State Based Encapsulation for Modular Reasoning about Behavior-Preserving Refactorings

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

[1] IMDEA Software Institute, Madrid, Spain
[2] Stevens Institute of Technology, Hoboken NJ 07030 USA

Abstract. A properly encapsulated data representation can be revised for refactoring or other purposes without affecting the correctness of client programs and extensions of a class. But encapsulation is difficult to achieve in object-oriented programs owing to heap based structures and reentrant callbacks. This chapter shows that it is achieved by a discipline using assertions and auxiliary fields to manage invariants and transferrable ownership. The main result is representation independence: a rule for modular proof of equivalence of class implementations.

1 Introduction

You are responsible for a library consisting of many Java classes. While fixing a bug or refactoring some classes, you revise the implementation of a certain class in a way that is intended not to change its observable behavior, e.g., an internal data structure is changed for reasons of performance. You are in no position to check, or even be aware of, the many applications that use the class via its instances or via instances of its subclasses if any. 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 —though progress has been made, as reported in the companion chapters [29,55,60,38]. One 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 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), it may be possible to show equivalence of behavior. If any specifications are available they can be taken into account using assert statements.

There is a standard technique for proving equivalence [36,46,27]: Define a *coupling relation* to connect the states of the two versions and prove that it has the *simulation property*, i.e., it holds initially and is preserved by parallel execution of the two versions of each method. In most cases, one would want to define a *local coupling* relation

for a single pair of instances of the class, as methods act primarily on a target object (self) and the *island* [6], i.e., a group of objects that comprise its internal representation. An *induced coupling* for complete states is then obtained by a general construction. A language with good encapsulation should enjoy a *representation independence* property 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. This means that from a client's point of view the behaviors of the two programs are the same. 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 object oriented 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 of interest. 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 chapter is a representation independence theorem using a state-based discipline for heap encapsulation and control of callbacks.*

Extant theories of data abstraction assume, in one way or another, a hierarchy of abstractions such that control does not reenter an encapsulation boundary while already executing inside it. It is commonplace in object oriented programs for a method m acting on some object o to invoke a method on some other object which in turn leads to invocation of some method on o —possibly m itself— while the initial invocation of m is in progress. This makes it difficult to reason about when an object's invariant holds [38,47]; we give an example later.

There is an analogous problem for reasoning with simulations. The first work on representation independence for programs with shared objects [6] provides an abstraction theorem that deals with sharing and is sound for programs with reentrant callbacks —but it is not easy to apply in cases where reentrant callbacks are possible. The theorem allows the programmer to assume that all methods preserve the coupling relation when proving simulation, i.e., when reasoning about parallel execution of two versions of a method of the class of interest. This assumption is like verifying a procedure implementation under the assumption that called procedures are correct. But the assumption that called methods preserve the coupling is of no use if the call is made in an uncoupled intermediate state. For the examples in [6], we resort to ad hoc reasoning for examples involving callbacks.

In the "Boogie methodology" [11,43], reentrancy is managed using an explicit auxiliary (or *ghost*) field *inv* to designate states in which an object invariant is to hold. The ghost field is extra instrumentation added to a program for reasoning purposes but does not influence data flow or control flow in the original program in any manner; thus the behavior of the annotated program is the same as that of the original program. Encapsulation is achieved using a notion of ownership represented by an auxiliary mutable field *own*. This is more flexible than type-based static analyses because the ownership

invariant need only hold in certain flagged states. Heap encapsulation is achieved not by disallowing boundary-crossing pointers but by limiting, in a state-dependent way, their use. Reasoning hinges on a global *program invariant* that holds in all states, using *inv* fields to track which object invariants are temporarily not in force because control is within their encapsulation boundary. When *inv* holds, the object is said to be *packed*; a field may only be updated when the object is unpacked.

In this chapter we adapt the *inv*/*own* discipline to proving class equivalence by simulation. The *inv* fields make it possible for an induced coupling relation to hold at some pairs of intermediate states during parallel execution of two alternative implementations. This means that the relation-preservation hypothesis of the abstraction theorem can be used at intermediate states even when the local coupling is not in force. So per-method modular reasoning is fully achieved. 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, but for representation independence we need to control reads as well. The discipline also allows ownership transfer quite freely, though as we will discuss later, 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" [28]) is allowed but must occur only in the code of the encapsulating class; even then, it poses a difficult technical challenge. The significance of our adaptations is discussed in Section 8.

A key insight is that, although transferring ownership and packing/unpacking involve only ghost fields that cannot affect program execution, it is useful to consider them to be observable. It is difficult to reason about two versions of a class, in a modular way, if they differ in the way objects cross the encapsulation boundary or in the points at which methods assume the invariant is in force. The requisite similarity can be expressed using assert statements, so we can develop a theory based on this insight without the need to require that the class under revision has any specifications.

The main contributions of this chapter are (a) formulation of a notion of instance-based coupling analogous to invariants in the *inv*/*own* discipline; (b) proof of a representation independence theorem for a language with inheritance and dynamic dispatch, recursive methods and callbacks, mutable objects, type casts, and recursive types; and (c) results on identity extension and use of the theorem to prove program equivalence. Together these constitute a rule by which the reasoner considers just the methods of the revised class and concludes that the two versions yield equivalent behavior for any program context.

The theorem allows ownership transfers that cross encapsulation boundaries: from client to abstraction [28], between instances of the abstraction, and even from abstraction to client [54,43,4]. The theorem supports the most important form of modularity: reasoning about one method implementation (or rather, one corresponding pair) at a time —on the assumption that all methods preserve the coupling (even the one in question, modulo termination). The theorem also supports local reasoning in the sense that a single instance (or pair of instances) is considered, together with the island comprised of its currently encapsulated representation objects.

The *inv*/*own* discipline can be used in any verification system that supports ghost variables and assertions in first order logic. Our formalism treats predicates in assertions semantically, avoiding ties to any particular logic or specification formalism. The original discipline is a core feature of the Spec# program verifier[1] for sequential C# programs [12] and our adaptations would not be difficult to implement. The original discipline has been adapted to concurrency and implemented as the core discipline for the VCC program verifier for multithreaded C programs [24].

This chapter is a revised version of the paper originally appearing in European Conference on Object-Oriented Programming [7]. Subsequent to the work reported in that paper, there have been further studies of representation independence using state-based notions of encapsulation. As discussed under related work (Section 7), these works include both foundational analyses and treatment of advanced language features (type polymorphism and higher order heaps —but so far not inheritance and nominal subtyping as in Java-like languages). Our work remains relevant not only for its conceptual perspective but also because it applies to a programming discipline for which there is tool support and which has been applied to substantial practical examples.

Outline. Sect. 2 sketches the *inv*/*own* discipline. It also sketches an example of the use of simulation to prove equivalence of two versions of a class involving reentrant callbacks, highlighting the problems and the connection between our solution and the *inv*/*own* discipline. Sect. 3 formalizes the language for which our result is given and Sect. 4 formalizes the discipline in our semantics. Sect. 5 gives the main definitions—proper annotation, coupling, simulation—and the abstraction theorem. Sect. 6 connects simulation with program equivalence. Sect. 7 discusses related work. Sect. 8 discusses future work and assesses our adaptation of the discipline.

2 Background and overview

2.1 The *inv*/*own* discipline

To illustrate the challenge of reentrant callbacks as well as the state based ownership discipline, we consider a class Queue that maintains a queue of tasks. Each task has an associated limit on the number of times it can be run. Method Queue.runAll runs each task that has not exceeded its limit. For simplicity we refrain from using interfaces; class Task in Fig. 1 serves as the interface for tasks. Class Qnode in the same figure is used by Queue which maintains a singly linked list of nodes that reference tasks. Field count tracks the number of times the task has been run.

In our illustrative language, all methods are dynamically dispatched and have public visibility; values of class type are references to mutable objects. For brevity we omit initialization and constructors throughout the examples. For reference to fields of self, we write f to abbreviate self.f, including the special fields *inv*, *com*, and *own*.

Fig. 2 gives class Queue; the annotations, in gray, will be discussed later. One intended invariant of Queue is that no task has been run more times than its limit. This

[1] Available at `http://specsharp.codeplex.com/`

```
class Task {   void run(){ }   }
class Qnode {
  Task tsk;   Qnode nxt;   int count, limit;
  invariant tsk ≠ null ∧ 0≤count≤limit;
  ... // constructor elided (in subsequent figures these ellipses are elided too)
  void run() { tsk.run(); count := count+1; }
  void setTsk(Task t, int lim) {
    unpack self from Qnode;
    tsk := t; limit := lim; count := 0; pack self as Qnode; }   }
  ... // other methods getNxt, getCount, getLimit, omitted
```

Fig. 1. Classes Task and Qnode. The unpack/pack statements are discussed later.

is expressed, in a decentralized way, by the invariant declared in Qnode. Some notation: we write $\mathscr{I}^{Qnode}$(o) for the predicate o.tsk≠null and o.count≤o.limit. That is, the declared invariant is considered to be a predicate parameterized on self.

Another intended invariant of Queue is that runs is the sum of the count fields of the nodes reached from tsks. This is the declared $\mathscr{I}^{Queue}$ of Fig. 2. (The reader may think of other useful invariants, e.g., that the list is null-terminated.) Note that at intermediate points in the body of Queue.runAll, $\mathscr{I}^{Queue}$ does not hold because runs is only updated after the loop. In particular, $\mathscr{I}^{Queue}$ does not hold at the point where p.run() is invoked.

For an example reentrant callback, consider tasks of the following type.

```
class RTask extends Task { Queue q;   ...
                           void run(){ q.runAll(); } }
```

Consider a state in which o points to an instance of Queue and the first node in the list, o.tsks of type RTask, has count=0 and limit=1. Moreover, suppose field q of the first node's task has value o. That is, o.tsks.q = o. Invocation of o.runAll diverges: before count is incremented to reflect the first invocation, the task makes a *reentrant call* on o.runAll —in a state where $\mathscr{I}^{Queue}$ does not hold. In fact runAll again invokes run on the first task and the program fails due to nonterminating recursion.

As another example, suppose RTask.run is instead `void run(){q.getRuns();}` . This seems harmless, in that the implementation of getRuns neither depends on $\mathscr{I}^{Queue}$ nor invokes any methods. Indeed this code is useful, returning a lower bound on the actual sum of runs. It typifies methods like state readers in the observer pattern, that are intended to be invoked as reentrant callbacks.

The examples illustrate that it is sometimes but not always desirable to allow a reentrant callback when an object's invariant is violated temporarily by an "outer" invocation. The ubiquity of method calls makes it impractical to require an object's invariant to be reestablished before making *any* call —e.g., the point between n.setTsk and n.setNxt of method add in Fig. 2— although this is sound and has been proposed in the literature on object oriented verification [35,45]. A better solution is to prevent just the undesirable reentrant calls.

One could make the invariant an explicit precondition, e.g., for runAll but not getRuns. This puts responsibility on the caller, e.g., RTask.run cannot establish the precondition

```
class Queue {
  Qnode tsks;
  int runs := 0;
  invariant runs = (Σp ∈ tsks.nxt* | p.count);
  int getRuns() { result := runs; }
  void runAll() {
    assert inv = Queue && ! com ;
    unpack self from Queue ;
    Qnode p := tsks;   int i := 0;
    while p ≠ null do {
      if p.getCount() < p.getLimit() then p.run(); i := i+1; fi;
      p := p.getNxt(); }
    runs := runs + i;
    pack self as Queue; }
  void add(Task t, int lim){
    assert inv = Queue && ! com;
    unpack self from Queue;
    Qnode n := new Qnode; setown n to (self,Queue);
    n.setNxt(tsks); n.setTsk(t,lim); tsks := n;
    pack self as Queue; } }
```

Fig. 2. Class Queue, with selected annotations. The assertions here serve as preconditions, as we refrain from formalizing method contracts.

and is thus prevented from invoking runAll. But an object invariant like $\mathscr{I}^{Queue}$ involves encapsulated state not suitable to be visible in a public specification.

The solution of the *inv*/*own* discipline [11,43] is to introduce a public ghost field, *inv*, that serves as a flag to explicitly indicate whether the invariant is in force[2] when $o.inv$ holds we say object o is *packed*. Special statements **pack** and **unpack** set and unset *inv*.

A given object is an instance not only of its class but of all its superclasses, each of which may have invariants. The methodology takes this into account as follows. Instead of *inv* being a boolean, as in the simplified explanation above, it ranges over class names C such that C is a superclass of the object's allocated type. That is, it is an invariant (enforced by typing rules) that $o.inv \geq$ *type* o, where *type* o is the dynamic type of o. Roughly, unpacking in a method of class C sets *inv* to $superC$. The discipline requires certain assertions preceding pack and unpack statements, as well as field updates, to ensure that the following is a *program invariant* (i.e., it holds in all reachable states, in the sense of small-step semantics).

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

[2] Without exposing the actual invariant. This resembles abstract predicates [55]. Our condition (1) is akin to the association between an abstract predicate and its definition in their work.

for all C and all allocated objects o. That is, if o is packed at least to class C then the invariant $\mathscr{I}^C$ for C holds. Perhaps the most important stipulated assertion is that $\mathscr{I}^C(o)$ is required as precondition for packing o to level C.

Fig. 2 shows how the discipline is used for class Queue. Assertions impose preconditions on runAll and add which require that the target object is packed to Queue. In runAll, the **unpack** statement sets *inv* to the superclass of Queue, putting the task in a position where it cannot establish the precondition for a reentrant call to runAll, although it can still call getRuns which imposes no precondition on *inv*. After the update to runs, $\mathscr{I}^{Queue}$ holds again as required by the precondition (not shown) of **pack**. The ghost field *com* is discussed below.

In order to maintain (1) as a program invariant, it is necessary to control updates to fields on which invariants depend. The idea is that, to update field f of some object p, all objects o whose invariant depends on $p.f$ must be unpacked. Put differently, $\mathscr{I}(o)$ should depend only on state encapsulated for o. The discipline uses a form of ownership for this purpose: $\mathscr{I}(o)$ may depend only on objects transitively owned by o. For example, an instance of Queue owns the Qnodes reached from field tsks.

Ownership is embodied in an auxiliary field *own*, so that if $p.own = (o,C)$ then o directly owns p and an admissible invariant $\mathscr{I}^D(o)$ may depend on p for types D with *type* $o \leq D \leq C$. Typically, o has a field, declared or inherited in C, that points to or reaches p, by way of which there is a dependency.

The objects transitively owned by o are called its *island*. For modular reasoning, it is not feasible to require as an explicit precondition for each field update that all transitive owners are unpacked. A third ghost field, *com*, is used to enforce a protocol whereby packing/unpacking is dynamically nested or bracketed. They need not be lexically nested, but typically that would be the case, for which purpose Spec# and VCC provide an 'expose block'.

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". The second says that a committed object is fully packed. These make it possible for an assignment to $p.f$ to be subject only to the precondition $p.inv > C$ where C is the class that declares f —because owing to the additional invariants the condition $p.inv > C$ implies that all dependents of $p.f$ are unpacked.

The invariants are formalized in Def. 4 in Sect. 4. The stipulated preconditions appear in Table 1, which also describes the semantics of the pack and unpack statements in detail.[3] The diligent reader may enjoy completing the annotation of Fig. 2 according to the rules of Table 1. Consult [11,43] for more leisurely introductions to the discipline.

2.2 Representation independence

At this point the reader may expect an alternate implementation of class Queue, perhaps using an array or doubly linked list of nodes. But recall that an invariant for a

[3] Preconditions like $e \neq$ **null** and "e not error" are needed for the rest of the precondition to be meaningful. Different verification systems make different choices in handling errors in assertions. Our formulation follows [51] and differs superficially from [11,43].

assert $x.inv > C$; /* where C is the class that declares f; i.e., $f \in dom(dfields C)$ */
$x.f := y$

assert $e.inv = superC \wedge \mathscr{I}^C(e) \wedge (\forall p \mid p.own = (e,C) \Rightarrow \neg p.com \wedge p.inv = type\, p)$;
pack e **as** C /* sets $e.inv := C$ and sets $p.com := true$ for all p with $p.own = (e,C)$ */

assert $e.inv = C \wedge \neg e.com$;
unpack e **from** C /* sets $e.inv := superC$ and $p.com := false$ for all p with $p.own = (e,C)$ */

assert $x.inv = \mathsf{Object} \wedge (e_2 = \mathbf{null} \vee e_2.inv > C)$;
setown x **to** (e_2,C) /* sets $x.own := (e_2,C)$ */

Table 1. Stipulated preconditions of field update and of the special commands. For brevity we leave implicit certain non-nullity conjuncts needed so the field references make sense: $x \neq \mathbf{null}$ and $e \neq \mathbf{null}$ above, but not for e_2.

class C may depend on objects owned either at C or at some superclass of C. We aim to generalize from invariants to coupling relations, and in treating them precisely we distinguish between objects owned at C and objects owned at some superclass. For the sake of an example, we consider a somewhat contrived subclass of Queue, with two alternate implementations.

The basic setup. Consider the subclass AQueue of Queue declared in Fig. 3. It maintains an array, actsks, of tasks which is used in an overriding declaration of runAll intended as an optimization for the situation where many tasks are inactive (i.e., have reached their limit).

Method add exhibits a typical pattern: unpack to establish the condition in which a super call can be made (since the superclass unpacks from its own level); after that call, reestablish the current class invariant. In some sense, this pattern is necessary in order to match the expectations of a caller —that the object is packed to its type— with assumptions of a method implementation in some class —that the object is packed to that class. It is implemented in Spec# by re-verifying inherited code, and explained in the original paper [11] by translating inheritance into 'stub' methods consisting either of a super call or the unpack/super/pack pattern. We return to this topic in Sect. 6.

The implementation of Fig. 3 does not set actsks[i] to null immediately when the task's count reaches its limit; rather, that situation is detected on the subsequent invocation of runAll. An alternative implementation is given in Fig. 4; it uses a different data structure and handles the limit being reached as soon as it happens. Both implementations maintain an array of Qnode, but in the alternative implementation, its array artsk is accompanied by a boolean array brtsk. Instead of setting entry i null when the node's task has reached its limit, brtsk[i] is set false.

We claim that the two versions are equivalent, in the context of arbitrary client programs and subclasses. We would like to argue as follows. Let $filt1(o.\mathsf{actsks})$ be the sequence of non-null elements of $o.\mathsf{actsks}$ with $\mathsf{count} < \mathsf{limit}$. Let $filt2(ts,bs)$ take an

```
class AQueue extends Queue {
  private Qnode[ ] actsks;
  private int  alen;
  void  add(Task t, int  lim) {
    assert inv = AQueue && ! com;
    unpack self from AQueue;
    super.add(t,lim); actsks[alen] := tsks; alen := alen+1;
    pack self as AQueue; }
  void  runAll() {
    assert inv = AQueue && ! com;
    unpack self from AQueue;
    int  i := alen - 1;
    while i ≥ 0 do  {
      Qnode qn := actsks[i];
      if  qn ≠ null then  if  qn.getCount() < qn.getLimit()
                          then  qn.run();
                                unpack self from Queue; runs++; pack self as Queue;
                          else  actsks[i] := null; fi; fi;
      i := i - 1; }
    pack self as AQueue; } }
```

Fig. 3. First version of Class AQueue. An invariant: actsks[0..alen-1] 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). The elided constructor allocates actsks and we ignore the issue of the array becoming full.

array ts of tasks and a same-length array bs of booleans and return the subsequence of those tasks n in ts where bs is true and n.count $<$ n.limit. Consider the following relation that connects a state for an instance o of the original implementation (Fig. 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. That is, for each method of AQueue, parallel execution of the two versions from a related pair of states results in a related pair of outcomes. (For this to hold we need to conjoin to the relation the invariants associated with the two versions, e.g., the second version requires artsk.length=brtsk.length.) In brief: the coupling relation is *preserved*.

Coupling relations. In general, a *local coupling* is a binary relation on islands. It relates the state of an island for one implementation of the class of interest with an island for the alternative.

Fig. 5 depicts local coupling involving two instances of AQueue. The left side of the figure is an instance of some subclass of AQueue, sliced into the fields of Queue, AQueue, and subclasses; dashed lines show the objects encapsulated at the two levels relevant to reasoning about AQueue —namely the Qnodes reached from tsks and the array actsks. On the right is an instance for the alternate implementation of AQueue. It

```
class AQueue extends Queue {
  private Qnode[ ] artsk;
  private boolean[ ] brtsk;
  private int len;
  void add(Task t, int lim) {
    assert inv = AQueue && ! com;
    unpack self from AQueue;
    super.add(t,lim); artsk[len] := tsks; brtsk[len] := true; len := len+1;
    pack self as AQueue; }
  void runAll() {
    assert inv = AQueue && ! com;
    unpack self from AQueue;
    int i := len - 1;
    while i ≥ 0 do {
      if brtsk[i] then
        Qnode n := artsk[i];
        int diff := n.limit - n.count;
        if diff ≠ 0 then n.run();
                        unpack self from Queue; runs++; pack self as Queue; fi;
        if diff = 1 then brtks[i] := false; fi; fi;
      i := i - 1; }
    pack self as AQueue; } }
```

Fig. 4. Alternative implementation of AQueue.

is the connection between these two islands that is of interest to the programmer. The 'a'... 'd' of the figure indicate that both versions reference the same sequence of tasks, although those tasks are not part of the islands.

A local coupling lifts to an *induced coupling* relation on the complete program state: Two heaps are related by the induced coupling provided that (a) they can be partitioned into islands and (b) the islands can be put into correspondence so that each corresponding pair is related by the local coupling. Moreover, the remaining objects (not in an island) are related by equality. More precisely, equality modulo a bijection on locations, to take into account differences in allocation between the two versions. For example, 'a'... 'd' on each side of the figure might well be different references, because the two runs might allocate different references, due to different allocation by the two implementations of AQueue. But the difference will be unobservable, because the programming language does not allow comparison of references except for equality with other references. The details of lifting a coupling are not obvious and are formalized later.

Abstraction theorem. The goal is to show that the two versions of AQueue are equivalent, when used by an arbitrary client. Because the induced coupling is a kind of identity relation —on the client-visible part of the state— the two versions of a complete program have equivalent behavior provided that they preserve the induced coupling. The

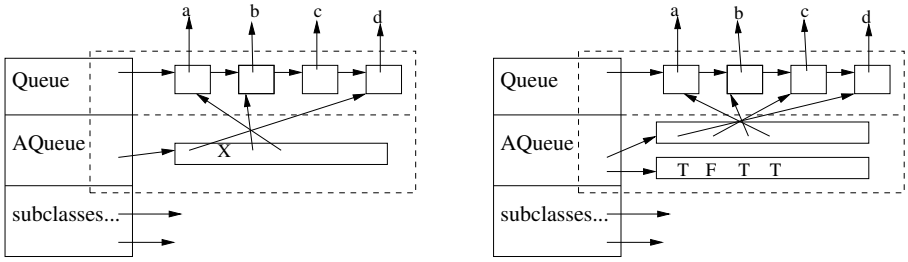


Fig. 5. Depiction of local coupling. This involves two instances of AQueue. Each is depicted as being sliced into the fields declared in class Queue, the fields declared in class AQueue, and those declared in subclasses there are down to the dynamic type of the instance.

abstraction theorem says that for a complete program to preserve the induced coupling, it is sufficient that the induced coupling is preserved by methods of AQueue —provided there is sufficient encapsulation, specifically an adaptation of the *inv*/*own* discipline.

At first glance one might expect the proof obligation to be that each method of AQueue preserves the local coupling, and indeed this will be the focus of reasoning in practice. But in general a method may act on more than just the island for self, e.g., by invoking methods on client objects or on other instances of AQueue. As a simple example, consider the version of RTask.run that calls q.getRuns(), and an execution where o.tsks.q≠o. So in general the proof obligation is formalized in terms of the induced coupling.[4]

In fact the proof obligation is not simply that each corresponding pair of method implementations preserves the coupling, but rather that they preserve the coupling *under the assumption that any method they invoke preserves the coupling*.[5] There is also a proof obligation for initialization but it is straightforward so we do not discuss it in connection with the examples.

For example, in the case of method runAll, one must prove that the implementations given in Fig. 3 and in Fig. 4 preserve the coupling on the assumption that the invoked methods getCount, getLimit, Qnode.run, etc. preserve the coupling. The assumption is not so important for getCount or getLimit. For one thing, it is possible to fully describe their simple behavior. For another, the alternative implementation of runAll does not even invoke these methods but rather accesses the fields directly.

The assumption about Qnode.run is crucial, however. Because run invokes, in turn, Task.run, essentially nothing is known about its behavior. For this reason both implementations of runAll invoke run on the same tasks in the same order; otherwise, it is hard to imagine how equivalence of the implementations could be verified in a modu-

[4] It also provides a technical simplification: we do not need to formulate a semantics of programs acting on heap fragments.

[5] The reason this is sound is similar to the justification for proof rules for recursive procedures: it is essentially the induction step for a proof by induction on the maximum depth of the method call stack.

lar way, i.e., reasoning only about class AQueue. But here we encounter the problem with simulation based reasoning that is analogous to the problem with invariants and reentrant callbacks. There is no reason for the coupling to hold at intermediate points of the methods of AQueue. If a method is invoked at such a point, the assumption that the called method preserves the coupling is of no use —just as the assumption of invariant-preservation is of no use if a method is invoked in a state where the invariant does not hold.

The *inv*/*own* discipline solves the invariant problem for an object o by replacing the declared invariant $\mathscr{I}(o)$ with an implication —see (1)— that is true in all states. As with invariants, so too with couplings: It does not make sense to ask a coupling to hold in every state, because two different implementations with nontrivial differences do not have lockstep correspondence of states. (For example, imagine that in the alternative version, the arrays are compressed every 100th invocation of runAll.) Our generalization of the *inv*/*own* idea is that the local coupling relation for a particular (pair of) island(s) is conditioned on an *inv* field so that the local coupling may hold in *some pairs of states* at intermediate points —in particular, at method calls that can lead to reentrant callbacks.

Proving the example. Consider corresponding instances o, o' of the two versions of AQueue. The local coupling serves to describe the corresponding pair of islands when o and o' are packed. So the induced coupling relation on all program states requires corresponding pairs of islands to satisfy the local coupling just when they are packed and the client visible states to be related by identity (modulo allocation behavior). 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) and thus include $o.inv = o'.inv$ as a conjunct of the induced coupling.

Consider the two implementations of runAll. To a first approximation, what matters is that each updates some internal state and then both reach a point where run is invoked. At that point, the *local* coupling does not hold —but the *induced* coupling relation on all states can and does hold, because the island is unpacked. In more detail it holds outside the island because the relation is the "identity" on client-visible states; note that unpacking the island does not preclude side effects outside it: however, the side effects must be the *same* for both versions. That the induced coupling holds for the island itself parallels the way $\mathscr{I}^C(o)$ can be false while $o.inv \leq C \Rightarrow \mathscr{I}^C(o)$ remains true, recall (1). So we can use the assumption about called methods to conclude that the coupling holds after the corresponding calls to run.

The hardest part of the proof for runAll is at the point where the two implementations pack self to AQueue. Just as both implementations invoke run (and on the same queue nodes), both need to pack in order to preserve the coupling. And at this point we have to argue that the local coupling is reestablished. To do so, we need to know the state of the internal structures that have been modified. We would like to argue that the only modifications are only those explicit in the code of runAll, but what about the effect of run? Owing to the preconditions on add and runAll, i.e., the requirement that Queue is packed, the only possible reentrant callbacks are to getRuns and this does no updates. (In other examples, modifies specifications would be needed at this point for modular reasoning.)

This concludes the informal sketch of how our abstraction theorem handles reentrant callbacks and encapsulation using the *inv*/*own* discipline. A more formal way of establishing the relation $o.inv = o'.inv$ between two implementations of runAll would involve reasoning in a relational program logic. The development of such a logic is a topic of active research [1,63] —no such widely accepted logic exists.

To justify reasoning along the lines sketched above, several features of the discipline need to be adapted —in ways which also make sense in terms of informal considerations of information hiding. The additional restrictions are formalized in Section 5 and their significance discussed in Section 8. As a preview we make the following remarks, using "*Abs*" as the generic name for a class for which two versions are considered.

Adapting the discipline for representation independence. We first describe the adaptations needed by field access, **pack** and hierarchical ownership. We then discuss the adaptations for ownership transfer. The adaptations are summarized in Table 2.

The discipline does not constrain field access, as reading cannot falsify an invariant predicate. However, for information hiding one expects visibility —and alias confinement —to prevent reading as well as writing encapsulated state. Information hiding is exactly what is formalized by representation independence and indeed the abstraction theorem fails if a client can read fields of encapsulated objects. For the fields of the class being revised, we can rely on scope, as indicated by the 'private' modifier on the fields of class AQueue. For representation objects, we augment the discipline by making every field access $y.f$ subject to a precondition: If y is transitively owned by some instance o of the class, *Abs*, under revision, then either the field access occurs in code of *Abs* or else self is transitively owned by o.

Another problematic feature is that "**pack** *e* **as** *C*" can occur in any class, so long as its preconditions are established. This means that, unlike traditional theories, an invariant is not simply established at initialization. In our theory the local coupling must be established preceding each "**pack** *e* **as** *Abs*". We aim for reasoning that is modular in the sense that the proof obligations are only for the two implementations of *Abs*, so we insist that **pack** *e* **as** *Abs* occurs only 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 9). To this end, it is convenient to prevent an instance of *Abs* from transitively owning another instance of *Abs*.[6] As a result, their islands are not nested. This can be achieved by a simple syntactic restriction. It does not preclude that, say, class AQueue can hold tasks that own AQueue objects, because an instance of AQueue owns its representation objects (the Qnodes), not the tasks they contain. Nor does it preclude hierarchical ownership, in general; e.g., *Abs* could own a hashtable that in turn owns some arrays. See Sect. 8 for futher discussion on this design decision.

Ownership transfer. Finally, consider ownership transfer across the encapsulation boundary. The case of transfer from client into the encapsulated abstraction is common in practice; for example, the main routine of a compiler could construct an input stream,

[6] A technical benefit is that the induced coupling does not need to be defined recursively.

assert $x.inv = \mathsf{Object} \wedge (e_2 = \textbf{null} \vee e_2.inv > C)$
/* Include the following conjunct, in contexts where the static type of self is not *Abs*. */
$\wedge\,((\exists o \mid o \succ_{Abs} x) \Rightarrow C = Abs \vee (\exists o \mid o \succ_{Abs} e_2))$;
setown x **to** (e_2, C)

/* Use the following in contexts where the static type of self is not *Abs*. */
assert $y \neq \textbf{null} \wedge (\forall o \mid o \succ_{Abs} y \Rightarrow o \succ_{Abs} \mathsf{self})$;
$x := y.f$

Table 2. Augmented preconditions for adapting the *inv*/*own* discipline for representation independence. Compare with Table 1 and see Def. 15. The preconditions for field update, **pack**, and **unpack** are unchanged from Table 1. For clarity we omit the obvious precondition $x \neq \textbf{null}$. An additional change is that **pack** *e* **as** *Abs* is disallowed outside class *Abs*.

then hand ownership to the lexer [11]. Our example can easily be adapted to such a scenario, by making the add method take ownership of its Task argument. Transfers in and out of an abstraction occur in case the abstraction is some sort of resource manager; such examples have been considered in [54,4]. In a setting like ours where there may be many instances of the abstraction, transfers may also occur between instances; in [6] we consider the example of queues that transfer owned tasks, as might be done for load balancing. We observe in [6] that the confinement invariant used there does not depend on the ownership relation being fixed. But the formalization there does not allow ownership transfer, basically because ownership confinement is formalized as a structural property of the heap rather than being explicitly encoded in the program state.

Technically, the most challenging case is where a hitherto-encapsulated object is released to a client, e.g., when a resource manager constructs fresh instances of a resource and later transfers ownership to a client. This can be seen as a deliberate exposure of representation and thus is observable behavior that must be retained in a revised version of the abstraction. Yet encapsulated data of the two versions can be in general quite different. To support modular reasoning about the two versions, it appears essential to restrict outward transfer of objects encapsulated for *Abs* to occur only in code of *Abs*. Given that restriction, it is part of the proof obligation for simulation that such transfers preserve coupling. That is, although in general encapsulated representations can be quite different between versions, any part of the representation that is transfered outward must be observably equivalent in the two versions, at the time of transfer.

We consider in detail an example that involves transfer of owned objects between instances of our example abstraction; see Fig. 6. In accord with the discipline [11,43], method xferFirst needs to be overridden in AQueue, with the unpack/pack forcing a check that the invariant of AQueue is maintained; in this case, additional code is needed to maintain the invariant. That additional code would be different for the two versions of AQueue, and the **pack** statements trigger an obligation to show that the local coupling is preserved for both queues.

```
/* In Queue */
void xferFirst(Queue q) {
  assert tsks ≠ null ∧ ...
  unpack self from Queue; unpack q from Queue;
  Qnode t := tsks;
  unpack t from Qnode;
  tsks := tsks.nxt; t.nxt := q.tsks; q.tsks := t;
  setown t to (q,Queue);
  pack t as Qnode; pack q as Queue; pack self as Queue; }

/* In AQueue */
void xferFirst(Queue q) {
  assert tsks ≠ null ∧ q is AQueue ∧ ...
  unpack self from AQueue; unpack q from AQueue;
  super();
  update the arrays to maintain invariants of AQueue;
  pack q as AQueue; pack self as AQueue;
```

Fig. 6. Possible addition to class Queue in which ownership of self's first task is transferred to another queue. Possible corresponding addition to class AQueue.

In this example, the transfer itself is in the superclass of *Abs*; indeed, the transferred node is owned at Queue. One can also imagine a variation that transfers all of the nodes from one queue to another, the latter having no tasks initially. In this variation, the overrides in the two versions of AQueue could also transfer the arrays owned at AQueue.

Finally, consider a variation where it is code in Queue that transfers the arrays. This would be rather strange. Indeed, we are assuming the fields of AQueue are not accessed in Queue. But it would be possible for some code in AQueue to "leak" the arrays, for example by assigning to a field of Queue. Our adaptation of the discipline restricts scenarios like this: transfer of an object p owned at *Abs*, in code outside *Abs*, is only allowed if afterwards p is still owned at *Abs*. The restriction is in the form of an added precondition for **setown**, see Table 2. The precondition would hold in the variation under discussion. But it would be falsified if the code in Queue transferred ownership of the arrays to a client, or to no owner at all.

3 An illustrative language

Following [11,43], we formalize the *inv*/*own* discipline in terms of a language in which fields have public visibility, to illuminate the conditions necessary for sound reasoning about invariants and simulations. In practice, private and protected visibility and perhaps lightweight alias control would serve to automatically check most of the conditions. This section formalizes the language, adapting notations and typing rules from Featherweight Java [37] and imperative features and the special commands from our

$C \in \mathit{ClassName}$	$m \in \mathit{MethName}$ $f \in \mathit{FieldName}$ $x \in \mathit{VarName}$	
T	::= **bool** | **void** | C	data type
M	::= $T\ m(\bar{T}\ \bar{x})\ \{S\}$	method declaration
S	::= $x{:=}e \mid x.f{:=}y$	assign to local var. or param., update field
	| $x{:=}$**new** $C \mid x{:=}e.m(\bar{e}) \mid x{:=}y.f$	object creation, method call, field access
	| $T\ x{:=}e$ **in** $S \mid S;\ S \mid$ **if** e **then** S **else** S **fi**	local variable, sequence, conditional
	| **pack** e **as** $C \mid$ **unpack** e **from** C	set *inv* to C, set *inv* to $\mathit{super}C$
	| **setown** x **to** (e,C)	set $x.\mathit{own}$ to (e,C)
	| **assert** $\mathscr{P}$	assert (semantic predicate $\mathscr{P}$)
e	::= $x \mid$ **null** | **true** | **false**	variable, constant
	| $e = e \mid e$ **is** $C \mid (C)\ e$	ptr. equality, type test, cast

Table 3. Grammar. The distinguished names self and result are in *VarName*.

previous papers [6,51]. We choose a denotational semantics, because it enables an elegant formulation of simulations and because it was used in the latter papers.

A complete program is given as a *class table*, CT, that maps class name C to a declaration $CT(C)$ of the form **class** C **extends** $D\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}$. The categories T, M are given by the grammar in Table 3. 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.

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 without loss of generality we omit loops). For convenience, we use the some auxiliary functions on syntax. For a class C, $\mathit{fields}C$ is defined as the inherited and declared fields of C; $\mathit{dfields}C$ is the fields declared in C; $\mathit{super}C$ is the direct superclass of C. We assume that a field name f uniquely determines the class, $\mathit{declClass}\,f$, that declares it; so $\mathit{declClass}\,f = C$ iff there is some T such that $(f : T)$ is in $\mathit{dfields}C$.

We use multi-letter identifiers, so one might read "$\mathit{dfields}C$" as a single identifier rather than, as intended, the application of a function. We use parentheses when there seems to be a risk of confusion, while avoiding them in cases where context or typography should suffice.

For a method declaration, $T\ m(\bar{T}_1\ \bar{x})\ \{S\}$ in class C, the method type $\mathit{mtype}(m,C)$ is $\bar{T}_1 \to T$ and list of parameter names, $\mathit{pars}(m,C)$, is $\bar{x}$. For m inherited in C, $\mathit{mtype}(m,C) = \mathit{mtype}(m,D)$ and $\mathit{pars}(m,C) = \mathit{pars}(m,D)$ where D is the direct superclass of C.

For use in the semantics, $\mathit{xfields}C$ extends $\mathit{fields}C$ by assigning "types" to the auxiliary fields: *com* : **bool**, *own* : owntyp, and *inv* : (invtyp$C$). Neither invtyp$C$ nor owntyp are types in the programming language but the slight notational abuse is convenient. These fields are present in every object, as if they were declared in class Object.

A *typing context* Γ is a finite function from variable names to types, such that self $\in \mathit{dom}\ \Gamma$. Selected typing rules for expressions and commands are given in Table 4. A

$$\frac{\Gamma \vdash e : T \qquad T \leq \Gamma\, x}{\Gamma \vdash x := e} \qquad\qquad \frac{\Gamma \vdash x : D_1 \qquad \Gamma \vdash e_2 : D_2 \qquad D_2 \leq C}{\Gamma \vdash \textbf{setown}\ x\ \textbf{to}\ (e_2, C)}$$

$$\frac{\Gamma \vdash e : D \quad D \leq C}{\Gamma \vdash \textbf{pack}\ e\ \textbf{as}\ C} \qquad\qquad \frac{\Gamma \vdash e : D \quad D \leq C}{\Gamma \vdash \textbf{unpack}\ e\ \textbf{from}\ C}$$

$$\frac{B \leq \Gamma\, x \qquad x \neq \mathsf{self} \qquad B \neq \mathsf{Object}}{\Gamma \vdash x := \textbf{new}\ B} \qquad\qquad \frac{(f : T) \in \mathit{fields}(\Gamma\, x) \qquad \Gamma\, y \leq T}{\Gamma \vdash x.f := y}$$

$$\frac{(f : T) \in \mathit{fields}(\Gamma\, y) \qquad T \leq \Gamma\, x}{\Gamma \vdash x := y.f}$$

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

Table 4. Typing rules for selected commands.

judgement of the form $\Gamma \vdash e : T$ says that expression e has type T in the context of a method of class $\Gamma\,\mathsf{self}$, with parameters and local variables declared by Γ. A judgement $\Gamma \vdash S$ says that S is a command in the same context. A class table CT is well formed if, for each class C, each method declaration M in $CT(C)$ is well formed in C; this is written $C \vdash M$ and defined by the following rule:

$$\frac{\begin{array}{c}\bar{x} : \bar{T}, \mathsf{self} : C, \mathsf{result} : T \vdash S \\ \text{if } \mathit{mtype}(m, \mathit{super}C) \text{ is defined then } \mathit{mtype}(m, \mathit{super}C) = \bar{T} \to T \text{ and } \mathit{pars}(m, \mathit{super}C) = \bar{x}\end{array}}{C \vdash T\ m(\bar{T}\ \bar{x})\{S\}}$$

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 abuse of notation: the syntax of **assert** uses a semantic predicate. We say $\Gamma \vdash \textbf{assert}\ \mathscr{P}$ is well formed provided that $\mathscr{P}$ is a set of program states for context Γ. We return to predicates later.

In the rest of the chapter we assume types and contexts are well formed, and that typings are derivable, without explicit mention.

Semantics. We assume that a countable set *Loc* is given, along with a distinguished value *nil* not in *Loc*. We assume given a function *type* from *Loc* to non-primitive types distinct from Object, such that for each C there are infinitely many locations o with $\mathit{type}\ o = C$. This is used in a way that is equivalent to tagging object states with their type, which is immutable. It serves to slightly streamline some definitions. The syntax is desugared, in the style of separation logic, so that access and update of mutable fields occurs only in commands ($x := y.f$ and $x.f := y$). So field read $x := y.f$ is considered a primitive command rather than an instance of ordinary assignment and $y.f$ is not a stand

$$
\begin{array}{lll}
\theta ::= T \mid \Gamma \mid \theta_\perp & & \\
\quad \mid \ \mathsf{ownotyp} \mid \mathsf{invtyp}\, C \mid \mathsf{state}\, C & \textit{own} \text{ and } \textit{inv} \text{ val., object state} \\
\quad \mid \ \mathsf{pre\text{-}heap} \mid \mathsf{heap} \mid \mathsf{heap} \otimes \Gamma & \text{heap fragment, closed heap, state} \\
\quad \mid \ (\Gamma \vdash T) \mid \Gamma \rightsquigarrow \Gamma' \mid \mathsf{menv} & \text{expression meaning, state transformer, method environment}
\end{array}
$$

$$
\begin{array}{ll}
[\![C]\!] & = \{nil\} \cup \{o \in Loc \mid type\, o \leq C\} \\
[\![\mathbf{bool}]\!] & = \{true, false\} \\
[\![\mathbf{void}]\!] & = \{it\} \\
[\![\mathsf{invtyp}\, C]\!] & = \{B \mid C \leq B\} \\
[\![\mathsf{ownotyp}]\!] & = \{(o, C) \mid o = nil \vee type\, o \leq C\} \\
[\![\theta_\perp]\!] & = [\![\theta]\!] \cup \{\perp\} \\
[\![\Gamma]\!] & = \{s \mid dom\, s = dom\, \Gamma \wedge s\, \mathsf{self} \neq nil \wedge \forall x \in dom\, s \mid s\, x \in [\![\Gamma\, x]\!]\} \\
[\![\mathsf{state}\, C]\!] & = \{s \mid dom\, s = dom(xfields\, C) \wedge \forall (f : T) \in xfields\, C \mid s\, f \in [\![T]\!]\} \\
[\![\mathsf{pre\text{-}heap}]\!] & = \{h \mid dom\, h \subseteq_{fin} Loc \wedge \forall o \in dom\, h \mid h\, o \in [\![\mathsf{state}\, (type\, o)]\!]\} \\
[\![\mathsf{heap}]\!] & = \{h \mid h \in [\![\mathsf{pre\text{-}heap}]\!] \wedge \forall s \in rng\, h \mid rng\, s \cap Loc \subseteq dom\, h\} \\
[\![\mathsf{heap} \otimes \Gamma]\!] & = \{(h, s) \mid h \in [\![\mathsf{heap}]\!] \wedge s \in [\![\Gamma]\!] \wedge rng\, s \cap Loc \subseteq dom\, h\} \\
[\![\Gamma \vdash T]\!] & = \{v \mid v \in ([\![\Gamma]\!] \to [\![T]\!]_\perp) \wedge \forall s \mid v\, s \in Loc \Rightarrow v\, s \in rng\, s\} \\
[\![\Gamma \rightsquigarrow \Gamma']\!] & = [\![\mathsf{heap} \otimes \Gamma]\!] \to [\![(\mathsf{heap} \otimes \Gamma')_\perp]\!] \\
[\![\mathsf{menv}]\!] & = \{\mu \mid \text{for all } C, m,\ \mu C m \text{ is defined iff } mtype(m, C) \text{ is defined,} \\
 & \qquad \text{and if so then } \mu C m \in [\![\mathsf{self} : C, \bar{x} : \bar{T} \rightsquigarrow \mathsf{result} : T_1]\!] \\
 & \qquad \text{where } pars(m, C) = \bar{x} \text{ and } mtype(m, C) = \bar{T} \to T_1\ \}
\end{array}
$$

Table 5. Semantic categories θ and domains $[\![\theta]\!]$. (Readers familiar with notation for dependent function spaces might prefer to write $[\![\mathsf{pre\text{-}heap}]\!] = (o : Loc \nrightarrow [\![\mathsf{state}\, (type\ o)]\!])$ and similarly for $[\![\mathsf{state}\, C]\!]$ and $[\![\Gamma]\!]$.)

alone expression; this choice is not essential but streamlines the formal development. In our semantics, type test and cast expressions do not depend on the heap.

Some semantic domains correspond directly to the syntax. For example, each data type T denotes a set $[\![T]\!]$ of values. The meaning of context Γ is a set $[\![\Gamma]\!]$ of stores; a *store* $s \in [\![\Gamma]\!]$ is a type-respecting assignment of locations and primitive values to the local variables and parameters given by a typing context Γ. The semantics, and later the coupling relation, is structured in terms of category names θ given in Table 5 which also defines the semantic domains. Subtyping is embodied in a simple way: if $T \leq U$ then $[\![T]\!] \subseteq [\![U]\!]$.

A *program state* for context Γ is a pair (h, s) where s is in $[\![\Gamma]\!]$ and h is a *heap*, i.e., a finite partial function from locations to object states. An *object state* is a type-respecting mapping of field names to values. A command typable in Γ denotes a function mapping each program 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. An *object state* is a mapping from (extended) 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

$$
\begin{array}{ll}
[\![\Gamma \vdash x{:=}y.f]\!]\mu(h,s) & = \mathsf{let}\ o = s\,y\ \mathsf{in}\ \mathsf{if}\ o = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else}\ (h, [s \mid x \mapsto h\,o.f]) \\
[\![\Gamma \vdash x{:=}e]\!]\mu(h,s) & = \mathsf{let}\ v = [\![\Gamma \vdash e : T]\!](s)\ \mathsf{in}\ (h, [s \mid x \mapsto v]) \\
[\![\Gamma \vdash x.f{:=}y]\!]\mu(h,s) & = \mathsf{let}\ o = s\,x\ \mathsf{in}\ \mathsf{if}\ o = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else}\ ([h \mid o.f \mapsto s\,y], s) \\
[\![\Gamma \vdash x{:=}\textbf{new}\ C]\!]\mu(h,s) & = \mathsf{let}\ o = \mathit{fresh}(C,h)\ \mathsf{in} \\
 & \quad \mathsf{let}\ h_0 = [h \mid o \mapsto [\mathit{xfields}C \mapsto \mathit{defaults}C]]\ \mathsf{in}\ (h_0, [s \mid x \mapsto o]) \\
[\![\Gamma \vdash x{:=}e.m(\bar{e})]\!]\mu(h,s) & = \mathsf{let}\ o = [\![\Gamma \vdash e : D]\!](s)\ \mathsf{in}\ \mathsf{if}\ o = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else} \\
 & \quad \mathsf{let}\ \bar{v} = [\![\Gamma \vdash \bar{e} : \bar{U}]\!](s)\ \mathsf{in}\ \mathsf{let}\ \bar{x} = \mathit{pars}(m,D)\ \mathsf{in} \\
 & \quad \mathsf{let}\ s_1 = [\bar{x} \mapsto \bar{v}, \mathsf{self} \mapsto o]\ \mathsf{in} \\
 & \quad \mathsf{let}\ (h_1, s_2) = \mu(\mathit{type}\,o)m(h,s_1)\ \mathsf{in}\ (h_1, [s \mid x \mapsto s_2\,\mathsf{result}]) \\
[\![\Gamma \vdash \textbf{assert}\ \mathscr{P}]\!]\mu(h,s) & = \mathsf{if}\ (h,s) \in \mathscr{P}\ \mathsf{then}\ (h,s)\ \mathsf{else}\ \bot \\
[\![\Gamma \vdash \textbf{pack}\ e\ \textbf{as}\ C]\!]\mu(h,s) = & \\
\quad \mathsf{let}\ q = [\![\Gamma \vdash e : D]\!](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) & \\
[\![\Gamma \vdash \textbf{unpack}\ e\ \textbf{from}\ C]\!]\mu(h,s) = & \\
\quad \mathsf{let}\ q = [\![\Gamma \vdash e : D]\!](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} & \\
\quad ([h_1 \mid q.\mathit{inv} \mapsto \mathit{super}C], s) & \\
[\![\Gamma \vdash \textbf{setown}\ x\ \textbf{to}\ (e_2,C)]\!]\mu(h,s) = & \\
\quad \mathsf{let}\ q = s\,x\ \mathsf{in}\ \mathsf{if}\ q = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else} & \\
\quad \mathsf{let}\ p = [\![\Gamma \vdash e_2 : D_2]\!](s)\ \mathsf{in}\ ([h \mid q.\mathit{own} \mapsto (p,C)],\ s) &
\end{array}
$$

Table 6. Semantics of selected commands. To streamline the treatment of $\bot$, the meta-language expression "let $\alpha = \beta$ in ..." denotes $\bot$ if β is $\bot$. We use notation $[h \mid o \mapsto st]$ for h extended or overridden at o with value st. For brevity the nested function extension for field update is written $[h \mid o.f \mapsto v]$.

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 ",".

The meaning of a derivable command typing $\Gamma \vdash S$ will be defined to be a function sending each method environment μ to an element of $[\![\Gamma \rightsquigarrow \Gamma]\!]$. That is, $[\![\Gamma \vdash S]\!]\mu$ is a state transformer $[\![\mathsf{heap} \otimes \Gamma]\!] \to [\![(\mathsf{heap} \otimes \Gamma)_\bot]\!]$. For a method m such that $\mathit{pars}(m,C) = \bar{x}$ and $\mathit{mtype}(m,C) = \bar{T} \to U$, the meaning $\mu\,C\,m$ will be a state transformer of type $(\mathsf{self} : C, \bar{x} : \bar{T}) \rightsquigarrow (\mathsf{result} : T_1)$.

Meanings for expressions and commands are defined, in Table 6, by recursion on typing derivation. In some of the defining equations, the right side refers to identifiers in the typing rules. For example, T in the semantics of $x{:=}e$ is the type of e as per the first rule in Table 4. The semantic definition for **pack** e **as** C refers to D which is the type of e in the typing rule for **pack**.

The semantics is defined for an arbitrary location-valued function *fresh* such that $\mathit{type}(\mathit{fresh}(C,h)) = C$ and $\mathit{fresh}(C,h) \notin \mathit{dom}\,h$.

Consider a method call $\Gamma \vdash x{:=}e.m(\bar{e})$, where $\Gamma \vdash e : D$. Consider execution of the call in initial state (h,s) and let $o = [\![\Gamma \vdash e : D]\!](h,s)$; so by type soundness $\mathit{type}\ o \leq D$.

The meaning of the method body, used for the semantics of the call, is found in the method environment μ as $\mu\,(type\ o)\,m$. It is applied to state (h,s_1) with argument store s_1 that maps self to o and parameters $\bar{x}$ to their values $\bar{v}$. It returns a state (h_1,s_2) where s_2 result provides the value assigned to x.

The meaning of a well typed method declaration M in class C, of the form $M = T\ m(\bar{T}\ \bar{x})\{S\}$ is the total function in $[\![\mathsf{menv}]\!] \to [\![\mathsf{self}:C,\bar{x}:\bar{T} \rightsquigarrow \mathsf{result}:T]\!]$ defined as follows: Given a method environment μ, a heap h and a store $s \in [\![\bar{x}:\bar{T},\mathsf{self}:C]\!]$,

$$\begin{array}{l}[\![M]\!]\mu(h,s) = \mathsf{let}\ s_1 = [s \mid \mathsf{result} \mapsto \mathit{default}\,T]\ \mathsf{in} \\ \qquad\qquad \mathsf{let}\ (h_0,s_0) = [\![\Gamma \vdash S]\!]\mu(h,s_1)\ \mathsf{in}\ (h_0,[\mathsf{result}:s_0\,\mathsf{result}])\end{array} \tag{2}$$

A method environment μ maps each C,m to a meaning obtained in this way or by inheritance. In more detail, we define for each i an environment μ_i; this is called the *approximation chain*. The basis is $\mu_0\,C\,m$ is the everywhere-$\bot$ function, for all C,m. We define $\mu_{i+1}\,C\,m$, for m declared as M in C, to be $[\![M]\!]\mu_i$. In case m is inherited in C from B, we define $\mu_{i+1}\,C\,m$ to be $\mu_{i+1}\,B\,m$.

Note that the ith element in the chain approximates $[\![CT]\!]$ in a way such that, in operational terms, it gives the correct semantics for executions with method call stack bounded in length by i. For well formed class table CT, the semantics $[\![CT]\!]$ is defined as the least upper bound of the approximation chain. For full details see [6] or the variation that was machine checked in PVS [49].

Predicates. A *predicate* for state type Γ is just a subset $\mathscr{P} \subseteq [\![\mathsf{heap} \otimes \Gamma]\!]$. For emphasis we can write $(h,s) \models \mathscr{P}$ for $(h,s) \in \mathscr{P}$. Note that $\bot \notin \mathscr{P}$. We give no formal syntax to denote predicates but rather use informal metalanguage for which the interpretation should be clear. For example, "self.$f \neq$ **null**" denotes the set of (h,s) with $h(s\,\mathsf{self}).f \neq nil$. and "$\forall o \mid \mathscr{P}(o)$" denotes the set of (h,s) such that $(h,s) \models \mathscr{P}(o)$ for all $o \in dom\ h$. Note that quantification over objects (e.g., in Table 1 and Def. 4) is interpreted to mean quantification over allocated locations; the range of quantification can include unreachable objects but this causes no problems.

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 1 (depends, new-closed) A predicate $\mathscr{P}$ *depends on* $o.f$ *in* (h,s) iff $(h,s) \in \mathscr{P}$, $o \in dom\ h$, and $([h \mid o.f \mapsto v],s) \notin \mathscr{P}$ for some v with $[h \mid o.f \mapsto v] \in [\![\mathsf{heap}]\!]$. We say $\mathscr{P}$ *depends on* $o.f$ iff there is some (h,s) such that $\mathscr{P}$ depends on $o.f$ in (h,s). We say $\mathscr{P}$ is *new-closed* iff $(h,s) \in \mathscr{P}$ implies $([h \mid o \mapsto defaults],s) \in \mathscr{P}$ for all $o \notin dom\,h$.

The condition $[h \mid o.f \mapsto v] \in [\![\mathsf{heap}]\!]$ merely ensures that v is not a dangling pointer or type-incorrect value.

4 The *inv*/*own* discipline

The discipline reviewed in Sect. 2.1 is designed to make Equation (1) a program invariant for every object. This is achieved by using additional program invariants that govern

ownership. We formalize this as a global predicate, *disciplined*, defined in three steps. Then we review prior results on how the discipline is enforced by proper annotation. Finally, we use ownership to partition the heap, in preparation for Sect. 5.

4.1 Ownership and invariants

The default values for the extended fields are $inv = \mathsf{Object}$, $own = (nil, \mathsf{Object})$, and $com = false$. So initially a new object is neither packed nor owned.

Definition 2 (transitive C- and $C\uparrow$-ownership) For any heap h, the relation $o \succ^h_C p$ on $dom\ h$, read "*o owns p at C in h*", holds iff either $(o,C) = h\,p.own$ or there are q and D such that $(o,C) = h\,q.own$ and $q \succ^h_D p$.

The relation $o \succ^h_{C\uparrow} p$ holds iff there is some D with $C \leq D$ and $o \succ^h_D p$. This may be read "*o owns p at or above C in h*".

The relations are transitive in this sense: $o \succ^h_C p$ and $p \succ^h_D q$ implies $o \succ^h_C q$.

The discipline ensures that if o owns p at C and p is not packed to its type then o is unpacked at least above C. This is formalized in Corollary 6.

Definition 3 (admissible invariant) A predicate $\mathscr{P} \subseteq [\![\mathsf{heap} \otimes (\mathsf{self} : C)]\!]$ is *admissible as an invariant for C* provided that it is new-closed and for every h, s, o, f such that $\mathscr{P}$ 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 $dom(xfields\,C)$ or $s\,\mathsf{self} \succ^h_{C\uparrow} o$.

For dependence on fields of self, the typing condition, $f \in dom(xfields\,C)$, 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) —while allowing dependence on fields declared or inherited in C. An invariant can depend on any fields of objects owned at C or above.

We refrain from formalizing syntax for declaring invariants. In the subsequent definitions, we assume that an admissible invariant $\mathscr{I}^C$ is given for every class C. We assume $\mathscr{I}^{Object} = \mathbf{true}$.

Definition 4 (disciplined, $\mathscr{J}$) A heap h is *disciplined* if $h \models \mathscr{J}$ where $\mathscr{J}$ is defined to be the conjunction of the following:

(D1) $\forall o, C \mid o.inv \leq C \Rightarrow \mathscr{I}^C(o)$
(D2) $\forall o, C, p \mid o.inv \leq C \wedge p.own = (o,C) \Rightarrow p.com$
(D3) $\forall o \mid o.com \Rightarrow o.inv = type\ o$

A state (h,s) is *disciplined* if h is.

Method environment μ *is disciplined* provided that every method *preserves* $\mathscr{J}$ in the following sense: For any C, m, h, s, if $h \models \mathscr{J}$ and $\mu\,C\,m\,(h,s) = (h_0, s_0)$ then $h_0 \models \mathscr{J}$.

In the sequel we refrain from reminding the reader that a hypothesis like $\mu\,C\,m\,(h,s) = (h_0,s_0)$ implies that $\mu\,C\,m\,(h,s) \neq \bot$.

Lemma 5 (transitive ownership) Suppose h is disciplined and $o \succ_C^h p$. Then

(a) $type\, o \leq C$, and
(b) $h\, o.inv \leq C$ implies $h\, p.com = true$.

Corollary 6 If h is disciplined, $o \succ_C^h p$, and $h\, p.inv > type\, p$, then $h\, o.inv > C$.

In a small-step semantics one would prove that every reachable state in a properly annotated program is disciplined. Instead, we will show that every command maps disciplined initial states to disciplined final states —just like methods in a disciplined environment. So our notion of *program invariant* is a predicate $\mathscr{P}$ that is *preserved* by commands in the sense that

$$(h,s) \models \mathscr{P} \text{ and } [\![\Gamma \vdash S]\!]\mu(h,s) = (h_0,s_0) \text{ implies } (h_0,s_0) \models \mathscr{P}$$

and similarly for predicates on the heap alone.

4.2 The discipline

To impose the stipulated preconditions of Table 1 we consider programs with the requisite syntactic structure (similar to formal proof outlines [3]).

Definition 7 (properly annotated) The *annotated commands* are the subset of the category of commands where each **pack**, **unpack**, **setown**, and field update is immediately preceded by an **assert**. A *properly annotated command* is an annotated command such that each of these assertions implies the precondition stipulated in Table 1. A *properly annotated class table* is one such that each method body is properly annotated.

For any class table and family of invariants there exists a proper annotation: just add **assert** commands with the stipulated preconditions. For practical interest, of course, one wants assertions that can collectively be proved correct.

For a properly annotated program, $\mathscr{J}$ is a program invariant. This property is shown in terms of small-step semantics in [11,43]. For our purposes, the initial state of a complete program has an empty heap, which satisfies $\mathscr{J}$ because the quantified objects in (D1–D3) range over allocated objects. So we focus on preservation in the following formulation.

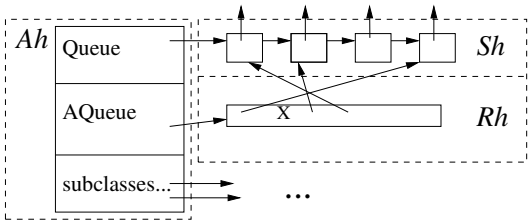
Proposition 8 If method environment μ is disciplined then any properly annotated command S preserves $\mathscr{J}$ in the sense that for all (h,s), if $h \models \mathscr{J}$ and $(h_0,s_0) = [\![\Gamma \vdash S]\!]\mu(h,s)$ then $h_0 \models \mathscr{J}$. If CT is a properly annotated class table then the method environment $[\![CT]\!]$ is disciplined, as is every method environment in the approximations of $[\![CT]\!]$.

The first statement is proved by induction on the structure of S. For the second statement, first we prove we go by induction on approximations of $[\![CT]\!]$, using the first statement. Then we show that being disciplined is preserved at the limit, so $[\![CT]\!]$ is disciplined. For details see [51]. The corollary is that S preserves $\mathscr{J}$, for any S that occurs as constituent of a method body in CT (interpreting S in $[\![CT]\!]$ or in any of the approximants).

4.3 Partitioning the heap

Next we show how ownership is used to partition the objects in the heap in order to formalize the encapsulation boundary depicted in Sect. 2.2.

Given an object $o \in dom\, h$ and class name A with $type\, o \leq A$ we can partition h into pre-heaps Ah (the A-object), Rh (the representation of o for 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 $o \succ^h_A p$, Sh is h restricted to the set of p with $o \succ^h_C p$ for some $C > A$, and Fh is the rest of h. Note that if $o \succ^h_B p$ for some proper subclass $B < A$ then $p \in dom\, Fh$. A pre-heap of the form $Ah * Rh * Sh$ is called an *island*. In these terms, dependency of admissible invariants is described in the following Proposition. As an illustration, here is the island for the left side of the situation depicted in Fig. 5 in Sect. 2.2:



Proposition 9 (island) Suppose $\mathscr{I}^A$ is an admissible invariant for A and $o \in dom\, h$ with $type\, o \leq A$. If $h = Fh * Ah * Rh * Sh$ is the partition defined above then $Fh_0 * Ah * Rh * Sh \models \mathscr{I}^C(o)$ iff $h \models \mathscr{I}^C(o)$, for all Fh_0 such that $Fh_0 * Ah * Rh * Sh$ is a heap.

In order to work with heap partitions it is convenient to have notation to extract the one object in a singleton heap. We define *pick* by $pick\, h = o$ where $dom\, h = \{o\}$; it is undefined if $dom\, h$ is not a singleton.

Prop. 9 considers a single object together with its owned representation; now we consider all objects of a given class.

Definition 10 (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:

- $dom\, Ah_i$ contains exactly one object o and $type\, o \leq A$ (for all i, $1 \leq i \leq k$);
- every $o \in dom\, h$ with $type\, o \leq A$ occurs in $dom\, Ah_i$ for some i;
- $dom\, Rh_i = \{p \mid o \succ^h_A p\}$ where $o = pick\, Ah_i$ (for all i);
- $dom\, Sh_i = \{p \mid o \succ^h_{(super A)\uparrow} p\}$ where $o = pick\, Ah_i$ (for all i);
- $dom\, Fh = dom\, h \setminus (\cup i \mid dom(Ah_i * Rh_i * Sh_i))$

The conditions determine a unique decomposition. However, the numbering is not unique. Note that each Rh_i and Sh_i is transitively closed under ownership: if $p \in dom\, Rh_i$ (resp. $dom\, Sh_i$) and $p \succ^h_C q$ for some C then $q \in dom\, Rh_i$ (resp. $q \in dom\, Sh_i$).

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 $o \succ^h_{(type\, o)\uparrow} p$ then $type\, p \not\leq A$. In this case decomposition partitions the

heap into separate islands of the form $Ah * Rh * Sh$. We use the term "partition" even though some blocks can be empty. Moreover, although it is the domain of the heap that is partitioned, we also use the term "partition" to refer to the corresponding factorization of the heap into a union of pre-heaps with disjoint domains.

Lemma 11 (A-partition) Suppose no A-object owns an A-object in h. Then the A-decomposition is a partition of h, that is, $h = Fh * Ah_1 * Rh_1 * Sh_1 * \ldots * Ah_k * Rh_k * Sh_k$.

We now define the function *encap* that removes from the heap the objects that are owned at *Abs*.

Definition 12 (encap) Suppose no *Abs*-object owns an *Abs*-object in h. Define the pre-heap $encap\,Abs\,h$ to be $Fh * Ah_1 * Sh_1 * \ldots * Ah_k * Sh_k$ where the *Abs*-partition of h is as in Lemma 11.

Def. 16 in Sect. 5 imposes a syntactic restriction to ensure that no *Abs*-object owns an *Abs*-object, where *Abs* is the class for which two representations are compared. The restriction is expressed by means of a static approximation of ownership.

Definition 13 (may own, $\succ^{\exists}$) Given well formed CT, define $\succ^{\exists}$ to be the least transitively closed relation such that

(M1) $D_2 \succ^{\exists} D_1$ for every occurrence of **setown** x **to** (e_2, D) in a method of CT, with static types $x : D_1$ and $e_2 : D_2$
(M2) if $C \succ^{\exists} D$, $C' \leq C$ and $D' \leq D$ then $C' \succ^{\exists} D'$

Lemma 14 (a) It is a program invariant that $o \succ^{h}_{C} p$ implies $type\,o \succ^{\exists} type\,p$.
(b) If $A \not\succ^{\exists} A$ then it is a program invariant that no A-object owns an A-object.

Proof. Part (b) is a direct consequence of (a). Part (a) is proved by structural induction on commands and then induction on the approximation chain, as in the proof of Prop. 8. The only command forms that affect ownership relations are **setown** and **new**. Because **new** constructs objects with no owner, the only interesting case is **setown**.

Suppose that some method body in CT contains **setown** x **to** (e_2, C), with static types $x : D_1$ and $e_2 : D_2$. Suppose $(h_0, s_0) = [\![\textbf{setown}\ x\ \textbf{to}\ (e_2, C)]\!](h, s)$, where the implication holds in (h, s) (for all o, p, C). We show that $o \succ^{h_0}_{C} p$ implies $type\,o \succ^{\exists} type\,p$, for all o, p, C, by induction on the relation $o \succ^{h_0}_{C} p$ in accord with Def. 2. (In essence, induction on the length of ownership chains.)

The base case is $h_0\,p.own = (o, C)$. If also $h\,p.own = (o, C)$ then we are done since the implication holds in (h, s). On the other hand, if $h\,p.own \neq (o, C)$ then we must have $o = [\![e_2]\!](h, s)$ and $p = [\![e_1]\!](h, s)$ since the command only changes the ownership of e_1. Hence by typing we have $type\,o \leq D_2$ and $type\,p \leq D_1$. By (M1) in Def. 13 we have $D_2 \succ^{\exists} D_1$ whence $type\,o \succ^{\exists} type\,p$ by (M2).

The inductive case is $(o, C) = h_0\,q.own$ and $q \succ^{h_0}_{D} p$ for some q, D. We have $type\,o \succ^{\exists} type\,q$ and $type\,q \succ^{\exists} type\,p$ by induction. Hence $type\,o \succ^{\exists} type\,p$ by the transitivity clause in Def. 13.

5 The abstraction theorem

The Abstraction Theorem generalizes Proposition 8. The Proposition considers a predicate on states and says the predicate is preserved by execution of command. The Theorem considers a relation between states and says it is preserved by a pair of executions. The Theorem builds on the use of *inv* and on ownership structure, and thus requires proper annotation. It does not directly depend on the chosen invariants, so in theory one may take $\mathscr{I}^C = \mathbf{true}$ for all C.

5.1 Comparing class tables

We compare two implementations of a designated class *Abs*, in the context of a fixed but arbitrary collection of other classes, such that both implementations give rise to a well formed class table. The two versions can have completely different declarations, so long as methods of the same signatures are present — declared or inherited — in both. It is mainly to simplify the additional precondition needed for reading fields that we consider programs desugared into a form like that used in separation logic.

Definition 15 (properly annotated for *Abs*) The *properly annotated commands for Abs* are those that are properly annotated according to Def. 7 and moreover

(A1) fields of *Abs* have private visibility (i.e., if $f \in dfields Abs$ then accesses and updates of f only occur in code of class *Abs*)
(A2) if $\Gamma\,\mathsf{self} \neq Abs$ then $\Gamma \vdash \mathbf{pack}\ e\ \mathbf{as}\ Abs$ is not allowed
(A3) if $\Gamma\,\mathsf{self} \neq Abs$ then field access $\Gamma \vdash x{:=}y.f$ is subject to stipulated precondition $y \neq \mathbf{null} \wedge (\forall o \mid o \succ_{Abs} y \Rightarrow o \succ_{Abs} \mathsf{self})$
(A4) if $\Gamma\,\mathsf{self} \neq Abs$ then $\Gamma \vdash \mathbf{setown}\ x\ \mathbf{to}\ (e_2, C)$ is subject to an additional precondition: $x \neq \mathbf{null} \wedge ((\exists o \mid o \succ_{Abs} x) \Rightarrow C = Abs \vee (\exists o \mid o \succ_{Abs} e_2))$

(A1) merely embodies our choice to focus on change of representation for a single class and its encapsulated representations, and in particular the most common form of encapsulation, namely, private visibility. For practical purposes, other visibilities are needed and would be treated in the same manner as fields of other classes.

(A2) is needed in reasoning about simulation for **pack** e **as** *Abs*. This command reasserts the local coupling and we want to confine the proof obligation of simulation to code in class *Abs* as explained in Sect. 2.2.

The effect of (A3) is that a method invocation on some q not of type *Abs*, but reading an object p owned by an *Abs* object o, is only allowed if q is itself owned by o. A client should not be reading owned objects and a rep should not read objects that do not belong to its own owner. Perhaps surprisingly, (A3) does not disallow that a method invocation on one instance of *Abs* reads objects owned by another instance. The effect of (A4) is that if x is initially owned at *Abs* then after a transfer (that occurs in code outside class *Abs*) it is still owned at *Abs*.

Conditions (A3) and (A4) use transitive ownership notation, but without an explicit superscript. This is informal notation for semantic predicates. For example, "$o \succ_{Abs} e$" means $\{(h,s) \mid o \succ^{h}_{Abs} e\}$. The annotations in Table 1 use no inductively defined predicates, which is a distinct advantage for program verifiers like Spec# based on SMT

provers. Direct use of transitive ownership in the program invariants (D1–D3) is no problem: these are part of the theory that justifies the discipline, not part of the verification conditions for programs. We have formulated (A3) and (A4) using transitive ownership for the sake of clarity.

Direct use of transitive ownership can be avoided by maintaining additional ghost state. For example, we can maintain an additional field, owns, of type *ClassName*→*Loc*, with invariant $\forall C, o \mid (o \succ_C \mathsf{self} \Leftrightarrow o \in \mathsf{self}.owns\,C)$. The invariant immediately gives an alternative way to formulate (A3) and (A4). To maintain the invariant we merely augment the semantics of **setown**. For any state (h,s), we define $[\![\textbf{setown}\ x\ \textbf{to}\ (y,C)]\!]\mu(h,s)$ to update owns as follows. Consider the case where initially $y \neq \textbf{null}$ and $x.own = (o,B)$ with $o \neq \textbf{null}$. So ownership of x is being transferred from o to y. We want to remove from $o.\mathsf{owns}B$ all p that is in $x.\mathsf{owns}D$ for any D; and add to $y.\mathsf{owns}C$ the union of $x.\mathsf{owns}D$ over all D. In the cases where the old or new owner is **null**, the corresponding removal/addition is not done.[7] This treatment of transitive ownership is straightforward and would be preferable for practical use, but for clarity of presentation we do not develop it in the sequel.

Definition 16 (comparable class tables) Well formed class tables CT and CT' are *comparable* with respect to class name *Abs* ($\neq \mathsf{Object}$) provided the following hold.

- $CT(C) = CT'(C)$ for all $C \neq Abs$.
- $CT(Abs)$ and $CT'(Abs)$ have the same direct superclass and declare the same methods with the same signatures.
- CT and CT' are properly annotated for *Abs*.[8]
- $Abs \not\succ^{\exists} Abs$ in both CT and CT'

The last condition ensures that the *Abs*-decomposition of any disciplined heap is a partition, by Lemmas 11 and 14. We write $\vdash, \vdash'$ for the typing relation determined by CT, CT' respectively; similarly we write $[\![-]\!], [\![-]\!]'$ for the respective semantics.

For properly annotated CT and CT', fields declared in *Abs* have "private scope" (see (A1)), so the two typing relations coincide except when $\Gamma\,\mathsf{self}$ is *Abs*.

In the rest of the chapter we assume CT, CT' are comparable.

5.2 Coupling relations

The definitions are organized as follows. A *local coupling* (Def. 20) is a suitable relation on islands. This induces a family of *coupling relations* $\mathscr{R}\,\beta\,\theta$ (Def. 21), one for each category name θ and typed bijection β (Def. 17). Each relation $\mathscr{R}\,\beta\,\theta$ 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.

[7] This semantics can be written as a bulk update, just like the updates to *com* in the semantics of pack/unpack, so the the axiomatic semantics used in a verifier does not need to use sets explicitly.

[8] Together with the requirement $CT(C) = CT'(C)$ for all $C \neq Abs$, this implies that the families of invariants $\mathscr{I}^C$ given for CT and CT' are the same.

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

Table 7. Value equivalence for the designated class *Abs*. The relation for heap is the same as for pre-heap. For object states, $\sim$ is independent from the declared fields of $CT(Abs)$ and $CT'(Abs)$.

Definition 17 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 p with $type\, p \leq Abs$, if $p \in dom\, h$ (resp. $p \in dom\, h'$) then $p \in dom\, \beta$ (resp. $p \in rng\, \beta$).

Lemma 18 (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*. Suppose no *Abs*- object owns an *Abs*-object in h or h' (so that Lemma 11 applies). 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 $j = k$.

A corollary is that, under the conditions of Lemma 18, we may w.l.o.g. assume that islands in the two heaps are numbered such that $\beta\, (pick Ah_i)\, (pick Ah'_i)$, for all i.

Definition 19 (equivalence for *Abs* modulo bijection) For any β we define a relation $\sim_\beta$ for data values, object states, heaps, and stores, in Table 7.

Equivalence hides the private fields of *Abs*. Later in the identity extension Lemma 31, it is used in conjunction with the *encap* function from Def. 12, to hide the objects owned at *Abs*.

The most important definition is of local coupling, which is analogous to an admissible object invariant but is formulated, somewhat differently, as a relation on pairs of pre-heaps. In Def. 3, we take an invariant $\mathscr{I}^C$ to be a predicate (set of states) and the program invariant $\mathscr{J}$ is based on the conjunction of these predicates for all objects and types —subject to *inv*, see Def. 4). By contrast, we define a local coupling $\mathscr{L}$ in terms of pre-heaps. And we are concerned with a single class, *Abs*, rather than all C. We impose the same dependency condition as in Def. 3, but in terms of pre-heaps of the form $h = Ah * Rh * Sh$. (Recall Proposition 9.)

Definition 20 (local coupling, $\mathscr{L}$) A *local coupling* is a function, $\mathscr{L}$, that assigns to each typed bijection β a binary relation $\mathscr{L}\,\beta$ on pre-heaps that satisfies the following. First, $\mathscr{L}\,\beta$ does not depend on *inv* or *com*. Second, $\beta \subseteq \beta_0$ implies $\mathscr{L}\,\beta \subseteq \mathscr{L}\,\beta_0$. Third, for any β, h, h', if $\mathscr{L}\,\beta\,h\,h'$ then there are locations o, o' with $\beta\,o\,o'$ and $type\,o \leq Abs$ such that the Abs partitions of h, h' are $h = Ah * Rh * Sh$ and $h' = Ah' * Rh' * Sh'$ with

- $pick\,Ah = o$ and $pick\,Ah' = o'$
- $o \succ^{h}_{Abs} p$ for all $p \in dom\,Rh$ and $o' \succ^{h'}_{Abs} p'$ for all $p' \in dom\,Rh'$
- $o \succ^{h}_{(super\,Abs)\uparrow} p$ for all $p \in dom\,Sh$ and $o' \succ^{h'}_{(super\,Abs)\uparrow} p'$ for all $p' \in dom\,Sh'$
- If $\mathscr{L}\,\beta$ depends on $o.f$ then f is in $xfields\,Abs$

The first three conditions ensure that $\mathscr{L}$ relates a single island, for an object of some subtype of Abs, to a single island for an object of the same type. Although $\mathscr{L}$ 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, nor *inv* or *com*). 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.

The restriction against dependence on *inv* or *com* is carried over from the *inv*/*own* discipline. We do not have an example to show it is necessary for representation independence. For friendship based invariants it is both useful and sound to allow dependence on *inv* (see [51]), so this point may merit further investigation.

In applications, $\mathscr{L}\,\beta\,h\,h'$ would be defined as something like this: h and h' partition as islands $Ah * Rh * Sh$ and $Ah' * Rh' * Sh'$ such that $Ah * Rh * Sh \models \mathscr{I}^{Abs}$ and $Ah' * Rh' * Sh' \models \mathscr{I}'^{Abs}$ and some condition links the data structures [36]. The bijection β would not be explicit but would be induced as a property of the formula language.

For an example, note that the property defined informally in Fig. 5 relates two instances of AQueue together with objects owned at AQueue and representation objects owned at Queue.

A local coupling $\mathscr{L}$ induces a relation on arbitrary heaps by requiring that corresponding islands are related by $\mathscr{L}$. This in turn gives rise to a relation on commands. Roughly, a pair of commands or methods are related if they send a related pair of initial states to a related pair of outcomes. Howover, we cannot expect this to hold in case of methods acting on receiver objects that are owned at Abs. (This exclusion is similar to a type-based exclusion of "non-Rep classes" in [6].) In order to make this precise, we define for Γ-states with self in Γ an abbreviation:

$$\mathsf{nonrep}(h, s) \Leftrightarrow \neg(\exists o \mid o \succ^{h}_{Abs} s\,\mathsf{self})$$

Definition 21 (coupling relation, $\mathscr{R}$) Given local coupling $\mathscr{L}$, we define for each θ and β a relation $\mathscr{R}\,\beta\,\theta \subseteq [\![\theta]\!] \times [\![\theta]\!]'$ by cases on θ.

Case θ is heap: Define $\mathscr{R}\,\beta\,\mathsf{heap}\,h\,h'$ iff

- h, h' are disciplined
- $\beta \subseteq dom\,h \times dom\,h'$
- β fully partitions h, h' for Abs

Moreover, suppose the *Abs*-partitions are

$$h = Fh * Ah_1 * Rh_1 * Sh_1 \ldots Ah_k * Rh_k * Sh_k \text{ and}$$
$$h' = Fh' * Ah'_1 * Rh'_1 * Sh'_1 \ldots Ah'_k * Rh'_k * Sh'_k$$

where, without loss of generality we assume $\beta\ (pick\, Ah_i)\ (pick\, Ah'_i)$ for all i (in accord with the remark following Lemma 18). Then we require:

(R1) β restricts to a total bijection between *dom Fh* and *dom Fh'* (recall Def. 17);
(R2) $Fh \sim_\beta Fh'$; and
(R3) for all i,
- **(i)** β restricts to a total bijection between *dom* Sh_i and *dom* Sh'_i
- **(ii)** $(Ah_i * Sh_i) \sim_\beta (Ah'_i * Sh'_i)$
- **(iii)** $h(pick\, Ah_i).inv \leq Abs \Rightarrow \mathscr{L}\ \beta\ (Ah_i * Rh_i * Sh_i)\ (Ah'_i * Rh'_i * Sh'_i)$

Case θ is any other category: $\mathscr{R}\ \beta\ \theta$ is defined as follows:

$$\begin{array}{ll}
\mathscr{R}\ \beta\ \theta\ \alpha\ \alpha' & \Leftrightarrow \alpha \sim_\beta \alpha' \text{ if } \theta \text{ is } T, \mathsf{invtyp}C, \mathsf{owntyp}, \Gamma, \text{ or } \mathsf{state}C \\
\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h,s)\ (h',s') & \Leftrightarrow \mathscr{R}\ \beta\ \mathsf{heap}\ h\ h' \wedge \mathscr{R}\ \beta\ \Gamma\ s\ s' \\
\mathscr{R}\ \beta\ (\theta_\perp)\ \alpha\ \alpha' & \Leftrightarrow (\alpha = \perp = \alpha') \vee (\alpha \neq \perp \neq \alpha' \wedge \mathscr{R}\ \beta\ \theta\ \alpha\ \alpha') \\
\mathscr{R}\ \beta\ (\Gamma \vdash T)\ v\ v' & \Leftrightarrow \forall s,s' \mid \mathscr{R}\ \beta\ \Gamma\ s\ s' \Rightarrow \mathscr{R}\ \beta\ T_\perp\ (v(s))\ (v'(s')) \\
\mathscr{R}\ \beta\ (\Gamma \rightsquigarrow \Gamma')\ t\ t' & \Leftrightarrow \forall h,s,h',s' \mid \\
 & \quad \mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h,s)\ (h',s') \wedge \mathsf{nonrep}(h,s) \wedge \mathsf{nonrep}(h',s') \\
 & \quad \Rightarrow \exists \beta_0 \supseteq \beta \mid \mathscr{R}\ \beta_0\ (\mathsf{heap} \otimes \Gamma)_\perp\ (t(h,s))\ (t'(h',s')) \\
\mathscr{R}\ \mathsf{menv}\ \mu\ \mu' & \Leftrightarrow \forall C,m,\beta \mid \mathscr{R}\ \beta\ (\mathsf{self} : C, \bar{x} : \bar{T} \rightsquigarrow \mathsf{result} : T_1)\ (\mu C m)\ (\mu' C m) \\
 & \quad \text{where } mtype(m,C) = \bar{T} \to T \text{ and } pars(m,C) = \bar{x}
\end{array}$$

Condition (R3)(iii) is the key connection with the *inv*/*own* discipline; compare Eqn. (1). All remaining items express that the relation is observable equivalence on everything on which a client can directly depend (see Lemma 31).

Given the other conditions in the definition of $\mathscr{R}\,\beta\,\mathsf{heap}$, we have that $(Ah_i * Sh_i) \sim_\beta (Ah'_j * Sh'_j)$ 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 (but not the declared fields of *Abs*); in particular, $type\, o = type\, o' \leq Abs$ and $Ah_i\, o.inv = Ah'_j\, o'.inv$.

By contrast with the definition of admissible invariant (Def. 3), there is no need to separately disallow that a coupling is falsifiable by allocation. Owing to the use of partial heaps, this property follows from the form of the definition, as becomes clear in the proof of Lemma 29.

The case that θ is $\Gamma \rightsquigarrow \Gamma'$ is for state transformers t, t' denoted by commands and method declarations. It says they preserve coupling, while possibly growing the bijection β, and excluding computations where self is owned by an instance of *Abs*.

The gist of the abstraction theorem (Theorem 30) is that if methods of *Abs* are related by $\mathscr{R}$ then all methods are related. In terms of the preceding definitions, we can express quite succinctly the conclusion that all methods are related: $\mathscr{R}$ menv $[\![CT]\!]\ [\![CT']\!]'$. We want the antecedent of the theorem to be that the meaning $[\![M]\!]$ is related to $[\![M']\!]'$, for any m with declaration M in $CT(Abs)$ and M' in $CT'(Abs)$. Moreover, $[\![M]\!]$ depends on a method environment. Thus the antecedent of the theorem is that $[\![M]\!]\mu$ is related

to $[\![M']\!]'\mu'$ for all related μ,μ'. (It suffices for μ,μ' to be in the approximation chains defining $[\![CT]\!]$ and $[\![CT']\!]'$.)

The rest of this subsection is devoted to technical results which may be skipped.

Lemma 22 If $\bar{U} \leq \bar{T}$ and $\mathscr{R}\ \beta\ \bar{U}\ \bar{v}\ \bar{v}'$ then $\mathscr{R}\ \beta\ \bar{T}\ \bar{v}\ \bar{v}'$.

Lemma 23 (closure under transitive owners) If $Ph \sim_\beta Ph'$ and $\beta\, p\, p'$, and $o \succ_C^{Ph} p$ then there is o' such that $\beta\, o\, o'$ and $o' \succ_C^{Ph'} p'$.

Proof. By induction on $o \succ_C^{Ph} p$. In the base case we have $(o,C) = Ph\,p.own$. Then by $Ph \sim_\beta Ph'$ and $\beta\, p\, p'$ we have some o' with $(o',C) = Ph'\,p'.own$ and $\beta\, o\, o'$, whence $o' \succ_C^{Ph'} p'$. In the inductive case, there is q,B with $(o,C) = Ph\,q.own$ and $q \succ_B^{Ph} p$. By induction hypothesis there is q' with $\beta\, q\, q'$ and $q' \succ_B^{Ph} p'$. By $Ph \sim_\beta Ph'$ and $(o,C) = Ph\,q.own$ there is o' such that $\beta\, o\, o'$ and $(o',C) = Ph'\,q'.own$, whence $o' \succ_C^{Ph'} p'$.

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 $o \succ_C^{Ph} p$ implies that o and p are in $dom\,Ph$ by definition of $\succ$.

Corollary 24 (splitting) Let Ph, Ph' be pre-heaps that are closed under transitive ownership, i.e., if $o \in dom\,Ph$ and $o \succ_{(type\,o)\uparrow}^{Ph} p$ then $p \in dom\,Ph$. Let β be a total bijection from Ph to Ph' and suppose $Ph \sim_\beta Ph'$. Let $o \in dom\,Ph$ and $\beta\, o\, o'$. Let Ph partition as $Ph^+ * Ph^-$ where Ph^+ contains o and the objects transitively owned by o, i.e.,

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

Let $Ph' = Ph'^+ * Ph'^-$ where Ph'^+ and Ph'^- are determined *mutatis mutandis* for Ph' with respect to o'. Then

(a) β is a total bijection from Ph^+ to Ph'^+ and a total bijection from Ph^- to Ph'^-
(b) $Ph^+ \sim_\beta Ph'^+$ and $Ph^- \sim_\beta Ph'^-$.

Proof. By Lemma 5(a) we can restrict to $(type\,o)\uparrow$. As a consequence of Lemma 23 we get that β is a total bijection from Ph^+ to Ph'^+. The rest follows from the definitions.

Lemma 25 If $o \succ_C^h p$ and $h\,p.own = (r,B)$ then either $(r,B) = (o,C)$ or $o \succ_C^h r$.

Proof. We first prove (by induction on $o \succ_C^h p$) that if $o \succ_C^h p$ then there is a series of one or more pairs (q_i,D_i) with (omitting h for clarity) $(o,C) = q_0.own$, $(q_0,D_0) = q_1.own$, $(q_1,D_1) = q_2.own$, $\ldots (q_n,D_n) = p.own$ and thus $q_i \succ_{D_i} q_{i+1}$ for each i; also $D_0 = C$ and $(r,B) = (q_n,D_n)$. Now an induction on i shows that $o \succ_C^h q_i$ for each i.

A consequence of this result is that if $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h,s)\ (h',s')$ then $\mathsf{nonrep}(h,s)$ iff $\mathsf{nonrep}(h',s')$. Hence the antecedent in the definition of $\mathscr{R}$ for $\Gamma \rightsquigarrow \Gamma'$ can be simplified.

Lemma 26 (partition and coupling) Suppose $\mathscr{R}\ \beta\ \mathsf{heap}\ h\ h'$ and $\beta\, o\, o'$. Let $h = Fh * \ldots Ah_k * Rh_k * Sh_k$ and $h' = Fh' * \ldots Ah'_k * Rh'_k * Sh'_k$ be the *Abs*-partitions, where w.l.o.g. we assume $pick Ah_i \sim_\beta pick Ah'_i$ for all i. Then $o \in dom\,Rh_i$ iff $o' \in dom\,Rh'_i$.

Proof. Let $o \in dom\,Rh_i$, to prove $o' \in dom\,Rh'_i$ (the reverse being symmetric). Using $\mathscr{R}\ \beta\ \mathsf{heap}\ h\ h'$ and Def. 21, 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, and by Lemma 23 it must be the j such that o' is transitively owned by $pick\,Ah'_j$.

A fine point about assertions. Def. 16 of comparable class tables says $CT(C) = CT'(C)$ for all $C \neq Abs$. But recall that we have used a "shallow embedding" formulation of assertions: we embed sets of states in code. Moreover, these are well-formed states — so, if $CT(Abs)$ declares different fields from those of $CT'(Abs)$, a state for CT is not a state for CT'. So it cannot be that $CT(T) = CT'(C)$ is literally true.

The intention is that CT and CT' have "the same" assertions, which in practice would be formulas. Consider any formula F that is well formed in both CT and CT', i.e., does not refer to private fields of Abs. Let $\mathscr{P}$ be the interpretation of F in CT and $\mathscr{P}'$ its interpretation in CT'. We claim the following,[9] for all h, s, h', s':

$$\text{If } \mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h,s)\ (h',s') \text{ then } h,s \models \mathscr{P} \text{ iff } h',s' \models \mathscr{P}'. \tag{3}$$

For ordinary assertions that may appear in code outside Abs, we expect that they respect scope, and so do not depend on private fields of Abs. For such assertions, and for the conditions in the stipulated preconditions —aside from pack— it is easy to prove the claim from usual semantics of the formulas (entirely independent from $fields\,Abs$). For pack, the precondition refers to $\mathscr{I}^C$ and our formulation of admissibility for invariants does not disallow dependence on private fields of Abs. However, the formula denoting such an invariant would not be well formed in both class tables.

In this chapter, we refrain from spelling out syntax and semantics of formulas. We simply assume that (3) holds for all corresponding assertions in classes other than Abs.

5.3 Simulation and the abstraction theorem

Definition 27 (simulation) A simulation is a coupling $\mathscr{R}$ such that the following hold.

- ($\mathscr{L}$ is initialized) For any $C \leq Abs$, and any o, o' with $\beta\, o\, o'$ and $type\, o = C$ we have $\mathscr{L}\ \beta\ h\ h'$ where $h = [o \mapsto [dom(xfields\, C) \mapsto defaults\, C]]$ and $h' = [o' \mapsto [dom(xfields'\, C) \mapsto defaults'\, C]]$.
- (methods of Abs preserve $\mathscr{R}$) For any disciplined μ, μ' such that $\mathscr{R}\ \mathsf{menv}\ \mu\ \mu'$ we have the following for every m declared in Abs. Let $\bar{U} \to U = mtype(m, Abs)$ and $\bar{x} = pars(m, Abs)$. For every β, we have

  $$\mathscr{R}\ \beta\ (\mathsf{self} : Abs, \bar{x} : \bar{U} \rightsquigarrow \mathsf{result} : U)\ ([\![M]\!]\mu)\ ([\![M']\!]'\mu')$$

  where M (resp. M') is the declaration of m in $CT(Abs)$ (resp. $CT'(Abs)$).

Perhaps surprisingly, there is no requirement for inherited methods. Rather, for practical application of the results —and indeed for the *inv*/*own* discipline itself— it is often

[9] Note that this does not mean there is a bijection between $\mathscr{P}$ and $\mathscr{P}'$. One or the other versions of Abs may have more states, owing to the type or number of fields of Abs.

necessary for inheritance to be "expanded" into stubs that perform suitable unpack/pack and super calls. The stubs are then subject to the proof obligation for declared methods. We return to this topic in Sect. 6.

The main theorem is that if $\mathscr{R}$ is a simulation for comparable class tables CT, CT' then $\mathscr{R}$ menv $[\![CT]\!]$ $[\![CT']\!]'$. The theorem is proved using the following Lemmas.

Because expressions do not include field access, an expression is typable in CT just if it is typable in CT', and preservation for expressions is straightforward.

Lemma 28 (preservation by expressions) For all expressions $\Gamma \vdash e : T$ and all β, we have $\mathscr{R}\ \beta\ (\Gamma \vdash T)\ ([\![\Gamma \vdash e : T]\!])\ ([\![\Gamma \vdash' e : T]\!]')$.

Proof. By induction on the structure of e, and by cases on e. In each case we assume $\mathscr{R}\ \beta\ \Gamma\ s\ s'$ (which amounts to $s \sim_\beta s'$) and must show

$$\mathscr{R}\ \beta\ T_\perp\ ([\![\Gamma \vdash e : T]\!](s))\ ([\![\Gamma \vdash' e : T]\!]'(s'))$$

For example, in case e is a variable x, from $\mathscr{R}\ \beta\ \Gamma\ s\ s'$ we get $\mathscr{R}\ \beta\ T\ (s\,x)\ (s'\,x)$ by definition. In case e is $(B)\ e$, then by induction we have $v \sim_\beta v'$ where $v = [\![\Gamma \vdash e]\!](s)$ and *mutatis mutandis* for v'. So either $v = \perp = v'$ and by semantics both cast expressions return $\perp$, or $\beta\, v\, v'$ and thus $type\, v = type\, v'$ (because β is a typed bijection) so the casts denote the same truth value. The other cases are similar.

For commands, the preservation lemma needs to rule out code in class *Abs*, since a field access or update in $CT(Abs)$ might not even be well formed in CT'.

Lemma 29 (preservation by commands) Let μ, μ' be disciplined method environments with $\mathscr{R}$ menv $\mu\ \mu'$. If $\Gamma \vdash S$ is a properly annotated command for *Abs*, with Γ self $\neq Abs$, then for all β we have $\mathscr{R}\ \beta\ (\Gamma \rightsquigarrow \Gamma)\ ([\![\Gamma \vdash S]\!]\mu)\ ([\![\Gamma \vdash' S]\!]'\mu')$.

Proof. In accord with the definition of $\mathscr{R}$ for category $\Gamma \rightsquigarrow \Gamma$, we consider arbitrary β and any (h,s), (h',s') such that $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h,s)\ (h',s')$ and $\mathsf{nonrep}(h,s)$ and $\mathsf{nonrep}(h',s')$. We show, by structural induction on S, that the outcomes $[\![\Gamma \vdash S]\!]\mu(h,s)$ and $[\![\Gamma \vdash' S]\!]'\mu'(h',s')$ are related at some β_0 with $\beta_0 \supseteq \beta$. For brevity we only consider a few cases on S such as field access and update. In each case we refer to the standard partitions $h = Fh * \ldots$ and $h' = Fh' * \ldots$ where w.l.o.g. $\beta(pickAh_i)(pickAh'_i)$ for each i (noting that by Def. 21 for heaps, we have that β fully partitions h, h' for *Abs*).

We never show that the result heaps are disciplined, because that follows in every case by Prop. 8.

Case of assignment $x := e$ Let $v = [\![\Gamma \vdash e : T]\!](h,s)$ and *mutatis mutandis* for v'. The outcomes are (h,s_0) and (h',s'_0) where $s_0 = [s \mid x \mapsto v]$ $s'_0 = [s' \mid x \mapsto v']$. We take $\beta_0 = \beta$. To show $\mathscr{R}\ \beta_0\ (\mathsf{heap} \otimes \Gamma)\ (h,s_0)\ (h',s'_0)$ it suffices to show $v \sim_\beta v'$ which we have by Lemma 28.

Case of assert For this we rely on assumption (3), from which we get the result directly by semantics of assert.

Case of field access $x := y.f$ (Recall that by typing, f is an ordinary field, not *inv*, *com*, or *own*.) In this case the output heap is the same as the input heap h. We choose $\beta_0 = \beta$ and note that $\mathscr{R}\ \beta\ \mathsf{heap}\ h\ h'$ holds by hypothesis. Let $p = s\ y$ and let the updated

store s_0 be $[s \mid x \mapsto h\,p.f]$ (and similarly for p', s'_0 for s' as per our convention). To prove $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s_0)\ (h', s'_0)$ it suffices to show $h\,p.f \sim_\beta h'\,p'.f$. By hypothesis $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s)\ (h', s')$ we have $p \sim_\beta p'$ i.e., either $\beta\,p\,p'$ or $p = nil = p'$. By stipulated precondition (A3) in Def. 15 neither is null, so $\beta\,p\,p'$.

Because $\Gamma\,\mathsf{self} \neq Abs$, by the stipulated precondition for field access we get

$$\forall o \mid o \succ^h_{Abs} p \Rightarrow o \succ^h_{Abs} s\,\mathsf{self} \quad \text{and} \quad \forall o \mid o \succ^{h'}_{Abs} p' \Rightarrow o \succ^{h'}_{Abs} s'\,\mathsf{self}$$

Because $\mathscr{R}\ \beta\ \mathsf{heap}\ h\ h'$, it suffices to consider the following cases.

- $p \in dom\,Fh$. Then $Fh \sim_\beta Fh'$ from $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s)\ (h', s')$, so by definition $h\,p.f \sim_\beta h'\,p.f$, using that f is not in $dfields\,Abs$ or $dfields'\,Abs$ because $type\,p \not\leq Abs$ by definition of decomposition. (Note that $Fh\,p = h\,p$ by decomposition, so we choose to write the shorter one, $h\,p$.)
- $p = pick\,Ah_i$ for some i. Hence $p' = pick\,Ah'_i$. By $\mathscr{R}\ \beta\ \mathsf{heap}\ h\ h'$ we have $Ah_i \sim_\beta Ah'_j$. By hypothesis of the Lemma, $\Gamma\,\mathsf{self} \neq Abs$, so by proper annotation Def. 15(A1), f is not in $dfields\,Abs$ or $dfields'\,Abs$. So $Ah_i \sim_\beta Ah'_j$ implies $h\,p.f \sim_\beta h'\,p.f$ as required.
- $p \in dom\,Rh_i$ for some i. We show by contradiction that this case cannot happen. Suppose $p \in dom\,Rh_i$. Then $pick\,Ah_i \succ^h_{Abs} p$ by definition of Abs-partition. By precondition we get $pick\,Ah_i \succ^h_{Abs} s\,\mathsf{self}$, which contradicts the antecedent $\mathsf{nonrep}(h, s)$.
- $p \in dom\,Sh_i$ for some i. Then by Def. 21 for heaps, item (R3), we have $\beta\,p\,p'$ and $Sh_i \sim_\beta Sh'_j$. By definition of decomposition, we have $type\,p \not\leq Abs$ so $declClass\,f \neq Abs$ and $declClass'\,f \neq Abs$, so $Sh_i \sim_\beta Sh'_j$ implies $h\,p.f \sim_\beta h'\,p.f$.

Case of field update $x.f := y$ Let $o = s\,x$ and $v = s\,y$. By typing we have $\Gamma\,x \leq declClass\,f$. The stipulated preconditions are $o \neq nil$ and $h\,o.inv > declClass\,f$. So the outcome from (h, s) is (h_0, s) where h_0 is $[h \mid o.f \mapsto v]$. *Mutatis mutandis* for o', v', h'_0. By hypothesis $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s)\ (h', s')$, we have $v \sim_\beta v'$ and $o \sim_\beta o'$; and thus $\beta\,o\,o'$. We take β_0 to be β. To show $\mathscr{R}\ \beta\ \mathsf{heap}\ h_0\ h'_0$ we have the following cases.

- $o \in dom\,Fh$. Then $\beta\,o\,o'$ and $Fh \sim_\beta Fh'$ from $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s)\ (h', s')$ (items (R1) and (R2) in Def. 21). It suffices to show $h_0\,o.f \sim_\beta h'_0\,o'.f$ which follows from $v \sim_\beta v'$.
- $o = pick\,Ah_i$ for some i. So we have $\beta\,o\,o'$ and $type\,o \leq Abs$. We have $Ah_i \sim_\beta Ah'_i$ by (R3)(ii) in Def. 21. By hypothesis of the Lemma, $\Gamma\,\mathsf{self} \neq Abs$, so by proper annotation Def. 15(A1), f is not in $dfields\,Abs$ or $dfields'\,Abs$. Thus for (R3)(ii) to hold for the updated heaps h_0, h'_0 we need the new values to be related; indeed, we already established $v \sim_\beta v'$.
  For (R3)(iii) we rely on the discipline. We have $type\,o \leq Abs$ by decomposition. By type soundness for x, $type\,o \leq \Gamma\,x \leq declClass\,f$. By the tree property of the subtype relation, one of these two cases must apply:
  - $Abs \leq declClass\,f$. Then $h\,o.inv > Abs$ from the stipulated precondition; this falsifies the antecedent in (R3)(iii) so we are done.
  - $declClass\,f < Abs$: Then $\mathscr{L}$ is not allowed to depend on $o.f$ (Def. 20); this means precisely that an update of $o.f$ cannot falsify $\mathscr{L}$, so (R3)(iii) is preserved.

– $o \in dom\,(Rh_i * Sh_i)$ for some i. By precondition, $h\,o.inv > declClass\,f$. By type soundness for x, $type\,o \leq \Gamma\,x \leq declClass\,f$; hence $h\,o.inv > type\,o$.
If $o \in dom\,Rh_i$, then $pickAh_i \succ^h_{Abs} o$, so by Corollary 6, $h(pickAh_i).inv > Abs$. If $o \in dom\,Sh_i$, $pickAh_i \succ^h_C o$ for $C > Abs$, so $h(pickAh_i).inv > C > Abs$, by Corollary 6 again. In either case, condition (R3)(iii) in Def. 21 holds in the updated heaps because its antecedent is false.
Finally, if $o \in dom\,Sh_i$ we must also show condition (R3)(ii) for the updated heaps. Because $Sh_i \sim_\beta Sh'_i$ holds initially, this follows from $\beta\,o\,o'$ and $v \sim_\beta v'$ already established.

Case of allocation $x := \mathbf{new}\ B$ Let $o = fresh(B,h)$, so we have $s_0 = [s \mid x \mapsto o]$ and $h_0 = [h \mid o \mapsto [xfields\,B \mapsto defaults\,B]]$ (and *mutatis mutandis* for o', h'_0, s'_0). We choose $\beta_0 = \beta \cup \{(o,o')\}$. To show $\mathscr{R}\ \beta_0\ (\mathsf{heap} \otimes \Gamma)\ (h_0,s_0)\ (h'_0,s'_0)$, first note that by the defaults we have $h_0\,o.inv = \mathsf{Object}$, $h_0\,o.com = false$, and $h_0\,o.own = (nil, \mathsf{Object})$. By the latter default, β_0 fully partitions h_0, h'_0 for Abs. To complete the argument, we consider the following two cases on B.

– $B \leq Abs$. Let $Ah_{new} = [o \mapsto [xfields\,B \mapsto defaults\,B]]$, so $h_0 = Ah_{new} * h$ and the partition of h_0 has the form $h_0 = Fh * \ldots * Ah_{new} * Rh_{new} * Sh_{new}$ where the old partition is unchanged but has an added island with $Rh_{new} = Sh_{new} = \varnothing$. Similarly for o' and h'_0.
Because o and o' are fresh, they are unreachable from objects in h and h' respectively, so $Fh \sim_{\beta_0} Fh'$ follows from $Fh \sim_\beta Fh'$, and similarly for other parts of h and h'. Thus to show $\mathscr{R}\ \beta_0\ \mathsf{heap}\ h_0\ h'_0$, we just use conditions (R1), (R2), (R3)(i), and (R3)(ii) in Def. 21 from the corresponding assumptions for β.
For condition (R3)(iii), consider any preexisting pair of corresponding islands $Ah_i * Rh_i * Sh_i$ and $Ah'_i * Rh'_i * Sh'_i$. The condition (R3)(iii) pertains only to objects in these islands, and is not falsifiable by the addition of another island. For the new island, we have $Ah_{new} \sim_{\beta_0} Ah'_{new}$ in virtue of the default field values (which have no preexisting references). Moreover, we have

$$Ah_{new}(o).inv \leq Abs \Rightarrow \mathscr{L}\,\beta_0(Ah_{new} * Rh_{new} * Sh_{new})(Ah'_{new} * Rh'_{new} * Sh'_{new})$$

because the antecedent is false. The reason being that $Ah_{new}(o).inv = \mathsf{Object} = Ah'_{new}(o).inv$ and by typing $B \neq \mathsf{Object}$.
– $B \not\leq Abs$. Because the default owner is (nil, Object), the partition is $h_0 = Fh_0 * \ldots$ where the islands are unchanged and $Fh_0 = Fh * [o \mapsto [xfields\,B \mapsto defaults\,B]]$. Because o and o' are fresh, $Fh \sim_{\beta_0} Fh'$ follows from $Fh \sim_\beta Fh'$. Moreover, we have $Fh_0 \sim_{\beta_0} Fh'_0$ because the default field values in the new objects do not refer to preexisting objects. Thus we have (R1) and (R2) in Def. 21. Condition (R3) holds for h_0, h'_0 because (R3)(iii) pertains to the unchanged part of the heap and conditions (R3)(i–ii) are preserved because o, o' are fresh.

Case of pack e **as** C Owing to (A2) in Def. 15, we have $C \neq Abs$. By semantics of **pack** e **as** C, what is changed is the value of $e.inv$ and the *com* fields of objects owned by e at C. Let $p = [\![e]\!](s)$ and $p' = [\![e]\!](s')$ Neither p nor p' is in an Rh part of the heap, for the same reasons as in the case of field access. The objects owned by p (resp. p')

at C are not owned at Abs so from $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h,s)\ (h',s')$ we get that the two states agree (modulo β on what is owned by p (resp. p') at C). So everything involved is related by $\sim_\beta$ and that remains true after the updates of *inv* and *com*.

Case of setown x **to** (e_2, C) Let $q = s\,x$ and note that $q \neq nil$ by stipulated precondition. Let $p = [\![e_2 : D_2]\!](h,s)$ and for future reference let $(r,B) = h\,q.own$. By typing we have $type\ p \leq D_2 \leq C$. The resulting heap h_0 is $[h \mid q.own \mapsto (p,C)]$. We take β_0 to be β. As usual, q', p', r', B', h'_0 are determined *mutatis mutandis* from (h',s'). The ownership structure is changed (unless $(r,B) = (p,C)$), but what is relevant to $\mathscr{R}$ is changes in Abs-partition. To reason carefully about these cases, we give notation for the Abs-partitions of the updated heaps: $h_0 = \hat{Fh} * \ldots * \hat{Ah}_n * \hat{Rh}_n * \hat{Sh}_n$ 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 the stipulated precondition for **setown** we obtain

(a) $q \sim_\beta q'$ and $p \sim_\beta p'$ (by hypothesis and by Lemma 28 on e_2, respectively);
(b) $h\ q.inv = \mathsf{Object} = h'\ q'.inv$ (by stipulated precondition)
(c) $p = nil = p'$ or else $h\ p.inv > C$ and $h'\ p'.inv > C$ (by stipulated precondition)
(d) both h and h' satisfy $(\exists o \mid o \succ_{Abs} q) \Rightarrow C = Abs \vee (\exists o \mid o \succ_{Abs} p)$ (owing to hypothesis $\Gamma\,\mathsf{self} \neq Abs$ and stipulated precondition (A4) in Def. 15)
(e) $r = nil = r'$ or else $h\ r.inv > B$ and $h'\ r'.inv > B'$ (from (b), Corollary 6, and hypothesis $\mathscr{R}\ \beta\ \mathsf{heap}\ h\ h'$)

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 $\mathscr{R}\ \beta\ \mathsf{heap}\ h\ h'$ and $q \sim_\beta q'$ we have $q' \in dom\,Fh'$. Moreover, either $r = nil = r'$ or $r \in dom\,Fh$ and also $r' \in dom\,Fh'$ by partitioning. There are the following subcases on p:

- If $p \in dom\,Fh$ or $p = nil = p'$ then the structure of the partition of h_0, h'_0 is unchanged from the initial one. We get $\mathscr{R}\ \beta\ \mathsf{heap}\ h_0\ h'_0$ because the only change is to set field *own* of q, q' to related values p, p', whence $\hat{Fh} \sim_\beta \hat{Fh}'$.
- If $p = pick\,Ah_i$ then by $\mathscr{R}\ \beta\ \mathsf{heap}\ h\ h'$ we have $p' = pick\,Ah'_i$. By type soundness, $type\ p \leq D_2$ and $D_2 \leq C$, and by definition of partition $type\ p \leq Abs$, 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 $\mathscr{R}\ \beta\ \mathsf{heap}\ h_0\ h'_0$ because the only change is to set field *own* of q, q' to related values p, p'.
  - If $Abs = C$ then q and the objects it transitively owns are being transferred into island i, i.e., the partition of h_0 has q in $dom\,\hat{Rh}_i$. Similarly for q' and $\hat{Rh}'_i$. By (c) we have $h\ p.inv > Abs$ and $h'\ p'.inv > Abs$, so the only condition in Def. 21 to check is (R3)(iii), which holds because the antecedent is false —"$\mathscr{L}$ is not in force".
  - 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 conditions (R3)(i) and (R3)(ii) $\hat{Sh}_i \sim_\beta \hat{Sh}'_j$. This follows from $\mathscr{R}\ \beta\ \mathsf{heap}\ h\ h'$ and $q \sim_\beta q'$ and $p \sim_\beta p'$.

– If $p \in dom(Rh_i * Sh_i)$ for some i then p' must be in $dom(Rh'_i * Sh'_i)$ owing to Lemma 26 and $Sh_i \sim_\beta Sh'_j$. So q (resp. q') is transferred into island i (resp. j). Let $o = pickAh_i$ (resp. $o' = pickAh'_j$) so that there is $B \geq Abs$ (resp. $B' \geq Abs$) such that $o \succ^h_B p$ (resp. $o' \succ^{h'}_{B'} p'$). By (c) and Corollary 6 we get $h\ o.inv > B$ (resp. $h'\ o'.inv > B'$) and thus $\mathscr{L}$ 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 $\mathscr{R}\ \beta\ \mathsf{heap}\ h\ h'$ and $p \sim_\beta p'$.

Case $q = pickAh_i$ for some i. So $q' = pickAh'_i$. Owing to the invariant that no Abs-object owns an Abs-object, we have $r \in dom\, Fh$ and for the same reason p is also in $dom\, Fh$ *Mutatis mutandis* for r, Fh'. So again the partition structure is unchanged. We get $\mathscr{R}\ \beta\ \mathsf{heap}\ h_0\ h'_0$ because the only change is to set field *own* of q, q' to related values p, p'.

Case $q \in dom\, Rh_i$ for some i. Then $pickAh_i \succ^h_{Abs} q$ by definition of Abs-partition. Now either $r = pickAh_i$ and $B = Abs$ or $pickAh_i \succ^h_{Abs} r$, by Lemma 25. By Lemma 26 we have $q' \in dom\, Rh'_i$. By (d), either $C = Abs$ or there is o such that $o \succ^h_{Abs} p$. In the case $C = Abs$, we have $type\ p \leq Abs$ (by the typing rule for **setown**) and thus $p = pickAh_k$ for some k, whence $p' = pickAh'_k$. In the other case, because $type\ o \leq Abs$ (by Lemma 5(a)) there is some k with $o = pickAh_k$ Similarly, there is some Abs-object o' that owns p' and some l with $o' = pickAh_l$. By $\beta\ p\ p'$ and Lemma 23 we have $\beta\ o\ o'$ and so $l = k$. 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 from Rh_i into $\hat{Rh}_k$; in parallel, q' and its transitively owned objects are transferred from Rh'_i to $\hat{Rh}'_k$. So couplings for i and k are at risk. By (b) and Corollary 6 we have $h(pickAh_i).inv > Abs$ and thus basic coupling $\mathscr{L}$ is not in force for i in h, nor is it in force for i in h'. By (c) and Corollary 6 we have $h\ o.inv > Abs$ (or $h\ p.inv > Abs$ in the case $C = Abs$) and thus $\mathscr{L}$ is not in force for k in either h or h'. Because the transfer of q is into $\hat{Rh}_k$ and q' into $\hat{Rh}_k$, Thus the condition (R3)(iii) holds in the final heap. The conditions (R3)(i–ii) are not affected.

Case $q \in dom\, Sh_i$ for some i. In regards to $\mathscr{L}$, this case is similar to the case for $q \in dom\, Rh_i$. By coupling of the initial states we have $q' \in dom\, Sh'_i$. The relevant islands are unpacked, by (e), so $\mathscr{L}$ is not in force and (R3)(iii) is preserved. We have to deal with the equivalence and bijection requirements (R3)(i–ii). From $\mathscr{R}\ \beta\ \mathsf{heap}\ h\ h'$ we have that β is a total bijection from Sh_i to Sh'_i and $Sh_i \sim_\beta Sh'_i$. By definition of Abs-partition, both Sh_i and Sh'_i 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 ${Sh'}^+_i$. Let Sh^-_i and ${Sh'}^-_i$ be the remainders so that $Sh_i = Sh^+_i * Sh^-_i$ and $Sh'_i = {Sh'}^+_i * {Sh'}^-_i$. Then by Corollary 24 we have that β is a total bijection from $dom\, Sh^+_i$ to $dom\, {Sh'}^+_i$ and from $dom\, Sh^-_i$ to $dom\, {Sh'}^-_i$; moreover $Sh^+_i \sim_\beta {Sh'}^+_i$ and $Sh^-_i \sim_\beta {Sh'}^-_i$. In the final heaps h_0, h'_0, $\hat{Sh}_i = Sh^-_i$ and $\hat{Sh}_i = Sh^-_i$. By the above considerations, these satisfy the bijection and equivalence conditions.

What remains is to account for Sh^+_i and ${Sh'}^+_i$ which get transferred into islands k. 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 $Sh^+_i \sim_\beta {Sh'}^+_i$.

- If p is in some $domAh_k$ (resp. $domSh_k$) we have p' is the $domAh'_k$ (resp. $p' \in domSh_k$). An argument like in the case for $q \in domRh_i$ shows that $\mathscr{L}$ is not in force so (R3)(iii) is preserved. And if q is going into Sh_k (resp. q' into Sh_k) then the coupling conditions for $\hat{Sh}_k = Sh_k * Sh_i^+$ and $\hat{Sh}'_k = Sh'_k * Sh'^+_i$ hold by the earlier considerations, e.g., $Sh_i^+ \sim_\beta Sh'^+_i$.
- If p is in some $domRh_k$ then the argument is similiar to the preceding case but simpler as there is no bijection or equivalence condition with which to be concerned.

Our main result says that if methods of *Abs* preserve the coupling then all methods do.

Theorem 30 (abstraction).
If $\mathscr{R}$ is a simulation for comparable class tables CT, CT' that are properly annotated for *Abs*, then $\mathscr{R}$ menv $[\![CT]\!]\ [\![CT']\!]'$.

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

$$\mathscr{R} \text{ menv } \mu_i\ \mu'_i \quad \text{for every } i \in \mathbb{N}$$

The result $\mathscr{R}$ menv $[\![CT]\!]\ [\![CT']\!]'$ then follows, because $[\![CT]\!]$ 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

$$\mathscr{R}\ \beta\ (\mathsf{self} : C, \bar{x} : \bar{T} \rightsquigarrow T)\ (\mu_0 C m)\ (\mu'_0 C m)$$

for every β, m, C, where $pars(m,C) = \bar{x}$ and $mtype(m,C) = \bar{T} \to T$. This holds by definition of μ_0, μ'_0, because $\lambda(h,s) \mid \bot$ relates to itself.

Induction step: Suppose $\mathscr{R}$ menv $\mu_i\ \mu'_i$. We must show $\mathscr{R}$ menv $\mu_{i+1}\ \mu'_{i+1}$, that is, for every β, every C, and every m with $mtype(m,C)$ defined:

$$\mathscr{R}\ \beta\ (\mathsf{self} : C, \bar{x} : \bar{T} \rightsquigarrow T)\ (\mu_{i+1} C m)\ (\mu'_{i+1} C m) \qquad (\dagger)$$

where $pars(m,C) = \bar{x}$ and $mtype(m,C) = \bar{T} \to T$.

For arbitrary m we show $(\dagger)$ for all C with $mtype(m,C)$ defined, using a secondary induction on inheritance chains.

The base case of the secondary induction is when class C declares m (i.e., m is declared in both $CT(C)$ and $CT'(C)$ by Def. 16). We go by cases on C.

- Case $C = Abs$. We get $(\dagger)$ from the assumption that $\mathscr{R}$ is a simulation. In detail: Using assumption $\mathscr{R}$ menv $\mu_i\ \mu'_i$ and Def. 27 we get

  $$\mathscr{R}\ \beta\ (\mathsf{self} : C, \bar{x} : \bar{T} \rightsquigarrow T)\ ([\![M]\!]\mu_i)\ ([\![M']\!]'\mu'_i)$$

  whence $(\dagger)$ by definition of μ_{i+1} and μ'_{i+1}.
- Case $C \neq Abs$. Then by Def. 16 of comparable class tables we have $CT(C) = CT'(C)$ and in particular both class tables have the same declaration $T\ m(\bar{T}\ \bar{x})\ \{S\}$, which we call M for short. To show $(\dagger)$, observe first that by semantics we have $\mu_{i+1} C m = [\![M]\!]\mu_i$ and $\mu'_{i+1} C m = [\![M]\!]'\mu'_i$. Unfolding the definition of $[\![M]\!]\mu_i$ and $[\![M]\!]'\mu'_i$ according to (2) in Sect.3, and the definition of $\mathscr{R}\,\beta\,(\mathsf{self} : C, \bar{x} : \bar{T} \rightsquigarrow T)$, it

suffices to proceed as follows. Consider any (h,s) and (h',s') such that $\mathscr{R}\,\beta\ (\mathsf{heap}\otimes\Gamma)\ (h,s)\ (h',s')$ where $\Gamma = (\mathsf{self}:C, \bar{x}:\bar{T})$. Because the class tables are properly annotated and μ_i, μ_i' are in the approximation chains, we have by Prop. 8 that μ_i and μ_i' are disciplined. So we can appeal to Lemma 29, using assumption $\mathscr{R}\ \mathsf{menv}\ \mu_i\ \mu_i'$, to get that the results from S are related. That is, either $[\![\Gamma \vdash S]\!]\mu_i(h,s) = \bot = [\![\Gamma \vdash' S]\!]'\mu_i'(h',s')$ 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 $(h_0,s_0) = [\![\Gamma \vdash S]\!]\mu_i(h,s)$ and $(h_0',s_0') = [\![\Gamma \vdash' S]\!]'\mu_i'(h',s')$. Then, by definition of $\mathscr{R}\,\beta\,\Gamma$, we get $\mathscr{R}\ \beta_0\ (\mathsf{result}:T)\ [\mathsf{result}\mapsto s_0\,\mathsf{result}]\ [\mathsf{result}\mapsto s_0'\,\mathsf{result}]$ from $\mathscr{R}\ \beta_0\ \Gamma\ s_0\ s_0'$. This concludes the argument that the outcomes are related by $\mathscr{R}\,\beta_0\,(\mathsf{heap}\otimes T)$.

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 $superC$. By semantics, $\mu_{i+1}\,C\,m = \mu_{i+1}\,(superC)\,m$ and $\mu_{i+1}'\,C\,m = \mu_{i+1}'\,(superC)\,m$ so we get (†) for C directly from the secondary induction hypothesis.

6 Using the theorem

A complete program is a command S in the context of a class table. To show equivalence between CT,S and CT',S, one proves simulation for *Abs* and then appeals to the abstraction theorem to conclude that $[\![S]\!]$ is related to $[\![S]\!]'$. Finally, one appeals to an *identity extension lemma* that says the relation is the identity for programs where the encapsulated representation is not visible. We choose simple formulations that can also serve to justify more specification-oriented formulations. We say that a state (h,s) is *Abs-free* if $type\,o \not\leq Abs$ for all $o \in dom\,h$.

Lemma 31 (identity extension) If $\mathscr{R}\ \beta\ (\mathsf{heap}\otimes\Gamma)\ (h,s)\ (h',s')$ then $encapAbs\,(h,s) \sim_\beta encapAbs\,(h',s')$.

Lemma 32 (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 $\mathscr{R}\ \beta\ (\mathsf{heap}\otimes\Gamma)\ (h,s)\ (h',s')$.

Definition 33 (program equivalence) Suppose programs $CT, (\Gamma \vdash S)$ and $CT', (\Gamma \vdash' S')$ are such that CT, CT' are comparable and properly annotated, and moreover S, S' are properly annotated. The programs are *equivalent* iff for all disciplined, *Abs*-free (h,s) and (h',s') in $[\![\mathsf{heap}\otimes\Gamma]\!]$ and all β with β total on h,h' and $(h,s) \sim_\beta (h',s')$, there is some $\beta_0 \supseteq \beta$ with $encapAbs\,([\![\Gamma \vdash S]\!]\mu(h,s)) \sim_{\beta_0} encapAbs\,([\![\Gamma \vdash' S']\!]'\mu'(h',s'))$ where $\mu = [\![CT]\!]$ and $\mu' = [\![CT']\!]'$.

Proposition 34 (simulation and equivalence) Suppose programs $CT, (\Gamma \vdash S)$ and $CT', (\Gamma \vdash' S)$ are properly annotated and $\mathscr{R}$ is a simulation from CT to CT'. If $\Gamma\,\mathsf{self} \neq Abs$ then the programs are equivalent.

On inheritance. The stipulated preconditions for **unpack**, **pack**, **setown** all include exact type tests on *inv*, i.e., conditions of the form self.$inv = C$. A typical pattern is for a method in class C to unpack itself from C —so, when inherited into a subclass $B < C$, the precondition of this code is bound to fail. This motivates a second pattern mentioned in Sect. 2.2: instead of simply inheriting a method, it may be better to override it with an implementation that merely unpacks, makes a super call, and repacks. In Figs. 3 and 4, method add has roughly this form, though with a little extra code.

The original paper [11] deals with programs that have explicit specifications, in which case this issue is manifest in terms of pre- and post-conditions for methods that may be inherited. That paper introduces a short-hand notation for specifications which is justified in terms of boilerplate method overrides as described just above. The Spec# tool achieves this effect, without requiring explicit method declarations, by re-verifying code inherited in subclasses.

In the present work, we are not concerned with specifications as such, but the issue is still present: in our semantics, a method of C inherited and acting on an object of dynamic type $B < C$ will return $\perp$ if it asserts $inv = C$. Our technical results are sound with no restrictions on inheritance —in particular, Def. 27 does not require inherited methods to preserve the coupling. But to be useful, an inherited method that unpacks needs to be replaced by boilerplate method implementations using the unpack/super/pack pattern as in method add. Consider a class table where inheritance has been "expanded" in this way. Methods inherited into *Abs* give rise to declarations in *Abs* that must be shown to preserve the coupling. On the other hand, a method that is inherited and does not touch state encapsulated for the subclass (e.g., getRuns in our example) necessarily preserves the coupling and is subject to no proof obligation.

7 Related work

Representation independence. Representation independence is proved in [6] for a language with shared mutable objects 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. Like the present chapter, our previous results are based on a semantics in which the semantics of primitive commands is given in straightforward operational terms. It is a denotational semantics in that a command denotes a state transformer function, defined by induction on program structure. To handle recursion, method calls are interpreted relative to a method environment that gives the semantics of all methods. This is constructed as the limit of approximations, each exact up to a certain maximum calling depth. This model directly matches the recursion rule of Hoare logic, of which the abstraction theorem is in some sense a generalization.

Representation independence is needed not only for modular proof of equivalence of class implementations but also for modular reasoning about improvements (called data refinement). Such reasoning is needed for correctness preserving refactoring. The refactoring rules of Borba et al. [15] were validated using the data refinement theory

of Cavalcanti and Naumann [20] which does not model sharing/aliasing. A recent paper [52] achieves correctness preserving refactoring for a class based language with shared mutable objects similar to the one considered in this chapter but adapted to encompass protected visibility and ownership of instances of library classes. The ownership regime of [6] is adapted to this setting. Their abstraction theorem accounts for changes of data representation in an inheritance hierarchy of classes. The theorem entails a data refinement law (similar to our Prop. 34) and facilitates correctness proofs of several refactorings that impact entire class trees.

Filipovic et al [33] provide an elegant semantic analysis of encapsulation and simulation, also in terms of refinement rather than equivalence. By contrast with the present chapter, they consider a 'static module' that owns internal state, rather than multiple instances of an abstraction, and their programming language allows pointer arithmetic. Using an instrumented semantics, they characterize client programs that do not read/write encapsulated locations, and show that such clients are representation independent. Like the *inv*/*own* discipline, and unlike type-based approaches, it is reads and writes, rather than the existence of pointers, that is controlled. Their theory makes minimal assumptions and thus provides a fundamental account of abstraction in the presence of shared mutable objects including those that are transferred across the encapsulation boundary. By contrast, our instrumentation is more intricate and less general, but provides a practical technique whereby clients can be proved to respect encapsulation. Filipovic et al require couplings to satisfy a condition, called 'growing relations'. Technically, it is needed due to non-determinacy of the allocator, but they argue that the semantic issue would arise even with a deterministic allocator, if specifications are taken into account, due to under-determinacy. Semantic analysis of this issue remains an open question.

In the realm of functional languages representation independence has been well studied, particularly for System F [59] and its extensions, e.g., with recursive functions [58] and general recursive types [2,25,44]. Several recent papers [62,40,14,31] consider representation independence for languages with references but do not address class based languages directly. An exception is the work of Koutavas and Wand [41] in which Kripke logical relations are used to verify the examples in our earlier work [6]. Ahmed et al. [1] achieve representation independence for a higher-order call-by-value λ-calculus with existential type abstraction as well as higher-order store (obtained by allowing general ML-style references). Their couplings are Kripke logical relations that involve 'islands' akin to ours, but which grow monotonically, disallowing ownership transfer. These semantic results are adapted by Dreyer et al [31] to a relational modal logic for the same programming language.

In a technical report [5], we generalize the results in the present chapter to a language with generic classes, but this is still first-order and quite different from ML-like languages.

Just as we show how the *inv*/*own* discipline for modular reasoning about data invariants gives rise to a form of representation indpendence, Birkedal and Yang [14] show how modular reasoning in separation logic gives rise to representation independence. They provide a relational interpretation of separation logic with higher order frame rules; such rules account for hiding of invariants on mutable state in higher order

programs. The relational interpretation shows that, if a client of an abstraction is proved correct then it is independent of the hidden representation. Thamsborg et al [63] develop similar results for a version of separation logic in which 'abstract predicates' (explained towards the end of this section) serve to isolate clients from invariants/relations on internal representations.

Methodologies. Several facets of the ownership methodology and its enforcement using static analysis techniques such as ownership types are presented in this volume. Here we sample some of the existing related work. Much of the literature concerns hierarchical ownership which is imposed by arranging the heap in a manner such that there is a single dominating owner [22] of representation objects. Clients are restricted from directly accessing representation objects. Ownership confinement is maintained in all reachable program states, hence it is a program invariant.

Static analyses for confinement such as ownership type systems [16,21,17] are a means to enforce hierarchical ownership. Most ownership type systems preclude ownership transfer; where allowed, it is achieved using nonstandard constructs such as destructive reads and restrictive linearity constraints (e.g., [18,61]). The overall objective of these static analyses is to provide some means of encapsulation for the purpose of modular reasoning. However they do not formalize exactly how the confinement invariant facilitates modular reasoning.

Müller and Rudich [48] extend Universe Types which provides encapsulation and has been adopted by JML for invariants, to solve the difficult problem of ownership transfer.

Drossopoulou et al. [32] introduce a general framework to describe verification techniques for invariants. The framework is based on variations on the idea that invariants hold exactly when control crosses module boundaries, e.g., *visible state semantics* requires all invariants to hold on all public method call/return boundaries. Several ownership disciplines are studied as instances of the framework.

While ownership is widely applicable, many programs involve local object structures which do not follow the ownership discipline. For example, friends and peer dependencies [43,13,51] exhibit non-hierarchical dependencies via cooperating classes of objects. Similarly, design patterns [34] involve local object structures which do not follow the ownership discipline. In the observer pattern, neither the subject nor its observers own each other; in the composite pattern, a client can have direct access to any node (not just the root) in a composite tree.

Cameron et al. [19] addressed the need for clusters without a single dominating owner. Ownership types are adapted to a system of "boxes" (clusters) which do not ensure encapsulation. However an effect system for disjointness of boxes is provided and proven sound.

The friendship discipline [51] that augments the *inv*/*own* discipline, can be used for modular reasoning about dependencies involving cooperating classes of objects. For concurrent programs, the *inv*/*own* discipline has been generalized by Locally Checked Invariants [24] which is implemented in the VCC tool [23]. In this case, ownership is complemented by non-hierarchical dependencies which are tracked in ghost state called "claims", generalizing the friendship discipline.

A key observation about the above examples is that reasoning about hierarchical and non-hierarchical dependencies ultimately involves the preservation of global program invariants. Local reasoning [53] about global invariants allows scalability of reasoning and ease of automation. For example, in the observer pattern, although there may be global invariants that hold over all objects in a heap, at any one point of time one can reason locally about a cooperating cluster of objects comprising of a single subject and its observers. The global invariant can be factored into two parts: one part that depends on this cluster of objects and another part that is independent of the cluster. The latter cannot be falsified by operations that affect the cluster, so it is enough to establish the former to show preservation of the global invariant.

Local reasoning as described above is embodied in *frame conditions* of a procedure specification that designates what part of the state is susceptible to change, together with frame-based reasoning that "all else is unchanged". The frame condition of a command is often termed its footprint (following separation logic [53]) and can be expressed using ghost state in the form of mutable auxiliary fields and variables. Use of ghost state in frame conditions was pioneered by Kassios, who dubbed it "dynamic framing" [39]. Region logic [10,8] is a Hoare logic for object-based programs that features local reasoning with frame conditions expressed in terms of sets of references (termed *regions*). VERL [64] is a verifier based on region logic that embodies local reasoning.

We have already seen the use of ghost states such as inv, own, com in the *inv*/*own* discipline. The implicit frame condition in the discipline is that clients cannot write fields of objects they do not own. Therefore writes to owned fields do not need to appear in the frame conditions of specifications. The Spec# tool [12] automatically generates such implicit frame conditions. In these tools particular methodologies such as friendship or ownership are an integral part of the verification conditions. The goal of verifiers such as VERL or Dafny [26,42] is to decouple methodologies from verification condition generation —both for tool modularity and for methodological flexibility.

To reason about non-ownership disciplines Parkinson and Bierman [57] propose abstraction instead of hiding, via second order assertions in separation logic. The jStar [30] tool implements this idea. Client reasoning can be done by means of "abstract predicates" —predicates whose concrete implementations are unknown to the client. For example a client may use an abstract predicate whose concrete implementation might be the layout of the heap. Parkinson [56] clearly articulates the case for specifications at the level of object clusters and shows an example specification of the Observer pattern that uses abstract predicates. For more insight on these issues we refer the reader to the companion chapter [55].

8 Discussion

Adaptations of the inv/own discipline. As compared with previous work on the discipline, we have imposed some additional restrictions to achieve sufficient information hiding to justify a modular rule for equivalence of class implementations. We argue that the restrictions are not onerous for practical application, though further practical experience is needed with the discipline and with our rule.

The first restriction is on field reads. Code in a client class cannot be allowed to read a field of an encapsulated representation object, although the discipline allows the existence of the reference; otherwise the client code could be representation dependent. On the other hand, a class such as *Hashtable* might be used both by clients and in the internal representation of the class *Abs* under revision; certainly the code of *Hashtable* needs to read its own fields. A distinction can be made on the basis of whether the current target object, i.e., self, is owned by an instance o of *Abs*. If it is, then we do not need the method invocation to preserve the coupling and we can allow reading of objects owned by o. If the target object is not owned by an instance of *Abs* then it should have no need to access objects owned by *Abs*. This distinction appears in the statement of Lemma 29 and it is used to stipulate a precondition for field access (see (A3) in Def. 15).[10]

Because the coupling relation imposes the user-defined local coupling only when an *Abs*-object is packed, it appears necessary to restrict **pack** e **as** *Abs* to occur only in code of *Abs* in order for simulation to be checked only for that code. In the majority of known examples, packing to a class C is only done in code of C, and this is required in Leino and Müller's extension of the discipline to handle static fields.

Similar considerations apply to **setown** x **to** (y,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 *Abs*-boundary at all, but it turns out that the only problematic case is transfer out from an *Abs* island. In the unusual case that **setown** x **to** (y,C) occurs in code outside *Abs* but x is initially inside the island for some *Abs*-object, x must end up in the island for some *Abs*-object. Our stipulated precondition says just this. In practice it seems that the obligation can be discharged by simple syntactic considerations of visibility and/or lightweight alias control.

The last restriction is that an *Abs* object cannot own other *Abs* objects. This does not preclude containers holding containers, because a container does not own its content (e.g., AQueue owns the Qnodes but not the tasks). It does preclude certain recursive situations. For example, we could allow Qnode instances to own their successors but then we could not instantiate the theory with *Abs*:=Qnode. This does not seem too important since it is Queue that is appropriate to view as an abstraction coupled by a simulation. The restriction is not needed for soundness of simulation. But absent the restriction, nested islands would require a healthiness condition on couplings (similar to the healthiness condition used by Cavalcanti and Naumann [20, Def. 5]); e.g., coupling for an instance of Qnode would need to recursively impose the same predicate on the nxt node. We disallow nested islands in the present work for simplicity and to highlight connections with separation logic.

---

[10] This is unattractive in that the other stipulated preconditions mention only direct ownership whereas this one uses transitive ownership. However, in practical examples code outside *Abs* rarely has references to encapsulated objects. We believe such references can be adequately restricted using visibility control and/or lightweight confinement analyses, e.g., [65,6]. Moreover, as noted following Def. 15, the transitive ownership relation can be maintained in ghost state so that the stipulated precondition can be expressed without induction.

Future work. The discipline may seem somewhat onerous in that it uses verification conditions rather than lighter weight static analysis for control of the use of aliases. (We have to say "use of", because whereas confinement disallows certain aliases, the invariant discipline merely prevents faulty exploitation of aliases.) The Spec# tool provides some support for inference of annotations [12]. For many situations, simple confinement rules and other checks are sufficient to discharge the proof obligations and this needs to be investigated for the additional obligations we have introduced. The advantage of a verification discipline over types is that, while simple cases can be checked automatically, complicated cases can be checked with additional annotations rather than simply rejected.

The generalization to a small group of related classes is important, as revisions often involve several related classes. One example would be a revision of our Queue example that involves revising Qnode as well. If nodes are used only by Queue then this is subsumed by our theory, as we can consider a renamed version of Qnode that coexists with it. The more interesting situations arise in refactoring and in design patterns with tightly related configurations of multiple objects. Naumann et al. explore this generalization in recent work [52].

We are currently working on a relational version of region logic as a basis for verification of e.g., soundness of refactorings as well as examples discussed here and in our earlier work [6]. A crucial part of this verification process is reasoning about representation independence using proof rules of the relational logic.

Acknowledgments

We thank Amal Ahmed, Mike Barnett, Lars Birkedal, Sophia Drossopoulou, Ivana Filipovic, Rustan Leino, Peter Müller, Peter O'Hearn, Uday Reddy, Wolfram Schulte, Noah Torp-Smith, and Hongseok Yang for discussions. Thanks to the anonymous ECOOP 2005 referees as well as the referees of this chapter for their detailed comments.

Banerjee was supported in part by CM Project S2009TIC-1465 Prometidos, MICINN Project TIN2009-14599-C03-02 Desafios, EU NoE Project 256980 Nessos. Naumann was supported in part by NSF grant CCF-0915611 and by Microsoft Research.

References

1. Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *ACM Symp. on Princ. of Program. Lang.*, pages 340–353, 2009.
2. Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming*, pages 69–83, 2006.
3. Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 3 edition, 2009.
4. Anindya Banerjee and David A. Naumann. Ownership transfer and abstraction. Technical Report TR 2004-1, Computing and Information Sciences, Kansas State University, 2003.
5. Anindya Banerjee and David A. Naumann. State based encapsulation and generics. Technical Report CS Report 2004-11, Stevens Institute of Technology, 2004.
6. Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM*, 52(6):894–960, 2005.

7. Anindya Banerjee and David A. Naumann. State based ownership, reentrance, and encapsulation. In *European Conference on Object-Oriented Programming*, pages 387–411, 2005.
8. Anindya Banerjee and David A. Naumann. Local reasoning for global invariants, part II: Dynamic boundaries. Extended version of [50]. `http://www.cs.stevens.edu/~naumann/pub/locResGloInvII.pdf`, 2011.
9. Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *European Conference on Object-Oriented Programming*, volume 5142 of *LNCS*, pages 387–411, 2008.
10. Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Local reasoning for global invariants, part I: Region logic. Extended version of [9]. `http://www.cs.stevens.edu/~naumann/pub/locResGloInvI.pdf`, 2011.
11. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
12. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *LNCS*, pages 49–69, 2004.
13. Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen, editor, *Mathematics of Program Construction*, pages 54–84, 2004.
14. Lars Birkedal and Hongseok Yang. Relational parametricity and separation logic. *Logical Methods in Computer Science*, 4(2), 2008.
15. Paulo Borba, Augusto Sampaio, and Márcio Cornélio. A refinement algebra for object-oriented programming. In *European Conference on Object-Oriented Programming*, number 2743 in LNCS, pages 457–482, 2003.
16. Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 211–230, 2002.
17. Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *ACM Symp. on Princ. of Program. Lang.*, pages 213–223, 2003. Invited paper.
18. John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 2–27, 2001.
19. Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple ownership. In *ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 441–460, 2007.
20. Ana Cavalcanti and David A. Naumann. Forward simulation for data refinement of classes. In *Formal Methods Europe*, volume 2391 of *LNCS*, pages 471–490, 2002.
21. David Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 292–310, November 2002.
22. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *SIGPLAN*, pages 48–64, October 1998.
23. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: a practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLS)*, volume 5674 of *LNCS*, pages 23–42, 2009.

24. Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *Computer Aided Verification*, volume 6174 of *LNCS*, pages 480–494, 2010.
25. Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types. *Electr. Notes Theor. Comput. Sci.*, 172:259–299, 2007.
26. Dafny. Available at `http://boogie.codeplex.com/`.
27. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
28. D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Research 156, DEC Systems Research Center, 1998.
29. Werner Dietl and Peter Müller. Object ownership in program verification. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-oriented Programming*, pages XXX–XXX. Springer, 2012.
30. Dino Distefano and Matthew J. Parkinson. jStar: Towards practical verification for Java. In *ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 213–226, 2008.
31. Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful adts. In *ACM Symp. on Princ. of Program. Lang.*, pages 185–198, 2010.
32. Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. A unified framework for verification techniques for object invariants. In *European Conference on Object-Oriented Programming*, volume 5142 of *LNCS*, pages 412–437, 2008.
33. Ivana Filipovic, Peter W. O'Hearn, Noah Torp-Smith, and Hongseok Yang. Blaming the client: on data refinement in the presence of pointers. *Formal Asp. Comput.*, 22(5):547–583, 2010.
34. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
35. John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.
36. C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
37. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–459, May 2001.
38. Bart Jacobs and Erik Poll. Java program verification at Nijmegen: Developments and perspective. In *International Symposium on Software Security*, volume 3233 of *LNCS*, pages 134–153, 2004.
39. Ioannis T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–288, 2011.
40. Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *ACM Symp. on Princ. of Program. Lang.*, pages 141–152, 2006.
41. Vasileios Koutavas and Mitchell Wand. Reasoning about class behavior. In *Informal proceedings of FOOL/WOOD.*, 2007.
42. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference*, volume 6355 of *LNCS*, pages 348–370, 2010.
43. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming*, pages 491–516, 2004.

44. Paul-André Melliès and Jerome Vouillon. Recursive polymorphic types and parametricity in an operational framework. In *IEEE Symp. on Logic in Computer Science*, pages 82–91, 2005.
45. Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, second edition, 1997.
46. John C. Mitchell. Representation independence and data abstraction. In *ACM Symp. on Princ. of Program. Lang.*, pages 263–276, 1986.
47. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.
48. Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 461–478, 2007.
49. David A. Naumann. Verifying a secure information flow analyzer. In *Theorem Proving in Higher Order Logics (TPHOLS)*, volume 3603 of *LNCS*, pages 211–226, 2005.
50. David A. Naumann and Anindya Banerjee. Dynamic boundaries: Information hiding by second order framing with first order assertions. In *European Symposium on Programming*, volume 6012 of *LNCS*, pages 2–22, 2010. Invited paper.
51. David A. Naumann and Mike Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theoretical Computer Science*, 365:143–168, 2006.
52. David A. Naumann, Augusto Sampaio, and Leila Silva. Refactoring and representation independence for class hierachies. *Theoretical Computer Science*, 433:60–97, 2012.
53. Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, volume 2142 of *LNCS*, pages 1–19, 2001.
54. Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. *ACM Trans. Program. Lang. Syst.*, 31(3), 2009.
55. Matthew Parkinson and Gavin Bierman. Separation logic for object-oriented programming. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-oriented Programming*, pages XXX–XXX. Springer, 2012.
56. Matthew J. Parkinson. Class invariants: the end of the road? In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, 2007.
57. Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *ACM Symp. on Princ. of Program. Lang.*, pages 247–258, 2005.
58. Andrew M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2005.
59. John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, 1984.
60. Jan Smans, Bart Jacobs, Frank Piessens, Willem Penninckx, Frédéric Vogels, and Pieter Philippaerts. Verifying java programs with VeriFast. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-oriented Programming*, pages XXX–XXX. Springer, 2012.
61. Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381, 2000.
62. Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. *J. ACM*, 54(5), 2007.
63. Jacob Thamsborg, Lars Birkedal, and Hongseok Yang. Two for the price of one: lifting separation logic assertions. *Logical Methods in Computer Science*, 2012. To appear.
64. Verl: VErifier for Region Logic. Software distribution, at `http://www.cs.stevens.edu/~naumann/pub/VERL/`.
65. Jan Vitek and Boris Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.