makes more sense for a bound, as $\lhd$ is used in other contexts for direct extension rather than subtype; but $\leq$ looks unpleasant written inside angle brackets.

does not follow from $T \leq U$. Sometimes we will be interested in the inheritance order on class-names, for which we define $C \leq D$ iff $\Delta \vdash C\langle\bar{T}\rangle \leq D\langle\bar{U}\rangle$ for some $\Delta, \bar{T}, \bar{U}$. To facilitate distinguishing between the two uses of "$\leq$" we define $name(C\langle\bar{T}\rangle) = C$. A *typing context* Γ is a finite mapping from variable and parameter names to types, such that $\mathsf{self} \in dom\,\Gamma$.

Definition 2 (well formed types and contexts). The judgement $\Delta \vdash -ok$ for *well formed types* is defined inductively by the following rules. Lists on the right side of judgements abbreviate multiple judgements: $\Delta \vdash \bar{N}, N\ ok$ means $\Delta \vdash N\ ok$ and $\Delta \vdash U\ ok$ for all U in $\bar{N}$. In the definition of $\Delta \vdash C\langle\bar{T}\rangle\ ok$, the conditions $\bar{X} \lhd \bar{N} \vdash \bar{N}, N\ ok$ ensure that all variables in the bounds $\bar{N}$ and superclass N of a class type are bound by the parameters $\bar{X}$ of the class.[5]

- $\Delta \vdash \mathsf{Object}\langle\rangle\ ok$
- if $X \in dom\,\Delta$ then $\Delta \vdash X\ ok$
- if C is declared by **class** $C\langle\bar{X} \lhd \bar{N}\rangle \lhd N\ \{\ \ldots\ \}$ then $\Delta \vdash C\langle\bar{T}\rangle\ ok$ provided that $\Delta \vdash \bar{T}\ ok$, $\Delta \vdash \bar{T} \leq [\bar{T}/\bar{X}]\bar{N}$, and $\bar{X} \lhd \bar{N} \vdash \bar{N}, N\ ok$.

We say Δ *is well formed* if $\Delta \vdash (\Delta\, X)\ ok$ for all $X \in dom\Delta$, and Γ *is well formed in* Δ if $\Delta \vdash (\Gamma\, x)\ ok$ for all $x \in dom\Gamma$.

We define the *ground class types* by $R ::= C\langle\bar{R}\rangle$ and also use identifiers P, Q for such types. The base cases of this recursion are classes with an empty argument list, e.g., $\mathsf{Object}\langle\rangle$. For ground types R and Q, $\Delta \vdash R \leq Q$ is independent from Δ and we write $R \leq Q$ without context. We can apply a substitution σ to context Γ by function composition: $(\sigma\Gamma)\, x = \sigma(\Gamma\, x)$.

A *ground instantiation* of Δ is a ground substitution σ with $dom\,\sigma = dom\,\Delta$ and $\sigma X \leq \sigma(\Delta X)$ for all $X \in dom\Delta$. We write $grnd\,\Delta$ for the set of ground instantiations of Δ. We write $instantiation(D, C\langle\bar{R}\rangle)$ for the $D\langle\bar{Q}\rangle$ given by (b) of the following.

Lemma 1. (a) If $\Delta \vdash C\langle\bar{X}\rangle \leq D\langle\bar{T}\rangle$ and $\Delta \vdash C\langle\bar{X}\rangle\ ok$ then all variables in $\bar{T}$ occur in $\bar{X}$. (b) If $C \leq D$ and $\vdash C\langle\bar{R}\rangle\ ok$ then there is unique $\bar{Q}$ such that $\vdash D\langle\bar{Q}\rangle\ ok$ and $C\langle\bar{R}\rangle \leq D\langle\bar{Q}\rangle$.

Lemma 2. Both the extension relation $\trianglelefteq$ and the subtyping relation $\leq$ has the tree property: if $\Delta \vdash U \leq T_1$ and $\Delta \vdash U \leq T_2$ then $\Delta \vdash T_1 \leq T_2$ or $\Delta \vdash T_2 \leq T_1$.

Well formed class tables are characterized using typing rules which are expressed using some auxiliary functions that in turn depend on the class table, allowing classes to make mutually recursive references to other classes, without restriction. In particular, this allows recursive methods (so we omit loops). Table 2 defines some auxiliary functions. Note that for well formed ground type $C\langle\bar{R}\rangle$, all types in *fields*$(C\langle\bar{R}\rangle)$ are ground.

For use in the semantics, we extend *fields*$(C\langle\bar{U}\rangle)$ to *xfields*$(C\langle\bar{U}\rangle)$ that also assigns "types" to the auxiliary fields: *com* : **bool**, *own* : owntyp, and *inv* : $(\mathsf{invtyp}\, C)$. Neither $\mathsf{invtyp}\, C$ nor owntyp are types in the programming language but there are corresponding semantic domains and the slight notational abuse is convenient.

Typing of commands for methods declared in class $C\langle\bar{X}\rangle$ is expressed using judgements $\Delta; \Gamma \vdash S$ where $\Gamma\,\mathsf{self} = C\langle\bar{X}\rangle$.

[5] In FGJ, this is not explicitly imposed in the definition of ok, but it may as well be because it is imposed in the typing rule for classes. It follows directly from our definition of ok that if $\Delta \vdash T\ ok$ then every variable in T has a bound in Δ. Different from FGJ, we use invariant subtyping for method overriding and omit renaming of parameters.

Given a class declaration, **class** $C\langle\bar{X} \lhd \bar{N}\rangle \lhd N\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}$, define

$\mathit{dfields}(C\langle\bar{U}\rangle) = (\bar{f} : [\bar{U}/\bar{X}]\bar{T})$	declared fields
$\mathit{fields}(C\langle\bar{U}\rangle) = \mathit{dfields}(C\langle\bar{U}\rangle) \cup \mathit{fields}([\bar{U}/\bar{X}]N)$	inherited and declared fields
$\mathit{super}(C\langle\bar{U}\rangle) = [\bar{U}/\bar{X}]N$	superclass instance
$\mathit{super}\, C = name\ N$	superclass name

In the context of the above class, define, for a method declaration $\langle\bar{X}_1 \lhd \bar{N}_1\rangle\ T\ m(\bar{T}_1\ \bar{x})\ \{S\}$

$\mathit{mtype}(m, C\langle\bar{U}\rangle) = [\bar{U}/\bar{X}](\langle\bar{X}_1 \lhd \bar{N}_1\rangle\bar{T}_1 \to T)$	instantiated method type
$\mathit{pars}(m, C) = \bar{x}$	parameter names
$\mathit{pars}(m, C\langle\bar{U}\rangle) = \bar{x}$	parameter names

For m inherited in C, define $\mathit{mtype}(m, C\langle\bar{U}\rangle) = \mathit{mtype}(m, [\bar{U}/\bar{X}]N)$ and $\mathit{pars}(m, C) = \mathit{pars}(m, N)$.

Table 2. Auxiliary functions on syntax.

Definition 3 (well formed class table). Class table CT is well formed if each class declaration $CT(C)$ is well formed according to the following rule.

$$\frac{\bar{X} \lhd \bar{N} \vdash N\ ok \qquad C\langle\bar{X} \lhd \bar{N}\rangle \lhd N\ \vdash\ M \quad \text{for each } M \in \bar{M}}{\vdash \textbf{class}\ C\langle\bar{X} \lhd \bar{N}\rangle \lhd N\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}}$$

The judgement for well formed M is expressed using the class header "$C\langle\bar{X} \lhd \bar{N}\rangle \lhd N$" as context and the rule is as follows:

$$\frac{\begin{array}{c}\bar{X} \lhd \bar{N}, \bar{X}_1 \lhd \bar{N}_1 \vdash \bar{T}, T, \bar{N}\ ok \qquad \bar{X} \lhd \bar{N}, \bar{X}_1 \lhd \bar{N}_1; \bar{x} : \bar{T}, \mathsf{self} : C\langle\bar{X}\rangle, \mathsf{result} : T \vdash S \\ \mathit{mtype}(m, N) \text{ is undefined or equals } \langle\bar{X}_1 \lhd \bar{N}_1\rangle\bar{T} \to T \\ \mathit{pars}(m, N) \text{ is undefined or equals } \bar{x}\end{array}}{C\langle\bar{X} \lhd \bar{N}\rangle \lhd N\ \vdash\ \langle\bar{X}_1 \lhd \bar{N}_1\rangle\ T\ m(\bar{T}\ \bar{x})\{S\}}$$

Selected rules for expressions and commands are given in Table 3. These rules may be instantiated only with well formed $\Delta; \Gamma$ and with types that are well formed in Δ. This stipulation allows us to omit explicit ok-judgements in the rules for cast, test, local variable, method call, and object construction where types occur in expressions or commands. The bound of a class type in a context Δ is defined by $bound\ \Delta\ X = \Delta\ X$ and $bound\ \Delta\ N = N$. [6]

In typing rules and elsewhere, we often write "$=$" between expressions involving partial functions. It means strong equality: both sides are defined and equal.

Selected typing rules appear in Table 3.

To formalize assertions, we prefer to avoid both the commitment to a particular formula language and the complication of an environment for declaring predicate names to be interpreted in the semantics. So we indulge in a mild and commonplace abuse of notation: the syntax of $\mathbf{assert}$ uses a semantic predicate, or rather a generic family thereof. We say $\Delta; \Gamma \vdash$ **assert** $\mathcal{P}$ is well formed provided that $\mathcal{P}\,\sigma$ is a set of program states for each σ. (In terms of the semantics defined

[6] Because identifiers T, U, V range over types including primitives, the rules for field access and method call might appear to allow treatment of primitives as objects. But in fact these rules cannot be instantiated at primitive types because for them the auxiliary functions (*dfields*, *mtype*) are not defined.

$$\Delta; \Gamma \vdash x : \Gamma x \qquad \frac{\Delta; \Gamma \vdash e : U \qquad (f : T) \in \mathit{fields}(\mathit{bound}\,\Delta\,U)}{\Delta; \Gamma \vdash e.f : T}$$

$$\frac{\Delta; \Gamma \vdash e : T \qquad \Delta \vdash N \leq \mathit{bound}\,\Delta\,T}{\Delta; \Gamma \vdash (N)\ e : N} \qquad \Delta; \Gamma \vdash \textbf{null} : N$$

$$\frac{\Delta; \Gamma \vdash e : T \qquad \Delta \vdash N \leq \mathit{bound}\,\Delta\,T}{\Delta; \Gamma \vdash e\ \textbf{is}\ N : \textbf{bool}} \qquad \frac{\Delta \vdash N \leq \Gamma x \qquad x \neq \mathsf{self} \qquad N \neq \mathsf{Object}\langle\rangle}{\Delta; \Gamma \vdash x := \textbf{new}\ N}$$

$$\frac{\Delta; \Gamma \vdash e_1 : V \qquad (f : T) \in \mathit{fields}(\mathit{bound}\,\Delta\,V) \qquad \Delta; \Gamma \vdash e_2 : U \qquad \Delta \vdash U \leq T}{\Delta; \Gamma \vdash e_1.f := e_2}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e : T \qquad \mathit{mtype}(m, (\mathit{bound}\,\Delta\,T)) = \langle \bar{X} \lhd \bar{N} \rangle \bar{T} \to U \\ x \neq \mathsf{self} \qquad \Delta; \Gamma \vdash \bar{e} : \bar{U} \qquad \Delta \vdash \bar{U} \leq [\bar{V}/\bar{X}]\bar{T} \qquad \Delta \vdash [\bar{V}/\bar{X}]U \leq \Gamma x \qquad \Delta \vdash \bar{V} \leq [\bar{V}/\bar{X}]\bar{N}\end{array}}{\Delta; \Gamma \vdash x := e.m\langle \bar{V} \rangle(\bar{e})}$$

$$\frac{\Delta; \Gamma \vdash e : N \quad \mathit{name}\,N \leq C}{\Delta; \Gamma \vdash \textbf{pack}\ e\ \textbf{as}\ C} \qquad \frac{\Delta; \Gamma \vdash e : N \quad \mathit{name}\,N \leq C}{\Delta; \Gamma \vdash \textbf{unpack}\ e\ \textbf{from}\ C}$$

$$\frac{\Delta; \Gamma \vdash e_1 : N_1 \qquad \Delta; \Gamma \vdash e_2 : N_2 \qquad \mathit{name}\,N_2 \leq C}{\Delta; \Gamma \vdash \textbf{setown}\ e_1\ \textbf{to}\ (e_2, C)}$$

Table 3. Typing rules for selected expressions and commands.

in the sequel, the precise statement is: $\mathcal{P}\,\sigma \subseteq [\![\mathsf{heap} \otimes \sigma\Gamma]\!]$ for all $\sigma \in \mathit{grnd}\,\Delta$.) This treatment of assertions is also convenient for taking advantage of a theorem prover's native logic [23, 25].

In the rest of the paper, we often assume types and contexts are well formed, and that typings are derivable, without explicit mention. By referring to "instances of $C\langle\bar{X}\rangle$" we mean that $\bar{X}$ are the declared parameters of C and that well formed instances are considered.

3.2 Semantics

A *global state* consists of a *heap* h, i.e., a finite partial function from locations to object states, and a *store* s, which assigns locations and primitive values to the local variables and parameters given by a typing context Γ. An *object state* is a mapping from field names to values. A *pre-heap* is like a heap except for possibly having dangling references. If h, h' are pre-heaps with disjoint domains then we write $h * h'$ for their union; otherwise $h * h'$ is undefined.

Function application associates to the left, so $h\,o\,f$ is the value of field f of the object $h\,o$ at location o. We also write $h\,o.f$. Application binds more tightly than binary operator symbols and ",".

A command denotes a function mapping each initial state (h, s) either to a final state (h_0, s_0) or to the distinguished value $\perp$ which represents runtime errors, divergence, and assertion failure.

For locations, we assume that a countable set Loc is given, along with a distinguished value nil not in Loc. Instances have ground type so we assume given a function type from Loc to

$$
\begin{array}{ll}
[\![R]\!] & = \{nil\} \cup \{p \in Loc \mid type\, p \leq R\} \\
[\![\textbf{bool}]\!] & = \{true, false\} \\
[\![\textbf{unit}]\!] & = \{it\} \\
[\![\mathsf{invtyp}\, C]\!] & = \{B \mid C \leq B\} \\
[\![\mathsf{owntyp}]\!] & = \{(o, C) \mid o = nil \vee name(type\, o) \leq C\} \\
[\![\mathsf{state}\, R]\!] & = \{s \mid dom\, s = dom(\mathit{xfields}\, R) \ \wedge\ \forall (f : R_1) \in \mathit{xfields}\, R \mid s\, f \in [\![R_1]\!]\} \\
[\![\mathsf{pre\text{-}heap}]\!] & = \{h \mid dom\, h \subseteq_{\mathit{fin}} Loc \ \wedge\ \forall o \in dom\, h \mid h\, o \in [\![\mathsf{state}\, (type\, o)]\!]\} \\
[\![\mathsf{heap}]\!] & = \{h \mid h \in [\![\mathsf{pre\text{-}heap}]\!] \ \wedge\ \forall s \in rng\, h \mid rng\, s \cap Loc \subseteq dom\, h\} \\
[\![\Gamma^r]\!] & = \{s \mid dom\, s = dom\, \Gamma^r \ \wedge\ s\, \mathsf{self} \neq nil \ \wedge\ \forall x \in dom\, s \mid s\, x \in [\![\Gamma^r\, x]\!]\ \} \\
[\![\mathsf{heap} \otimes \Gamma^r]\!] & = \{(h, s) \mid h \in [\![\mathsf{heap}]\!] \ \wedge\ s \in [\![\Gamma^r]\!] \ \wedge\ rng\, s \cap Loc \subseteq dom\, h\} \\
[\![\mathsf{heap} \otimes T]\!] & = \{(h, v) \mid h \in [\![\mathsf{heap}]\!] \ \wedge\ v \in [\![T]\!] \ \wedge\ (v \in Loc \Rightarrow v \in dom\, h)\} \quad \text{for ground } T \\
[\![(\Gamma^r \vdash \mathsf{command})]\!] & = [\![\mathsf{heap} \otimes \Gamma^r]\!] \to [\![(\mathsf{heap} \otimes \Gamma^r)_\perp]\!] \\
[\![(\Gamma^r \vdash R)]\!] & = \{v \mid v \in ([\![\mathsf{heap} \otimes \Gamma^r]\!] \to [\![R]\!]_\perp) \ \wedge\ \forall h, s \mid v(h, s) \in Loc \Rightarrow v(h, s) \in dom\, h\} \\
[\![(R,\ \bar{R} \to R_1)]\!] & = [\![\mathsf{heap} \otimes (\bar{x} : \bar{R}, \mathsf{self} : R)]\!] \to [\![(\mathsf{heap} \otimes R_1)_\perp]\!] \quad \text{where } \bar{x} = \mathit{pars}(m, R)
\end{array}
$$

Table 4. Semantic domains. See Definition 4 for [[meth-env]]. For state R, the range of $f : R_1$ includes com : **bool**, inv : invtyp($name\ R_1$), and own : owntyp.

ground, non-primitive types distinct from Object<>, such that for each C<$\bar{R}$> there are infinitely many locations o with $type\ o = C$<$\bar{R}$>. This is used in a way that is equivalent to tagging object states with their type.

Some semantic domains correspond directly to (ground instantiations of) the syntax. For example, each ground data type R denotes a set $[\![R]\!]$ of values. For lack of a symbol, we use decorated Γ^r for ground context, which denotes a set $[\![\Gamma^r]\!]$ of stores. The semantics, and later the coupling relation, is structured in terms of category names θ given as follows.[7]

$\theta ::= R \mid \Gamma^r \mid \theta_\perp$

$\mid$	owntyp $\mid$ invtyp C	lift, *own* and *inv* val.
$\mid$	state $R \mid$ pre-heap	obj. state, heap frag.
$\mid$	heap $\mid$ heap $\otimes\ \Gamma^r \mid$ heap $\otimes\ R$	closed heap, state
$\mid$	$(\Gamma^r \vdash \mathsf{command}) \mid (\Gamma^r \vdash R)$	command, expr.
$\mid$	$(R,\ \bar{R} \to R_1) \mid$ meth-env	method, methods

The semantic domains are defined in Table 4. They embody important invariants: the value in a field has its declared type and there are no dangling pointers.

Subtyping is embodied in a simple way: if $\Delta \vdash T \leq U$ and $\sigma \in grnd\, \Delta$ then $[\![\sigma\, T]\!] \subseteq [\![\sigma\, U]\!]$.

The meaning of a derivable command typing $\Delta; \Gamma \vdash S$ will be defined to be a (curried) function sending each $\sigma \in grnd\, \Delta$ and method environment μ to an element of $[\![(\sigma\Gamma \vdash \mathsf{command})]\!]$. That is, $[\![\Delta; \Gamma \vdash S]\!]\sigma\mu$ is a state transformer $[\![\mathsf{heap} \otimes \sigma\Gamma]\!] \to [\![(\mathsf{heap} \otimes \sigma\Gamma)_\perp]\!]$. Similarly, for $\sigma \in grnd\, \Delta$ the meaning $[\![\Delta; \Gamma \vdash e : T]\!]\sigma$ is an element of $[\![\mathsf{heap} \otimes \sigma\Gamma]\!] \to [\![\sigma T_\perp]\!]$.

[7] The notation $\Gamma^r \vdash R$ for the category of expressions uses ground class type R but we also use the $\Gamma^r \vdash T$ with T a primitive type and likewise for method arguments/results $\bar{R} \to R_1$. Rather than clutter notation, we leave it to the reader to note situations where primitive types are included.

$[\![\Delta; \Gamma \vdash x : T]\!]\sigma(h,s) = s\,x$

$[\![\Delta; \Gamma \vdash e.f : T]\!]\sigma(h,s) = \mathsf{let}\ o = [\![\Delta; \Gamma \vdash e : U]\!]\sigma(h,s)\ \mathsf{in}\ \mathsf{if}\ o = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else}\ h\,o.f$

$[\![\Delta; \Gamma \vdash (N)\ e : N]\!]\sigma(h,s) = \mathsf{let}\ o = [\![\Delta; \Gamma \vdash e : D]\!]\sigma(h,s)\ \mathsf{in}$
$\quad \mathsf{if}\ o = \mathit{nil} \vee \mathit{type}\ o \leq \sigma N\ \mathsf{then}\ o\ \mathsf{else}\ \bot$

$[\![\Gamma \vdash e\ \mathbf{is}\ N : \mathbf{bool}]\!]\sigma(h,s) = \mathsf{let}\ o = [\![\Gamma \vdash e : D]\!]\sigma(h,s)\ \mathsf{in}$
$\quad \mathsf{if}\ o \neq \mathit{nil} \wedge \mathit{type}\ o \leq \sigma N\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ \mathit{false}$

$[\![\Delta; \Gamma \vdash x := e]\!]\sigma\mu(h,s) = \mathsf{let}\ v = [\![\Delta; \Gamma \vdash e : T]\!]\sigma(h,s)\ \mathsf{in}\ (h, [s \mid x \mapsto v])$

$[\![\Delta; \Gamma \vdash e_1.f := e_2]\!]\sigma\mu(h,s) = \mathsf{let}\ o = [\![\Delta; \Gamma \vdash e_1 : V]\!]\sigma(h,s)\ \mathsf{in}$
$\quad \mathsf{if}\ o = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else}$
$\quad \mathsf{let}\ v = [\![\Delta; \Gamma \vdash e_2 : U]\!]\sigma(h,s)\ \mathsf{in}\ ([h \mid o.f \mapsto v], s)$

$[\![\Delta; \Gamma \vdash x := \mathbf{new}\ N]\!]\sigma\mu(h,s) = \mathsf{let}\ o = \mathit{fresh}(\sigma N, h)\ \mathsf{in}$
$\quad \mathsf{let}\ h_0 = [h \mid o \mapsto [\mathit{xfields}(\sigma N) \mapsto \mathit{defaults}(\sigma N)]]\ \mathsf{in}\ (h_0, [s \mid x \mapsto o])$

$[\![\Delta; \Gamma \vdash x := e.m\langle \bar{V} \rangle(\bar{e})]\!]\sigma\mu(h,s) = \mathsf{let}\ o = [\![\Delta; \Gamma \vdash e : T]\!]\sigma(h,s)\ \mathsf{in}\ \mathsf{if}\ o = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else}$
$\quad \mathsf{let}\ \bar{v} = [\![\Delta; \Gamma \vdash \bar{e} : \bar{U}]\!](h,s)\ \mathsf{in}$
$\quad \mathsf{let}\ \bar{x} = \mathit{pars}(m,T)\ \mathsf{in}\ \mathsf{let}\ s_1 = [\bar{x} \mapsto \bar{v}, \mathsf{self} \mapsto o]\ \mathsf{in}$
$\quad \mathsf{let}\ (h_1, v_1) = \mu(\mathit{type}\ o)m(\sigma\bar{V})(h, s_1)\ \mathsf{in}\ (h_1, [s \mid x \mapsto v_1])$

$[\![\Delta; \Gamma \vdash \mathbf{assert}\ \mathcal{P}]\!]\sigma\mu(h,s) = \mathsf{if}\ (h,s) \in \mathcal{P}\,\sigma\ \mathsf{then}\ (h,s)\ \mathsf{else}\ \bot$

$[\![\Delta; \Gamma \vdash \mathbf{pack}\ e\ \mathbf{as}\ C]\!]\sigma\mu(h,s) =$
$\quad \mathsf{let}\ q = [\![\Delta; \Gamma \vdash e : N]\!]\sigma(h,s)\ \mathsf{in}\ \mathsf{if}\ q = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else}$
$\quad \mathsf{let}\ h_1 = \lambda p \in \mathit{dom}\ h \mid \mathsf{if}\ h\,p.\mathit{own} = (q, C)\ \mathsf{then}\ [h\,p \mid \mathit{com} \mapsto \mathit{true}]\ \mathsf{else}\ h\,p\ \mathsf{in}\ ([h_1 \mid q.\mathit{inv} \mapsto C],\ s)$

$[\![\Delta; \Gamma \vdash \mathbf{unpack}\ e\ \mathbf{from}\ C]\!]\sigma\mu(h,s) =$
$\quad \mathsf{let}\ q = [\![\Delta; \Gamma \vdash e : N]\!]\sigma(h,s)\ \mathsf{in}\ \mathsf{if}\ q = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else}$
$\quad \mathsf{let}\ h_1 = \lambda p \in \mathit{dom}\ h \mid \mathsf{if}\ h\,p.\mathit{own} = (q, C)\ \mathsf{then}\ [h\,p \mid \mathit{com} \mapsto \mathit{false}]\ \mathsf{else}\ h\,p\ \mathsf{in}\ ([h_1 \mid q.\mathit{inv} \mapsto \mathit{super}\,C],\ s)$

$[\![\Delta; \Gamma \vdash \mathbf{setown}\ e_1\ \mathbf{to}\ (e_2, C)]\!]\sigma\mu(h,s) = \mathsf{let}\ q = [\![\Delta; \Gamma \vdash e_1 : N_1]\!]\sigma(h,s)\ \mathsf{in}\ \mathsf{if}\ q = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else}$
$\quad \mathsf{let}\ p = [\![\Delta; \Gamma \vdash e_2 : N_2]\!]\sigma(h,s)\ \mathsf{in}\ ([h \mid q.\mathit{own} \mapsto (p, C)],\ s)$

Table 5. Semantics of selected expressions and commands. To streamline the treatment of $\bot$, the metalanguage expression "$\mathsf{let}\ \alpha = \beta\ \mathsf{in}\ \ldots$" denotes $\bot$ if β is $\bot$.

Meanings for expressions and commands are defined, in Table 5, by recursion on typing derivation. For expressions, the semantics is independent from the ground instantiation σ except for casts and type tests. For commands, σ is used only for method invocation and object construction. The semantics is defined for an arbitrary *allocator*, i.e., location-valued function *fresh* such that $\mathit{type}(\mathit{fresh}(R,h)) = R$ and $\mathit{fresh}(R,h) \notin \mathit{dom}\ h$.

The method environment is used only to interpret the method call command. The semantics for method meanings and method environments, is based on the following considerations. Like any command, a method call $\Delta; \Gamma \vdash x := e.m\langle \bar{V} \rangle(\bar{e})$, where $\Delta; \Gamma \vdash e : U$, is interpreted with respect to a substitution $\sigma \in \mathit{grnd}\,\Delta$. By typing, the free variables in $\bar{V}$ and in Γ are bound by Δ, so $\sigma\,\Gamma$ and $\sigma\,\bar{V}$ are ground. Consider execution of the call in initial state (h,s) and suppose that R is the type of the target object. That is, $R = \mathit{type}([\![\Delta; \Gamma \vdash e : U]\!]\sigma(h,s))$ and thus by type soundness $R \leq \sigma U$. Suppose $\mathit{mtype}(m,R) = \langle \bar{Y} \lhd \bar{N}_1 \rangle \bar{T} \to T$. Let $\bar{Q} = \sigma\,\bar{V}$. Consider the substitution $[\bar{Q}/\bar{Y}]$. By typing, both $[\bar{Q}/\bar{Y}]\bar{T}$ and $[\bar{Q}/\bar{Y}]T$ are ground. These give the ground

types at which the method body is interpreted. The meaning of the method body, used for the semantics of the call, is found in the method environment μ as $\mu\, R\, m\, \bar{Q}$.

Definition 4 (method environment). We define the set $[\![\mathsf{meth\text{-}env}]\!]$ to be the set of partial functions μ such that $\mu R m \bar{Q}$ is in $[\![R,\ [\bar{Q}/\bar{Y}](\bar{T}{\to}T)]\!]$ whenever $\mathit{mtype}(m, R) = {<}\bar{Y} \lhd \bar{N}_1{>}\bar{T}{\to}T$ and $\bar{Q} \leq [\bar{Q}/\bar{Y}]\bar{N}_1$.

Definition 5 (method declaration). Suppose M is a typable method declaration $C{<}\bar{X}{\lhd}\bar{N}{>}{\lhd}N \vdash M$ where $M = {<}\bar{Y} \lhd \bar{N}_1{>}\ T\ m(\bar{T}\ \bar{x})\{S\}$. For any $\bar{R}$ and $\bar{Q}$ such that $\bar{R} \leq [\bar{R}/\bar{X}]\bar{N}$ and $\bar{Q} \leq [\bar{Q}/\bar{Y}]\bar{N}_1$, define $[\![M]\!](C{<}\bar{R}{>})\bar{Q}$ to be a total function $[\![\mathsf{meth\text{-}env}]\!] \to [\![C{<}\bar{R}{>},\ [\bar{R}, \bar{Q}/\bar{X}, \bar{Y}](\bar{T}{\to}T)]\!]$ as follows.

$$
\begin{array}{l}
[\![M]\!](C{<}\bar{R}{>})\bar{Q}\mu(h, s) = \\
\quad \mathsf{let}\ \Delta = [\bar{X} \lhd \bar{N}, \bar{Y} \lhd \bar{N}_1]\ \mathsf{in} \\
\quad \mathsf{let}\ \sigma = [\bar{R}/\bar{X}, \bar{Q}/\bar{Y}]\ \mathsf{in} \\
\quad \mathsf{let}\ \Gamma = [\bar{x} : \sigma\bar{T}, \mathsf{self} : C{<}\bar{R}{>}, \mathsf{result} : \sigma T]\ \mathsf{in} \\
\quad \mathsf{let}\ s_1 = [s \mid \mathsf{result}{\mapsto}\, \mathit{defaults}(\sigma\, T)]\ \mathsf{in} \\
\quad \mathsf{let}\ (h_0, s_0) = [\![\Delta; \Gamma \vdash S]\!]\sigma\mu(h, s_1)\ \mathsf{in}\ (h_0, s_0\ \mathsf{result})
\end{array}
$$

Definition 6 (semantics of class table). For well formed class table CT, the semantics $[\![CT]\!]$ is the least upper bound of the ascending chain $\mu \in \mathbb{N} \to [\![\mathsf{meth\text{-}env}]\!]$ of method environments defined as follows.

$$
\begin{array}{l}
\mu_0\ (C{<}\bar{R}{>})\, m\, \bar{Q} = \lambda(h, s) \mid \bot \\
\mu_{j+1}\ (C{<}\bar{R}{>})\, m\, \bar{Q} = [\![M]\!](C{<}\bar{R}{>})\bar{Q}\mu_j \\
\quad \text{if } m \text{ is declared as } M \text{ in } C. \\
\mu_{j+1}\ (C{<}\bar{R}{>})\, m\, \bar{Q} = \mathit{restr}((\mu_{j+1}\ ([\bar{R}/\bar{X}]N)\, m\, \bar{Q}), C{<}\bar{R}{>}) \\
\quad \text{if } m \text{ is inherited in } C{<}\bar{X}{>} \text{ from } N.
\end{array}
$$

Here *restr* restricts the function $\mu_{j+1}\ ([\bar{R}/\bar{X}]N)\, m\, \bar{Q}$, which is defined on stores with $\mathsf{self} : [\bar{R}/\bar{X}]N$, to stores with $\mathsf{self} : C{<}\bar{R}{>}$. This works because $R_1 \leq R_2$ implies $[\![R_1]\!] \subseteq [\![R_2]\!]$ which induces an inclusion for stores.

Proposition 1 (type soundness). The semantic clauses for expressions, commands, method bodies, and method environment define elements of the designated semantic domains.

By contrast with [5, 30], we have taken care to separate the annotations required by the *inv*/*own* discipline from the semantics of commands. The invariants encoded in the semantic domains depend in no way on assertions, only typing. To prove type soundness, elements of the domains $(\Gamma^r \vdash \mathsf{command})$ and $(R,\ \bar{R}{\to}R_1)$ are subject to an additional requirement we omit from Table 4 for clarity: the domain of a result heap contains the domain of the initial heap. Type soundness has been machine checked in PVS for the same language and semantics but without generics, and we plan to extend that to generics.

3.3 Predicates

A *predicate* for some ground state type Γ^r is just a subset $\mathcal{P} \subseteq [\![\mathsf{heap} \otimes \Gamma^r]\!]$. For emphasis, $\models$ can be written instead of $\in$. Note that $\bot \notin \mathcal{P}$. We frequently convert a predicate on the single variable self into one independent of the heap: if $\mathcal{P} \subseteq [\![\mathsf{heap} \otimes \mathsf{self} : R]\!]$ and o is a location of type $\leq R$ then we write $\mathcal{P}(o)$ for the set of h such that $(h, [\mathsf{self} \mapsto o]) \in \mathcal{P}$. We give no formal

syntax to denote predicates but rather use informal metalanguage for which the correspondence should be clear. For example, $\mathsf{self}.f \neq \textbf{null}$ denotes the set of (h, s) with $h(s\,\mathsf{self}).f \neq nil$.

N.B.: quantification over objects (e.g., in Table 1 and Definition 10) is interpreted to mean quantification over allocated locations: $(h, s) \models \forall o \mid \mathcal{P}(o)$ iff for all $o \in dom\ h$, $(h, s) \models \mathcal{P}(o)$.

To formalize encapsulation we need precise semantic formulations concerning dependence. In terms of formulas, a predicate depends on $e.f$ if it can be falsified by some update of $e.f$. Some predicates are falsifiable by creation of new objects; an example is the predicate

$$\forall o \mid type\, o = C \;\Rightarrow\; o = \mathsf{self}$$

Definition 7 (depends, new-closed). Predicate $\mathcal{P}$ *depends on* f iff it depends on some $o.f$ in some (h, s). Predicate $\mathcal{P}$ *depends on* $o.f$ *in* (h, s) iff $(h, s) \in \mathcal{P}$, $o \in dom\ h$, and $([h \mid o.f \mapsto v], s) \notin \mathcal{P}$ for some v with $[h \mid o.f \mapsto v] \in [\![\mathsf{heap}]\!]$.

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

As in the Boogie papers and elsewhere [11], we require object invariants to be new-closed. This does not seem to be a restriction in practice, but see Pierik et al. [41] for arguments to the contrary.

4 The *inv*/*own* discipline

Definition 8 (transitive C- and $C\uparrow$-ownership). For any heap h, the relation $o \succeq^h_C p$ on $dom\ h$, read "*o owns p at C in h*", holds iff either $(o, C) = h\,p.\mathit{own}$ or there are q and D such that $(o, C) = h\,q.\mathit{own}$ and $q \succeq^h_D p$. The relation $o \succeq^h_{C\uparrow} p$ holds iff there is some D with $C \leq D$ and $o \succeq^h_D p$.

Definition 9 (admissible (generic) invariant). A predicate $\mathcal{P} \subseteq [\![\mathsf{heap} \otimes (\mathsf{self} : R)]\!]$ is *admissible as an invariant for* R 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 neither *inv* nor *com*, and one of the following conditions holds: $o = s(\mathsf{self})$ and f is in $dom(\mathit{xfields}\, C)$ or $s(\mathsf{self}) \succeq^h_{C\uparrow} o$ where $C = name\ R$.

An *admissible generic invariant for* $C\langle\bar{X}\rangle$ is a family, typically named $\mathcal{I}^C$, of predicates $\mathcal{I}^C_R$ indexed by ground instances R of $C\langle\bar{X}\rangle$, such that $\mathcal{I}^C_R$ is an admissible invariant for R.

For dependence on fields of self, the typing condition prevents an invariant for C from depending on fields declared in a subclass of C (which could be expressed in a formula using a cast). Similarly, an invariant can depend on any fields of objects owned at C or above.

We refrain from introducing syntax for declaring invariants.

Assumption 1 In the subsequent definitions, an admissible generic predicate $\mathcal{I}^C$ is assumed given for every class C. For Object, there is only one ground instance, Object$\langle\rangle$; we assume its invariant is everywhere true.

Definition 10 (disciplined, $\mathcal{PI}$). A heap h is *disciplined* if h satisfies the program invariant $\mathcal{PI}$ defined to be the conjunction of:

$$\forall o, C \mid o.\mathit{inv} \leq C \;\Rightarrow\; \mathcal{I}^C_{instantiation(C, type\, o)}(o) \tag{4}$$

$$\forall o, C, p \mid o.\mathit{inv} \leq C \wedge p.\mathit{own} = (o, C) \;\Rightarrow\; p.\mathit{com} \tag{5}$$

$$\forall o \mid o.\mathit{com} \;\Rightarrow\; o.\mathit{inv} = name(type(o)) \tag{6}$$

A state (h, s) is *disciplined* if h is. Method environment μ *is disciplined* provided that it maintains $\mathcal{PI}$ in the following sense: for any $R, m, \bar{Q}, h, s$, if $h \in \mathcal{PI}$ and $\mu\, R\, m\, \bar{Q}(h, s) = (h_0, v)$ (and thus $\mu\, R\, m\, \bar{Q}(h, s) \neq \bot$) then $h_0 \in \mathcal{PI}$.

Lemma 3 (transitive ownership). Suppose h is disciplined and $o \succeq^h_C p$. Then (a) $name(type\, o) \leq C$ and (b) $h\, o.inv \leq C$ implies $h\, p.com = true$.

Corollary 1. If h is disciplined, $o \succeq^h_C p$, and $h\, p.inv > name(type\, p)$ then $h\, o.inv > C$.

Proof. By $\mathcal{PI}(6)$, $h\, p.inv > name(type\, p)$ implies $h\, p.com = false$. Then by part (b) of the Lemma we have $h\, o.inv \not\leq C$. We have $name(type\, o) \leq C$ by part (a) and $name(type\, o) \leq h\, o.inv$ by type soundness so $h\, o.inv > C$ follows by the tree property of $\leq$.

Given an object $o \in dom\, h$ and class name A with $name(type\, o) \leq A$ we can partition h into pre-heaps Ah (the A-object), Rh (the representation of class A), Sh (objects owned by o at a superclass), and Fh (free from o) determined by the following conditions: Ah is the singleton $[o \mapsto h\, o]$, Rh is h restricted to the set of p with $o \succeq^h_A p$, Sh is h restricted to the set of p with $o \succeq^h_C p$ for some $C > A$, and Fh is the rest of h. Note that if $o \succeq^h_B p$ for some proper subclass $B < A$ then $p \in dom\, Fh$.

In these terms, dependency of admissible invariants can be described as follows.

Proposition 2 (island). Suppose $\mathcal{I}^C$ is an admissible invariant for C and $o \in dom\, h$ with $type\, o \leq R$ and $name\, R \leq C$. If $h = Fh * Ah * Rh * Sh$ is the partition defined above then $Fh_0 * Ah * Rh * Sh \models \mathcal{I}^C_R(o)$ iff $h \models \mathcal{I}^C_R(o)$, for all Fh_0 such that $Fh_0 * Ah * Rh * Sh$ is a heap.

To impose the stipulated preconditions of Table 1 we consider programs with the requisite syntactic structure (similar to formal proof outlines [1]).

Definition 11 (properly annotated). The *annotated commands* are the subset of the category of commands defined by the following grammar:

$$
\begin{array}{rl}
S ::= & \textbf{assert}\ \mathcal{P};\ \textbf{pack}\ e\ \textbf{as}\ C \\
\mid & \textbf{assert}\ \mathcal{P};\ \textbf{unpack}\ e\ \textbf{from}\ C \\
\mid & \textbf{assert}\ \mathcal{P};\ \textbf{setown}\ e\ \textbf{to}\ (e, C) \\
\mid & \textbf{assert}\ \mathcal{P};\ e.f := e \\
\mid & \textbf{assert}\ \mathcal{P} \\
\mid & x := \textbf{new}\ N \mid x := e.m\langle \bar{T} \rangle(\bar{e}) \\
\mid & T\ x := e\ \textbf{in}\ S \mid S;\ S \mid x := e \\
\mid & \textbf{if}\ e\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{fi}
\end{array}
$$

A *properly annotated command* is an annotated command such that each designated assertion implies (at each ground instance) the precondition stipulated in Table 1. A *properly annotated class table* is one such that each method body is a properly annotated command.

The effect of asserting the stipulated precondition is that when proving something about the associated command, we can confine attention to initial states (h, s) that satisfy that precondition.

For any class table and family of generic invariants there exists a proper annotation: just add **assert** commands with the stipulated preconditions. For practical purposes, of course, one wants assertions that can collectively be proved correct. For the abstraction theorem, $\mathcal{I}^C_R$ may as well be everywhere true for all C, R.

Theorem 2. If CT is a properly annotated class table then $[\![CT]\!]$ is disciplined.

The proof is similar to the one in [36], using the following.

Lemma 4 (disciplined commands). If μ is disciplined then any properly annotated command S maintains $\mathcal{PI}$ in the sense that for all (h, s), if $h \models \mathcal{PI}$ and $(h_0, s_0) = [\![\Delta; \Gamma \vdash S]\!]\sigma\mu(h, s)$ then $h_0 \models \mathcal{PI}$.

Proof. **Case of new.** Suppose $h \models \mathcal{PI}$ and $(h_0, s_0) = [\![\Delta; \Gamma \vdash x{:=}\,\textbf{new}\ C]\!]\sigma\mu(h, s)$. Suppose q is the fresh object, so that $h_0 = [h \mid q \mapsto \mathit{defaults}]$. We consider each of the predicates in $\mathcal{PI}$ in turn. For (4): $h_0\ q.\mathit{inv} = \mathsf{Object}$ by definition of the defaults. Because every admissible $\mathcal{I}_R^C$ is new-closed, adding q to the heap does not falsify (4) for existing objects. For (5) and (6) the argument is similar, noting that the default values give $h_0\ q.\mathit{com} = \mathit{nil}$ and $h_0\ q.\mathit{own} = (\mathit{nil}, \mathsf{Object})$.

Case of pack. Suppose that (h, s) is disciplined and $(h_0, s_0) = [\![\Delta; \Gamma \vdash \textbf{pack}\ e\ \textbf{as}\ C]\!]\sigma\mu(h, s)$. Let $q = [\![\Delta; \Gamma \vdash e : N]\!]\sigma(h, s)$ and $R = \mathit{instantiation}(C, \mathit{type}\, q)$. (Note that the latter equation is implicit in the stipulated precondition as given in Table 1.) Suppose moreover that (h, s) satisfies the stipulated preconditions, interpreted with respect to R, i.e., $q \neq \mathit{nil}$, $h\, q.\mathit{inv} = \mathit{super}\, C$, $h \models \mathcal{I}_R^C(q)$, $h \models \forall p \mid p.\mathit{own} = q \Rightarrow \neg p.\mathit{com} \wedge p.\mathit{inv} = \mathit{name}(\mathit{type}\, p)$. The static typing gives $e : N$ with $\mathit{name}\, N \leq C$.

For Definition 10(4): For any o, if $o \neq q$ then $\mathcal{I}(o)$ by $h \models \mathcal{PI}$, because **pack** only changes *inv* and *com* on which admissible invariants do not depend. If $o = q$ we have $\mathcal{I}_R^B(q)$, for all $B > C$, by precondition $h\, q.\mathit{inv} = \mathit{super}\, C$, as (4) holds in the initial heap h. And $\mathcal{I}_R^C(q)$ holds by precondition.

For (5): The only object for which the antecedent gets truthified in h_0 is q. And *com* is set true for the objects owned by q, by semantics of **pack**.

For (6): If $h\, o.\mathit{com} = \mathit{false}$ but $h_0\ o.\mathit{com} = \mathit{true}$ then $h\, o.\mathit{own} = q$ by semantics of **pack**; and $h_0\ o.\mathit{inv} = \mathit{name}(\mathit{type}\, o)$ by precondition.

Case of setowner Let $q = [\![e_1 : N_1]\!]\sigma(h, s)$ and $p = [\![e_2 : N_2]\!]\sigma(h, s)$, so that

$$[\![\Delta; \Gamma \vdash \textbf{setown}\ e_1\ \textbf{to}\ (e_2, C)]\!]\sigma\mu(h, s) = ([h \mid q.\mathit{own} \mapsto (p, C)],\ s)$$

The stipulated preconditions give $h\, q.\mathit{inv} = \mathsf{Object}$ and either $h\, p.\mathit{inv} > C$ or $p = \mathit{nil}$.

For (4): The update can only falsify (4) for some o, C' with $h\, o.\mathit{inv} \leq C'$ and $\mathcal{I}^{C'}$ dependent on $q.\mathit{own}$ in h. Then by admissibility of $\mathcal{I}^{C'}$ either $o = q$ or $o \succeq^h_{R\uparrow} q$ where R is the appropriate instantiation for C'. In the case $o = q$, $h\, o.\mathit{inv} \leq C'$ contradicts the precondition $h\, q.\mathit{inv} = \mathsf{Object}$. (Note that C' cannot be Object as it's invariant is true.) If $o \succeq^h_{R\uparrow} q$ and $h\, o.\mathit{inv} \leq C'$ then $h\, q.\mathit{com} = \mathit{true}$ by transitive ownership, Lemma 3(b). But $h\, q.\mathit{com} = \mathit{false}$ by precondition $h\, q.\mathit{inv} = \mathsf{Object}$ and program invariant (6).

Note that this applies in particular for o the previous owner; no special precondition is needed for it.

For (5), the instance to consider is $p.\mathit{inv} \leq C \wedge q.\mathit{own} = (p, C) \Rightarrow q.\mathit{com}$ but $p.\mathit{inv} > C$ by precondition.

For (6), this is independent from the *own* field.

Case of field update.

Let $o = [\![e_1 : V]\!]\sigma(h, s)$ and $v = [\![e_2]\!]\sigma(h, s)$. So $[\![e_1.f{:=}\, e_2]\!]\sigma\mu(h, s) = ([h \mid o.f \mapsto v], s)$. The preconditions are $o \neq \mathit{nil}$ and $h\, o.\mathit{inv} > \mathit{name}\, V'$ where V', with $V' \geq V$, is the class that declares f.

Neither (5) nor (6) depends on declared fields so they are not falsified by field update.

For (4): consider any o', C', R such that $\mathcal{I}_R^{C'}(o')$ depends on $o.f$ in h, to show that $h\,o'.\mathit{inv} > C'$. By admissibility of $\mathcal{I}^{C'}$, there are two cases.

- $o = o'$: We have precondition $h\,o.\mathit{inv} > name\,V'$ so it suffices to show that $name\,V' \geq C'$. Admissibility for $\mathcal{I}_R^{C'}(o')$ requires $f \in dom(\mathit{fields}\,C')$ and thus $C' \leq nameV'$.
- $o' \succeq_B^h o$ for some $B \geq C'$. From precondition $h\,o.\mathit{inv} > V'$, the semantic typing property $\mathit{type}\,o \leq name\,V$, and $V \leq V'$ we get $h\,o.\mathit{inv} > name(\mathit{type}\,o)$. This implies, by transitive ownership Corollary 1, $h\,o'.\mathit{inv} > B$ whence $h\,o'.\mathit{inv} > C'$.

5 The abstraction theorem

5.1 Comparing class tables.

We compare two implementations of a designated class Abs. They can have completely different declarations, so long as methods of the same signatures are present —declared or inherited— in both.

To simplify the precondition needed for reading fields, we consider programs desugared into a form like that used in Separation Logic.

Definition 12 (properly annotated for Abs). The *annotated commands for Abs* are those of Definition 11 with the additional restriction that no expression of the form $e.f$ occurs except in commands of the form

$$\textbf{assert}\ \mathcal{P}; x := e.f$$

(in particular, no field access appears in this e).

The *properly annotated commands for Abs* are those such that fields of Abs have private visibility (i.e., if $f \in \mathit{dfields}\,Abs$ then accesses and updates of $e.f$ only occur in code of class Abs) and moreover the designated preconditions imply the preconditions stipulated in Table 1 and in addition

- if $name(\Gamma\,\mathsf{self}) \neq Abs$ then for $\Delta; \Gamma \vdash x := e.f$, the stipulated precondition is $\neg(\exists o \mid o \succeq_{Abs} e)$ (for code of Abs, the stipulated precondition is **true**)
- if $name(\Gamma\,\mathsf{self}) \neq Abs$ then $\Delta; \Gamma \vdash$ **pack** e **as** Abs is not allowed
- if $name(\Gamma\,\mathsf{self}) \neq Abs$ then $\Delta; \Gamma \vdash$ **setown** e_1 **to** (e_2, C) is subject to an additional precondition:
  $(\exists o \mid o \succeq_{Abs} e_1) \Rightarrow C = Abs \vee (\exists o \mid o \succeq_{Abs} e_2)$

The effect of the last precondition is that if e is initially owned at Abs then after a transfer (that occurs in code outside class Abs) it is still owned at Abs.

For convenience in working with singleton heaps, we define $pickdom$ by $pickdom\,h = o$ where $dom\,h = \{o\}$; it is undefined if $dom\,h$ is not a singleton.

Definition 13 (A-decomposition). For any class A and heap h, the *A-decomposition* of h is the set $Fh, Ah_1, Rh_1, Sh_1 \ldots, Ah_k, Rh_k, Sh_k$ (for some $k \geq 0$) of pre-heaps, all subsets of h, determined by the following conditions:

- each $dom\,Ah_i$ contains exactly one object o and $name(\mathit{type}\,o) \leq A$

- every $o \in dom\, h$ with $name(type\, o) \leq A$ occurs in $dom\, Ah_i$ for some i;
- $dom\ Rh_i = \{p \mid o \succeq^h_A p\}$ where $pickdom\, Ah_i = o$;
- $dom\ Sh_i = \{p \mid o \succeq^h_{(\mathit{super}\, A)\uparrow} p\}$ with $pickdom\, Ah_i = o$;
- $dom\, Fh = dom\, h - (\cup i \mid dom(Ah_i * Rh_i * Sh_i))$

Except for the Ah_i, any of these named subheaps can be empty.

We say that *no A-object owns an A-object in h* provided for every o, p in $dom\, h$ if $type\, o \leq A$ and $o \succeq^h_{(type\, o)\uparrow} p$ then $type\, p \not\leq A$. Definition 15 in the sequel imposes a syntactic restriction to maintain this property as an invariant, where A is the class for which two representations are compared. A consequence is that there is a unique decomposition of the heap into separate *islands* of the form $Ah * Rh * Sh$. We use the term "partition" even though some blocks can be empty.

Lemma 5 (A-partition). Suppose no A-object owns an A-object in h. Then the A-decomposition is a partition of h, i.e.,

$$h = Fh * Ah_1 * Rh_1 * Sh_1 * \ldots * Ah_k * Rh_k * Sh_k \tag{7}$$

A partition in this sense determines —and is determined by— a partition on $dom\, h$. Thus it makes sense, given a heap h_0 that is updated from h but has no new objects, to consider the partition on h_0 with *the same structure as* a given partition on h. In fact, the only primitive commands that can change the structure are **setown**, **new**, and consequently method call. Whereas **new** merely adds an object that owns nothing and is not owned, **setown** changes the ownership structure. In particular, it can be used to transfer an object in or out of an island —and this implicitly transfers the transitively owned objects.

To maintain the invariant that no Abs-object owns an Abs-object, we formulate a mild syntactic restriction expressed using a static approximation of ownership.

Definition 14 (can own, $\preceq^\exists$). Given well formed CT, define $\preceq^\exists$ to be the least relation such that

- $(name\, N_2) \preceq^\exists (name\, N_1)$ for every occurrence of **setown** e_1 **to** (e_2, D) in a method of CT, with static types $e_1 : N_1$ and $e_2 : N_2$
- if $C \preceq^\exists D$, $C' \leq C$ and $D' \leq D$ then $C' \preceq^\exists D'$
- it is transitively closed

Suppose $Abs \not\preceq^\exists Abs$. Then it is a program invariant that no Abs-object owns an Abs-object (recall the definition preceding Lemma 5). This is a direct consequence of the following result.

Lemma 6. It is a program invariant that if $o \succeq^h_C p$ then $name(type\, o) \preceq^\exists name(type\, p)$.

Proof. We assume this is true initially. The only command that updates ownership is **setown**. Consider **setown** e_1 **to** (e_2, C) with static types $e_i : N_i$ that occurs in CT. Suppose $h_0 = [\![\textbf{setown}\ e_1\ \textbf{to}\ (e_2, C)]\!]$ and $o \succeq^{h_0} p$ but not $o \succeq^h p$.

We argue by cases on the definition of $o \succeq^{h_0} p$ (or the length of the transitive ownership chain, if you like).

The base case is $h_0\, p.own = (o, B)$. As this was not true in h, we have that $o = [\![e_2]\!]$ and $p = [\![e_1]\!]$ and thus, by typing of the semantics, $type\, o \leq N_2$ and $type\, p \leq N_1$. Then by definition of $\preceq^\exists$ we have $name(type\, o) \preceq^\exists name(type\, p)$.

For the induction case, suppose there are o', p' such that $o \succeq^h o'$, $p' \succeq^h p$, $h\,p'.own \neq (o', -)$ but $h_0\,p'.own = (o', -)$. (Well, I omit the cases $o = o'$ and $p' = p$ but they work.) So $o' = [\![e_2]\!]$ and $p' = [\![e_1]\!]$ and thus, by typing of the semantics, $type\,o' \leq N_2$ and $type\,p' \leq N_1$. Thus $name(type\,o') \preceq^\exists name(type\,p')$ by definition of $\preceq^\exists$. By induction we have $name(type\,o) \preceq^\exists name(type\,o')$ and $name(type\,p') \preceq^\exists name(type\,p')$ so $name(type\,o) \preceq^\exists name(type\,p)$ by transitivity.

Definition 15 (comparable class tables). Well formed class tables CT and CT' are *comparable* with respect to class name Abs ($\neq$ Object) provided the following hold.

- $CT(C) = CT'(C)$ for all $C \neq Abs$.
- For Abs, CT and CT' agree on parameterization and superclass, so the declarations have the form
$$CT(Abs) = \textbf{class}\ Abs{<}\bar{X} \lhd \bar{N}{>} \lhd N\ \{\ \bar{T}\,\bar{g};\ \bar{M}\ \}$$
$$CT'(Abs) = \textbf{class}\ Abs{<}\bar{X} \lhd \bar{N}{>} \lhd N\ \{\ \bar{T}'\,\bar{g}';\ \bar{M}'\ \}$$
We write $\vdash, \vdash'$ for the typing relations determined by CT, CT' respectively, and similarly for the auxiliary functions, such as *mtype*, *mtype'*. We also write $[\![-]\!], [\![-]\!]'$ for the respective semantics, with $fresh, fresh'$ the allocators.
- For every method m declared in $CT(Abs)$, m is declared in $CT'(Abs)$ and has the same signature; *mutatis mutandis* for m declared in CT'.
- CT and CT' are properly annotated (for Abs) with respect to their given families of invariants. There is no requirement that $\mathcal{I}^C$ has any relationship to $\mathcal{I}'^C$ for any C.
- $Abs \not\preceq^\exists Abs$ in both CT and CT'

The last condition ensures that the Abs-decomposition of any disciplined heap is a partition, by Lemmas 5 and 6. Note that $[\![R]\!] = [\![R]\!]'$ for all R, and $[\![\Gamma^r]\!] = [\![\Gamma^r]\!]'$ for all Γ^r.

5.2 Coupling relations and simulation

The definitions are organized as follows. A *basic coupling* BC is a suitable relation on islands. This induces a family of *coupling relations*, $\mathcal{R}\,\beta\,\theta$ for each category name θ and typed bijection β. Each relation $\mathcal{R}\,\beta\,\theta$ is from $[\![\theta]\!]$ to $[\![\theta]\!]'$. Here β is a bijection on locations, used to connect a heap in $[\![\mathsf{heap}]\!]$ to one in $[\![\mathsf{heap}]\!]'$. The idea is that β relates all objects except those in the Rh_i or Rh_i' blocks that have never been exposed. Finally, a *simulation* is a coupling that is preserved by all methods of Abs and holds initially.

Definition 16. A *typed bijection* is a bijective relation, β, from Loc to Loc, such that $\beta\,o\,o'$ implies $type\,o = type\,o'$ for all o, o'. A *total bijection* on h, h' is a typed bijection with $dom\,h = dom\,\beta$ and $dom\,h' = rng\,\beta$. Finally, β *fully partitions* h, h' *for* Abs if, for all $o \in dom\,h$ (resp. $o \in dom\,h'$) with $name(type\,o) \leq Abs$, o is in $dom\,\beta$ (resp. $rng\,\beta$).

Lemma 7 (typed bijection and Abs-partition). Suppose β is a typed bijection with $\beta \subseteq dom\,h \times dom\,h'$ and β fully partitions h, h' for Abs. If h, h' are disciplined and partition as $h = Fh * \ldots Ah_j * Rh_j * Sh_j$ and $h' = Fh' * \ldots Ah_k' * Rh_k' * Sh_k'$ then $j = k$.

Definition 17 (equivalence modulo bijection). For any β and designated class name Abs, we define a relation $\sim_\beta$ for data values, object states, heaps, and stores, as in Table 6.

$o \sim_\beta o'$	in $[\![R]\!]$	$\Leftrightarrow\ \beta\, o\, o' \vee o = nil = o'$
$v \sim_\beta v'$	in $[\![T]\!]$	$\Leftrightarrow\ v = v'$ for primitive types T
$s \sim_\beta s'$	in $[\![\mathsf{state}\ R]\!]$	$\Leftrightarrow\ \forall (f:T) \in \mathit{xfields}\ R \mid s f \sim_\beta s' f \vee (f:T) \in \mathit{dfields}\ Abs$
$s \sim_\beta s'$	in $[\![\Gamma^r]\!]$	$\Leftrightarrow\ \forall x \in dom\, \Gamma^r \mid s\, x \sim_\beta s'\, x$
$h \sim_\beta h'$	in $[\![\mathsf{pre\text{-}heap}]\!]$	$\Leftrightarrow\ \forall o \in dom\, h, o' \in dom\, h' \mid \beta\, o\, o' \Rightarrow h\, o \sim_\beta h'\, o'$
$(h,s) \sim_\beta (h',s')$	in $[\![\mathsf{heap} \otimes \Gamma^r]\!]$	$\Leftrightarrow\ h \sim_\beta h' \wedge s \sim_\beta s'$
$v \sim_\beta v'$	in $[\![\theta_\perp]\!]$	$\Leftrightarrow\ v = \perp = v' \vee (v \neq \perp \neq v' \wedge v \sim_\beta v'$ in $[\![\theta]\!])$
$(o,C) \sim_\beta (o',C')$	in $[\![\mathsf{owntyp}]\!]$	$\Leftrightarrow\ (o = nil = o') \vee (\beta\, o\, o' \wedge C = C')$
$B \sim_\beta B'$	in $[\![\mathsf{invtyp}\, C]\!]$	$\Leftrightarrow\ B = B'$

Table 6. Value equivalence for designated class Abs. The relation for heap is the same as for pre-heap. Note that the relation on object states is independent from the declared fields of both $CT(Abs)$ and $CT'(Abs)$.

Equivalence hides the private fields of Abs. In the identity extension lemma, it is used in conjunction with the following which hides objects owned at Abs.

Definition 18 (encap). Suppose no A-object owns an A-object in h. Define $encap\, A\, h$ to be the pre-heap $Fh * Ah_1 * Sh_1 * \ldots * Ah_k * Sh_k$ where the A-partition of h is as in (7) above.

The most important definition is of basic coupling, which is analogous to an object invariant but is a relation on pairs of pre-heaps. For such a relation to *left-depend* on some $o.f$ in some related pair h, h' simply means that there is some v such that $[h \mid o.f \mapsto v]$ is not related to h' (*mutatis mutandis* for right-depend). To *depend on* some f means there exist o, h, h' as above. (This can be formalized as in Definition 7.)

In Definition 9, we take an invariant $\mathcal{I}_R^C$ to be a predicate (set of states) and the program invariant $\mathcal{PI}$ is based on the intersection of these predicates for all objects and types —subject to *inv*, see (4). By contrast, we define a basic coupling BC in terms of pre-heaps. We are concerned with a single class name, Abs, rather than all names C, and instead of a family indexed on ground instances of $Abs\langle\bar{X}\rangle$ we use a single set of pre-heaps that include instance objects of all ground $R \leq Abs\langle\bar{X}\rangle$. We impose the same dependency condition as in Definition 9, but in terms of pre-heaps of the form $h = Ah * Rh * Sh$. (Recall Proposition 2.)

Although the simulation on global states is analogous to the program invariant $\mathcal{PI}$, we do not use intersection over all instances of Abs. Instead, the definition has a form that, in separation logic (over two states [49]), could appear as an iterated separating conjunction $BC * BC * BC * \ldots * true$. Recall Lemma 5.) The "$*true$" has an effect like enforcing new-closure by fiat, but a monotonicity condition is also needed for β in the following.

Definition 19 (basic coupling). Given comparable class tables, a *basic coupling* is a function, BC, that assigns to each typed bijection β a binary relation $BC\ \beta$ on pre-heaps that satisfies the following. First, $BC\ \beta$ does not depend on *inv* or *com*. Second, $\beta \subseteq \beta_0$ implies $BC\ \beta \subseteq BC\ \beta_0$. Third, for any β, h, h', if $BC\ \beta\, h\, h'$ then there are locations o, o' with $\beta\, o\, o'$ and $name(type\, o) \leq Abs$ such that the Abs partitions of h, h' are $h = Ah * Rh * Sh$ and $h' = Ah' * Rh' * Sh'$ with

- $pickdom\, Ah = o$ and $pickdom\, Ah' = o'$
- $o \succeq^h_{Abs} p$ for all $p \in dom(Rh)$ and $o' \succeq^{h'}_{Abs} p'$ for all $p' \in dom(Rh')$

- $o \succeq^{h}_{(\mathit{super}\,Abs)\uparrow} p$ for all $p \in dom(Sh)$ and $o' \succeq^{h'}_{(\mathit{super}\,Abs)\uparrow} p'$ for all $p' \in dom(Sh')$
- If $BC\ \beta$ left-depends on $o.f$ in h, h', or right-depends on $o'.f$, then f is in *xfields* Abs

The first three conditions ensure that BC relates a single island, for an object with some ground subtype of $Abs\langle\bar{X}\rangle$, to a single island for an object of the same type. Although BC is unconstrained for the private fields of $CT(Abs)$ and $CT'(Abs)$, it may also depend on fields inherited from a superclass of Abs (but not on subclass fields).

The induced coupling relation, defined below, imposes the additional constraint that fields of proper sub- and super-classes of Abs are linked by equivalence modulo β.

Although superficially different, the notion of basic coupling is closely related to admissible invariant, as indicated by the following.

Proposition 3. Let BC be a basic coupling, β a typed bijection, and h' a pre-heap. Define the set $J = \{h \mid BC\,\beta\,h\,h'\}$ and define $\mathcal{P} = \{h * h_0 \mid h \in J \wedge (h * h_0) \in [\![\mathsf{heap}]\!]\}$. Then $\mathcal{P}$ is an admissible invariant for Abs.

Set J is the "h'-projection" and $\mathcal{P}$ could be written in separation logic as $J * \mathbf{true}$ —it is new-closed.

In applications, $BC\,\beta\,h\,h'$ is typically defined as something like this: h and h' partition as islands $Ah{*}Rh{*}Sh$ and $Ah'{*}Rh'{*}Sh'$ such that $Ah{*}Rh{*}Sh \models \mathcal{I}^{Abs}$ and $Ah'{*}Rh'{*}Sh' \models \mathcal{I}'^{Abs}$ and some condition linking the data structures Rh and Rh' [22].

A basic coupling BC induces a relation on arbitrary heaps by requiring that they partition such that islands can be put in correspondence so that pairs are related by BC.

Definition 20 (coupling relation, $\mathcal{R}$). Given basic coupling BC, we define for each θ and β a relation $\mathcal{R}\,\beta\,\theta \subseteq [\![\theta]\!] \times [\![\theta]\!]'$ as follows.

For heaps h, h', we define $\mathcal{R}\ \beta\ \mathsf{heap}\ h\ h'$ iff h, h' are disciplined, $\beta \subseteq dom\,h \times dom\,h'$, and β fully partitions h, h' for Abs; moreover, if the Abs-partitions are $h = Fh * Ah_1 * Rh_1 * Sh_1 \ldots Ah_k * Rh_k * Sh_k$ and $h' = Fh' * Ah'_1 * Rh'_1 * Sh'_1 \ldots Ah'_k * Rh'_k * Sh'_k$ then (recall Lemma 7) (a) β restricts to a total bijection between $dom(Fh)$ and $dom(Fh')$; (b) $Fh \sim_\beta Fh'$; and (c) for all i, j, if $\beta\,(pickdom Ah_i)\,(pickdom Ah'_j)$ then

- β restricts to a total bijection between $dom(Sh_i)$ and $dom(Sh'_j)$
- $(Ah_i * Sh_i) \sim_\beta (Ah'_j * Sh'_j)$
- $h(pickdom Ah_i).\mathit{inv} \leq Abs$
  $\Rightarrow BC\ \beta\ (Ah_i * Rh_i * Sh_i)\ (Ah'_j * Rh'_j * Sh'_j)$

For other categories θ we define $\mathcal{R}\,\beta\,\theta$ in Table 7.

Under the antecedent in the definition, $(Ah_i{*}Sh_i) \sim_\beta (Ah'_j{*}Sh'_j)$ is equivalent to the conjunction of $Ah_i \sim_\beta Ah'_j$ and $Sh_i \sim_\beta Sh'_j$. And $Ah_i \sim_\beta Ah'_j$ means that the two objects o, o' agree on superclass and subclass fields; in particular, $name(type\,o) = name(type\,o') \leq Abs$ and $Ah_i\,o.\mathit{inv} = Ah'_j\,o'.\mathit{inv}$.

(Note: a better notation might be $\sim^{Abs}_\beta$ to remind what is excluded.)

The gist of the abstraction theorem is that if methods of Abs are related by $\mathcal{R}$ then all methods are. In terms of the preceding definitions, we can express quite succinctly the conclusion that all methods are related: $\mathcal{R}$ meth-env $[\![CT]\!]\ [\![CT']\!]'$. We want the antecedent of the theorem to be

$$
\begin{array}{ll}
\mathcal{R}\ \beta\ \theta\ \alpha\ \alpha' & \Leftrightarrow \alpha \sim_\beta \alpha' \text{ for } \theta = \textbf{bool}, R, \Gamma^r, \text{and } \mathsf{state}\ R \\
\mathcal{R}\ \beta\ (\mathsf{heap} \otimes \Gamma^r)\ (h,s)\ (h',s') & \Leftrightarrow \mathcal{R}\ \beta\ \mathsf{heap}\ h\ h' \wedge \mathcal{R}\ \beta\ \Gamma^r\ s\ s' \wedge \mathit{disciplined}(h,s) \wedge \mathit{disciplined}(h',s') \\
\mathcal{R}\ \beta\ (\mathsf{heap} \otimes T)\ (h,v)\ (h',v') & \Leftrightarrow \mathcal{R}\ \beta\ \mathsf{heap}\ h\ h' \wedge \mathcal{R}\ \beta\ T\ v\ v' \\
\mathcal{R}\ \beta\ (\theta_\perp)\ \alpha\ \alpha' & \Leftrightarrow (\alpha = \perp = \alpha') \vee (\alpha \neq \perp \neq \alpha' \wedge \mathcal{R}\ \beta\ \theta\ \alpha\ \alpha') \\
\mathcal{R}\ \beta\ (\Gamma^r \vdash T)\ v\ v' & \Leftrightarrow \forall h,s,h',s' \mid \mathcal{R}\ \beta\ (\mathsf{heap} \otimes \Gamma^r)\ (h,s)\ (h',s') \\
 & \quad \Rightarrow \mathcal{R}\ \beta\ T_\perp\ (v(h,s))\ (v'(h',s')) \\
\mathcal{R}\ \beta\ (\Gamma^r \vdash \mathsf{command})\ v\ v' & \Leftrightarrow \forall h,s,h',s' \mid \mathcal{R}\ \beta\ (\mathsf{heap} \otimes \Gamma^r)\ (h,s)\ (h',s') \\
 & \quad \Rightarrow \exists \beta_0 \supseteq \beta \mid \mathcal{R}\ \beta_0\ (\mathsf{heap} \otimes \Gamma^r)_\perp\ (v(h,s))\ (v'(h',s')) \\
\mathcal{R}\ \beta\ (R,\ \bar{R}{\to}R_1)\ v\ v' & \Leftrightarrow \forall h,s,h',s' \mid \mathcal{R}\ \beta\ (\mathsf{heap} \otimes \Gamma^r)\ (h,s)\ (h',s') \\
 & \quad \Rightarrow \exists \beta_0 \supseteq \beta \mid \mathcal{R}\ \beta_0\ (\mathsf{heap} \otimes R_1)_\perp\ (v(h,s))\ (v'(h',s')) \\
 & \quad \text{where } \bar{x} = \mathit{pars}(m,R) \text{ and } \Gamma^r = [\bar{x} : \bar{R}, \mathsf{self} : R] \\
\mathcal{R}\ \mathsf{meth\text{-}env}\ \mu\ \mu' & \Leftrightarrow \forall C,m,\beta,R,\bar{Q} \mid \mathcal{R}\ \beta\ (R,\ [\bar{Q}/\bar{Y}](\bar{T}{\to}T))\ (\mu R m \bar{Q})\ (\mu' R m \bar{Q}) \\
 & \quad \text{where } R \text{ instantiates } C{<}\bar{X}{>} \text{ and } \mathit{mtype}(m,R) = {<}\bar{Y} \triangleleft \bar{N}{>}\bar{T}{\to}T
\end{array}
$$

Table 7. The induced coupling relation for Definition 20.

that the meaning $[\![M]\!]$ is related to $[\![M']\!]'$, for any m with declaration M in $CT(Abs)$ and M' in $CT'(Abs)$. But the relation is defined for ground instantiations. Moreover, $[\![M]\!]$ depends on a method environment. Thus the antecedent of the theorem is that $[\![M]\!](Abs{<}\bar{R}{>})\bar{Q}\mu$ is related to $[\![M']\!]'(Abs{<}\bar{R}{>})\bar{Q}\mu'$ for all $\bar{Q}$ and all related μ, μ'.

Definition 21 (simulation). A simulation is a coupling $\mathcal{R}$ such that the following hold.

- (BC is initialized) For any $C \leq Abs$, any instance R of $C{<}\bar{X}{>}$, and any o, o' with $\beta\, o\, o'$ and $\mathit{type}\, o = R$ we have $BC\ \beta\ h\ h'$ where
  $h = [o \mapsto [dom(\mathit{xfields}\, R) \mapsto \mathit{defaults}\, R]]$
  $h' = [o' \mapsto [dom(\mathit{xfields}'\, R) \mapsto \mathit{defaults}'\, R]]$.
- (methods of Abs preserve $\mathcal{R}$) For any disciplined μ, μ' such that $\mathcal{R}$ meth-env $\mu\ \mu'$ we have the following for every m declared in Abs. Let $\mathit{mtype}(m, Abs{<}\bar{X}{>}) = {<}\bar{Y} \triangleleft \bar{N}{>}\bar{U}{\to}U$. For every instance $\bar{R}$ of $\bar{X}$, every instance $\bar{Q}$ of $\bar{Y}$, and every β, let $\theta = (Abs{<}\bar{R}{>},\ [\bar{Q}/\bar{Y}](\bar{U}{\to}U))$ in
  $$\mathcal{R}\ \beta\ \theta\ ([\![M]\!](Abs{<}\bar{R}{>})\bar{Q}\mu)\ ([\![M']\!]'(Abs{<}\bar{R}{>})\bar{Q}\mu')$$
  where M (resp. M') are as above.

5.3 Abstraction theorem

The main theorem is that if $\mathcal{R}$ is a simulation for comparable class tables CT, CT' then

$$\mathcal{R}\ \mathsf{meth\text{-}env}\ [\![CT]\!]\ [\![CT']\!]'$$

It is proved using Preservation Lemmas 12, 10, 11. The most difficult case in the preservation lemma for commands is that of **setown**, for which we need some technical results that are best skipped on first reading.

Technical results.

Lemma 8. Consider pre-heaps Ph and Ph' such that $Ph \sim_\beta Ph'$. If $\beta\, o\, o'$, $\beta\, p\, p'$, and $o \succeq_C^{Ph} p$ for some C then $o' \succeq_C^{Ph'} p'$.

Note that there may be locations not in $dom\, Ph$ or $dom\, Ph'$ that are related by β and thus enter into whether $Ph \sim_\beta Ph'$ holds. But $o \succeq_C^{Ph} p$ implies that o and p are in $dom\, Ph$ by definition of $\succeq$.

Proof. We show $o' \succeq_C^{Ph'} p'$ by induction on $\succeq$. For the base case, suppose $o \succeq_C^{Ph} p$ because $Ph\ p.own = (o, C)$. From $\beta\, p\, p'$ and $Ph \sim_\beta Ph'$ we obtain $(Ph\ p.own) \sim_\beta (Ph'\ p'.own)$ and hence $Ph'\ p'.own = (o', C)$. So $o' \succeq_C^{Ph'} p'$.

For the induction step, suppose $o \succeq_C^{Ph} p$ because there are q and B with $q \in dom\, Ph$ and $Ph\ p.own = (q, B)$ and $o \succeq_C^{Ph} q$. As in the base case, we obtain q', B' with $q \in dom\, Ph$, $Ph'\ p'.own = (q', B')$, and $\beta\, q\, q'$. Now we get $o' \succeq_C^{Ph'} p'$ by induction, using $\beta\, o\, o'$, $\beta\, q\, q'$, and $o \succeq_C^{Ph} q$.

Corollary 2 (splitting). Let Ph, Ph' be pre-heaps that are closed under transitive ownership (i.e., if $o \in dom\, Ph$ and $o \succeq^{Ph}_{(type\ o)\uparrow} p$ then $p \in dom\, Ph$). Suppose β is a total bijection from Ph to Ph' and $Ph \sim_\beta Ph'$. Suppose $o \in dom\, Ph$ and $\beta\, o\, o'$. Define Ph^+ by domain restriction from Ph so that

$$dom\, Ph^+ = \{p \mid p = o \vee o \succeq^{Ph}_{(type\ o)\uparrow} p\}$$

and *mutatis mutandis* for Ph'^+ with respect to o'. Define Ph^- to be the remaining pre-heap so that $Ph = Ph^+ * Ph^-$ and likewise $Ph' = Ph'^+ * Ph'^-$. Then β is a total bijection from Ph^+ to Ph'^+ and a total bijection from Ph^- to Ph'^-. Moreover $Ph^+ \sim_\beta Ph'^+$ and $Ph^- \sim_\beta Ph'^-$.

Proof. As a consequence of Lemma 8 we get that β is a total bijection from Ph^+ to Ph'^+. The rest follows from the definitions.

Lemma 9 (partition and coupling). Suppose $\mathcal{R}\ \beta$ heap $h\ h'$ and $\mathcal{R}\ \beta\ R\ o\ o'$. Let $h = Fh * \ldots Ah_k * Rh_k * Sh_k$ and $h' = Fh' * \ldots Ah'_k * Rh'_k * Sh'_k$ be the Abs-partitions. Then

(a) $o \in dom\, Fh$ implies $o' \in dom\, Fh'$
(b) $o = pickdom\, Ah_i$ implies $o' = pickdom\, Ah'_j$ for some j
(c) $o \in dom\, Sh_i$ implies $o' \in dom\, Sh'_j$ for some j (namely the j such that o' is owned by $pickdom\, Ah'_j$)
(d) $o \in dom\, Rh_i$ implies $o' \in dom\, Rh'_j$ for some j (namely the j such that o' is owned by $pickdom\, Ah'_j$)

Proof. By definition of $\mathcal{R}$ we have $\beta\, o\, o'$. Now (a) holds by conditions (a) and (b) in Definition 20 of coupling for the heap.

For (b) the argument is straightforward, using the definitions of typed bijection and Abs-partition.

For (c), let j index the island in h' that corresponds to i, i.e., $\beta(pickdom\, Ah_i)(pickdom\, Ah'_j)$. By Definition 20 we have that β is a total bijection from $dom\, Sh_i$ to $dom\, Sh'_j$, so $\beta\, o\, o'$ implies $o' \in dom\, Sh_j$.

For (d), let $o \in dom\, Rh_i$. Using $\mathcal{R}\ \beta$ heap $h\ h'$ and Definition 20, o' is not in any $dom\, Ah'_j$ or $dom\, Sh'_j$, nor is it in $dom\, Fh'$, as these parts of h' are connected to h bijectively. Thus by partitioning o' must be in some Rh'_j.

Note that the argument in case (d) does not say that o' is in the island k with $\beta(pickdom\ Ah_i)(pickdom\ Ah'_k)$.

Preservation results.

Lemma 10 (preservation by expressions). For all expressions $\Delta; \Gamma \vdash e : T$ that contain no field access subexpressions, all $\sigma \in grnd\,\Delta$, and all β

$$\mathcal{R}\ \beta\ (\Gamma \vdash T)\ (\llbracket \Delta; \Gamma \vdash e : T \rrbracket \sigma)\ (\llbracket \Delta; \Gamma \vdash e : T \rrbracket' \sigma)$$

Proof. Omitted.

Lemma 11 (preservation by guarded field access). Consider $\Delta; \Gamma \vdash e.f : T$ where e contains no field access. Consider any $\sigma \in grnd\,\Delta$, any β, and any h, s, h', s' such that $\mathcal{R}\ \beta\ (\mathsf{heap} \otimes \sigma\,\Gamma)\ (h, s)\ (h', s')$. Let $o = \llbracket \Delta; \Gamma \vdash e.f : T \rrbracket \sigma(h, s)$ and $o' = \llbracket \Delta; \Gamma \vdash e.f : T \rrbracket \sigma(h', s')$ with $o \neq \bot \neq o'$. If $\neg(\exists p \mid p \succeq^{h}_{Abs} o)$ and $\neg(\exists p \mid p \succeq^{h'}_{Abs} o')$ then $\mathcal{R}\ \beta\ T\ (h\,o.f)\ (h'\,o'.f)$.

Proof. Omitted.

Lemma 12 (preservation by commands). Suppose that μ, μ' are disciplined method environments and $\mathcal{R}$ meth-env $\mu\ \mu'$. Suppose $\Delta; \Gamma \vdash S$ is a properly annotated command for Abs with $name(\Gamma\ \mathsf{self}) \neq Abs$. Then for all $\sigma \in grnd\,\Delta$ and all β

$$\mathcal{R}\ \beta\ (\Gamma \vdash \mathsf{command})\ (\llbracket \Delta; \Gamma \vdash S \rrbracket \sigma\mu)\ (\llbracket \Delta; \Gamma \vdash S \rrbracket' \sigma\mu')$$

Proof. By definition of $\mathcal{R}$ for category $\Gamma \vdash \mathsf{command}$ we are to show that the meaning of S preserves the relation at the level of states. So we argue in terms of arbitrary initial states (h, s) and (h', s') that are related at β. Note in particular that by Definition 20 for heaps, β fully partitions h, h' for Abs.

Moreover, because S is a properly annotated command for Abs and $name(\Gamma\ \mathsf{self}) \neq Abs$, all occurrences of $f accessed$ occurs in commands of the form $x{:=}\, e.f$. For such a command, the precondition $\neg(\exists p \mid p \succeq^{h}_{Abs} o)$ holds, where $o = \llbracket e.f \rrbracket$. Thus for all expressions e, appearing as constituents of S, we can apply Lemma 10 and Lemma 11 and conclude that

$$\mathcal{R}\ \beta\ (\Gamma \vdash T)\ (\llbracket \Delta; \Gamma \vdash e : T \rrbracket \sigma)\ (\llbracket \Delta; \Gamma \vdash e : T \rrbracket' \sigma)$$

Case of field update.

Let $o = \llbracket e_1 : V \rrbracket \sigma(h, s)$ and $v = \llbracket e_2 \rrbracket \sigma(h, s)$ (and o', v' for h', s' as per our convention). By typing, f is declared in some $V' \geq V$. The resulting heap h_0 is $[h \mid o.f \mapsto v]$. The preconditions are $o \neq nil$ and $h\,o.\mathit{inv} > name(V')$.

Assume $\mathcal{R}\ \beta\ \mathsf{heap}\ h\ h'$ to show $\mathcal{R}\ \beta\ \mathsf{heap}\ h_0\ h'_0$. Let the Abs-partitions of h and h' be $h = Fh * \ldots$ and $h' = Fh' * \ldots$. The Abs-partitions of h_0 and h'_0 have the same structure as those of h and h' respectively.

By Lemmas 10,11 on e_1, $o \sim_\beta o'$, i.e., either $\beta\,o\,o'$ or $o = nil = o'$. For the latter case, both semantics are $\bot$ and we are done. So we suppose $\beta\,o\,o'$. We have the following cases.

- $o \in dom\,Fh$. Because β is a typed bijection, $o' \in dom\,Fh'$. In the update heaps h_0 and h'_0, $Fh \sim_\beta Fh'$ holds because $v \sim_\beta v'$ by Lemma 10 applied to e_2.

- $o = pickdom Ah_i$ for some i. Then $type\, o \leq Abs$ and because β is a typed bijection, $type\, o' \leq Abs$. Hence $o' = pickdom Ah'_j$ for some j. To show $Ah_i \sim_\beta Ah'_j$ in the updated heaps note that $v \sim_\beta v'$ by Lemmas 10, 11 on e_2.
  By type soundness for e_1, $name(type\, o) \leq name(V)$, hence $name(type\, o) \leq name(V')$. By the tree property of $\leq$, we have two subcases. In both subcases, according to Def. 20, we must show that BC is preserved if o (hence o') is packed. (a) $Abs < name(V')$: so $h\, o.inv > Abs$ from the precondition and there is nothing to prove about $BC\, \beta$. (b) $name(V') < Abs$: so f is a declared field in a proper subclass of Abs. Here it is possible that $h\, o.inv \leq Abs$, and we must show that $BC\, \beta$ is maintained. The only way $BC\, \beta$ can be falsified is if it depended on f; but this is impossible as $BC\, \beta$ is independent of fields in proper subclasses of Abs.
- $o \in dom\, (Rh_i * Sh_i)$ for some i. By precondition, $h\, o.inv > name(V')$. By type soundness for e_1, $type\, o \leq name(V')$; hence $h\, o.inv > type\, o$. If $o \in dom\, Rh_i$, then $pickdom\, Ah_i \succeq^h_{Abs} o$, so by Corollary 1, $h(pickdom\, Ah_i).inv > Abs$. If $o \in dom\, Sh_i$, $pickdom\, Ah_i \succeq^h_C o$ for $C > Abs$, so $h(pickdom\, Ah_i).inv > C > Abs$, also by Corollary 1. Because *inv* is not updated, $(pickdom\, Ah_i).inv > Abs$ holds in the updated heaps also. Thus the implication $Ah_i.inv \leq Abs \Rightarrow BC\, \beta \ldots$ holds.
  Finally, if $o \in dom, Sh_i$, must show $Sh_i \sim_\beta Sh'_j$ holds in the updated heaps, which follows by $v \sim_\beta v'$, using Lemma 10 on e_2.

Case of new. Let $o = \mathit{fresh}(\sigma\, N, h)$ and let $h_0 = [h \mid o \mapsto [\mathit{xfields}(\sigma\, N) \mapsto \mathit{defaults}(\sigma\, N)]]$. Assume $\mathcal{R}\ \beta\ (\mathsf{heap} \otimes \sigma\Gamma)\ (h, s)\ (h', s')$ to show that there exists $\beta_0 \supseteq \beta$ such that $\mathcal{R}\ \beta_0\ (\mathsf{heap} \otimes \sigma\Gamma)\ (h_0, s_0)\ (h'_0, s'_0)$, where $s_0 = [s \mid x \mapsto o]$ and $s'_0 = [s' \mid x \mapsto o']$.

Let the Abs-partition of h be $Fh * \ldots$. We choose $\beta_0 = \beta \cup \{(o, o')\}$. Then $\mathcal{R}\ \beta_0\ (\sigma\Gamma)\ s_0\ s'_0$ follows. To show $\mathcal{R}\ \beta_0\ \mathsf{heap}\ h_0\ h'_0$, note first that β fully partitions h_0, h'_0 for Abs. For the remaining conditions, we have the following cases:

- $\sigma\, N \leq Abs$. The resulting heap has an island $Ah_{new} * Rh_{new} * Sh_{new}$ where $Ah_{new} = [o \mapsto [\mathit{xfields}(\sigma\, N) \mapsto \mathit{defaults}(\sigma\, N)]]$, and $o = pickdom(Ah_{new})$, $o' = pickdom(Ah'_{new})$ and $Rh_{new} = Sh_{new} = \varnothing$. All the other islands are the same as in h. Note that $Ah_{new} \sim_\beta Ah'_{new}$. To show $Ah_{new}(o).inv \leq Abs \Rightarrow BC\, \beta_0 (Ah_{new} * Rh_{new} * Sh_{new})(Ah'_{new} * Rh'_{new} * Sh'_{new})$, note that $Ah_{new}(o).inv = \mathsf{Object} = Ah'_{new}(o).inv$, because $name\, N \neq \mathsf{Object}$. Thus the antecedent of the implication is falsified. For all other islands, Ah_i, if $\beta\, q\, q'$ where $q = pickdom\, Ah_i$ and $q' = pickdom\, Ah'_j$, then $Ah_i \sim_{\beta_0} Ah'_j$ follows by $Ah_i \sim_\beta Ah'_j$ and monotonicity of $\sim_\beta$. Finally, $Ah_i\, o.inv \leq Abs \Rightarrow BC\, \beta_0 (Ah_i * Rh_i * Sh_i)(Ah'_j * Rh'_j * Sh'_j)$ follows because initially $Ah_i\, o.inv \leq Abs \Rightarrow BC\, \beta (Ah_i * Rh_i * Sh_i)(Ah'_j * Rh'_j * Sh'_j)$ and $BC\, \beta \subseteq BC\, \beta_0$ by monotonicity in Def. 19.
  Next, $Sh_i \sim_{\beta_0} Sh'_j$ follows by $Sh_i \sim_\beta Sh'_j$ and monotonicity of $\sim_\beta$. Finally, because β restricts to a total bijection between $dom\, Fh$ and $dom\, Fh'$, so does β_0.
- Otherwise, the resulting heap is $Fh_0 * (Ah_1 * Rh_1 * Sh_1) * \ldots$, where $Fh_0 = Fh * [o \mapsto [\mathit{xfields}(\sigma\, N) \mapsto \mathit{defaults}(\sigma\, N)]]$.
  The condition on islands holds because $BC\, \beta$ is monotonic. We have that β_0 restricts to a total bijection between $dom\, Fh \cup \{o\}$ and $dom\, Fh' \cup \{o'\}$; and $Fh_0 \sim_{\beta_0} Fh'_0$ holds by monotonicity of $\sim_\beta$.

Case of method call, $x := e.m\langle \bar{V} \rangle(\bar{e})$**.** Let $o = [\![e : U]\!]\sigma(h, s)$, $\bar{v} = [\![\bar{e} : \bar{U}]\!]\sigma(h, s)$, $\bar{x} = \mathit{pars}(m, \sigma\, U)$; let $s_1 = [\bar{x} \mapsto \bar{v}, \mathsf{self} \mapsto o]$, $(h_1, v_1) = \mu(type\, o)m(\sigma\, \bar{V})(h, s_1)$.

The resulting heap h_0 is h_1 and the resulting store s_0 is $[s \mid x \mapsto v_1]$. Note that $o', \bar{v'}$ for h', s' are as per our convention.

Assume $\mathcal{R}\ \beta\ (\mathsf{heap} \otimes \Gamma^r)\ (h, s)\ (h', s')$ to show $\mathcal{R}\ \beta_0\ (\mathsf{heap} \otimes \Gamma^r)\ (h_0, s_0)\ (h'_0, s'_0)$ for some $\beta_0 \supseteq \beta$.

Let R be the type of the target object, i.e., $R = \mathit{type}\, o$. Suppose $\mathit{mtype}(m, R) = <\bar{Y} \triangleleft \bar{N}>\bar{U} \to U$. Let $\bar{Q} = \sigma\bar{V}$. Because $\mathcal{R}$ meth-env $\mu\ \mu'$, we have,

$$\mathcal{R}\ \beta\ (R, \bar{x}, [\bar{Q}/\bar{Y}](\bar{T} \to T))\ (\mu R m \bar{Q})\ (\mu' R m \bar{Q})$$

Hence by assumption $\mathcal{R}\ \beta\ (\mathsf{heap} \otimes \Gamma^r)\ (h, s)\ (h', s')$, we have, $\mathcal{R}\ \beta_0\ (\mathsf{heap} \otimes [\bar{Q}/\bar{Y}]T)\ (h_1, v_1)\ (h'_1, v'_1)$, for some β_0. Hence $v_1 \sim_\beta v'_1$. Hence $\mathcal{R}\ \beta_0\ (\mathsf{heap} \otimes \Gamma^r)_\perp\ (h_0, s_0)\ (h'_0, s'_0)$, for some β_0.

Case of setowner.

Let $q = [\![e_1 : N_1]\!]\sigma(h, s)$ and $p = [\![e_2 : N_2]\!]\sigma(h, s)$ and $h\, q.\mathit{own} = (r, B)$. The resulting heap h_0 is $[h \mid q.\mathit{own} \mapsto (p, C)]$. As usual, q', p', r', B', h'_0 are determined *mutatis mutandis* from (h', s'). Assume $\mathcal{R}\ \beta$ heap $h\ h'$ to show $\mathcal{R}\ \beta$ heap $h_0\ h'_0$. Let the Abs-partitions of h and h' be $h = Fh * \ldots * Ah_n * Rh_n * Sh_n$ and $h' = Fh' * \ldots * Ah'_n * Rh'_n * Sh'_n$. Although ownership structure is changed (except in the case $(r, B) = (p, C)$), what is relevant to $\mathcal{R}$ is changes in Abs-partition. To reason carefully about these cases, we give notation for the Abs-partitions of the updated heaps as well: $h_0 = \hat{Fh} * \ldots * \hat{Ah}_n * \hat{Rh}_n * \hat{Sh}_n$ and $h'_0 = \hat{Fh}' * \ldots * \hat{Ah}'_n * \hat{Rh}'_n * \hat{Sh}'_n$. Although case analysis is needed, we begin with general considerations.

By hypothesis of the Lemma, Γ self $\neq Abs$, so the expression lemma applies, yielding that p, p' and q, q' are related values and thus $q \sim_\beta q'$ and either $p = nil = p'$ or $p \sim_\beta p'$.

By the stipulated precondition for **setown** we obtain

(a) $q \neq nil \neq q'$
(b) $h\ q.\mathit{inv} = \mathsf{Object} = h'\ q'.\mathit{inv}$
(c) $p = nil = p'$ or else $h\ p.\mathit{inv} > C$ and $h'\ p'.\mathit{inv} > C$

Recall that by typing we have $name(\mathit{type}\, p) \leq C$.

Owing to Γ self $\neq Abs$ we also have another precondition, by Definition 12:

(d) $(\exists o \mid o \succeq_{Abs} q) \Rightarrow C = Abs \vee (\exists o \mid o \succeq_{Abs} p)$

From (b) and the transitive ownership corollary and $\mathcal{R}\ \beta$ heap $h\ h'$ it follows that

(e) $r = nil = r'$ or else $h\ r.\mathit{inv} > B$ and $h'\ r'.\mathit{inv} > B'$

We complete the proof by cases on whether q is in the domain of Fh or one of Ah_i, Rh_i, or Sh_i for some i.

CASE $q \in dom\, Fh$. By $\mathcal{R}\ \beta$ heap $h\ h'$ and the definitions we have $q' \in dom\, Fh'$. [Now that Lemma 9 is explicit, we should use it here too.] Moreover, either $r = nil = r'$ or $r \in dom\, Fh$ and also $r' \in dom\, Fh'$. There are the following subcases on p:

- If $p \in dom\, Fh$ or $p = nil = p'$ then structure of the partition of h_0, h'_0 is unchanged. We get $\mathcal{R}\ \beta$ heap $h_0\ h'_0$ because the only change is to set field *inv* of q, q' to related values p, p'.
- If $p = pickdom\, Ah_i$ for some i then by Lemma 9 there is j with $p' = pickdom\, Ah'_j$. By type soundness, $\mathit{type}\, p \leq \sigma N_2$ and $name\, N_2 \leq C$, and by definition of partition $name(\mathit{type}\, p) \leq Abs$, so by the tree property of $\leq$ either $C < Abs$ or $Abs \leq C$.

- If $C < Abs$ then, in the partition of h_0 we have $q \in dom\, \hat{Fh}$, i.e., the partition has the same structure as initially and the same is true for q', $\hat{Fh'}$. Then we get $\mathcal{R}\ \beta$ heap $h_0\ h'_0$ because the only change is to set field *inv* of q, q' to related values p, p'.
- If $Abs = C$ then q and the objects it transitively owns are being transferred into Rh_i. By (c) we have $h\ p.inv > Abs$ and $h'\ p'.inv > Abs$, so BC is not in force and coupling holds for the updated islands i, j.
- If $Abs < C$ then q and the objects it transitively owns are transferred from Fh into $\hat{Sh}_i$ and q' into $\hat{Sh}'_i$. Again, by (c) we have $h\ p.inv > Abs$ and $h'\ p'.inv > Abs$. To show coupling for the updated islands i, j it remains to show $\hat{Sh}_i \sim_\beta \hat{Sh}'_j$. This follows from $\mathcal{R}\ \beta$ heap $h\ h'$ and $q \sim_\beta q'$. [Note: this is similar to the case of transfer from Rh_i to Sh_k that I've spelled out in more detail later.]

– If $p \in dom(Rh_i * Sh_i)$ for some i then p' must be in $dom(Rh'_j * Sh'_j)$ for some j, by Lemma 9. So q (resp. q') is transferred into island i (resp. j). Let $o = pickdom\, Ah_i$ (resp. $o' = pickdom\, Ah'_j$) so that there is $B \geq Abs$ (resp. $B' \geq Abs$) such that $o \succeq^h_B p$ (resp. $o' \succeq^{h'}_{B'} p'$). By (c) and the transitive ownership Corollary we get $h\ o.inv > B$ (resp. $h'\ o'.inv > B'$) and thus BC is not currently in force for these islands. It remains to show that if q, q' are being transferred into Sh_i, Sh'_j (because $p \in dom\, Sh_i$ and thus $p' \in dom\, Sh_j$) then $Sh_i \sim_\beta Sh'_j$ and this follows from $\mathcal{R}\ \beta$ heap $h\ h'$ because p, p' are related.

CASE $q = pickdom\, Ah_i$ for some i. Then, by Lemma 9, $q' = pickdom\, Ah'_j$ for some j. Because no Abs-object owns an Abs-object, we get $r \in dom\, Fh$ (resp. $r' \in dom\, Fh'$) and moreover p, p' are also in $dom\, Fh$, $dom\, Fh'$. So again the partition structure is unchanged [details similar to preceding case].

CASE $q \in dom\, Rh_i$ for some i. Then $pickdom\, Ah_i \succeq^h_{Abs} q$ by definition of Abs-partition. Now either $r = pickdom\, Ah_i$ and $B = Abs$ or $pickdom\, Ah_i \succeq^h_{Abs} r$, by properties of ownership. By Lemma 9(d) we have $q' \in dom\, Rh'_j$ for some j (but not necessarily $\beta(pickdom\, Ah_i)(pickdom\, Ah'_j)$). By (d), either $C = Abs$ or there is o such that $o \succeq^h_{Abs} p$. In the case $C = Abs$, we have $name(type\, p) \leq Abs$ and thus $p = pickdom\, Ah_k$ and $p' = pickdom\, Ah'_l$ for some k, l. In the other case, because $name(type\, o) \leq Abs$ there is some k with $o = pickdom\, Ah_k$. Similarly, there is some Abs-object o' that owns p' and some l with $o' = pickdom\, Ah_l$. [There is nothing to force that $\beta\, o\, o'$ i.e. that islands k and l correspond.] In the rest of the argument, the two cases are treated together.

Informally, the sub-heap consisting of q and the objects it transitively owns are being transferred from island i to island k, and in particular into Rh_k. In parallel, the sub-heap rooted at q' is being transferred from island j to island l. So couplings for i, j, k, l are all at risk and it need not be the case that i corresponds to j or k to l. By (b) and transitive ownership Corollary we have $h(pickdom\, Ah_i).inv > Abs$ and thus basic coupling BC is not in force for i. Similarly, it is not in force for j. By (c) and transitive ownership Corollary we have $h\ o.inv > Abs$ (or $h\ p.inv > Abs$ in the case $C = Abs$) and thus BC is not in force for k. Similarly, it is not in force for l. Because the transfer of q is into $\hat{Rh}_k$ and q' into $\hat{Rh}_l$, the "Sh part" of coupling is not at risk. This concludes the proof for $q \in dom\, Rh_i$.

CASE $q \in dom\, Sh_i$ for some i. In regards to BC, this case is similar to the case for $q \in dom\, Rh_i$. By Lemma 9(c) we have $q' \in dom\, Sh'_j$ for some j, and unlike the case for Rh we do have $\beta(pickdom\, Ah_i)(pickdom\, Ah'_j)$. [Well, that's not how the Lemma is currently stated, but it's in the proof.] And these islands are unpacked, by (e), so BC is not in force. As q, q'

are in the "Sh part", we also have to deal with the equivalence and bijection requirements. From $\mathcal{R}\ \beta$ heap $h\ h'$ we have that β is a total bijection from Sh_i to Sh'_j and $Sh_i \sim_\beta Sh'_j$. By definition of Abs-partition, both Sh_i and Sh'_j are closed under transitive ownership. Let Sh_i^+ be the sub-heap of Sh with domain consisting of q and objects transitively owned by q; *mutatis mutandis* for q' and Sh'^+_j. Let Sh_i^- and Sh'^-_j be the remainders so that $Sh_i = Sh_i^+ * Sh_i^-$ and $Sh'_j = Sh'^+_j * Sh'^-_j$. Then by Corollary 2 we have that β is a total bijection from $dom\, Sh_i^+$ to $dom\, Sh'^+_j$ and from $dom\, Sh_i^-$ to $dom\, Sh'^-_j$; moreover $Sh_i^+ \sim_\beta Sh'^+_j$ and $Sh_i^- \sim_\beta Sh'^-_j$.

The updated islands i and j are unchanged except that $\hat{Sh}_i = Sh_i^-$ and $\hat{Sh}_j = Sh_j^-$. By the above considerations, these satisfy the bijection and equivalence conditions.

What remains is to account for Sh_i^+ and Sh'^+_j which get transferred into islands k, l. We go by cases on p.

- If $p \in dom\, Fh$ then the partition for h_0 has $\hat{Fh} = Fh * Sh_i^+$ and similarly for h'_0. The coupling conditions hold because $Sh_i^+ \sim_\beta Sh'^+_j$.
- If p is in some $dom\, Ah_k$ (resp. $dom\, Sh_k$) we have p' is the $dom\, Ah'_l$ (resp. $p' \in dom\, Sh_l$) such that $\beta(pickdom\, Ah_k)(pickdom\, Ah'_l)$. An argument like in the case for $q \in dom\, Rh_i$ shows that BC is not in force. And if q is going into Sh_k (resp. q' into Sh_l) then the coupling conditions for $\hat{Sh}_k = Sh_k * Sh_i^+$ and $\hat{Sh}'_l = Sh'_l * Sh'^+_i$ hold by the earlier considerations, e.g., $Sh_i^+ \sim_\beta Sh'^+_j$.
- If p is in some $dom\, Rh_k$ then the argument is similiar to the preceding case but simpler as there is no bijection or equivalence condition with which to be concerned.

Main result.

Theorem 3 (abstraction).
If $\mathcal{R}$ is a simulation for comparable class tables CT, CT' then $\mathcal{R}$ meth-env $[\![CT]\!]\ [\![CT']\!]'$.

Proof. Assume that $\mathcal{R}$ is a simulation. We show that $\mathcal{R}$ holds for each step in the approximation chain in the semantics of class tables. That is, we show by induction on i that

$$\mathcal{R} \text{ meth-env } \mu_i\ \mu'_i \quad \text{for every } i \in \mathbb{N}$$

The result $\mathcal{R}$ meth-env $[\![CT]\!]\ [\![CT']\!]'$ follows as $[\![CT]\!]$ and $[\![CT']\!]'$ are the least upper bounds of these ascending chains and the relation distributes over lubs of chains.

Base case, $i = 0$: We must show $\mathcal{R}\ \beta\ (R, mtype(m, R)\ (\mu_0\, R\, m\, \bar{Q})\ (\mu'_0\, R\, m\, \bar{Q})$ for every $\beta, m, R, \bar{Q}$ where R instantiates $C\langle\bar{X}\rangle$. This holds by definition of μ_0, μ'_0, because $\lambda(h, s) \mid \bot$ relates to itself.

Induction step: Suppose

$$\mathcal{R} \text{ meth-env } \mu_i\ \mu'_i \qquad (*)$$

We must show $\mathcal{R}$ meth-env $\mu_{i+1}\ \mu'_{i+1}$, that is, for every β, every R instantiating $C\langle\bar{X} \lhd \bar{N}\rangle$ and every m with $mtype(m, R)$ defined and every $\bar{Q}$:

$$\mathcal{R}\ \beta\ (R, [\bar{Q}/\bar{Y}](\bar{T} \to T))\ (\mu_{i+1}\, R\, m\, \bar{Q})\ (\mu'_{i+1}\, R\, m\, \bar{Q}) \qquad (\dagger)$$

where *mtype*$(m, R) = <\bar{Y} \lhd \bar{N}_1>\bar{T} \to T$.

For arbitrary m we show (†) for all R with *mtype*(m, R) defined as above, using a secondary induction on inheritance chains.

The base case of the secondary induction is where R instantiates a class $C<\bar{X} \lhd \bar{N}>$ that declares m (so m is declared in both $CT(C)$ and $CT'(C)$). We go by cases on C. If $C = Abs$, we get (†) from the assumption that $\mathcal{R}$ is a simulation. In detail: Using $(*)$ and Def. 21 we get

$$\mathcal{R}\ \beta\ (R,\ [\bar{Q}/\bar{Y}](\bar{T} \to T))\ (\llbracket M \rrbracket R\bar{Q}\mu_i)\ (\llbracket M' \rrbracket' R\bar{Q}\mu'_i)$$

whence (†) by definition of μ_{i+1} and μ'_{i+1}. The other case is $C \neq Abs$. Then by Def. 15 of comparable class tables we have $CT(C) = CT'(C)$ and in particular both class tables have the same declaration; since R is an instantiation of $C<\bar{X}>$, $R = C<\bar{R}>$, and both class tables have the same declaration

$$<\bar{Y} \lhd \bar{N}_1>T\ m(\bar{T}\ \bar{x})\ \{S\}$$

To show (†), suppose $\sigma = [\bar{R}/\bar{X}, \bar{Q}/\bar{Y}]$; suppose (h, s) and (h', s') are disciplined, $\mathcal{R}\ \beta$ heap $h\ h'$, and $\mathcal{R}\ \beta\ \Gamma\ s\ s'$, where $\Gamma = (\mathsf{self} : R,\ \bar{x} : \sigma\bar{T})$. Let $\Delta = [\bar{X} \lhd \bar{N}, \bar{Y} \lhd \bar{N}_1]$. Because the class tables are properly annotated, we may appeal to Lemma 12, using $\mathcal{R}$ meth-env $\mu_i\ \mu'_i$, to get that the results from S are related. That is, either $\llbracket \Delta; \Gamma \vdash S \rrbracket \sigma\mu_i(h, s) = \bot = \llbracket \Delta; \Gamma \vdash' S \rrbracket' \sigma\mu'_i(h', s')$ or neither is $\bot$. In the latter case, (h_0, s_0) is related to (h'_0, s'_0) for some $\beta_0 \supseteq \beta$, where $(h_0, s_0) = \llbracket \Delta; \Gamma \vdash S \rrbracket \sigma\mu_i(h, s)$ and $(h'_0, s'_0) = \llbracket \Delta; \Gamma \vdash' S \rrbracket' \sigma\mu'_i(h', s')$. Then, by definition of $\mathcal{R}\,\beta\,\Gamma$, $\mathcal{R}\ \beta\ \Gamma\ s_0\ s'_0$ implies $\mathcal{R}\ \beta\ T\ (s_0\,\mathsf{result})\ (s'_0\,\mathsf{result})$. Thus (†) holds by definition of μ_{i+1} and μ'_{i+1}. This concludes the base case of the secondary induction.

The induction step is for m inherited in $CT(C)$ and $CT'(C)$. By the secondary induction hypothesis we have (†) for *super* R, from which the result follows by semantics of inherited methods.

6 Using the theorem

The standard use of an abstraction theorem is to show equivalence between two versions of a program, one using CT and the other CT'. One proves simulation for Abs and then appeals to the abstraction theorem to conclude that $\llbracket S \rrbracket$ is related to $\llbracket S \rrbracket'$ for any client program S. Finally, one appeals to an *identity extension lemma* that says the relation is the identity for programs where the encapsulated representation is not visible.

For heap encapsulation, which does not have a simple correspondence to program structure, it is not obvious how best to formulate identity extension. Moreover, in this paper we have not formalized the notion of "main program" so in particular this means there is not a standard initial state. (The obvious initial state is with a single object of type Main; it is disciplined and has no reachable Abs.)

We have found straightforward but elegant formulations of program equivalence and identity extension in terms of specifications, which can express encapsulation via partitioning. The identity extension and program equivalence results are based on results expressed more directly in terms of the semantics; in this report we include only the latter.

Lemma 13 (identity extension). If $\mathcal{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s)\ (h', s')$ then $encap\ Abs\,(h, s) \sim_\beta encap\ Abs\,(h', s')$.

A state (h, s) is *Abs-free* if no $o \in dom\, h$ has type name $\leq Abs$.

Lemma 14 (inverse identity extension). Suppose (h, s) and (h', s') are Abs-free. If $(h, s) \sim_\beta (h', s')$ and β is total on h, h' then $\mathcal{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s)\ (h', s')$.

Definition 22 (program equivalence). Suppose programs $CT, (\Delta; \Gamma \vdash S)$ and $CT', (\Delta; \Gamma \vdash' S')$ are such that CT, CT' are comparable and properly annotated, and moreover S, S' are properly annotated. The programs are equivalent iff for all σ instantiating Δ, and for all disciplined, Abs-free (h, s) and (h', s') in $[\![\mathsf{heap} \otimes \sigma\Gamma]\!]$ and all β with β total on h, h' and $(h, s) \sim_\beta (h', s')$, there is some $\beta_0 \supseteq \beta$ with

$$encap\ Abs\ ([\![S]\!]\sigma\hat{\mu}(h, s)) \sim_{\beta_0} encap\ Abs\ ([\![S']\!]'\sigma\hat{\mu}'(h', s'))$$

where $\hat{\mu} = [\![CT]\!]$ and $\hat{\mu'} = [\![CT']\!]'$.

Proposition 4 (simulation and equivalence). Suppose programs $CT, (\Delta; \Gamma \vdash S)$ and $CT', (\Delta; \Gamma \vdash' S)$ are properly annotated and $\mathcal{R}$ is a simulation from CT to CT'. If $name(\Gamma\ \mathsf{self}) \neq Abs$ then the programs are equivalent.

7 Discussion

As compared with previous work on the Boogie discipline, we have imposed some additional restrictions that merit consideration.

- An object of class Abs can only be packed in class Abs. Because the coupling relation imposes the user-defined basic coupling only when an Abs-object is packed, this restriction is necessary for modular reasoning about class Abs.
  The most common use of **pack** is to pack, in code of class C, self to class C. In examples where some class C is used in the representation of a class Abs it makes sense for code of Abs to pack an object to C —if the code is in procedural style, manipulating C objects as records rather than invoking methods on them in OO style. Our restriction then precludes instantiating our theory with $Abs := C$. But if C is being manipulated by code of other classes then it is not surprising that a per-class theory of encapsulation is inapplicable.
- Similarly, for **setown** o **to** (p, C), care must be taken to prevent arbitrary code from moving objects across the encapsulation boundary for Abs in ways that do not admit modular reasoning. One would expect that code outside Abs cannot move objects across the boundary at all. But we have chosen the least restrictive condition for which the abstraction theorem holds: if **setown** o **to** (p, C) occurs in code outside Abs, and o is initially inside the island for some Abs-object, then it must end up in the island for some Abs-object.
- The restriction that an Abs object cannot own other Abs objects does not preclude containers holding containers, because a container does not own its content. It does preclude certain recursive situations. Consider a class $Node$ used as nodes of a linked list storing values in sorted order. The sorting invariant can be expressed by a decentralized invariant by using a peer/friend discipline [30, 36] but this is not encompassed by our theory. Another alternative considered in the Boogie papers is for each $Node$ to own its successor, so its invariant can mention the successor's fields. Our theory does not preclude recursive ownership in general, but cannot be instantiated with $Abs := Node$. This does not seem too onerous, because to

reason about change of representation the typical situation is that $Node$ is an internal data structure and it is then a non-recursive container class C that is manipulated by clients. Our theory admits the instantiation $C := Abs$ even $Node$s can own $Node$s.
This technical restriction does not appear essential for soundness. As an alternative, invariants could be required to satisfy a healthiness condition that amounts to saying that the condition on an island inside an island is recursively structured. Technically this would look similar to the healthiness condition used by Cavalcanti and Naumann [12, Def. 5] to deal with recursive class types. In the present work we prefer to avoid nested islands, in part to highlight connections with separation logic.

–

8 Related and future work

Soundness for the *inv*/*own* discipline without subclassing or generics is proved on a semantic basis in [36] which includes a discipline for "friend" invariants. For subclasses but not generics, soundness is argued in detail in [5] and also in [30] which deals with peers; but no semantic basis is given nor is it entirely clear which language features are considered. In this paper we use a denotational semantics to validate the discipline in the presence of generics and subclassing. It turns out that there is little interaction with generics.

Representation independence is proved in [2] for a language with subclassing but not generics on the basis of ownership confinement imposed using restrictions expressed in terms of ordinary types; but these restrictions disallow ownership transfer. The results are extended to encompass ownership transfer in [4] but at the cost of substantial technical complications and the need for reachability analysis at transfer points, which are designated by explicit annotations. In this paper we prove representation independence using the discipline to control ownership which may be transferred flexibly.

The discipline may seem somewhat onerous, but the Boogie project is exploring the inference of annotations. The advantage of this over types is that, while simple cases can be checked automatically, complicated cases can be checked rather than simply rejected (though that's fodder for further research papers). For representation independence it suffices to take each $\mathcal{I}^C$ to be everywhere true; the remaining conditions involve fields with finite domain types (*inv* and *com*, though not *own*). This and the simple discipline for unpack/pack seems promising for automated checking.

An invariant justifies or explains a program and is thus suitable for inclusion in the program text. A coupling connects two programs, and is used only to justify a revision. Thus we only focus on a single class to be revised and its basic coupling, whereas we expect an invariant $\mathcal{I}^C$ to be given for every class C. The generalization to a small group of related classes is important, as revisions often involve several related classes; the "friend" and "peer" dependencies of [30, 6, 36] may play a key role.

For proof of preservation by the methods of the revised class one can use ordinary program reasoning together with special rules for how code with the same structure preserves relations [16, 20, 34, 7, 49]. In future work we give specific rules focusing on the particular challenges of our discipline: re-establishing the basic coupling with **pack** and transferring ownership across abstraction boundaries.

The higher order frame rule of separation logic [38] also supports ownership transfer, though for invariants rather than simulations and in a very simple language. It would be interesting to see

how our semantic development could be realized in separation logic, in particular how the induced coupling's multiple islands could be expressed using iterated separation. Although our couplings disallow "nested" islands, in general invariants and instantiable classes give rise to nesting of sub-heaps and it is not obvious how this can be handled.

References

1. K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 2 edition, 1997.
2. A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 2002. Accepted, subject to revision. Extended version of [3].
3. A. Banerjee and D. A. Naumann. Representation independence, confinement and access control. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 166–177, 2002.
4. A. Banerjee and D. A. Naumann. Ownership transfer and abstraction. Technical Report TR 2004-1, Computing and Information Sciences, Kansas State University, 2003.
5. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.
6. M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In D. Kozen, editor, *Mathematics of Program Construction*, pages 54–84, 2004.
7. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25, 2004.
8. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.
9. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 213–223, 2003.
10. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *OOPSLA '98 Conference Proceedings*, volume 33(10) of *SIGPLAN*, pages 183–200. ACM, Oct. 1998.
11. C. Calcagno, P. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Comput. Sci.*, 298(3):557–581, 2003.
12. A. L. C. Cavalcanti and D. A. Naumann. Forward simulation for data refinement of classes. In L. Eriksson and P. A. Lindsay, editors, *Formal Methods Europe*, volume 2391 of *LNCS*, pages 471–490, 2002.
13. D. Clarke. Object ownership and containment. Dissertation, Computer Science and Engineering, University of New South Wales, Australia, 2001.
14. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, Nov. 2002.
15. D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In J. L. Knudsen, editor, *ECOOP 2001 - Object Oriented Programming*, 2001.
16. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
17. D. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Technical Report 156, COMPAQ Systems Research Center, July 1998.
18. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.
19. P. Gardiner and C. Morgan. Data refinement of predicate transformers. *Theoretical Comput. Sci.*, 87:143–162, 1991.

20. D. Gries. Data refinement and the tranform. In *Program Design Calculi*. Springer, 1993. International Summer School at Marktoberdorf.
21. E. Gunnerson. *A Programmer's Introduction to C#*. Apress, Berkeley, CA, 2000.
22. C. A. R. Hoare. Proofs of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
23. M. Hofmann and F. Tang. Implementing a program logic of objects in a higher-order logic theorem prover. In *TPHOLs*, pages 268–282, 2000.
24. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. Syst.*, 23(3):396–459, May 2001.
25. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. Technical Report NIII-R0318, Computing Science Institute, University of Nijmegen, 2003. To appear in International Symposium on Software Security, November 2003.
26. A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *PLDI*, pages 1–12, 2001.
27. A. Kennedy and D. Syme. Transposing F to C#: Expressivity of polymorphism in an object-oriented language. *Concurrency and Computation: Practice and Experience*, 16(7), 2004.
28. D. Lea. *Concurrent Programming in Java*. Addison-Wesley, second edition, 2000.
29. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, Oct. 2000.
30. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 491–516, 2004.
31. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Trans. Prog. Lang. Syst.*, 24(5):491–553, 2002.
32. B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.
33. R. Milner. An algebraic definition of simulation between programs. In *Proceedings of Second Intl. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.
34. C. Morgan. *Programming from Specifications, second edition*. Prentice Hall, 1994.
35. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. Number 2262 in LNCS. Springer, 2002.
36. D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *IEEE Symp. on Logic in Computer Science (LICS)*, pages 313–323, 2004.
37. G. C. Necula. Proof-carrying code. In *POPL*, 1997.
38. P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 268–280, 2004.
39. P. W. O'Hearn and R. D. Tennent. *Algol-like Languages (Two volumes)*. Birkhäuser, Boston, 1997.
40. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6, 1976.
41. C. Pierik, D. Clarke, and F. S. de Boer. Creational invariants. In *Proceedings of ECOOP workshop on Formal Techniques for Java-like Programs*. 2004. Technical Report NIII-R0426, University of Nijmegen.
42. G. Plotkin. Lambda definability and logical relations. Technical Report SAI-RM-4, University of Edinburgh, School of Artificial Intelligence, 1973.
43. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Defaulting generic java to ownership. In *ECOOP Workshop on Formal Techniques for Java Programs*, 2004.
44. J. C. Reynolds. Types, abstraction, and parametric polymorphism. In R. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, 1984.
45. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
46. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, NY, second edition, 2002.

47. R. D. Tennent. Correctness of data representations in Algol-like languages. In A. W. Roscoe, editor, *A Classical Mind: Essays Dedicated to C. A. R. Hoare*. Prentice-Hall, 1994.
48. J. Vitek and B. Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.
49. H. Yang. Relational separation logic. submitted, 2004.
50. D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET Common Language Runtime. In *POPL*, pages 39–51, 2004.
51. T. Zhao, J. Palsberg, and J. Vitek. Lightweight confinement for featherweight Java. In *OOPSLA*, 2003.