STEVENS
Institute of Technology

Ownership Confinement Ensures Representation Independence for Object-Oriented Programs

Anindya Banerjee
Computing and Information Science
Kansas State University

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

Technical Report CS-2004-14

(also appears as Kansas State University CIS TR 2004-6)

Ownership Confinement Ensures Representation Independence for Object-Oriented Programs

February 11, 2005

ANINDYA BANERJEE
Department of Computing and Information Sciences
Kansas State University
Manhattan KS 66506 USA
and
DAVID A. NAUMANN
Department of Computer Science
Stevens Institute of Technology
Hoboken NJ 07030 USA

Dedicated to the memory of Edsger W. Dijkstra.

Representation independence or relational parametricity formally characterizes the encapsulation provided by language constructs for data abstraction and justifies reasoning by simulation. Representation independence has been shown for a variety of languages and constructs but not for shared references to mutable state; indeed it fails in general for such languages. This paper formulates representation independence for classes, in an imperative, object-oriented language with pointers, subclassing and dynamic dispatch, class oriented visibility control, recursive types and methods, and a simple form of module. An instance of a class is considered to implement an abstraction using private fields and so-called representation objects. Encapsulation of representation objects is expressed by a restriction, called confinement, on aliasing. Representation independence is proved for programs satisfying the confinement condition. A static analysis is given for confinement that accepts common designs such as the observer and factory patterns. The formalization takes into account not only the usual interface between a client and a class that provides an abstraction but also the interface (often called "protected") between the class and its subclasses.

Categories and Subject Descriptors: D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*; F.3.1 [**Theory of Computation**]: Logics and Meanings of Programs—*Specifying and Verifying and Reasoning about Programs*

General Terms: Languages, Verification

Additional Key Words and Phrases: Alias control, confinement, relational parametricity, simulation

Contents

[1]

[1]This is an expanded and revised version of a paper originally appearing in *ACM Symposium on Principles of Programming Languages*, 2002.
This technical report is largely the same as a submission to JACM which has been accepted subject to revisions to reduce the length and improve the exposition (arXiv:cs.PL/0212003 v1). Some clarifications have been made and typos corrected in accord with the reviews.
Banerjee was supported by NSF grants EIA-9806835, CCR-0209205 and NSF Career award CCR-0093080/CCR-0296182.
Naumann was supported by NSF grants INT-9813854 and CCR-0208984 and a grant from the New Jersey Commission on Science and Technology.
Banerjee and Naumann were also supported by EPSRC grant GR/S03539, "Abstraction, Confinement and Heap Storage", for which we thank the principal investigator, Peter O'Hearn.

1. INTRODUCTION

You have implemented a class [Dahl and Nygaard 1966; Arnold and Gosling 1998], FIFO, whose instances are FIFO queues with public methods enqueue and dequeue as well as method size that reports the number of elements in the queue. The class, implemented in some Java-like object-oriented language, is part of a library and is used by many programs, most unknown to you. The queue is *represented* using a singly linked chain of nodes that point to elements of the queue. There is also a sentinel node [Cormen et al. 1990]. Each instance of FIFO has a field num with the number of nodes and a field snt that references the sentinel. You realize that a simpler, more efficient implementation can be provided without the sentinel, using two fields, head and tail, pointing to the end nodes in the chain. You revise method size to return num instead of num-1 and revise the other methods suitably. You are guided to the necessary revisions by thinking about the correspondence, sometimes called a *simulation relation*, between the representations for the two versions.

Can the revisions affect the behavior of clients, that is, programs that use class FIFO in some way or other? The answer would be yes, if some client determined the number of nodes by reading field num directly. A client that refers to field name snt would no longer compile. But you have taken care to *encapsulate* the queue's representation: the fields are declared to be private. By using programming language constructs like private fields you aim to ensure that client programs depend only on the *abstraction* provided by the class, not on its representation. If client behavior is independent from the representation of FIFO, it is enough for you to ensure equivalent visible behavior of the revised methods.

For scalable systems, scalable system-building tools, and scalable development methods, abstraction is essential. For reasoning about a single component, e.g., a class, module, or local block, abstraction makes it possible to consider other components in terms of their behavioral interface rather than their internal representation.[2] Abstraction is needed for the automated reasoning embodied in static analysis tools [Cousot and Cousot 1977] and it is needed for formal and informal reasoning about functional correctness during development and evolution [Milner 1971; Hoare 1972]. Modular reasoning has always been a central issue in software engineering and in static analysis. With the ascendancy of mobile code it has become absolutely essential. For example, it is possible for clients of FIFO to be linked to it only at runtime, so it is impossible to check all uses to determine whether the revisions affect them.

The need for flexible but robust encapsulation mechanisms to support data abstraction has been one of the driving forces in the evolution of programming language design, from type safety and scoped local variables to module and abstract data type constructs [Liskov and Guttag 1986]. There is a rich theoretical literature on the subject (e.g., [Plotkin 1973; Reynolds 1974; Donahue 1979; Haynes 1984; Reynolds 1984; He et al. 1986; Mitchell 1996; Lynch and Vaandrager 1995; de Roever and Engelhardt 1998]). Many different language constructs have been studied. There is considerable variation in the details of these theories, partly because the intended applications vary from justifying general tools for program

[2]Even a primitive type like **int** is an abstraction from the machine representation.

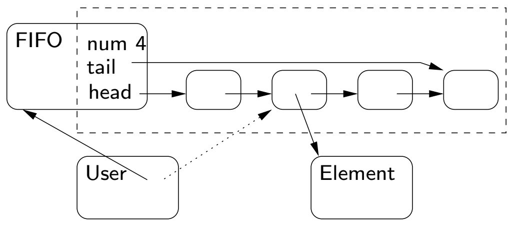

Fig. 1. A FIFO object with its encapsulated representation: private fields and nodes of a list (within the dashed rectangle). One element of the queue is shown as well as a user of the queue, but other objects and references are omitted. The dotted reference is an example of representation exposure.

analysis and transformation to justifying proof rules to be applied to specific programs as in the FIFO example. The common thread is that two implementations of a component are linked by a simulation relation between the two representations.

Unfortunately, these theories are inadequate for object-oriented programs. They deal well with the encapsulation of data structures that correspond directly to some language construct, such as modules, local variables, or private fields. But the FIFO example also involves encapsulation of a data structure composed of heap cells and pointers, including aliasing with the tail field as depicted in Fig. 1.

The problem is that encapsulation provided by language constructs often runs afoul of aliasing. For variables and parameters, aliasing can be prevented through syntactic restrictions that are tolerable in practice (and often assumed in formal logics and theories). Aliasing via pointers is an unavoidable problem in object oriented programming where shared mutable objects are pervasive. Yet unintended aliasing can be catastrophic. A version of the Java access control system was rendered insecure because a leaked reference to an internal data structure made it possible to forge crytographic authentication [Vitek and Bokowski 2001]. In simply typed languages, types offer limited help: variables x, y are not aliased if they have different types. Even this help is undercut by subclass polymorphism: in Java, a variable x of type **Object** can alias y of any type.

The ubiquity and practical significance of the issue is articulated well in the manifesto of Hogg et al. [1992]. A number of subsequent papers in the object-oriented programming literature propose disciplines to control aliasing. Of particular relevance are disciplines that impose some form of *ownership confinement* that restricts access to designated "representation objects" except via their "owners", to prevent *representation exposure* [Leino and Nelson 2002]. A good survey on confinement, especially ownership, can be found in the dissertation of Clarke [2001]; see also Lea [2000],Vitek and Bokowski [2001], Clarke et al. [2001], Müller and Poetzsch-Heffter [2000b], Boyland [2001], Aldrich et al. [2002], and the related work section of this paper.

In Figure 1, an instance of class FIFO (the owner) uses private fields to point to objects intended to be part of its encapsulated representation, as indicated by the dashed rectangle. The contribution of this paper is a theory of representation independence for encapsulation of data in the heap, using ownership confinement. We follow Reynolds [1984] in calling our main result an *abstraction theorem.* Some readers may prefer the term *relational parametricity.*

The literature on confinement is largely concerned with static or dynamic checks to ensure invariance of various confinement properties. One of our contributions is to show how established semantic techniques can be used to evaluate confinement disciplines. To prove our abstraction theorem, we use a semantic formulation of confinement. Separately, we give a modular, syntax-directed static analysis for confinement and show that it accepts some interesting example programs that embody important object-oriented design patterns.

There are a number of ways in which abstractions can be expressed using constructs of contemporary object-oriented languages, including modules, classes, local variables, object instances, not to mention heap structures such as object groups. We treat the most common situation: an instance of some class is viewed as representing an abstraction, possibly using some other objects as part of its representation.

We are aware of no previous results on representation independence that address encapsulation of objects in the heap. Thus it is tempting to present the ideas in the setting of a simple idealized language, say a simple imperative language with pointers to mutable heap cells. But this would leave open some challenging issues, such as how class-based scoping rules fit with instance-based abstraction. We have chosen to consider a rich imperative object-oriented language with class-based visibility, inheritance and dynamic binding, type casts and tests, recursive types, and other features sufficient for programs that fit common design patterns such as observer and factory [Gamma et al. 1995].

Previous work on representation independence has been concerned with relating two versions of a component with respect to programs that use the component. But the designer of a class needs to consider not only users (the client interface) but also subclasses (the *protected interface*). This is a source of complication in our treatment of confinement and, to a lesser extent, in our treatment of representation independence. Our results consider replacement of one version of a class by another with the same public interface, in the context of arbitrary classes that use it or are subclasses of it.

Overview and readmap. Sect. 2 introduces the language for which our results are proved and describes a simple example with which we review the formalization of representation independence using simulation relations. The example is extended to one showing how representation independence can be invalidated by leaked references to representation objects. The section concludes with an informal statement of our abstraction theorem.

Sect. 3 discusses more elaborate examples that typify object-oriented programs. A version of a Meyer-Sieber [1988] example shows how higher order programs can be expressed. Versions of the observer pattern [Gamma et al. 1995] illustrate challenges in formulating robust but practical notions of confinement. The section concludes with an informal description of our notion of ownership confinement.

Sect. 4 formalizes the syntax and typing rules. Sect. 5 gives a surprisingly simple denotational semantics in the manner of Strachey [2000]. The reader is expected to be familiar with elementary domain theory and fixpoints but nothing what is found in introductory textbooks [Davey and Priestley 1990].

Confinement, the semantic notion, is defined formally in Sect. 6. Sect. 7 gives the first main result, an abstraction theorem for confined programs. Sect. 8 shows in detail how the theorem applies to the examples in Sect. 3 and to further variations on the observer pattern. Sect. 9 considers examples of the interface between an owner class and its subclasses. To achieve a sufficiently flexible form of confinement for subclasses of the owner class, while avoiding irrelevant complication, we add a simple module construct to the language. Sect. 10 proves a second abstraction theorem, for this extended language and for a generalized notion of simulation needed for owner subclasses. Sect. 11 wraps up the technical development by defining a static analysis for confinement that accepts the examples of Sections 2, 3, 8, and 9; soundness with respect to (semantic) confinement is shown. Sect. 12 discusses related work and open challenges.

Detailed proofs are given, as the complexity of similar languages has led to errors in published proofs, e.g., of type soundness. Appendices give some additional proofs.

The organization of the paper is intended to make it possible for the casual reader to skip some technical material and still get the gist of the results. Readers who wish to study the details may still prefer to skip, on a first reading, material concerning object constructors and proofs that involve fixpoints and inheritance.

Differences from the preliminary version. Outgoing references, from representation objects to client objects, were disallowed in the preliminary version of this paper [Banerjee and Naumann 2002a]. We conjectured that they could be allowed if restricted to read-only access as in [Müller and Poetzsch-Heffter 2000b; Leino and Nelson 2002]. Here we allow them without restriction, as is needed to handle examples such as the observer pattern where observers may well change state in response to events. We have also added constructors to the language, at the cost of some complexity in proofs due to the interdependence of semantics for commands and for constructors. The benefit is succinct formulation of an abstraction theorem sufficient for transparent application to realistic examples. The other major additions are as follows: module-scoped methods, the generalized abstraction theorem, substantial worked examples, and the static analysis for confinement.

In [Banerjee and Naumann 2002a] we discuss simulation proofs of the equivalence of "security passing style" [Wallach et al. 2000] with the lazy "stack inspection" implementation of Java's privilege-based access control mechanism [Gong 1999], and then extend our language to include access control. We give an abstraction theorem for this extended language. It was this study that led us to the main results but in retrospect it seems tangential and is omitted.

2. REPRESENTATION INDEPENDENCE

We begin this expository section with a very simple example of representation independence, contrived mainly to introduce the Java-like language that we will use. Building on this example we show how pointer aliasing can invalidate representation independence. We conclude with an informal statement of the main results. Sect. 3 deals with more challenging examples including the observer pattern [Gamma et al. 1995] and gives a more precise description of ownership confinement.

2.1 A first example

The concrete syntax for classes is based on that of Java [Arnold and Gosling 1998] but using more conventional notation for simple imperative constructs. Keywords are typeset in bold font and comments are preceded by double slash. A program consists of a collection of class declarations like the following one.

```
class Bool extends Object {
   bool f;                             // private field
   con{ skip }                         // public constructor
   unit set(bool x){ self.f := x }    // public method
   bool get(){ result := self.f }     // public method
}
```

There are two associated methods: set takes a boolean parameter and returns nothing; get takes no parameter and returns a boolean value. Methods are considered to be public, that is, visible to methods in all classes. (Module-scoped methods are added in Sect. 10.) Every method has a return type; the primitive type **unit**, with only a single value (**it**), corresponds to Java's "void" and is used for methods like set that are called only for their effect on state.

Instances of class Bool have a field f of (primitive) type bool. A field f is accessed in an expression of the form e.f, and in particular self.f is used for fields of the current object; a bare identifier like x is either a parameter or a local variable. The distinguished variable result provides the return value; it is initialized with the default for its type (*false* for **bool** and *nil* for class types). Fields are considered to be private, that is, visible only to the methods declared in the class. Visibility is class-based, as in many mainstream object-oriented languages: an object can directly access the private fields of another object of the same class.

When a new object is constructed, each field is initialized with the default value for its type. Then the constructor commands are executed: the constructors declared in superclasses are executed before the declared one which is designated by keyword **con**. We refrain from considering constructors with parameters. In subsequent examples we omit the constructor if it is **skip**.

The observable behavior of a Bool object can be achieved using an alternate implementation in which the complement is stored in a field:

```
class Bool extends Object {
   bool f;
   con{ self.f := true }
   unit set(bool x){ self.f := ¬x }
   bool get(){ result := ¬(self.f) } }
```

We do not formalize class types ("interfaces" in Java) separately from class declarations. Class names are used as types and we use the term *class* loosely to mean the name of a declared class. But we are concerned with relating comparable versions of a class: as in the example above, a comparable version has the same name and methods with the same names and signatures.

We claim that no client program using Bool can distinguish one implementation from the other; thus we are free to replace one by the other. Of course this is not the case if we consider aspects of client behavior such as real time or the size of object code —but these are not at the level of abstraction of source code. Moreover, input and output for end users is of some limited type like **int** or String. If a Bool could be output directly, say displayed in binary on the screen, then an end user could distinguish between the implementations. So we consider only clients that use Bool objects in temporary data structures and not as input or output data.

An example of such a client is method main in the following class. It declares a local variable b of type Bool, with scope beginning at the keyword **in**. In the absence of explicit braces, the scope of a local variable extends to the end of the method body.

```
class Main extends Object {
  String inout;
  unit main(){ Bool b := new Bool in
                 if ...self.inout... then b.set(true) else b.set(false) fi;
                 self.inout := convertToString(b.get()) } }
```

We may consider method main as a main program for which the observable state consists of field inout. Its final value depends on some condition "...self.inout..." on its initial value. No object of type Bool is reachable in the state of a Main object after invocation of main, so there is no observable difference between its behavior using one implementation of Bool and its behavior using the other.

The claim is that we need not consider specific clients; there is no use of Bool that can distinguish between the two implementations. The standard reasoning goes as follows.

(1) Suppose o is an object of type Bool for the first implementation and o′ an object for the second. The correspondence between their states is described by the *basic coupling relation*

$$\mathsf{o.f} = \neg(\mathsf{o}'.\mathsf{f}) \ .$$

(2) This relation has the *simulation property*:
—it holds initially (once the constructor has been executed), and
—if the two versions of set (respectively, get) are executed from related states then the outcomes are related. (As we consider sequential programs, the outcome is the updated heap and the return value if any.)

In short, the relation is established by the constructor and preserved by the methods of Bool.

(3) To consider client programs we must consider program states consisting of local variables (and parameters) along with the heap, which may contain many instances of Bool as well as other objects. For states, we define the *induced coupling relation.* Primitive values and locations are related by equality (later we refine this to a bijection, to account for differences in allocation.) A pair of heaps are related if there is a one-to-one correspondence between Bool objects such that they are pairwise related by the basic coupling of (1), and everything else is related by equality.

The induced coupling relation is preserved by all commands in methods of all classes. This is the *abstraction theorem.*

(4) For a pair of states related by the (induced) coupling, if no Bool objects are reachable then the states are equal. This is the *identity extension lemma*, which follows from the definition of the induced coupling.

It is a consequence of (3) and (4) that the two implementations cannot be distinguished by a client that does not input or output Bool objects. Any initial state for such a client is related to itself, by (4). We can consider an execution of the client using either of the two implementations of Bool; the final states are related, according to (3). And thus they are equal, by (4).

Identity extension confirms that the chosen notion of coupling relation is suited to the chosen form of encapsulation. (Here, encapsulation means private fields and objects, not input or output.) It is typically a straightforward consequence of the definitions.

For program refinement, identity can be replaced by inequality in step (4). In this paper we do not emphasize refinement, but the requisite adaptation of our results is straightforward. For applications in program analysis, other relations are used in step (4), e.g., for secure information flow the relation expresses equivalence from the point of low-security observers [Volpano et al. 1996].[3]

The abstraction theorem is a non-trivial property of the language. It would fail, for example, if the language had constructs that allowed client programs to read the private fields of Bool —or to enumerate the names of the private fields, or to query the number of boolean fields that are currently true. In fact, similar facilities can be found in some reflection libraries and in the implementation of Java's inner classes, but are considered to be design flaws.

Familiar operations on pointers, however, can also violate abstraction. For example, with pointer arithmetic one can distinguish between two representations that differ only in the size of storage used (e.g., representing a boolean value using one bit of an integer versus one bit of a character). Even in the absence of pointer arithmetic, shared references lead to the following problem.

2.2 Representation exposure

Consider the following class OBool which provides functionality similar to that of Bool, in fact using Bool. For clarity we have chosen different method names, to emphasize that we are not comparing this class with Bool.

```
class OBool extends Object {
  Bool g;
  unit init(){ self.g := new Bool; self.g.set(true) }
  unit setg(bool x){ self.g.set(x) }
  bool getg(){ result := self.g.get() } }
```

To simplify the formal development, we sidestep the complicated interactions between subclassing and method calls in constructors by confining attention to con-

[3]Our formulation of the abstraction theorem can be applied directly to prove command and class equivalences for a specific program. For applications of simulation in static analysis, the problem is usually to show that a syntax directed system of types and effects approximates some property like secure information flow, for all programs in a language. We have not attempted to formulate an abstraction theorem general enough to apply directly in such analyses; they use analysis-specific typing systems rather than the language's own types and syntax. But the essence of our result is that the language is relationally parametric, given suitable confinement conditions. Indeed, in work subsequent to this paper, Banerjee and Naumann [2002b] use the same language and semantic model for a relational analysis of secure information flow.

structors without parameters or method calls. In cases where this is inadequate, an ordinary method can be used (like init in this example).

Here is an alternate implementation of OBool.

```
class OBool extends Object {
  Bool g;
  unit init(){ self.g := new Bool; self.g.set(false) }
  unit setg(bool x){ self.g.set(¬ x) }
  bool getg(){ result := ¬(self.g.get()) } }
```

To describe the connection between the two implementations a suitable basic coupling (recall (1) in Sect. 2.1) is the following relation between an object state o for the first implementation of OBool and o′ for the alternate one:

$$(\mathsf{o.g} = \mathit{nil} = \mathsf{o'.g}) \vee (\mathsf{o.g} \neq \mathit{nil} \neq \mathsf{o'.g} \wedge \mathsf{o.g.f} = \neg(\mathsf{o'.g.f})) \quad . \qquad (*)$$

If o and o′ are newly constructed, the first disjunct holds; method init establishes the second disjunct. Invocations of setg and getg maintain the relation: From related initial states, either both abort (due to dereferencing *nil* because init has not been called) or both terminate in related states.

For these implementations, it is not just a private field that is to be encapsulated, but also the object referenced by that field. This is apparent in the coupling (∗) which involves both. To describe the roles of the objects involved, we call class OBool an *owner* class. Its instances "own" objects of class Bool, their representation objects, which are called *reps* for short. Together, an owner and its reps constitute what we call an *island* (cf. Fig. 1), following Hogg [1991].

Here is a suitable client for OBool.

```
class Main extends Object {
  String inout;
  unit main(){ OBool z := new OBool in z.init();
               if ...self.inout... then z.setg(true) else z.setg(false) fi;
               self.inout := convertToString(z.getg()) } }
```

This does not distinguish between the two implementations of OBool nor does it violate the intended encapsulation boundary.

Suppose we add to both versions of OBool the following method which "leaks" a reference to the rep object.

```
Bool bad(){ result := self.g }
```

The method gives its caller an alias to the object pointed to by the private field g. This makes the location of the encapsulated object visible to clients. In and of itself, access to this location is not harmful.[4] Like the other methods, method bad preserves (∗). But a client class C can exploit the leak as in the following command.

[4]To make this clear, one could assume that, for both versions of OBool, the Bool object is allocated at the same location. The assumption can be formalized by adding a conjunct $\mathsf{o.g} = \mathsf{o'.g}$ to coupling (∗) and assuming that method init preserves this equality. It is then preserved by all the methods of OBool including bad. Another justification is given in Sect. 10 where we show formally how the language is "parametric in locations".

$\mathsf{OBool\ z := }\ \mathbf{new}\ \mathsf{OBool}\ \mathbf{in}\ \mathsf{z.init();}$
$\mathsf{Bool\ w := z.bad()}\ \mathbf{in\ if}\ \mathsf{w.get()}\ \mathbf{then\ skip\ else\ abort\ fi}$

The command aborts if the new OBool is an object o′ for the second implementation of OBool, but it does not abort for an object o for the first implementation. An attempt to argue using the steps in Sect. 2.1 breaks down because this difference in behavior violates the abstraction theorem, step (3).

Identity extension, step (4), also fails. The relation is not the identity for the rep object states, because we have $\mathsf{o.g} = \mathsf{o'.g}$ yet o.g.f is not equal to o′.g.f. So the relation is not the identity for the client to which the reps are visible.

The client in the example above does preserve the relation (∗), up to the point where the program does or does not diverge, because it does not alter the state of the objects it accesses. For an example where the abstraction theorem, step (3), fails with terminating computations, consider the following client command.

$\mathsf{OBool\ z := }\ \mathbf{new}\ \mathsf{OBool}\ \mathbf{in}\ \mathsf{z.init();\ Bool\ w := z.bad()}\ \mathbf{in}\ \mathsf{w.set(true)}$

This does not preserve (∗). To see why, suppose o, o′ are a related pair of OBool objects assigned to z and satisfying (∗). After the assignment to w, the effect of w.set(true) is to make $\mathsf{o.g.f} = \mathsf{o'.g.f}$, contrary to the relation (∗). This is very different from the effect of z.setg(true).

The examples show that both ingredients of representation independence — identity extension and preservation— can fail if a rep is leaked. The challenge is to confine pointers in a way that disallows harmful leaks and thus admits a robust representation independence property —without imposing impractical restrictions. The challenge is made more difficult by various features of Java-like languages, for example, type casts. We consider casts now; other challenges are deferred to Sect. 3.

Suppose we change the return type for method bad, attempting to hide the type of the rep object.

$\mathbf{Object}\ \mathsf{bad()\{\ result := self.g\ \}}$

Class **Object** is the root of the subclassing hierarchy so by subsumption it allows references to objects of any class. The client can use a (Bool) cast to assert that the result of z.bad() has type Bool. (In a state where the assertion is false, the cast would cause abortion.)

$\mathsf{OBool\ z := }\ \mathbf{new}\ \mathsf{OBool}\ \mathbf{in}\ \mathsf{z.init();}$
$\mathsf{Bool\ w := z.bad()}\ \mathbf{in\ if}\ \mathsf{w.get()}\ \mathbf{then\ skip\ else\ abort\ fi}$

Again, the client is dependent on representation.

Note that the cast could not be used if the scope of class name Bool did not include the client. This suggests a focus on modules ("packages" in Java) for confinement of pointers, as has been studied by Vitek and Bokowski [2001] among others (see Sect. 12). But in our example the field has private scope, each rep is associated with a single owner, and the coupling relation is expressed in terms of a single owner. Our results account for this sort of instance-based encapsulation. Instance-based encapsulation facilitates more local or modular reasoning. It is suited to many common design patterns, as we illustrate in the sequel, and it is similar to the value-oriented notions used for representation independence in functional languages [Reynolds 1984; Mitchell 1986; 1991].

2.3 Overview of results

In the examples above, class OBool is viewed as providing an abstraction. It is just as sensible to consider Bool as providing an abstraction for which OBool is a client. We do not annotate programs with a fixed designation of owners and reps. Rather, we study how to reason about a class, say *Own*, one has chosen to view as an abstraction with encapsulated representation. Objects of any subclass of *Own* are also considered to be owners. A second class, say *Rep*, is designated as the type of reps for *Own*. In practice, *Rep* could be an interface or class type, and there could be multiple *Rep* classes; these generalizations are straightforward but would complicate the formalization.

A complete program is a closed collection of class declarations, called a *class table*. We consider an idealized Java-like language similar to the sequential fragment of C++ (without pointer arithmetic), Modula-3, Oberon, C#, Eiffel, and other class-based languages. It includes subclassing and dynamic dispatch, class oriented visibility control, recursive types and methods, type casts and tests (Java's `instanceof`), and a simple form of module.

Roughly speaking, a class table CT is *confined*, for *Own* and *Rep*, if all of its methods preserve confinement. A confined heap is one where the objects can be partitioned into some owner islands (recall Fig. 1) along with a block of client objects as in Fig. 5. Furthermore, there are no references from clients to reps. (We use the term *client* for all objects except owners and reps.)

Sect. 3 discusses confinement in more detail and the formal definitions are the subject of Sect. 6. The full significance of the definitions does not become clear until Sect. 9 where we study subclasses of *Own*: an object of such a type inherits the methods and private fields of *Own*, which manipulate reps. To be useful, owner subclasses must have some access to reps. On the other hand, full access cannot be granted; to do so would be to study not the class as unit of encapsulation but a class together with its subclasses, which would be revised in concert.

Our objective is to compare versions of *Own* that may use different reps. We say CT and CT' are *comparable* if they are identical except for having different versions of class *Own*, and those two versions declare the same public methods. The two versions of *Own* may well use different rep classes, say *Rep* and *Rep*$'$. Without loss of generality, our formalization has both *Rep* and *Rep*$'$ present in CT and in CT'.

An interesting question is how to formalize basic couplings, step (1) of the proof method outlined in Sect. 2.1. To allow useful data structures, we need to allow representations to include pointers to client objects (e.g., elements of the FIFO queue in Fig. 1). But if the programmer is required to define a relation involving the state of objects outside the encapsulated data, how can this be done in a modular way? We have chosen to use relations on the encapsulated state only. Put differently: those things on which a coupling depends are considered as part of the island. Although other alternatives merit study, this one makes for transparent application of the formal results to interesting examples (this is done in Sects. 8 and 9). Moreover, it is straightforward to define the induced coupling.

A *basic coupling* is a relation between a pair of owner islands for comparable CT and CT'. A simple example is given by $(*)$ above in Sect. 2.2. More interesting is the observer example, discussed in Sect. 3, which uses a linked list of client objects (the observers). In Fig. 7 on page 44, a basic coupling is depicted in which the observer objects occur as dangling pointers from the corresponding islands. The point is that both versions are manipulating the same observer objects in the same way, including the invocation of methods on those objects. So the state of the observer objects is not relevant in the basic coupling —nor could it be, if the argument is to be carried out in a modular way independent of the particular clients.

In a related pair of islands, both owners have the same class, which may well be a proper subclass of Own.

The induced *coupling relation* for heaps relates h to h' just if there are confining partitions for which corresponding islands are pairwise related by the basic coupling. Moreover, there is an exact correspondence between client objects in h and h'. Primitive values are related by equality. Locations are related by an arbitrary bijective renaming, which is needed to account for differences in allocation behavior.

The induced relation is a *simulation* if it is preserved by the methods of class Own in CT and in CT'. A method declared in one version of Own may be inherited in the other version; it is the behavior of those methods that matters.

The abstraction theorem says that a simulation is preserved by all methods of all classes, provided that both class tables are confined. The identity extension lemma says that the induced relation is the identity, after garbage collection, for client states in which no owners are reachable.

Sect. 7 gives the formal definitions for coupling and simulation in the special case where locations of objects other than reps are related by equality. The abstraction and identity extension results are proved there in detail. Sect. 10 generalizes the definitions to allow an arbitrary bijection on locations; abstraction and identity extension are proved for the general case. The special case is of interest because it is adequate for some applications in program analysis (e.g., [Banerjee and Naumann 2002b]) and for non-trivial examples like those of Sect. 3 (as shown in Sect. 8). Examples that require the general case are given in Sect. 9; they are subclasses of Own that construct reps and pass them to methods of Own as in the factory pattern [Gamma et al. 1995]. Notation is more complicated for the general case but the proofs are not very different from the special case.

These results are proved in terms of a semantic formulation of confinement; indeed, the details of this formulation come directly from what is needed in the proofs. Sect. 11 gives a syntax-directed static analysis: typing rules that characterize *safe* programs and a proof that safety implies confinement (soundness). Our objective is to round out the story by showing how confinement can be achieved in practice, not to give a definitive treatment of static analyses. But our analysis accepts many natural examples and the constraints are clearly motivated in the proof of soundness. The analysis is modular: It does not require code annotations and the only constraint it imposes on client programs is that they cannot manufacture representation objects.

3. OWNERSHIP CONFINEMENT

This section considers two examples of representation independence. The first is an object-oriented version of an example given by Meyer and Sieber [1988] as a challenge for semantics of Algol. It illustrates the expressiveness of object-oriented constructs, specifically the use of *callbacks* which go against the hierarchical calling structure which typifies the simplest forms of procedural and data abstraction.

The second example is an instance of the observer pattern [Gamma et al. 1995] which is widely used in object-oriented programs. In addition to callbacks it involves a non-trivial data structure and outgoing references from representation objects to clients. Note that we use the term *client* not just for objects that use an abstraction (by instantiating it or calling its methods) but for any objects except instances of the abstraction of interest or its encapsulated representation.

The section concludes with an overview of our semantic notion of confinement.

3.1 Callbacks

Meyer and Sieber [1988] consider the following pair of Algol commands:

$$\mathbf{var}\ \mathsf{n} := 0;\ \mathsf{P}(\mathsf{n} := \mathsf{n}{+}2);\ \mathbf{if}\ \mathsf{n}\ \mathbf{mod}\ 2 = 0\ \mathbf{then\ abort\ else\ skip\ fi} \qquad (*)$$

$$\mathbf{var}\ \mathsf{n} := 0;\ \mathsf{P}(\mathsf{n} := \mathsf{n}{+}2);\ \mathbf{abort} \qquad (\dagger)$$

Both invoke some procedure P, passing to it the command n := n+2 that acts on local variable n. (That is, P is passed a parameterless procedure whose calls have the effect n := n+2.) For any P, the commands are equivalent: both abort. The reason is that in the first example n is invariably even: P is declared somewhere not in the scope of n so the variable can only be affected by (possibly repeated) executions of n := n+2 and this maintains the invariant.

The difficulty in formalizing this argument is due to the difficulty of capturing the semantics of lexically scoped local variables and procedures in a language where local variables can be free in procedures that can be passed as arguments to other procedures. A formalization based on operational It appears even more difficult, and remains an open problem, to cope with assignment of such procedures to variables (see Sect. 12.1).

Now we consider a Java-like adaptation of the example, due to Peter O'Hearn. In place of local variable n it uses a private field g in a class A. Instead of passing the command n := n+2 as argument, an A-object passes a reference to itself; this gives access to a public method inc that adds 2 to the field.

```
class A extends Object {
  int g; // (the default integer value is 0)
  unit callP(C y){ y.P(self); if self.g mod 2 = 0 then abort else skip fi }
  unit inc(){ self.g := self.g + 2 } }
```

In the context of this class and some declaration of class C with method P, the Algol command (∗) corresponds to the command

$$\mathsf{C\ y} := \mathbf{new}\ \mathsf{C}\ \mathbf{in}\ \mathsf{A\ x} := \mathbf{new}\ \mathsf{A}\ \mathbf{in}\ \mathsf{x.callP(y)} \qquad (\ddagger)$$

This aborts because after calling y.P, method callP aborts. The command (†) also corresponds to (‡) but in the context of an alternative implementation of class A:

```
class A extends Object {
  int g;
  unit callP(C y){ y.P(self); abort }
  unit inc(){ self.g := self.g + 2 } }
```

In Example 8.3, we use the abstraction theorem to prove equivalence of the two versions using coupling relation

$$\mathsf{o.g} = \mathsf{o'.g} \wedge \mathsf{o.g}\ \mathbf{mod}\ 2 = 0\ .$$

This relation is preserved by arbitrary P because P can affect the private field g only by calls to inc.

As Reynolds [1978] shows (see also [Reddy 1998]), instance-based object-oriented constructs can be expressed in Algol-like languages, but the latter are in some ways significantly more powerful. The Java version of the example can be seen as giving an explicit closure to represent the command $\mathsf{n} := \mathsf{n}{+}2$ in the form of method inc. Indeed the simplicity of the semantic model for our language can be explained by saying the language is defunctionalized [Reynolds 1972; Banerjee et al. 2001] and lacks true higher order constructs. If the example is written in such a language, P ranges over more limited procedures than in Algol. The root problem for Algol semantics [Reynolds 1981b; O'Hearn and Tennent 1995] and proof rules [Olderog 1983; German et al. 1989] is the interaction between arbitrary nesting of variable and procedure declarations and possibility of passing procedures as arguments. In imperative languages like C and Modula-3, procedures can be passed as arguments and even stored in variables, but only if their free variables are in outermost scope. This restriction greatly simplifies implementation of the language, and it suffices to admit simple but adequate semantic models.[5] The constructs of a Java-like language offer similar expressive power and also admit simple models.

The example also illustrates what are known as *callbacks* in object-oriented programs. When an A-object invokes y.P(self) it passes a reference to itself, by which y may invoke a method on the A-object which is in the middle of executing method callP —a callback to A. If in (‡) we replace x.callP(y) by x.callP(self), and assume that (‡) is a constituent of a method of class C, then we get a callback to C.

The point of the Algol example is modular reasoning about (∗) and (†) independent from the definition of P. For the object-oriented version we can also consider reasoning independent from subclasses of A. If instead of (‡) we consider a method

```
    unit m(C y, A x){ x.callP(y) }
```

then there is the possibility that m is passed an argument x of some subtype of A that overrides inc. By dynamic binding, the overriding implementation would be invoked by callP and our reasoning above would no longer be sound. For modular reasoning, we could require that any overriding declaration of inc must preserve the intended invariant that g is even. To impose such a requirement —and a corresponding one for callP— is to require behavioral subclassing [Liskov and Wing 1994; Dhara and Leavens 1996]. One important application of simulations is in the formalization of behavioral subclassing but that is beyond the scope of this paper.

Unlike much work on reasoning about object-oriented programs, our results do not depend on behavioral subclassing. Representation independence holds for clients and abstractions that do not exhibit behavioral subclassing (see Sect. 9.2).

[5]Naumann [2002] uses such a model to prove an abstraction theorem and apply it to Meyer-Sieber examples. The simpler of their examples can be proved directly in the model without use of simulations [Naumann 2001].

```
class Observer extends Object { // "abstract class" to be overridden in clients
  unit notify(){ abort } }

class Node extends Object { // rep for Observable
  Observer ob;
  Node nxt; // next node in list
  unit setOb(Observer o){ self.ob := o }
  unit setNext(Node n){ self.nxt := n }
  Observer getOb(){ result := self.ob }
  Node getNext(){ result := self.nxt } }

class Observable extends Object { // owner
  Node fst; // first node in list
  unit add(Observer ob){ Node n := new Node; n.setOb(ob); n.setNext(self.fst); self.fst := n }
  unit notifyAll(){ Node n := self.fst; while n ≠ null do n.getOb().notify(); n := n.getNext() od } }
```

Fig. 2. First version of observer pattern, in procedural style.

3.2 The observer pattern

In this subsection we consider variations on an often-used design known as the observer pattern [Gamma et al. 1995] which involves a non-trivial recursive data structure using multiple rep objects and outgoing references to client objects. Further variations are given in Sect. 8.

We focus attention on the abstraction provided by an Observable object (sometimes called the "subject"). It maintains a list of so-called observers to be notified when some event occurs. Its public method add allows the addition of an observer object to the list. The public method notifyAll represents the event of interest; its effect is to invoke method notify on each observer in the list. What notify does is not relevant, so long as it is confined.[6]

The abstraction involves a collection of objects, a well-worn example for data representations. Simple collections are essentially mutable sets of pointers to client objects. Testing whether a reference is in the set requires only pointer equality. To facilitate lookup by key, and to facilitate implementations like binary search trees, it may be necessary for the abstraction to invoke a comparison method on the client objects in the collection. This is similar to the call to notify in the observer pattern.

In the first version of the observer example, Fig. 2, most of the work is done by the owner class Observable, which uses rep class Node to store observers in a singly linked list. A more object-oriented version appears in Fig. 8 of Sect. 8; it exemplifies the use of class-based visibility.

Fig. 3 gives example client classes AnObserver and Main. Class AnObserver records notifications in its state. Method main constructs and initializes an Observable, installs an observer, and invokes notifyAll; upon termination, ob.count $= 1$ and no Observable is reachable.

Fig. 4 gives another version of Observable, using a sentinel node [Cormen et al. 1990], for the sake of an example. A more compelling use of sentinels is the version

[6]In Java, class `Object` declares methods notify and notifyAll. Here we assume that no superclass of Observer declares notify and no superclass of Observable declares notifyAll. In the Java versions of our examples we use different names.

```
class AnObserver extends Observer {
    int count;
    unit notify(){ self.count := self.count+1 } }

class Main extends Object {
    AnObserver ob;
    unit main(){
        ob := new AnObserver; Observable obl := new Observable; obl.add(ob); obl.notifyAll() } }
```

Fig. 3. Example client for Observable.

```
class Node2 extends Object { // rep for Observable
    Observer ob;
    Node2 nxt;
    unit setOb(Observer o){ self.ob := o }
    unit setNext(Node2 n){ self.nxt := n }
    Observer getOb(){ result := self.ob }
    Node2 getNext(){ result := self.nxt } }
class Observable extends Object // owner {
    Node2 snt; // sentinel node pointing to list
    con{ self.snt := new Node2 }
    unit add(Observer ob){
        Node2 n := new Node2; n.setOb(ob); n.setNext(self.snt.getNext()); self.snt.setNext(n); }
    unit notifyAll(){
        Node2 n := self.snt.getNext(); while n ≠ null do n.getOb().notify(); n := n.getNext() od } }
```

Fig. 4. Version of observable that uses sentinel node, in procedural style

of Fig. 9 (in Sect. 8), which also uses subclassing and dynamic dispatch.

In Sect. 8 we show equivalence of the versions of Figs. 2 and 4 as an application of the abstraction theorem and identity extension. The coupling relation describes the correspondence between a pair of lists, one with and one without a sentinel node (see Fig. 7). It is enough to say that the same Observer locations are stored in the lists, in the same order. The state of the Observer is not relevant —nor could it be in a modular treatment, as class Observer has no fields. To reason about outgoing calls, namely to notify, it is enough to show that the two implementations make the same calls. Those calls may lead to calls back to the Observable, but encapsulation ensures that those calls are the only way the behavior of notify can depend on, or affect, the Observable.

Except for the bad method of Sect. 2.1, all of the examples discussed so far satisfy the confinement conditions discussed next.

3.3 Confinement

We need a notion of confinement to prevent representation exposures that invalidate simulation-based reasoning, as discussed in Sect. 2.1. A related issue is how to formulate simulation. In all the examples, our discussion centered on a corresponding pair of instances for two implementations of the owner class. In particular, the coupling relations are described for a pair of instances as discussed in Sect. 2.3. A class- or module-based notion of confinement might rule out leaks, but we aim for an instance-based notion of simulation suited to the kind of examples we have

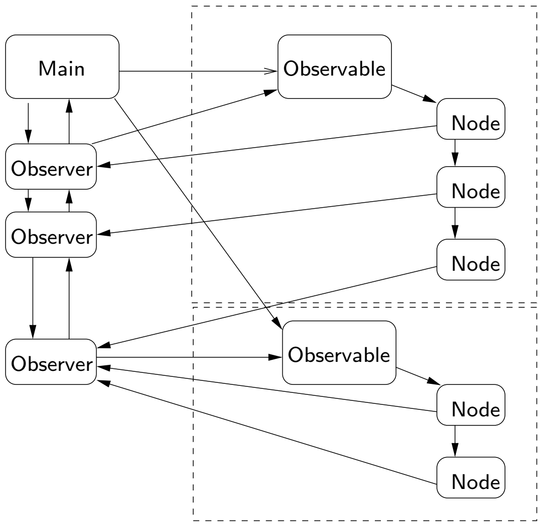

Fig. 5. Confinement example. Rounded boxes are instances of the indicated class. Solid arrows represent allowed pointers. Dashed boxes indicate owner islands, each consisting of one owner and its reps.

discussed. These involve an abstraction provided by a single instance (the owner object) using a representation accessed via its private fields. So we need to prevent problematic sharing not only between client and owner but also between different instances of the owner class.

Fig. 5 illustrates instance-based owner confinement; in this case Nodes are confined to their owning Observable. Following Hogg [1991], we use the term *island* for the sub-heap consisting of an owner and its reps. Dashed lines in the Figure depict two islands. Our notion of owner confinement imposes four conditions on islands; here are the first three:

(1) there are no references from a client object to a rep;
(2) there are no references from an owner to reps in a different island;
(3) there are no references from a rep into a different island.

The Figure exhibits most allowed references, but we also allow an owner to reference another owner (see Fig. 6 on page 33). An example is given in Sect. 9.1. Note that heap confinement is a state predicate. The full definition, formalized in Sect. 6, deals with preservation of this predicate by commands and also with leaks via parameter passing in outgoing method calls from island to client.

In class-based languages with inheritance, there is a subclass (or "protected") interface in addition to the public one. This raises the possibility of expressing encapsulation of reps for not only (instances of) the owner class but also its subclasses. We have chosen the alternative that subclasses are like clients in that fields they declare may *not* point to reps. To the list of conditions above we add:

(4) references from an owner's fields to its reps are only in the private fields of the owner class.

In order not to abandon the expressiveness of subclassing, however, we allow subclass methods to manipulate reps: they may be constructed, stored in local variables, and passed to the owner. This fits well with the factory pattern [Gamma et al. 1995] which allows owner behavior to be adapted in owner subclasses without violating encapsulation. To balance the paper, we have deferred the relevant examples to Sect. 8.

Confinement is formulated using class names. Two incomparable class names, *Own* and *Rep*, are designated. An object is considered to be an owner (respectively, a rep) if its type is *Own* (resp. *Rep*) or a subtype thereof. Incomparability is a mild restriction that enforces a widely-followed discipline of distinguishing between rep objects (e.g., nodes in a linked list) and objects representing abstractions (e.g., a list). The technical benefit of incomparability is that if C and D are incomparable, which we write $C \not\lesseqgtr D$, then an expression of type C never has a value of type D.

We aim for clear separation between the semantic property needed for the abstraction theorem —restrictions on the heap as described above— and the syntactic conditions used by the static analysis to enforce the confinement property. For perspicuity, the separation is not absolute: the "semantic" property includes conditions on method signatures. For example, we impose the restriction that the return type of a public owner method is incomparable to *Rep*.

Our use of types to formulate alias restrictions allows heterogeneous data structures, but is slightly restrictive in that there is a single common superclass for all reps. For more flexibility in practical applications, our theory could be adapted by taking *Own* and *Rep* to be "class types" ("interfaces" in Java), rather than class implementations, and also by allowing multiple *Rep* types. The generalization is straightforward and not illuminating.

The more substantial restriction is due to the fact that class **Object** is comparable to all classes. Because Java lacks parametric polymorphism, **Object** is often used to express generics, e.g., a list containing elements of arbitrary type. A method to enumerate the list would have return type **Object**, which violates our restriction on owner methods. This restriction could be dropped in favor of more sophisticated conditions to ensure that no rep is returned (see Sect. 12). But in practice many generics have some sort of constraint expressed by a class or interface type —like Observer in our examples, or Comparable for data structures that depend on an ordering. These do not run afoul of our restriction. In any case, the use of **Object** for generics is widely deplored because it undercuts the benefits of typing; parametric types are clearly preferable.

Some works on confinement have considered all the confinement properties intended to be satisfied by a program, using hierarchical notions of ownership [Clarke et al. 2001; Müller 2002]. For example, a Set could own the header of a list which in turn owns the nodes of the list. This is not necessary for our purposes (see Sect. 12). To analyse the abstraction provided by the set, we would consider both the header and nodes to be reps, with a common superclass *Rep*. On the other hand, to replace one header implementation by another, Set is irrelevant; we choose *Own* to be the header and *Rep* for the nodes.

4. SYNTAX

This section formalizes the language, for which purpose we adapt some notations from Featherweight Java [Igarashi et al. 2001].[7] To avoid burdening the reader with

[7]But the languages differ, e.g., ours has imperative features and private fields.

straightforward technicalities we deliberately confuse surface syntax with abstract syntax. We do not distinguish between classes and class types. We confuse syntactic categories with names of their typical elements. Barred identifiers like $\overline{T}$ indicate finite lists, e.g., $\overline{T}\,\overline{f}$ stands for a list $\overline{f}$ of field names with corresponding types $\overline{T}$. The bar has no semantic import; $\overline{T}$ has nothing to do with T.

The grammar is based on given sets of class names (with typical element C and including at least **Object**), field names (f), method names (m), and names (x) for parameters and local variables. In most respects self and result are like any other variables but self cannot be the target of assignment.

Grammar

T	::=	$\mathbf{bool} \mid \mathbf{unit} \mid C$	data type
CL	::=	$\mathbf{class}\ C\ \mathbf{extends}\ C\ \{\ \overline{T}\ \overline{f};\ \mathbf{con}\{\ S\ \}\ \overline{M}\ \}$	class declaration
M	::=	$T\ m(\overline{T}\ \overline{x})\ \{S\}$	method declaration
S	::=	$x := e \mid e.f := e$	assign to variable, to field
	$\mid$	$x := \mathbf{new}\ C$	object construction
	$\mid$	$x := e.m(\overline{e}) \mid x := \mathbf{super}.m(\overline{e})$	method calls
	$\mid$	$T\ x := e\ \mathbf{in}\ S$	local variable block
	$\mid$	$\mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S\ \mathbf{fi} \mid S;\ S$	conditional, sequence
e	::=	$x \mid \mathbf{null} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{it}$	variable, constant
	$\mid$	$e.f \mid e = e$	field access, equality test
	$\mid$	$e\ \mathbf{is}\ C \mid (C)\ e$	type test, cast

Class **Object** has no fields, no methods, and no proper superclass. Additional primitive types, such as integers, can be treated in the same way as **bool** and **unit** (integers can also be represented, e.g., in unary using linked lists).

In the formal language, expressions do not have side effects. Object construction, **new**, occurs only as a command $x := \mathbf{new}\ C$ that assigns to a local variable. Method calls are not expressions but rather occur in special assignments $x := e.m(\overline{e})$ to allow both heap effects and a return value.

Remark* 4.1 (*syntactic sugar) In examples we use several abbreviations:

—A method call command $e.m(\overline{e})$, e.g., self.g.set(true), abbreviates a call assigning to an otherwise unused local variable.

—Assignment of a new object to a field abbreviates a local block assigning the new object to a variable that is then assigned to the field.

—Object construction in local variable initialization abbreviates initialization to **null** followed by object construction.

—Methods that return values but do not mutate state are used in expressions, e.g., the argument in self.inout := convertToString(z.getg()) and the target object in n.getOb().notify(). These are easily desugared using fresh variables and suitable assignments.

—**skip** abbreviates some no-op assignment $x := x$.

As the language has general recursion, we omit loops. For desugaring loops it would be convenient to have local or private method declarations, but the module-scoped methods added in Sect. 10 suffice. The issue is discussed in Sect. 8.1. □

A program is given as a *class table* CT, a finite partial function sending class name C to its declaration $CT(C)$ which may make mutually recursive references to other classes. Well formed class tables are characterized using typing rules which are expressed using some auxiliary functions that in turn depend on the class table, as is needed to allow mutual recursion. Consider a declaration

$$CT(C) = \textbf{class } C \textbf{ extends } D \ \{ \ \overline{T}_1 \ \overline{f}; \ \textbf{con}\{ \ S_1 \ \} \ \overline{M} \ \} \ .$$

To refer to the constructor, we define $\mathit{constr}\, C = S_1$. For the direct superclass of C, we define $\mathit{super}\, C = D$. Let M be in the list $\overline{M}$ of method declarations, with

$$M = T \ m(\overline{T}_2 \ \overline{x}) \ \{S_2\} \ .$$

We record the typing information by defining $\mathit{mtype}(m, C) = \overline{T}_2 {\rightarrow} T$. (Note that $\overline{T}_2 {\rightarrow} T$ is not a data type in the language.) For the parameter names we define $\mathit{pars}(m, C) = \overline{x}$. If m has no declaration in $CT(C)$ but $\mathit{mtype}(m, D)$ is defined then m is an inherited method, for which we define $\mathit{mtype}(m, C) = \mathit{mtype}(m, D)$ and $\mathit{pars}(m, C) = \mathit{pars}(m, D)$. For the declared fields, we define $\mathit{type}(\overline{f}, C) = \overline{T}_1$ and $\mathit{dfields}\, C = (\overline{f} : \overline{T}_1)$. Here $\overline{f} : \overline{T}_1$ denotes a finite mapping of field names to types. To include inherited fields, we define $\mathit{fields}\, C = \mathit{dfields}\, C \cup \mathit{fields}\, D$ and assume $\overline{f}$ is disjoint from the names in $\mathit{fields}\, D$. The distinguished class **Object** has no methods, $\mathit{fields}(\textbf{Object})$ is the empty list, and $\mathit{super}(\textbf{Object})$ is undefined.

A *typing context* Γ is a finite mapping from variable and parameter names to data types, such that $\mathsf{self} \in \mathit{dom}\,\Gamma$. Whereas the Java format T x is used in code to give x type T, it is written x:T in typing contexts. Typing of commands for methods declared in class C is expressed using judgements $\Gamma \vdash S$ where $\Gamma\,\mathsf{self} = C$. Moreover, if $\mathit{mtype}(m, C) = \overline{T} {\rightarrow} T$ and $\mathit{pars}(m, C) = \overline{x}$ then $\Gamma\,\overline{x} = \overline{T}$ and $\Gamma\,\mathsf{result} = T$.[8] We sometimes say "command" rather than the more precise "command in context" to refer to a derivable judgement $\Gamma \vdash S$. The judgement $\Gamma \vdash e : T$ says that expression e has type T. The constructor is typed using a judgement $\mathsf{self} : C \vdash S : \textbf{con}$ which is distinguished from the typing of S as a command, as the former is used to define the semantics of S as a constructor, which in turn is used in the semantics of object construction (**new**).

***Definition* 4.2 (subtyping , $\leq$)** The class table determines a subtyping relation $\leq$, where $T \leq U$ means T is a subtype of U, as follows. If T or U is **bool** or **unit** then define $T \leq U$ iff $T = U$. For class types C and D, define $C \leq D$ iff either $C = D$ or $\mathit{super}\, C \leq D$. □

The definition of well formed class table, in the sequel, requires that $\leq$ is acyclic and as a consequence we have $C \leq \textbf{Object}$ for all C.

Subsumption is built into the rules for specific constructs. For example, the assignment rule allows $x : D, y : E, \mathsf{self} : C \vdash x := y$ provided that $E \leq D$.

[8]In [Banerjee and Naumann 2002a] we make C an explicit, and redundant, part of the judgement, and we use separate return statements rather than variable result.

The constructor for one class may construct objects of other classes (Fig. 4 is an example). But for simplicity we disallow cyclic constructor dependencies as in the following.

$$\begin{array}{l}\textbf{class } B \textbf{ extends Object } \{\ B\ f;\ \textbf{con}\{\ \mathsf{self}.f := \textbf{new } C\}\ \} \\ \textbf{class } C \textbf{ extends } B\ \{\ \textbf{con}\{\ \textbf{skip}\ \}\ \}\end{array}$$

(Recall that to initialize a C object both the B- and C-constructor are applied.)

***Definition* 4.3 (constructor dependence, $\sqsubset$)** For B, C ranging over declared classes, we say that C has constructor dependence on B, written $B \sqsubset C$, iff $B \sqsubset (\mathit{super}\, C)$ or $x := \textbf{new } B$ occurs in $\mathit{constr}\, C$, for some x. □

Note that $B \sqsubset C$ just if the constructor of C or one of its ancestor classes contains **new** B (by which we mean $x := \textbf{new } B$ for some x). Thus, writing $\sqsubset^+$ for the transitive closure, we have $B \sqsubset^+ C$ just if construction of a C-object entails construction of a B-object. For the example above we have $C \sqsubset B$ and $C \sqsubset C$.

***Definition* 4.4 (well formed class table)** A class table is well formed provided it satisfies the following conditions.

—Each class declaration $\textbf{class } C \textbf{ extends } D\ \{\ \overline{T}\ \overline{f};\ \textbf{con}\{\ S\ \}\ \overline{M}\ \}$ is well formed, that is, each method declaration M in $\overline{M}$ is well formed, and $\mathsf{self}\!:\!C \vdash S : \textbf{con}$, according to the rules to follow.

—If C occurs as the type of a field, parameter, or local variable in some class then $CT(C)$ is defined. No field or method has multiple declarations in a class.

—The subclass relation $\leq$ is antisymmetric.

—Transitive constructor dependence, $\sqsubset^+$, is irreflexive (hence antisymmetric). □

The rules are straightforward renderings of the typing rules for Java, for private fields, public methods and public classes [Arnold and Gosling 1998].

Typing of constructors

$$\frac{S = \mathit{constr}\, C \quad \mathsf{self}\!:\!C \vdash S \quad \text{no method calls occur in } S}{\mathsf{self}\!:\!C \vdash S : \textbf{con}}$$

Typing of method declarations

$$\frac{\begin{array}{c}\overline{x}:\overline{T}, \mathsf{self}\!:\!C, \mathsf{result}\!:\!T \vdash S \\ \mathit{mtype}(m, \mathit{super}\, C) \text{ is undefined or equals } \overline{T}\!\rightarrow\! T \\ \mathit{pars}(m, \mathit{super}\, C) \text{ is undefined or equals } \overline{x}\end{array}}{C \vdash T\ m(\overline{T}\ \overline{x})\{S\}}$$

In this method rule, the condition on *mtype* is the standard invariance restriction on method types, as in Java [Arnold and Gosling 1998; Abadi and Cardelli 1996]. The last antecedent in the rule, concerning $\mathit{pars}(m, D)$, ensures that all declarations of a method use the same parameter names. This loses no generality and slightly streamlines the formalization of the semantic domains in the sequel.

Typing of expressions

$$\Gamma \vdash x : \Gamma x \quad \Gamma \vdash \mathbf{null} : B \quad \Gamma \vdash \mathbf{it} : \mathbf{unit} \quad \Gamma \vdash \mathbf{true} : \mathbf{bool} \quad \Gamma \vdash \mathbf{false} : \mathbf{bool}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 = e_2 : \mathbf{bool}} \qquad \frac{\Gamma \vdash e : (\Gamma\, \mathsf{self}) \quad (f : T) \in \mathit{dfields}(\Gamma\, \mathsf{self})}{\Gamma \vdash e.f : T}$$

$$\frac{\Gamma \vdash e : D \quad B \leq D}{\Gamma \vdash (B)\ e : B} \qquad \frac{\Gamma \vdash e : D \quad B \leq D}{\Gamma \vdash e\ \mathbf{is}\ B : \mathbf{bool}}$$

The rule for equality test allows comparison of arbitrary data types, and is reference equality in the case of class types. But if e_1 and e_2 have types not related by $\leq$, the test $e_1 = e_2$ is false except when both are null. The rule for field access enforces private visibility: only a method declaration in class C can access fields declared in $CT(C)$. It can access those fields on any object of its type; to access its own fields the expression is $\mathsf{self}.f$. The rule for cast is standard.[9]

Typing of commands

$$\frac{\Gamma \vdash e : T \quad T \leq \Gamma\, x \quad x \neq \mathsf{self}}{\Gamma \vdash x := e} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : (\Gamma\, \mathsf{self}) \quad (f : T) \in \mathit{dfields}(\Gamma\, \mathsf{self}) \\ \Gamma \vdash e_2 : U \quad U \leq T\end{array}}{\Gamma \vdash e_1.f := e_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : D \quad \mathit{mtype}(m, D) = \overline{T} \to T \\ \Gamma \vdash \overline{e} : \overline{U} \quad \overline{U} \leq \overline{T} \quad x \neq \mathsf{self} \quad T \leq \Gamma\, x\end{array}}{\Gamma \vdash x := e.m(\overline{e})} \qquad \frac{\begin{array}{c}\mathit{mtype}(m, \mathit{super}(\Gamma\, \mathsf{self})) = \overline{T} \to T \\ \Gamma \vdash \overline{e} : \overline{U} \quad \overline{U} \leq \overline{T} \quad x \neq \mathsf{self} \quad T \leq \Gamma\, x\end{array}}{\Gamma \vdash x := \mathbf{super}.m(\overline{e})}$$

$$\frac{B \leq \Gamma x \quad x \neq \mathsf{self} \quad B \neq \mathbf{Object}}{\Gamma \vdash x := \mathbf{new}\ B} \qquad \frac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1;\ S_2}$$

$$\frac{\Gamma \vdash e : \mathbf{bool} \quad \Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}} \qquad \frac{\begin{array}{c}x \neq \mathsf{self} \quad x \notin \mathit{dom}\,\Gamma \\ \Gamma \vdash e : U \quad U \leq T \quad (\Gamma, x : T) \vdash S\end{array}}{\Gamma \vdash T\ x := e\ \mathbf{in}\ S}$$

In some of the command rules, the hypothesis involves a partial functions which must be defined for the hypothesis to be satisfied. For example, in the rule for super calls, $\mathit{mtype}(m, \mathit{super}\, C)$ must be defined and equal to $\overline{T} \to T$.

[9]It is not adequate for expressions that arise through substitutions used in program logic (see Cavalcanti and Naumann [1999]) and in small-step semantics (see Igarashi et al. [2001]); the latter source uses the term "stupid cast" for the typing rule that allows $(B)\ e$ when B is not a subclass of the static type of e.

Each expression and command construct is the conclusion of exactly one typing rule, and there are no other rules. Thus we have the following.[10]

Lemma 4.5 A typing $\Gamma \vdash S$ or $\Gamma \vdash e : T$ has at most one derivation. □

***Definition* 4.6 (inheritance)** Method m is *inherited in* C *from* B if $C \leq B$, there is a declaration for m in B, and there is no declaration for m in any D such that $C \leq D < B$. To make the class table explicit, we also say m is inherited from B in $CT(C)$. □

Because the language has single inheritance, the subtyping relation $\leq$ is a tree: if $D \leq B$ and $D \leq C$ then $B \leq C$ or $C \leq B$. If $mtype(m, C)$ is defined for some C then it is defined for all subclasses of C and there is a unique ancestor class declaring m that is least with respect to $\leq$.

Lemma 4.5 allows proofs by structural induction on typings. The following notion facilitates induction on inheritance chains.

***Definition* 4.7 (method depth)** For any m and C such that $mtype(m, C)$ is defined, the *method depth of* C *for* m *in* CT is defined by $depth(m, C) = 1 + depth(m, super\,C)$ if $mtype(m, super\,C)$ is defined; otherwise, $depth(m, C) = 0$. □

An immediate consequence is that if $mtype(m, C)$ is defined and $depth(m, C) = 0$ then $CT(C)$ has a declaration for m.

Finally, we consider ramifications of constructor dependence. Note that **Object** $\not\sqsubset$ C for all C, by the typing rule for **new**.

***Definition* 4.8 (semantic dependence, $\ll$)** As an auxiliary notation, we define $B \precsim C$ iff $\{D \mid D \sqsubset^+ B\} \subseteq \{D \mid D \sqsubset^+ C\}$ and write $B \prec C$ if this inclusion is proper. For classes B, C declared in the class table, define $B \ll C$ iff $B \prec C$ or both $B \precsim C$ and $B > C$. □

Lemma 4.9 For a well formed class table we have the following.

(1) $\ll$ is well founded.

(2) $super\,C \ll C$ for all C.

(3) $B \sqsubset C$ implies $B \ll C$ for all B and C.

Proof. Note that $\precsim$ is a preorder but not antisymmetric, so $\ll$ is not a lexicographic order per se. To prove (1), define $deps\,C = \{D \mid D \sqsubset^+ C\}$ for any C. Then we have $B \ll C$ iff $(deps\,B, B) \lessdot (deps\,C, C)$, where $\lessdot$ is defined by $(X, B) \lessdot (Y, C)$ iff $X \subsetneq Y$ or $X \subseteq Y$ and $B > C$ (where $\subsetneq$ means proper subset). This is logically equivalent to: $X \subsetneq Y$ or $X = Y$ and $B > C$, which shows that the definition is the lexicographic coupling of $\subsetneq$ and $>$. As $\subsetneq$ here is for finite subsets of declared class names, both $\subsetneq$ and $>$ are well founded, hence so is their lexicographic coupling.

For (2), if $D \sqsubset^+ super\,C$ then $D \sqsubset^+ C$ by definition of $\sqsubset$; hence $super\,C \precsim C$. Also, $super\,C > C$, so (2) holds by definition of $\ll$.

For (3), suppose $B \sqsubset C$. Then, by transitivity, $\{D \mid D \sqsubset^+ B\} \subseteq \{D \mid D \sqsubset^+ C\}$. Also, we have $B \sqsubset^+ C$ but $B \not\sqsubset^+ B$, by well formedness of the class table, so the inclusion is proper. That is, $B \prec C$, whence $B \ll C$ by definition of $\ll$. □

[10]Strictly speaking this is not quite true, because in a context where **null** is typed as C it can also be typed as some subtype of C. But this has no bearing on semantics. There are several straightforward solutions to the problem and we leave it to the interested reader.

5. SEMANTICS

This section defines the semantic domains, then the semantics of expressions and commands, and finally the semantics of well formed class tables.

Because methods are associated with classes rather than with instances, the semantic domains are rather simple. There are no recursive domain equations to be solved: subclassing ($\leq$) is acyclic and the cycle of recursive references via class fields is broken via the heap. Mutually recursive method invocations can arise through direct calls on a single object and also through callbacks between reachable objects, as for example in the observer pattern. We impose no restrictions on such calls. A fixpoint construction is used for the method environment which comprises the semantics of the class table.

The interdependence between constructors and object construction commands (**new**) is a bit complex; things pertaining to constructors may be skipped on first reading. As a way of explaining the fine points, we prove in some detail that the semantics is well defined (Lemma 5.7).

Often we write $=$ between expressions involving partial functions such as those used in typing. Unless otherwise indicated, it means strong equality: both sides are defined and equal.

5.1 Semantic domains

The state of a method in execution is comprised of a *heap* h, which is a finite[11] partial function from locations to object states, and a *store* η, which assigns locations and primitive values to the local variables and parameters given by a typing context Γ.[12] An *object state* is a mapping from field names to values. Function application associates to the left, so $h\,\ell\,f$ is the value of field f of the object $h\,\ell$ at location ℓ.

A command denotes a function mapping each initial state (h, η) either to a final state (h_0, η_0) or to the distinguished value $\bot$. We use the term *global state* for (h, η), to distinguish it from object states. The improper value $\bot$ represents nontermination as well as runtime errors: attempts to dereference *nil* or cast a location to a type it does not have.

In some languages it is a runtime error to dereference a dangling pointer, i.e., one not in the domain of the heap. In Java dangling pointers cannot arise: there is no command for deallocation and a correct garbage collector never deallocates reachable objects. For our purposes, garbage collection need not be modelled. Commands act on heaps and stores that are closed in the sense that all locations that occur are in the domain of the heap. The following paragraphs formalize our assumptions about locations and then define the semantic domains.

For locations, we assume that a countable set *Loc* is given, along with a distinguished value *nil* not in *Loc*. To track each object's class we assume given a function $\mathit{loctype} : \mathit{Loc} \to \mathit{ClassNames}$ such that for each C there are infinitely many locations ℓ with $\mathit{loctype}\ \ell = C$. We use the term *heap* for any partial function h

[11]The preliminary version [Banerjee and Naumann 2002a] of this paper has a bug: infinite heaps are allowed, and it is not required that there be unallocated locations at every type.

[12]In [Banerjee and Naumann 2002a] we use the term "environment" for η, wishing to avoid the irrelevant connotations of "stack"; here we use "store", following Reynolds [2001].

such that $dom\, h \subseteq_{fin} Loc$ and each $h\,\ell$ is an object state of type $loctype\ \ell$. Object states are formalized later. Because the domain of a heap is finite, the assumption about *loctype* ensures an adequate supply of fresh locations.

We write $locs\ C$ for $\{\ell \in Loc \mid loctype\,\ell = C\}$, and $locs(C{\downarrow})$ for $\{\ell \mid loctype\,\ell \leq C\}$. There is no independent meaning for $C{\downarrow}$.

***Definition* 5.1 (allocator, parametric)** An *allocator* is a location-valued function *fresh* such that $loctype(fresh(C,h)) = C$ and $fresh(C,h) \notin dom\,h$, for all C, h. An allocator is *parametric* if $dom\,h_1 \cap locs\,C = dom\,h_2 \cap locs\,C$ implies $fresh(C,h_1) = fresh(C,h_2)$. $\square$

For example, taking $Loc = \mathbb{N}$, a parametric allocator is given by the function $fresh(C,h) = min\{\ell \mid loctype\,\ell = C \wedge \ell \notin dom\,h\}$.

Typical implementations encode the object class as part of its state. One could uncurry this representation of heaps and take *Loc* to be $\mathbb{N} \times ClassNames$. Then $fresh(C,h)$ could return (n, C) where n is the least address of an unused memory segment of sufficient size for the state of C. This is an allocator but not parametric because the presence of objects of one class affect the availability of memory for objects of other classes.

We define the semantics in terms of an arbitrary allocator *fresh*. The assumption of parametricity is stated explicitly where it is needed, namely for the first abstraction theorem (Sect. 7) but not the second (Sect. 10). Parametricity of the allocator is a reasonable assumption for some applications but not all. The assumption streamlines the proof of the abstraction theorem, allowing us to highlight other issues. For the second abstraction theorem, we drop parametricity and complicate the definitions of coupling and simulation by adding a bijective renaming of locations.

In addition to heaps, it is convenient to name a number of other semantic categories that are explained in due course.

Semantic categories

$$\theta ::= T \mid \Gamma \mid state\,C \mid Heap \mid Heap \otimes \Gamma \mid Heap \otimes T \mid \theta_\bot \mid C, \overline{x}, \overline{T}{\to}T \mid MEnv$$

In order to define the more complicated semantic domains, we need to define closed stores. Stores are among the simpler semantic domains, which are defined as follows.

Semantics of types, object states, and stores

$$\begin{array}{lcl}
[\![\mathbf{bool}]\!] & = & \{true, false\} \\
[\![\mathbf{unit}]\!] & = & \{it\} \\
[\![C]\!] & = & \{nil\} \cup locs(C{\downarrow}) \\
[\![state\,C]\!] & = & \{s \mid dom\,s = dom(fields\,C) \wedge \forall (f : T) \in fields\,C \bullet s f \in [\![T]\!]\} \\
[\![\Gamma]\!] & = & \{\eta \mid dom\,\eta = dom\,\Gamma \wedge \eta\,\mathsf{self} \neq nil \wedge \forall x \in dom\,\eta \bullet \eta\,x \in [\![\Gamma\,x]\!]\}
\end{array}$$

As small dot has another use, we use the fat dot $\bullet$ to separate a bound variable from its scope. Note that $[\![\Gamma]\!]$ is defined for Γ both with and without result in its domain.

***Definition* 5.2 (closed heap and store)** A heap h is *closed*, written $\mathit{closed}\, h$, iff $\mathit{rng}(h\,\ell) \cap \mathit{Loc} \subseteq \mathit{dom}\, h$, for all $\ell \in \mathit{dom}\, h$. A store $\eta \in [\![\Gamma]\!]$ is *closed in heap h*, written $\mathit{closed}(h, \eta)$, iff $\mathit{rng}\, \eta \cap \mathit{Loc} \subseteq \mathit{dom}\, h$. $\Box$

Note that $\mathit{rng}(h\,\ell)$ is the set of values in fields of the object state $h\,\ell$.

Recall that fresh locations should occur nowhere in the global state. For a closed store and heap, this follows from the requirement that $\mathit{fresh}(C, h) \notin \mathit{dom}\, h$.[13]

Semantics of global states and methods

$$\begin{array}{lcl}
[\![\mathit{Heap}]\!] & = & \{h \mid \mathit{dom}\, h \subseteq_{\mathit{fin}} \mathit{Loc} \wedge \mathit{closed}\, h \wedge \forall \ell \in \mathit{dom}\, h \bullet h\ell \in [\![\mathit{state}\,(\mathit{loctype}\ \ell)]\!]\} \\
[\![\mathit{Heap} \otimes \Gamma]\!] & = & \{(h, \eta) \mid h \in [\![\mathit{Heap}]\!] \wedge \eta \in [\![\Gamma]\!] \wedge \mathit{closed}(h, \eta)\} \\
[\![\mathit{Heap} \otimes T]\!] & = & \{(h, d) \mid h \in [\![\mathit{Heap}]\!] \wedge d \in [\![T]\!] \wedge (d \in \mathit{Loc} \Rightarrow d \in \mathit{dom}\, h)\} \\
[\![\theta_\bot]\!] & = & [\![\theta]\!] \cup \{\bot\} \quad \text{(where $\bot$ is some fresh value not in $[\![\theta]\!]$)} \\
[\![C, \overline{x}, \overline{T} \to T]\!] & = & [\![\mathit{Heap} \otimes (\overline{x} : \overline{T}, \mathsf{self} : C)]\!] \to [\![(\mathit{Heap} \otimes T)_\bot]\!] \\
[\![\mathit{MEnv}]\!] & = & \{\mu \mid \forall C, m \bullet \mu C m \text{ is defined iff } \mathit{mtype}(m, C) \text{ is defined,} \\
 & & \quad \text{and } \mu C m \in [\![C, \mathit{pars}(m, C), \mathit{mtype}(m, C)]\!] \text{ if } \mu C m \text{ defined } \}
\end{array}$$

Just as a class declaration $CT(C)$ gives a collection of method declarations, the semantics of a class table is a *method environment* that assigns to each class C a method meaning $\mu\, C\, m$ for each m declared or inherited in C.

For the fixpoint construction of the method environment denoted by a class table, we need to impose order on the semantic domains. We use the term *complete partial order* for a poset with least upper bounds of countable ascending chains [Davey and Priestley 1990]. The degenerate case is ordering by equality, which is the order we use for the semantics of T, Γ, $\mathit{state}\, C$, Heap, $(\mathit{Heap} \otimes \Gamma)$, and $(\mathit{Heap} \otimes T)$. Then $[\![(\mathit{Heap} \otimes \Gamma)_\bot]\!]$ and $[\![(\mathit{Heap} \otimes T)_\bot]\!]$ are complete partial orders with the "flat" order: $\bot$ is below anything and other comparable elements are equal. The set $[\![C, \overline{x}, \overline{T} \to T]\!]$ is defined to be the space of total functions $[\![\mathit{Heap} \otimes (\overline{x} : \overline{T}, \mathsf{self} : C)]\!] \to [\![(\mathit{Heap} \otimes T)_\bot]\!]$, all of which are continuous because $\mathit{Heap} \otimes (\overline{x} : \overline{T}, \mathsf{self} : C)$ is ordered by equality. The function space itself is ordered pointwise, making it a complete partial order with minimum element $\lambda(h, \eta) \bullet \bot$. Finally, we order $[\![\mathit{MEnv}]\!]$ pointwise. All method environments μ in $[\![\mathit{MEnv}]\!]$ have the same domain, determined by CT, so this is also a complete partial order, taken pointwise. It has a minimum element, namely $\lambda C \bullet \lambda m \bullet \lambda(h, \eta) \bullet \bot$.

[13] If dangling pointers were allowed, the definition of freshness would need to be with respect to both the store and all object states in the heap. The issue becomes apparent in the proof of Lemma 6.16 in the sequel, which uses closure. Most of the other definitions and results can be formulated without restricting heaps to be closed, so we mistakenly neglected closure in [Banerjee and Naumann 2002a].

Whereas $[\![\mathit{state}\ C]\!]$ consists of the states for objects of exactly class C, the set $[\![C]\!]$ is downward closed. For data types T_1, T_2 we have $T_1 \leq T_2 \Rightarrow [\![T_1]\!] \subseteq [\![T_2]\!]$.

***Definition* 5.3 (incomparable, $\not\lesseqgtr$)** We write $C \not\lesseqgtr B$ for $C \not\leq B \wedge C \not\geq B$. For a list $\overline{C}$, $\overline{C} \not\lesseqgtr B$ means $C \not\lesseqgtr B$ for all C in $\overline{C}$. □

Lemma 5.4 For classes C, B, if $C \not\lesseqgtr B$ then $[\![C]\!] \cap [\![B]\!] = \{\mathit{nil}\}$. For primitive T we have $[\![T]\!] \cap [\![B]\!] = \varnothing$. □

The result is a direct consequence of the definitions. We often use the contrapositive: if there is a non-*nil* location in both $[\![B]\!]$ and $[\![C]\!]$ then $B \leq C$ or $C \leq B$.

5.2 Semantics of expressions, commands, constructors and methods

For expressions and commands, the semantics is defined by induction on typing derivations. As a consequence of uniqueness of typing derivations, Lemma 4.5, the semantics is a function of typings. The meaning of a command $\Gamma \vdash S$ will be defined to be a function

$$[\![\Gamma \vdash S]\!] \in [\![\mathit{MEnv}]\!] \rightarrow [\![\mathit{Heap} \otimes \Gamma]\!] \rightarrow [\![(\mathit{Heap} \otimes \Gamma)_\bot]\!] \ .$$

The meaning of an expression $\Gamma \vdash e : T$ will be defined to be a function

$$[\![\Gamma \vdash e : T]\!] \in [\![\mathit{Heap} \otimes \Gamma]\!] \rightarrow [\![T_\bot]\!]$$

such that the result value is always in the domain of the heap if it is a location.[14] This is part of Lemma 5.7, the proof of which serves as an exposition for some details of the semantic definitions.

The command and expression constructs are strict in $\bot$, except, as usual, for the then- and else-commands in **if**$-$**fi**. To streamline the treatment of $\bot$ in the semantic definitions we use a metalanguage construct which some readers will recognize as the bind operation of the lifting monad [Moggi 1991]. The construct

$$\mathsf{let}\ d = E_1\ \mathsf{in}\ E_2$$

has the following meaning: If the value of E_1 is $\bot$ then that is the value of the entire let expression; otherwise, its value is the value of E_2 with d bound to the value of E_1.

We let $(h, \eta) \in [\![\mathit{Heap} \otimes \Gamma]\!]$ in the following definitions. Identifiers are as in the corresponding typing rules. For semantic values we use the identifier d, but sometimes ℓ for elements of the sets $[\![C]\!]$.

For expressions the semantics is straightforward; we choose the Java semantics for casts and tests.

[14]We have chosen a simple but slightly inelegant formulation. We express closure of the result for commands in the semantic domain whereas for expressions there is no returned heap and we express closure as a property of the semantic function. The presentation could be made more elegant by introducing categories $\mathbf{exp}(\Gamma, T)$ and $\mathbf{com}(\Gamma)$ with $[\![\mathbf{com}(\Gamma)]\!] = [\![\mathit{MEnv}]\!] \rightarrow [\![\mathit{Heap} \otimes \Gamma]\!] \rightarrow [\![(\mathit{Heap} \otimes \Gamma)_\bot]\!]$ and imposing the restriction on return values in the definition of $[\![\mathbf{exp}(\Gamma, T)]\!]$ as a subset of $[\![\mathit{Heap} \otimes \Gamma]\!] \rightarrow [\![T_\bot]\!]$. We could even restrict the meanings to those that are confined, but the gain in elegance would come at the expense of complexity that not all readers would find illuminating. We have chosen to treat confinement and parametricity as properties to be proved after the semantics is defined, downplaying the model as an independent structure. Thus little would be gained by naming categories $\mathbf{exp}(\Gamma, T)$ and $\mathbf{com}(\Gamma)$.

Semantics of expressions

$$
\begin{array}{lcl}
[\![\Gamma \vdash x : T]\!](h,\eta) & = & \eta x \\
[\![\Gamma \vdash \mathbf{null} : B]\!](h,\eta) & = & \mathit{nil} \\
[\![\Gamma \vdash \mathbf{it} : \mathbf{unit}]\!](h,\eta) & = & \mathit{it} \\
[\![\Gamma \vdash \mathbf{true} : \mathbf{bool}]\!](h,\eta) & = & \mathit{true} \\
[\![\Gamma \vdash \mathbf{false} : \mathbf{bool}]\!](h,\eta) & = & \mathit{false} \\
[\![\Gamma \vdash e_1 = e_2 : \mathbf{bool}]\!](h,\eta) & = & \mathsf{let}\ d_1 = [\![\Gamma \vdash e_1 : T_1]\!](h,\eta)\ \mathsf{in} \\
 & & \mathsf{let}\ d_2 = [\![\Gamma \vdash e_2 : T_2]\!](h,\eta)\ \mathsf{in} \\
 & & \mathsf{if}\ d_1 = d_2\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ \mathit{false} \\
[\![\Gamma \vdash e.f : T]\!](h,\eta) & = & \mathsf{let}\ \ell = [\![\Gamma \vdash e : (\Gamma\,\mathsf{self})]\!](h,\eta)\ \mathsf{in} \\
 & & \mathsf{if}\ \ell = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else}\ h\,\ell\,f \\
[\![\Gamma \vdash (B)\ e : B]\!](h,\eta) & = & \mathsf{let}\ \ell = [\![\Gamma \vdash e : D]\!](h,\eta)\ \mathsf{in} \\
 & & \mathsf{if}\ \ell = \mathit{nil} \lor \mathit{loctype}\ \ell \le B\ \mathsf{then}\ \ell\ \mathsf{else}\ \bot \\
[\![\Gamma \vdash e\ \mathbf{is}\ B : \mathbf{bool}]\!](h,\eta) & = & \mathsf{let}\ \ell = [\![\Gamma \vdash e : D]\!](h,\eta)\ \mathsf{in} \\
 & & \mathsf{if}\ \ell \ne \mathit{nil} \land \mathit{loctype}\ \ell \le B\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ \mathit{false}
\end{array}
$$

The semantics of commands is defined by structural induction on the command, except for object construction $x := \mathbf{new}\ C$ which also depends on the constructor semantics of the constructor, $\mathit{constr}\,C$, of C. That in turn depends on the constructor of $\mathit{super}\,C$, and on the command semantics of $\mathit{constr}\,C$. Well foundedness of this dependence is part of the proof of Lemma 5.7.

In the semantics of commands, we write $[\mathit{fields}\,B \mapsto \mathit{defaults}]$ as an abbreviation for the function sending each $f \in \mathit{dom}(\mathit{fields}\,B)$ to the default value for $\mathit{type}(f,B)$. The defaults are *false* for **bool**, *it* for **unit**, and *nil* for classes. Function update or extension is written like $[\eta \mid x \mapsto d]$. We write $\downharpoonright$ for domain restriction: if x is in the domain of η then $\eta \downharpoonright x$ is the function like η but with x dropped from its domain.

Method calls of the form $x := e.m(\overline{e})$ are dynamically bound: the method meaning is determined by $\mathit{loctype}\,\ell$ in the semantic definition, where ℓ is the value of e. By typing, $\mathit{loctype}\,\ell \le D$ and $\mathit{pars}(m, \mathit{loctype}\,\ell) = \mathit{pars}(m, D)$. Super-calls are statically bound: the method meaning used, $\mu(\mathit{super}\,C)m$, is determined by the static class C. Note that if $\mathit{mtype}(m, \mathit{super}\,C)$ is defined, as required by the typing rule, then $\mathit{pars}(m, C) = \mathit{pars}(m, \mathit{super}\,C)$.

Semantics of commands

$$\begin{array}{lcl}
[\![\Gamma \vdash x := e]\!]\mu(h,\eta) & = & \mathsf{let}\ d = [\![\Gamma \vdash e : T]\!](h,\eta)\ \mathsf{in}\ (h, [\eta \mid x \mapsto d]) \\
[\![\Gamma \vdash e_1.f := e_2]\!]\mu(h,\eta) & = & \mathsf{let}\ \ell = [\![\Gamma \vdash e_1 : (\Gamma\,\mathsf{self})]\!](h,\eta)\ \mathsf{in} \\
 & & \mathsf{if}\ \ell = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else} \\
 & & \mathsf{let}\ d = [\![\Gamma \vdash e_2 : U]\!](h,\eta)\ \mathsf{in} \\
 & & ([h \mid \ell \mapsto [h\ell \mid f \mapsto d]], \eta) \\
[\![\Gamma \vdash x := \mathbf{new}\ B]\!]\mu(h,\eta) & = & \mathsf{let}\ \ell = \mathit{fresh}(B,h)\ \mathsf{in} \\
 & & \mathsf{let}\ h_1 = [h \mid \ell \mapsto [\mathit{fields}\,B \mapsto \mathit{defaults}]]\ \mathsf{in} \\
 & & \mathsf{let}\ \eta_1 = [\mathsf{self} \mapsto \ell]\ \mathsf{in} \\
 & & \mathsf{let}\ h_0 = [\![\mathsf{self} : B \vdash \mathit{constr}\,B : \mathbf{con}]\!]\mu(h_1,\eta_1)\ \mathsf{in} \\
 & & (h_0, [\eta \mid x \mapsto \ell]) \\
[\![\Gamma \vdash x := e.m(\overline{e})]\!]\mu(h,\eta) & = & \mathsf{let}\ \ell = [\![\Gamma \vdash e : D]\!](h,\eta)\ \mathsf{in} \\
 & & \mathsf{if}\ \ell = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else} \\
 & & \mathsf{let}\ \overline{x} = \mathit{pars}(m,D)\ \mathsf{in} \\
 & & \mathsf{let}\ \overline{d} = [\![\Gamma \vdash \overline{e} : \overline{U}]\!](h,\eta)\ \mathsf{in} \\
 & & \mathsf{let}\ \eta_1 = [\overline{x} \mapsto \overline{d}, \mathsf{self} \mapsto \ell]\ \mathsf{in} \\
 & & \mathsf{let}\ (h_1,d_1) = \mu(\mathit{loctype}\,\ell)m(h,\eta_1)\ \mathsf{in} \\
 & & (h_1, [\eta \mid x \mapsto d_1]) \\
[\![\Gamma \vdash x := \mathbf{super}.m(\overline{e})]\!]\mu(h,\eta) & = & \mathsf{let}\ \ell = \eta\,\mathsf{self}\ \mathsf{in} \\
 & & \mathsf{let}\ \overline{x} = \mathit{pars}(m, \Gamma\,\mathsf{self})\ \mathsf{in} \\
 & & \mathsf{let}\ \overline{d} = [\![\Gamma \vdash \overline{e} : \overline{U}]\!](h,\eta)\ \mathsf{in} \\
 & & \mathsf{let}\ \eta_1 = [\overline{x} \mapsto \overline{d}, \mathsf{self} \mapsto \ell]\ \mathsf{in} \\
 & & \mathsf{let}\ (h_1,d_1) = \mu(\mathit{super}(\Gamma\,\mathsf{self}))m(h,\eta_1)\ \mathsf{in} \\
 & & (h_1, [\eta \mid x \mapsto d_1]) \\
[\![\Gamma \vdash S_1;\ S_2]\!]\mu(h,\eta) & = & \mathsf{let}\ (h_1,\eta_1) = [\![\Gamma \vdash S_1]\!]\mu(h,\eta)\ \mathsf{in} \\
 & & [\![\Gamma \vdash S_2]\!]\mu(h_1,\eta_1) \\
[\![\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}]\!]\mu(h,\eta) & = & \mathsf{let}\ b = [\![\Gamma \vdash e : \mathbf{bool}]\!](h,\eta)\ \mathsf{in} \\
 & & \mathsf{if}\ b\ \mathsf{then}\ [\![\Gamma \vdash S_1]\!]\mu(h,\eta)\ \mathsf{else}\ [\![\Gamma \vdash S_2]\!]\mu(h,\eta) \\
[\![\Gamma \vdash T\ x := e\ \mathbf{in}\ S]\!]\mu(h,\eta) & = & \mathsf{let}\ d = [\![\Gamma \vdash e : U]\!](h,\eta)\ \mathsf{in} \\
 & & \mathsf{let}\ \eta_1 = [\eta \mid x \mapsto d]\ \mathsf{in} \\
 & & \mathsf{let}\ (h_1,\eta_2) = [\![(\Gamma, x : T) \vdash S]\!]\mu(h,\eta_1)\ \mathsf{in} \\
 & & (h_1, (\eta_2 \upharpoonright x))
\end{array}$$

The meaning of a command S as a constructor is a function

$$[\![\mathsf{self} : C \vdash S : \mathbf{con}]\!] \in [\![\mathit{MEnv}]\!] \to [\![\mathit{Heap} \otimes \mathsf{self} : C]\!] \to [\![\mathit{Heap}_\bot]\!] \ .$$

Dependence on $[\![\mathit{MEnv}]\!]$ is a formal technicality: the semantic definition uses the command semantics of S, but the typing rule disallows method calls in S.

Semantics of constructor

$$\begin{aligned}
[\![\mathsf{self}\!:\!C \vdash S\!:\!\mathbf{con}]\!]\mu(h,\eta) \;=\; & \mathsf{let}\ B = \mathit{super}\,C\ \mathsf{in}\\
& \mathsf{let}\ S_0 = \mathit{constr}\,B\ \mathsf{in}\\
& \mathsf{let}\ h_1 = \ \mathsf{if}\ B \neq \mathbf{Object}\\
& \qquad\quad \mathsf{then}\ [\![\mathsf{self}\!:\!B \vdash S_0\!:\!\mathbf{con}]\!]\mu(h,\eta)\ \mathsf{else}\ h\ \mathsf{in}\\
& \mathsf{let}\ (h_0,-) = [\![\mathsf{self}\!:\!C \vdash S]\!]\mu(h_1,\eta)\ \mathsf{in}\\
& h_0
\end{aligned}$$

Note that if $[\![\mathsf{self}\!:\!B \vdash S_0\!:\!\mathbf{con}]\!]\mu(h,\eta)$ or $[\![\mathsf{self}\!:\!C \vdash S]\!]\mu(h_1,\eta)$ is $\bot$ then so is $[\![\mathsf{self}\!:\!C \vdash S\!:\!\mathbf{con}]\!]\mu(h,\eta)$. The result $\bot$ is possible due to *nil* dereferences and cast failures but not divergence (because there are no method calls or cyclic constructor dependencies).

Semantics of method declaration

Suppose M is a method declaration in $CT(C)$, with $M = T\ m(\overline{T}\ \overline{x})\{S\}$. Its meaning $[\![M]\!]$ is the total function $[\![\mathit{MEnv}]\!] \to [\![C,\overline{x},\overline{T}\to T]\!]$ defined by

$$\begin{aligned}
[\![M]\!]\mu(h,\eta) = \ & \mathsf{let}\ \eta_1 = [\eta \mid \mathsf{result} \mapsto \mathit{default}]\ \mathsf{in}\\
& \mathsf{let}\ (h_0,\eta_0) = [\![\overline{x}\!:\!\overline{T}, \mathsf{self}\!:\!C, \mathsf{result}\!:\!T \vdash S]\!]\mu(h,\eta_1)\ \mathsf{in}\\
& (h_0, \eta_0\ \mathsf{result})
\end{aligned}$$

For precision in the semantics of a method inherited in C from B we make an explicit definition for the domain-restriction of a method meaning in $[\![B,\overline{x},\overline{T}\to T]\!]$ to the global states (h,η) in $[\![\mathit{Heap} \otimes \overline{x}\!:\!\overline{T}, \mathsf{self}\!:\!C]\!]$.

***Definition* 5.5 (*restr*)** For $d \in [\![B,\overline{x},\overline{T}\to T]\!]$ and $C \leq B$, define $\mathit{restr}(d,C)$, an element of $[\![C,\overline{x},\overline{T}\to T]\!]$, by $\mathit{restr}(d,C)(h,\eta) = d(h,\eta)$. $\Box$

Semantics of class table and its approximation chain μ_j

The semantics of a well formed class table CT, written $[\![CT]\!]$, is the least upper bound of the ascending chain $\mu \in \mathbb{N} \to [\![\mathit{MEnv}]\!]$ defined as follows.

$$\begin{array}{lcll}
\mu_0\,C\,m & = & \lambda(h,\eta)\bullet\bot & \text{if } m \text{ is declared or inherited in } C\\
\mu_{j+1}\,C\,m & = & [\![M]\!]\mu_j & \text{if } m \text{ is declared as } M \text{ in } C\\
\mu_{j+1}\,C\,m & = & \mathit{restr}((\mu_{j+1}\,B\,m),C) & \text{if } m \text{ is inherited in } C \text{ from } B
\end{array}$$

***Remark* 5.6 (*On proofs*)** We give some proofs in considerable detail. To avoid repetition, we use the same identifiers as in the relevant semantic definition for each case —often different from those in the statement of the result being proved— taking care to avoid ambiguity. This saves explicit introduction of the identifiers or mention of the ranges and scopes of quantification. But it requires the reader to keep an eye on the semantic clauses. Often, without remark, we consider only the case where the outcome and various intermediate values are non-$\bot$, as the $\bot$ cases are straightforward.

Lemma 5.7 (semantics is well defined and typed) Let CT be well formed.

(1) If $C \leq B$ then for any Γ with $\mathsf{self} \notin dom\,\Gamma$ we have $[\![Heap \otimes \Gamma, \mathsf{self}\colon C]\!] \subseteq [\![Heap \otimes \Gamma, \mathsf{self}\colon B]\!]$.

(2) If $\Gamma \vdash e : T$ then $[\![\Gamma \vdash e : T]\!] \in [\![Heap \otimes \Gamma]\!] \to [\![T_\bot]\!]$.

(3) If $(h, \eta) \in [\![Heap \otimes \Gamma]\!]$ and $d = [\![\Gamma \vdash e : T]\!](h, \eta)$ with $d \neq \bot$ then $(h, d) \in [\![Heap \otimes T]\!]$.

(4) If $\Gamma \vdash S$ then $[\![\Gamma \vdash S]\!] \in [\![MEnv]\!] \to [\![Heap \otimes \Gamma]\!] \to [\![(Heap \otimes \Gamma)_\bot]\!]$.

(5) $[\![CT]\!]$ is well defined.

Proof. (1) follows easily from the fact that $C \leq B$ implies $[\![C]\!] \subseteq [\![B]\!]$.

For (2), inspection of the definitions shows that $[\![\Gamma \vdash e : T]\!](h, \eta)$ is in $[\![T_\bot]\!]$. It is property $(h, d) \in [\![Heap \otimes T]\!]$, i.e., (3), that we need explicitly in some proof steps. This holds because (h, η) is closed and no expression creates fresh locations.

Property (4) requires a straightforward but not entirely trivial check that, for any μ, $[\![\Gamma \vdash S]\!]\mu(h, \eta)$ is in $[\![(Heap \otimes \Gamma)_\bot]\!]$. For example, in the case of method call $x := e.m(\overline{e})$ we need the fact that $\mu C m$ is in $[\![C, pars(m, C), mtype(m, C)]\!]$ regardless of whether m is declared or inherited in C. The store η_1 is passed to the method meaning $\mu(loctype\,\ell)m$ determined by the type, $loctype\,\ell$, of the target. Note that $\mu(loctype\,\ell)m$ is from a declaration in $loctype\,\ell$ or a superclass thereof. So, as $\eta_1\,\mathsf{self} = \ell$, we have η_1 in the domain of $\mu(loctype\,\ell)m$ by (1). Of course the call aborts if $\ell = nil$.

For (5), acyclicity of $\leq$ ensures that the semantics of the class table is well founded on inheritance depth. And (1) ensures that the definition $\mu_{j+1}\,C\,m$ for an inherited method yields a value in the semantic domain $[\![C, pars(m, C), mtype(m, C)]\!]$. We only take fixpoints for method environments, which form a complete partial order with bottom. The fixpoint is well defined because the meaning $[\![M]\!]$ of a method declaration M is a continuous functions of the method environment. This is because each $[\![\Gamma \vdash S]\!]$ is a continuous function on method environments —which in turn depends on the fact that the semantic definitions for commands are continuous in their constituent commands and expressions.

The semantics of object construction commands (**new**) is mutually dependent on the semantics of constructors. This is resolved as follows.

First, the semantics of constructors is defined by well founded recursion on the order $\ll$ on classes. For semantics of $\mathsf{self}\colon B \vdash constr\,B : \mathbf{con}$ we use both (a) the constructor semantics of $\mathsf{self}\colon (super\,B) \vdash constr(super\,B) : \mathbf{con}$ and (b) the command semantics for $constr\,B$. For (a), note that $super\,C \ll C$ by Lemma 4.9. For (b), note that if $constr\,C$ uses **new** B for other classes B, we have $B \sqsubset C$ by a condition on well formed class tables; then $B \ll C$ by Lemma 4.9. Note that there is no dependence on the method environment.

Finally, for semantics of methods we need all constructors as there is no restriction on which objects can be constructed. The semantics of methods is by structural recursion on method bodies, using the semantics of constructors. $\square$

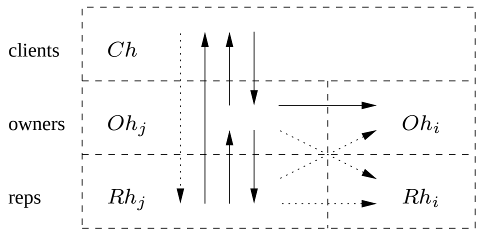

Fig. 6. Confinement scheme for island j. Dashed boxes are partition blocks. Solid lines indicate allowed references and dotted lines indicate prohibited ones. There is no restriction within blocks.

6. CONFINEMENT RAMIFIED

Our aim is to support reasoning where simulations are specified on a per-island basis, where an island consists of a single owner and its reps,[15] as discussed in Sect. 3.3. This section formalizes a semantic notion of confinement suited to this purpose. In particular, it takes into account the limited access to reps allowed for owner subclasses, which is discussed further in Sect. 9. Our notion does not allow multiple owners or transfer of ownership; this is discussed in Sect. 12.2.

6.1 Confinement of states

As discussed in Sect. 3.2 we assume that class names Own and Rep are given, such that $Own \not\leq Rep$ and thus $[\![Own]\!] \cap [\![Rep]\!] = \{nil\}$. As an abbreviation, we write $locs(Own{\downarrow}, Rep{\downarrow})$ for $locs(Own{\downarrow}) \cup locs(Rep{\downarrow})$.

We say heaps h_1 and h_2 are *disjoint* if $dom\, h_1 \cap dom\, h_2 = \varnothing$. Let $h_1 * h_2$ be the union of h_1 and h_2 if they are disjoint, and undefined otherwise.

We shall partition the heap as $h = Ch * \ldots$ where Ch contains client objects and the rest is partitioned into islands of the form $Oh * Rh$ consisting of a singleton heap Oh with an owner object and a heap Rh of its representation objects. In such a partition, the heaps Ch, Oh, and Rh need not be closed. An example is Fig. 5 in Sect. 3.3; the general scheme is depicted in Fig. 6. Our use of the word "partition" is slightly non-standard: we allow the blocks Rh_i and Ch to be empty.

***Definition* 6.1 (admissible partition)** An *admissible partition* of heap h is a

[15]In particular, this entails describing how a simulation is established by an owner constructor acting on a single owner object. As constructors have no parameters, one could define the semantics in terms of constructors applied to a single object and yielding a small heap. But such a constructor will in fact be executed in a larger heap. Suppose $(h, \eta) \in [\![Heap \otimes \Gamma]\!]$, so that everything reachable from η is already in h. If h' is a heap, not necessarily closed, such that $h' * h$ is in $[\![Heap]\!]$, then it is immediate from the definitions that $(h' * h, \eta)$ is in $[\![Heap \otimes \Gamma]\!]$. (See Sect. 6.1 for $*$.) For any S and μ we have $[\![\Gamma \vdash S]\!]\mu(h, \eta) = \bot$ iff $[\![\Gamma \vdash S]\!]\mu(h' * h, \eta) = \bot$, as can be shown using the fact that $[\![\Gamma \vdash e : T]\!](h' * h, \eta) = [\![\Gamma \vdash e : T]\!](h, \eta)$. (Strictly speaking, this depends on μ having the property; and then one shows that $[\![CT]\!]$ has the property.) What is not true is the following: if $(h_0, \eta_0) = [\![\Gamma \vdash S]\!]\mu(h, \eta)$ then $[\![\Gamma \vdash S]\!]\mu(h' * h, \eta) = (h' * h_0, \eta_0)$. The reason is that the allocator *fresh* depends on the domain of the entire heap, and we have made no assumptions to relate its behavior on h and $h' * h$.

We have not checked the details but it seems clear that if $(h_0, \eta_0) = [\![\Gamma \vdash S]\!]\mu(h' * h, \eta)$ then there is h'_0 such that $h_0 = h' * h'_0$ and $(h'_0, \eta_0) \in [\![Heap \otimes \Gamma]\!]$. Also, for S without method calls and satisfying the dependency condition for constructors (Def. 4.4), if $h_0 = [\![\Gamma \vdash S : \mathbf{con}]\!]\mu(h' * h, \eta)$ then there is h'_0 such that $h_0 = h' * h'_0$. But to be useful for our purposes this property would have to be strengthened to take partitions into account.

set of pairwise disjoint heaps $Ch, Oh_1, Rh_1, \ldots, Oh_k, Rh_k$, for $k \geq 0$, with

$$h = Ch * Oh_1 * Rh_1 * \ldots * Oh_k * Rh_k$$

and for all i $(1 \leq i \leq k)$

—$dom\, Oh_i \subseteq locs(Own\downarrow)$ and $size(dom\, Oh_i) = 1$ (owner blocks)
—$dom\, Rh_i \subseteq locs(Rep\downarrow)$ (rep blocks)
—$dom\, Ch \cap locs(Own\downarrow, Rep\downarrow) = \varnothing$ (client blocks)

***Definition* 6.2 (confined heap, confining partition, $\not\hookrightarrow$, $\not\hookrightarrow^{\overline{f}}$)** To say that no object in h_1 contains a reference to an object in h_2, we define $\not\hookrightarrow$ by

$$h_1 \not\hookrightarrow h_2 \;\Leftrightarrow\; \forall \ell \in dom\, h_1 \bullet rng(h_1\, \ell) \cap dom\, h_2 = \varnothing \;\; .$$

To say that no object in h_1 contains a reference to an object in h_2 *except via a field in* $\overline{f}$, we define $\not\hookrightarrow^{\overline{f}}$ by

$$h_1 \not\hookrightarrow^{\overline{f}} h_2 \;\Leftrightarrow\; \forall \ell \in dom\, h_1 \bullet rng((h_1\, \ell)\lfloor \overline{f}) \cap dom\, h_2 = \varnothing \;\; .$$

A heap h *is confined*, written *conf* h, iff it has a confining partition. A *confining partition* is an admissible partition such that for all j, i with $j \neq i$ we have

(1) $Ch \not\hookrightarrow Rh_j$ (clients do not point to reps)
(2) $Oh_j \not\hookrightarrow Rh_i$ (owners do not share reps)
(3) $Oh_j \not\hookrightarrow^{\overline{g}} Rh_j$ where $\overline{g} = dom(dfields(Own))$ (reps are private to Own)
(4) $Rh_j \not\hookrightarrow Oh_i * Rh_i$ (reps are confined to their islands)

A heap may have several admissible partitions, because there is no inherent order on islands and because unreachable reps can be put in any island. The definitions and results do not depend on choice of partition. We have not found a workable formulation that determines unique partitions. To describe the effect of confined commands on partitions we use the following.

***Definition* 6.3 (extension of confining partition, $\trianglelefteq$)** Define $h \trianglelefteq h_0$ iff h is confined and for any confining partition of h,

$$h = Ch * Oh_1 * Rh_1 * \ldots * Oh_k * Rh_k \quad (k \geq 0),$$

there is a confining partition of h_0,

$$h_0 = Ch^0 * Oh^0_1 * Rh^0_1 * \ldots * Oh^0_n * Rh^0_n \;\; ,$$

that is an extension in the sense that it satisfies the following:

—$n \geq k$
—$dom(Ch) \subseteq dom(Ch^0)$
—$dom(Oh_j) \subseteq dom(Oh^0_j)$ for all $j \leq k$
—$dom(Rh_j) \subseteq dom(Rh^0_j)$ for all $j \leq k$ □

Confinement of a store depends on the class in which it may occur. For owners and reps it depends on the domain of the heap as well.

***Definition* 6.4 (confined store, global state)** Let h be a confined heap and η be a store in $[\![\Gamma, \mathsf{self}:C]\!]$ for some Γ. We say η is *confined in h for C* iff

(1) $C \not\leq Rep \wedge C \not\leq Own \;\Rightarrow\; rng\,\eta \cap locs(Rep{\downarrow}) = \varnothing$

(2) $C \leq Own \;\Rightarrow\; rng\,\eta \cap locs(Rep{\downarrow}) \subseteq dom(Rh_j)$
for some confining partition and j with $\eta\,\mathsf{self} \in dom(Oh_j)$

(3) $C \leq Rep \;\Rightarrow\; rng\,\eta \cap locs(Own{\downarrow}, Rep{\downarrow}) \subseteq dom(Oh_j * Rh_j)$
for some confining partition and j with $\eta\,\mathsf{self} \in dom(Rh_j)$

A global state (h, η) is *confined*, written $conf\,C\,(h, \eta)$, iff h is confined and η is confined in h for C. □

Apropos the examples in Sect. 2.1, take Rep to be Bool and suppose the sequence z := **new** OBool; w := z.bad() occurs in a method of some client class. Executed in a confined initial state, the state after assignment of a new OBool to z is still confined. The assignment to w then yields a state where the heap is confined but the client's store is not.

6.2 Confinement of commands and methods

A confined command is one that preserves confinement of global states. Because command meanings depend on the method environment and expression meanings, confinement for those is formalized first. We need to ensure that a method call yields a heap confined for the caller. This is achieved using the condition $h \trianglelefteq h_0$ in the following Definition, together with Lemma 6.13 to follow.

***Definition* 6.5 (confined method environment)** Method environment μ is confined, written $conf\,\mu$, if and only if the following holds for all C and m with $mtype(m, C)$ defined. Let $mtype(m, C) = \overline{T} \to T$ and $pars(m, C) = \overline{x}$. For all $(h, \eta) \in [\![Heap \otimes \overline{x}:\overline{T}, \mathsf{self}:C]\!]$, if $conf\,C\,(h, \eta)$ and $\mu C m (h, \eta) \neq \bot$ then

(1) $C \not\leq Rep \;\Rightarrow\; h \trianglelefteq h_0 \wedge d \notin locs(Rep{\downarrow})$

(2) $C \leq Rep \;\Rightarrow\; h \trianglelefteq h_0 \wedge\; (d \in locs(Own{\downarrow}, Rep{\downarrow}) \;\Rightarrow\; d \in dom(Oh_j * Rh_j))$
for some confining partition $h_0 = Ch * Oh_1 * Rh_1 \ldots$
and j with $\eta\,\mathsf{self} \in dom(Rh_j)$

where $(h_0, d) = \mu C m (h, \eta)$. □

Note that the consequent $h \trianglelefteq h_0$ implies $conf\,h_0$, by definition of $\trianglelefteq$ using $conf\,h$ which follows from the antecedent $conf\,C\,(h, \eta)$.

Condition (1) fails for method bad of the example in Sect. 2.1, regardless of whether the return type of bad is taken to be **Object** or Bool.

The conditions for confinement of expressions are like those for confined stores —after all, a store provides the meaning for the expression x. The conditions are somewhat different for confined method environments, because methods are public and can be called both by clients and from within an owner island. (In Sect. 9, Def. 6.5 is refined to allow module-scoped owner methods to return reps.) Also, confinement of commands does not explicitly require heap extension $h \trianglelefteq h_0$ like Def. 6.5 does, because it is a consequence of the other conditions (see Lemma 6.16).

***Definition* 6.6 (confined expression)** Let $C = \Gamma\,\mathsf{self}$. Expression $\Gamma \vdash e:T$ is confined iff for any (h, η), if $\mathit{conf}\; C\,(h, \eta)$ and $[\![\Gamma \vdash e:T]\!](h, \eta) \neq \bot$ then the following hold, where $d = [\![\Gamma \vdash e:T]\!](h, \eta)$.

(1) $C \nleq \mathit{Rep} \wedge C \nleq \mathit{Own} \;\Rightarrow\; d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$

(2) $C \leq \mathit{Own} \;\Rightarrow\; (d \in \mathit{locs}(\mathit{Rep}{\downarrow}) \;\Rightarrow\; d \in \mathit{dom}(\mathit{Rh}_j))$
for some confining partition and j with $\eta\,\mathsf{self} \in \mathit{dom}(\mathit{Oh}_j)$

(3) $C \leq \mathit{Rep} \;\Rightarrow\; (d \in \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow}) \;\Rightarrow\; d \in \mathit{dom}(\mathit{Oh}_j * \mathit{Rh}_j))$
for some confining partition and j with $\eta\,\mathsf{self} \in \mathit{dom}(\mathit{Rh}_j)$ □

***Definition* 6.7 (confined command)** Let $C = \Gamma\,\mathsf{self}$. Command $\Gamma \vdash S$ is confined iff

— $\mathit{conf}\;\mu \wedge \mathit{conf}\; C\,(h, \eta) \wedge [\![\Gamma \vdash S]\!]\mu(h, \eta) \neq \bot \;\Rightarrow\; \mathit{conf}\; C\,(h_0, \eta_0)$, for any μ and any (h, η), where $(h_0, \eta_0) = [\![\Gamma \vdash S]\!]\mu(h, \eta)$

—if S is a method call then it has confined arguments (see below). □

Confinement of arguments means that the store η_1 passed in the semantics of method call is confined for the callee.

***Definition* 6.8 (confined arguments)** Let $C = \Gamma\,\mathsf{self}$. A call $\Gamma \vdash x := e.m(\overline{e})$ has *confined arguments* provided the following holds. Suppose $\overline{U}$ is the static type of $\overline{e}$ and D the static type of e. For any (h, η) with $\mathit{conf}\; C\,(h, \eta)$, let

$$\overline{d} = [\![\Gamma \vdash \overline{e}:\overline{U}]\!](h, \eta) \qquad \ell = [\![\Gamma \vdash e:D]\!](h, \eta) \qquad \eta_1 = [\overline{x} \mapsto \overline{d}, \mathsf{self} \mapsto \ell] \;.$$

If $\ell \neq \bot$, $\ell \neq \mathit{nil}$, and $\overline{d} \neq \bot$ (i.e., $\bot$ does not occur in $\overline{d}$) then $\mathit{conf}\;(\mathit{loctype}\;\ell)\,(h, \eta_1)$.

A super-call $\Gamma \vdash x := \mathbf{super}.m(\overline{e})$ has *confined arguments* provided the following holds. Suppose $\overline{U}$ is the static type of $\overline{e}$. For any (h, η) with $\mathit{conf}\; C\,(h, \eta)$, let

$$\overline{d} = [\![\Gamma \vdash \overline{e}:\overline{U}]\!](h, \eta) \qquad \ell = \eta\,\mathsf{self} \qquad \eta_1 = [\overline{x} \mapsto \overline{d}, \mathsf{self} \mapsto \ell] \;.$$

If $\overline{d} \neq \bot$ then $\mathit{conf}\;(\mathit{super} C)\,(h, \eta_1)$. □

A purely semantic formulation would call class table CT confined just if $[\![CT]\!]$ is a confined method environment. But under simple restrictions, confinement of $[\![CT]\!]$ follows from confinement of method bodies and constructors. Thus we choose the following.

***Definition* 6.9 (confined class table)** Class table CT is confined iff for every C and every m with $\mathit{mtype}(m, C) = \overline{T} \to T$ the following hold.

(1) If m is declared in C by $T\; m(\overline{T}\;\overline{x})\{S\}$ then S and all its constituents are confined.

(2) If the constructor declaration in C is $\mathbf{con}\{\,S\,\}$ then S and all its constituents are confined.

(3) If $C \leq \mathit{Own}$ then $T \ngeq \mathit{Rep}$.

(4) If m is inherited in Own from some $B > \mathit{Own}$ then $\overline{T} \ngeq \mathit{Rep}$.

(5) No method m is inherited in Rep from any $B > \mathit{Rep}$. □

In Sect. 10 we add module-scoped methods on which condition (3) need not be imposed. This condition ensures that owner methods do not return reps, which is not ensured by confinement of the method body. Condition (5) is needed because confinement of a method inherited from $B > Rep$ depends on the arguments, including self, being confined at B where reps are disallowed. Invocation of such a method on an object of type Rep (or a subclass) would yield a store with self a rep. A more refined restriction is to disallow inheritance into Rep only for methods which leak self; see Sect. 12.

***Example* 6.10** Condition (3) precludes the bad method of Sect. 2.1, for both return types **Object** and Bool. Except for this, all examples in Sect. 2 yield confined class tables (e.g., the well formed class table obtained by combining Figs. 2 and 3). One way to prove confinement for these examples is to check that they are safe according to the static analysis of Sect. 11. For this one uses the desugarings of Remark 4.1. □

6.3 Properties of confinement

We need a number of results about confinement. The most important is that the semantics of a confined class table is a confined method environment (Theorem 6.17). This depends on Lemma 6.16 which says that confined commands extend heap partitions, provided that method meanings have this property.

Lemma 6.11 If T is **bool** or **unit**, then every $\Gamma \vdash e : T$ is confined.

Proof. Direct from the definitions: confinement only pertains to locations. □

Lemma 6.12 Suppose $rng\,\eta \cap locs(Rep{\downarrow}) = \varnothing$ and $C \leq B$. Then for any h and any $\eta \in [\![\Gamma, \mathsf{self} : C]\!]$ we have $conf\, C\,(h, \eta)$ iff $conf\, B\,(h, \eta)$.

Proof. Straightforward. See Appendix. □

Lemma 6.13 If $conf\, C\,(h, \eta)$ and $h \trianglelefteq h_0$ then $conf\, C\,(h_0, \eta)$.

Proof. Straightforward. See Appendix. □

Although confining partitions are not unique, a given confining partition of an initial state can be extended to one on the final state for any command. This is Lemma 6.16 below, which depends on the analogous property for constructors, Lemma 6.15. From the proof of the latter, we factor out the induction step as a somewhat complicated separate result, Lemma 6.14, because it is also used in Sect. 11 to show soundness of the static analysis. Skip on first reading!

Lemma 6.14 Let μ be a method environment. Suppose we have the following:

(1) $\mathsf{self} : C \vdash S$ is a confined command.

(2) for any B with an occurrence of **new** B in S we have $B \sqsubset C$ and moreover no method calls occur in S.

(3) for any B with an occurrence of **new** B in S, and also for $B = super C$ unless $super C =$ **Object**, the following holds for any (h, η) with $conf\, B\,(h, \eta)$:

$$[\![\mathsf{self} : B \vdash S_0 : \mathbf{con}]\!]\mu(h, \eta) \neq \bot \;\Rightarrow\; h \trianglelefteq h_1 \;\;,$$

where $S_0 = constr\, B$ and $h_1 = [\![\mathsf{self} : B \vdash S_0 : \mathbf{con}]\!]\mu(h, \eta)$.

Then for any (h,η) with $\mathit{conf}\,C\,(h,\eta)$, if $[\![\mathsf{self}\!:\!C \vdash S : \mathbf{con}]\!]\mu(h,\eta) \neq \bot$ then $h \trianglelefteq h_0$ where $h_0 = [\![\mathsf{self}\!:\!C \vdash S : \mathbf{con}]\!]\mu(h,\eta)$.

Proof. Assume (1–3) hold. To show the conclusion for the non-$\bot$ case, consider any (h,η) with $\mathit{conf}\,C\,(h,\eta)$ and let h_1 be as in the semantics of S as a constructor. If $\mathit{super}C = \mathbf{Object}$ then $h_1 = h$ and thus $h \trianglelefteq h_1$. Otherwise, $h_1 = [\![\mathsf{self}\!:\!\mathit{super}C \vdash \mathit{constr}(\mathit{super}C) : \mathbf{con}]\!]\mu(h,\eta)$ and $h \trianglelefteq h_1$ holds by hypothesis (3). Now by semantics, $h_0 = [\![\mathsf{self}\!:\!C \vdash S]\!]\mu(h_1,\eta)$.

To show that $h \trianglelefteq h_0$, we can argue by induction on the structure of S. Note that S has no method calls, by hypothesis (2), so μ is not relevant. Moreover, for any object construction the result holds by hypothesis (3). We omit the rest of the argument, which uses hypothesis (1): it is exactly the same as in the proof of Lemma 6.16 below, except for appealing to hypothesis (2) for the case of **new**, where that proof appeals to Lemma 6.15. $\square$

Lemma 6.15 (extension by constructors) Suppose $\mathsf{self}\!:\!C \vdash \mathit{constr}\,C$ is confined, for all C. Then for any (h,η) with $\mathit{conf}\,C\,(h,\eta)$ we have

$$[\![\mathsf{self}\!:\!C \vdash S : \mathbf{con}]\!]\mu(h,\eta) \neq \bot \;\Rightarrow\; h \trianglelefteq h_0 \quad \text{where } h_0 = [\![\mathsf{self}\!:\!C \vdash S : \mathbf{con}]\!]\mu(h,\eta) \ .$$

Proof. This is exactly the conclusion of Lemma 6.14. We prove it by well founded induction on C, using $\ll$ which is well founded by Lemma 4.9(1). For any C and S, it suffices to show that the hypotheses (1–3) of Lemma 6.14 hold for classes smaller than C with respect to $\ll$. First, (1) holds by hypothesis of the present Lemma. By well formedness of the class table, there are no method calls in $\mathit{constr}C$, and moreover if **new** B occurs in S then $B \sqsubset C$; this is hypothesis (2). Now from Lemma 4.9 and well formedness of the class table we have that $B \ll C$ for every **new** B that occurs in S and also $\mathit{super}C \ll C$. Thus by the induction hypothesis we have (3). $\square$

Lemma 6.16 (extension by commands) Suppose $\Gamma \vdash S$ is confined and all its constituents are confined. Suppose moreover that $\mathsf{self}\!:\!B \vdash \mathit{constr}B$ is confined, for all B. Let $C = \Gamma\,\mathsf{self}$. For any μ, h, η with $\mathit{conf}\,\mu$ and $\mathit{conf}\,C\,(h,\eta)$

$$[\![\Gamma \vdash S]\!]\mu(h,\eta) \neq \bot \;\Rightarrow\; h \trianglelefteq h_0 \quad \text{where } (h_0,-) = [\![\Gamma \vdash S]\!]\mu(h,\eta) \ .$$

Proof. By structural induction on S. Let $C = \Gamma\,\mathsf{self}$. We assume a confining partition $h = \mathit{Ch} * \mathit{Oh}_1 * \mathit{Rh}_1 * \ldots * \mathit{Oh}_k * \mathit{Rh}_k$ is given (k may be 0, i.e., there need not be any islands). We show how to construct confining partition $h_0 = \mathit{Ch}^0 * \mathit{Oh}^0_1 * \mathit{Rh}^0_1 * \ldots$ that extends the given one.

Case $\Gamma \vdash e_1.f := e_2$. From $[\![\Gamma \vdash e_1.f := e_2]\!]\mu(h,\eta) \neq \bot$ and Lemma 5.7(3) we have that $\ell \in \mathit{dom}\,h$ where $\ell = [\![\Gamma \vdash e_1 : C]\!](h,\eta)$. By semantics, $h_0 = [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]$. We partition h_0 using the given partition for h. That is, the domain for each block of the updated heap h_0 is the same as the corresponding block for h. Clearly this extends the partition for h. To show that this partition is confining for h_0, it suffices to show that the update of $h\ell f$ to d satisfies the confinement property for ℓ. We argue by cases on $\mathit{loctype}\,\ell$

—$\mathit{loctype}\,\ell \not\leq \mathit{Own} \wedge \mathit{loctype}\,\ell \not\leq \mathit{Rep}$. Then Def. 6.2(1) applies; it requires $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$. By typing, $\mathit{loctype}\,\ell \leq C$, so $C \not\leq \mathit{Own} \wedge C \not\leq \mathit{Rep}$. Thus by confinement of e_1 (a constituent of $e_1.f := e_2$ and therefore confined by hypothesis), we have by Def. 6.6(1) that $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$.

—$\mathit{loctype}\,\ell \leq \mathit{Own}$. Def. 6.2(2) and (3) apply here. Letting j be the index of the island with $\{\ell\} = \mathit{dom}(\mathit{Oh}^0_j) = \mathit{dom}(\mathit{Oh}_j)$, we must show both $\mathit{Oh}^0_j \not\rightsquigarrow \mathit{Rh}^0_i$ (for $i \neq j$) and $\mathit{Oh}^0_j \not\rightsquigarrow^{\overline{g}} \mathit{Rh}^0_j$. By typing, $\mathit{loctype}\,\ell \leq C$, so $C \leq \mathit{Own}$ or $\mathit{Own} \leq C$ by the tree property of $\leq$. We argue by cases on C.

—$\mathit{Own} < C$. By $\mathit{Own} \not\geq \mathit{Rep}$, we have $C \not\leq \mathit{Rep}$ so confinement of e_2 at C yields $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$. Thus $\mathit{Oh}^0_j \not\rightsquigarrow \mathit{Rh}^0_i$ and $\mathit{Oh}^0_j \not\rightsquigarrow^{\overline{g}} \mathit{Rh}^0_j$.

—$C \leq \mathit{Own}$. By confinement of e_2, if $d \in \mathit{locs}(\mathit{Rep}{\downarrow})$ then $d \in \mathit{dom}(\mathit{Rh}^0_j)$ so $\mathit{Oh}^0_j \not\rightsquigarrow \mathit{Rh}^0_i$ for $i \neq j$. If $C = \mathit{Own}$ then, by the typing rule for field update, f is in the private fields $\overline{g}$ of Own, so the update cannot violate $\mathit{Oh}^0_j \not\rightsquigarrow^{\overline{g}} \mathit{Rh}^0_j$. If $C < \mathit{Own}$ then $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ because if d is a rep then there would be no confining partition, contradicting confinement of h_0 which holds by confinement of S.

—$\mathit{loctype}\,\ell \leq \mathit{Rep}$. Def. 6.2(4) applies in this case: we need to show $\mathit{Rh}^0_j \not\rightsquigarrow \mathit{Oh}^0_i * \mathit{Rh}^0_i$ where $i \neq j$ and j is the island for ℓ in the partition of h. By typing, $\mathit{loctype}\,\ell \leq C$, hence $C \leq \mathit{Rep}$ or $\mathit{Rep} < C$. But if $\mathit{Rep} < C$ then $C \not\geq \mathit{Own}$ and the confinement condition for e_1 (Def. 6.6(1)) at C contradicts $\mathit{loctype}\,\ell \leq \mathit{Rep}$, so we have $C \leq \mathit{Rep}$. Now confinement of e_2 yields $d \in \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow}) \Rightarrow d \in \mathit{dom}(\mathit{Oh}_j * \mathit{Rh}_j)$. This proves $\mathit{Rh}^0_j \not\rightsquigarrow \mathit{Oh}^0_i * \mathit{Rh}^0_i$, because $\mathit{dom}(\mathit{Oh}_j * \mathit{Rh}_j) = \mathit{dom}(\mathit{Oh}^0_j * \mathit{Rh}^0_j)$.

Case $\Gamma \vdash x := \mathbf{new}\ B$. In the semantic definition, $h_1 = [h \mid \ell \mapsto [\mathit{fields}\,B \mapsto \mathit{defaults}]]$ where $\ell = \mathit{fresh}(B, h)$. Define $\mathit{Bh} = [\ell \mapsto [\mathit{fields}\,B \mapsto \mathit{defaults}]]$ so $h_1 = h * \mathit{Bh}$. Let $\eta_1 = [\mathsf{self} \mapsto \ell]$. Next, we argue that $h \trianglelefteq h_1$ and $\mathit{conf}\,B\,(h_1, \eta_1)$. Because h is closed, ℓ is not in the range of any object state in h. To construct an extending partition it suffices to deal with the new object, as its addition cannot violate confinement of existing objects. (This would not be the case if dangling pointers were allowed, unless further restrictions are imposed on *fresh*.) We define the extension and argue by cases on B.

—$B \not\leq \mathit{Own} \wedge B \not\leq \mathit{Rep}$. For a confining partition of h_1 we extend that for h by defining $\mathit{Ch}^0 = \mathit{Ch} * \mathit{Bh}$ and using the given partition of owner islands. Because *defaults* contains no locations, this is a confining partition and we have $\mathit{conf}\,B\,(h_1, \eta_1)$.

—$B \leq \mathit{Own}$. We extend the partition by adding an island $\mathit{Oh}^0_{k+1} * \mathit{Rh}^0_{k+1}$ with $\mathit{Oh}^0_{k+1} = \mathit{Bh}$ and $\mathit{Rh}^0_{k+1} = \varnothing$. This is a confining partition because *defaults* has no locations and we have $\mathit{conf}\,B\,(h_1, \eta_1)$ because $\mathit{rng}\,\eta_1$ has no reps.

—$B \leq \mathit{Rep}$. We can obtain a confining extension by adding Bh to any of the Rh_i, as *defaults* has no locations. As $\mathit{rng}\,\eta_1 = \{\ell\}$, we have $\mathit{conf}\,B\,(h_1, \eta_1)$ by definition.[16]

[16]In this case we have $C \leq \mathit{Own}$ or $C \leq \mathit{Rep}$, as otherwise the command would not be confined. To show $\mathit{conf}\,C\,(h_1, \eta)$ in this case, we would have to we put ℓ in Rh_j, choosing j such that $\eta\,\mathsf{self}$ is in the jth island. But we are only showing the extension of the partition for this lemma. For soundness of the static analysis, we do have to show $\mathit{conf}\,C\,(h_1, \eta)$.

This concludes the argument for $h \trianglelefteq h_1$ and $\mathit{conf}\, B\,(h_1, \eta_1)$. These let us apply Lemma 6.15 for *constrB* to get $h_1 \trianglelefteq h_0$ where $h_0 = [\![\mathsf{self} : B \vdash \mathit{constr}\, B : \mathbf{con}]\!]\mu(h_1, \eta_1)$. Then $h \trianglelefteq h_0$ by transitivity of $\trianglelefteq$.

Case $\Gamma \vdash x := e.m(\overline{e})$. As $e.m(\overline{e})$ is confined, its argument values are confined. Thus we can obtain the result directly from the semantics of $e.m(\overline{e})$ and confinement of μ —which explicitly stipulates $h \trianglelefteq h_0$.

The remaining cases are straightforward. See Appendix. □

Theorem 6.17 Suppose that CT is confined. Then the semantics $[\![CT]\!]$ is confined, as is each μ_j in the approximation chain used to define it.

The proof uses fixpoint induction, which is only sound for inclusive predicates, i.e., those closed under limits of ascending chains. For confinement of method environments the definition is given pointwise, ultimately unfolding to the property that the semantics of each method body preserves confinement. This definition, as well as the one for the simulation $\mathcal{R}$ later, is in the usual form of logical relations. By the structure of the definition, and continuity of the semantics, the property is an inclusive predicate.[17]

Proof. Confinement of $[\![CT]\!]$ follows by fixpoint induction from confinement of μ_i for all i, which we show by induction on i. The base case holds because $\mu_0 Cm = \lambda(h, \eta) \bullet \bot$, for any C, m, and this is confined by definition.

For the induction step, suppose $\mathit{conf}\ \mu_i$, to show $\mathit{conf}\ \mu_{i+1}$. Consider an arbitrary m. We argue for all C with $\mathit{mtype}(m, C)$ defined, by induction on method depth (Def. 4.7) of C for m. The base case is C such that $\mathit{depth}(m, C) = 0$. In this case, $CT(C)$ has a declaration

$$T\ m(\overline{T}\ \overline{x})\{S\}\ .$$

Suppose $\mathit{conf}\, C\,(h, \eta)$ and $\mu_{i+1} Cm(h, \eta) \neq \bot$. Let $(h_0, d) = \mu_{i+1} Cm(h, \eta)$, which by definition of μ_{i+1} is obtained as

$$\begin{aligned} \eta_1 &= [\eta \mid \mathsf{result} \mapsto \mathit{default}] \\ (h_0, \eta_0) &= [\![\overline{x} : \overline{T}, \mathsf{self} : C, \mathsf{result} : T \vdash S]\!]\mu_i(h, \eta_1) \\ d &= \eta_0\, \mathsf{result} \end{aligned}$$

Default values do not violate confinement so $\mathit{conf}\, C\,(h, \eta_1)$. As CT is confined, S and its constituents are confined. By Lemma 6.16 we have $h \trianglelefteq h_0$, so by Lemma 6.13 we have $\mathit{conf}\, C\,(h_0, \eta)$. To show the confinement condition for $\mu_{i+1} Cm$ it remains to deal with the result value d. We have $\mathit{conf}\, C\,(h_0, \eta_0)$ by confinement of S. We argue by cases on C.

—$C \not\leq \mathit{Own} \wedge C \not\leq \mathit{Rep}$. We need $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$, for Def. 6.5(1), and this follows from $\mathit{conf}\, C\,(h_0, \eta_0)$ by Def. 6.4(1).

—$C \leq \mathit{Own}$. We need $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$, and since by typing we have $d \in [\![T_\bot]\!]$, Def. 6.9(3) ensures $T \not\leq \mathit{Rep}$ and hence $d \notin \mathit{locs}(\mathit{Reps}{\downarrow})$. (Note that semantic confinement of η_0 at $C \leq \mathit{Own}$ allows reps, so it is not enough for this case).

[17]See Ploto's notes.

—$C \leq \mathit{Rep}$. Then we need $d \in \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow})$ to imply that d is in the domain of $\mathit{Oh}_j * \mathit{Rh}_j$ for some partition and island j such that $\eta\,\mathsf{self} \in \mathit{dom}(\mathit{Oh}_j * \mathit{Rh}_j)$. This follows from $\mathit{conf}\, C\,(h_0, \eta_0)$ by Def. 6.4(3).

This concludes the base case of the induction on depth.

For the induction step, i.e., $\mathit{depth}(m, C) > 0$, m may be inherited or declared in C. If it is declared in C the argument is the same as for the case $\mathit{depth}(m, C) = 0$ above. Suppose m is inherited in C from B. Now $\mu_{i+1}Cm = \mathit{restr}((\mu_{i+1}Bm), C)$ by definition of μ_{i+1}. By induction on depth $\mu_{i+1}Bm$ satisfies the confinement condition for m, B. To show the condition for $\mu_{i+1}Cm$, suppose $\mathit{conf}\, C\,(h, \eta)$. We claim that $\mathit{conf}\, B\,(h, \eta)$. Using the claim, we argue as follows. If $\mu_{i+1}Bm(h, \eta) \neq \bot$, let $(h_0, d) = \mu_{i+1}Bm(h, \eta)$. By induction on depth we have $\mathit{conf}\, B\,(h_0, \eta)$ and $h \trianglelefteq h_0$. By Lemma 6.13 we obtain $\mathit{conf}\, C\,(h_0, \eta)$. It remains to show the confinement condition for d and to prove the claim. We argue by cases on C.

In the following non-rep cases, the claim holds by Lemma 6.12. To apply the Lemma, we just need to show that $\mathit{rng}\,\eta \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \varnothing$.

—$C \not\leq \mathit{Own} \wedge C \not\leq \mathit{Rep}$. In this case, we have $\mathit{rng}\,\eta \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \varnothing$ by confinement of η at C, Def. 6.4(1).

—$C \leq \mathit{Own} < B$. Then Own inherits m from $B > \mathit{Own}$, so by confinement of the class table, Def. 6.9(4), we have $\overline{T} \not\geq \mathit{Rep}$. Also, $\mathit{Own} \not\geq \mathit{Rep}$, so by Lemma 5.4 we have no reps in $\mathit{rng}\,\eta$.

In the preceding cases, the condition imposed on d by Def. 6.5(1) for class C is $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$. But this same condition is imposed for class B, and it holds by induction on depth. For the remaining cases we prove the claim $\mathit{conf}\, B\,(h, \eta)$ as follows.

—$C < B \leq \mathit{Own}$. Both B and C impose the same condition (Def. 6.4(2)).

—$C < B \leq \mathit{Rep}$. Both C and B impose the same conditions on η (Def. 6.4(3)).

In these two cases the requirement for d at C, Def. 6.5(2) or (1), is the same as for B, so it holds by induction on depth.

The case $C \leq \mathit{Rep} < B$ cannot occur in a confined class table. If m is inherited in $C \leq \mathit{Rep}$ from B then it is inherited in Rep from B, and this is explicitly disallowed in Def. 6.9(5). □

7. FIRST ABSTRACTION THEOREM

This section formulates and proves the central result of the paper. First, we make precise the idea of comparing two class tables that differ only in their implementation of class Own. Then we define basic coupling: a relation between single instances of class Own for the two implementations. This induces the coupling relations for other data types, for heaps containing multiple instances of Own, and for method meanings. Related method meanings have the simulation property: if initial states are coupled, then so are outcomes. The main theorem says that if methods of Own have the simulation property, then so do all methods of all classes.

7.1 Comparing class tables

We compare two implementations of a designated class Own. They can have completely different declarations, so long as methods of the same signatures are present —declared or inherited— in both. They can use different reps, distinguished by class name Rep for one implementation and Rep' for the other. We allow $Rep = Rep'$. For simplicity, we assume that both Rep and Rep' are in each of the two compared class tables.[18]

***Definition* 7.1 (comparable class tables, non-rep classes)** Suppose class names Own, Rep, Rep' are given, such that $Own \not\leq Rep$ and $Own \not\leq Rep'$. We say C is a *non-rep* class iff $C \not\leq Rep$ and $C \not\leq Rep'$. Well formed class tables CT and CT' are *comparable* provided the following hold.

(1) CT and CT' are identical except for their values on Own. (In particular, $CT(Rep) = CT'(Rep)$ and $CT(Rep') = CT'(Rep')$.)
We write $\vdash, \vdash'$ for the typing relations determined by CT, CT' respectively, and similarly for the auxiliary functions, such as $mtype, mtype'$. We also write $[\![-]\!], [\![-]\!]'$ for the respective semantics and assume that the same allocator, $fresh$, is used for both $[\![-]\!]$ and $[\![-]\!]'$.

(2) $super\, Own = super'\, Own$.

(3) For any m, either $mtype(m, Own)$ and $mtype'(m, Own)$ are both undefined or both are defined and equal. □

***Example* 7.2** Let CT be given by Figs. 2 and 3. Let CT' be given by Figs. 4 and 3 together with Observer from Fig. 2. These are comparable. □

Instead of condition (3), one could require that $CT(Own)$ and $CT'(Own)$ declare the same methods. But that would disallow some situations that occur in practice. Suppose class C extends B by adding a method m implemented using calls to methods inherited from B. This might be the easiest way to achieve desired functionality for m, but there could be an alternative data structure that is more efficient for m and for the methods of B. An alternative implementation of C could add that data structure and override the methods of B to use it. One can argue that the program is poorly designed, e.g., because space for attributes of B is wasted in C objects. Better designs are possible. Nonetheless, such examples do arise in practice; allowing them complicates the proof of Theorem 7.20 but none of the other results. The main consequence we need from condition (3) is the following.

Lemma 7.3 If $mtype(m, C)$ is defined then $depth(m, C) = depth'(m, C)$.

PROOF. Straightforward. See Appendix. □

One can imagine a theory in which an owner subclass $C < Own$ has different declarations in CT and CT'. But we are concerned with an abstraction provided by a single class rather than by a collection of classes, so $CT(C) = CT'(C)$ here. In Sect. 7.3 we impose a restriction on owner subclasses that is needed for the first abstraction theorem. The issue is explored in Sect. 8 and the restriction lifted in Sect. 10.

[18]An alternative formulation would consider different declarations of Own together with associated class tables in which Rep or Rep' but not both are declared. But these could be combined into class tables fitting our formulation.

7.2 Coupling relations and simulation

The definitions are organized as follows. A *basic coupling* R is a suitable relation on islands. This induces a family of *coupling relations*, $\mathcal{R}\,\theta$ for each semantic category θ. Then comes the definition of *simulation*, a coupling that is preserved by all methods of Own and established by the constructor.

***Definition* 7.4 (basic coupling)** Given comparable class tables, a *basic coupling* is a binary relation R on heaps —not necessarily closed— such that the following holds: For any h, h', if $R\,h\,h'$ then there is a location ℓ with $loctype\,\ell \leq Own$ and partitions $h = Oh * Rh$ and $h' = Oh' * Rh'$ such that

(1) $dom\ Oh = \{\ell\} = dom\ Oh'$

(2) $dom(Rh) \subseteq locs(Rep{\downarrow})$ and $dom(Rh') \subseteq locs(Rep'{\downarrow})$

(3) $h\ell f = h'\ell f$ for all $f \in dom(fields(loctype\,\ell))$ with $f \notin \overline{g}$ and $f \notin \overline{g}'$, where $\overline{g} = dom(dfields(Own))$ and $\overline{g}' = dom(dfields'(Own))$ □

Example 7.7 below shows why we allow R to act on heaps that are not closed.

Although R is unconstrained for the private fields and reps, condition (3) determines it for fields of subclasses of Own (while allowing R to depend on these fields). Once we have defined the induced relation $\mathcal{R}$, item (3) will be equivalent to the condition $\mathcal{R}\ (type(f, loctype\,\ell))\ (h\ell f)\ (h'\ell f)$.

Because CT and CT' are well formed, the declared field names $\overline{g}$ and $\overline{g}'$ do not occur as fields of subclasses or superclasses of Own. In (3), f ranges over fields of both subclasses and superclasses; excluding $\overline{g}$ and $\overline{g}'$, i.e., $dfields\,Own$, we have $dfields\,C = dfields'\,C$ for all other C. The typing relations $\vdash$ and $\vdash'$ are also the same except for class Own.

***Example* 7.5** Sect. 2.2 discusses this coupling relation:

$$(\mathsf{o.g} = nil = \mathsf{o'.g}) \vee (\mathsf{o.g} \neq nil \neq \mathsf{o'.g} \wedge \mathsf{o.g.f} = \neg(\mathsf{o'.g.f}))\ .$$

For this example we take both Rep and Rep' to be Bool, and Own to be OBool. The displayed formula can be interpreted as relation R which relates h to h' just if either $h = [\ell_1 \mapsto [g \mapsto nil]]$ and $h' = [\ell_1 \mapsto [g \mapsto nil]]$ or else

$$h = [\ell_1 \mapsto [g \mapsto \ell_2], \ell_2 \mapsto [f \mapsto d]] \text{ and } h' = [\ell_1 \mapsto [g \mapsto \ell_3], \ell_3 \mapsto [f \mapsto \neg d]]$$

for some boolean d and locations ℓ_1 in $locs$ OBool and ℓ_2, ℓ_3 in $locs$ Bool. We assume that the class table contains only Bool, OBool, and some client classes. If OBool had subclasses, the relation on their fields would be determined by condition (3) above. □

***Example* 7.6** Sect. 3.1 uses the formula $\mathsf{o.g} = \mathsf{o'.g} \wedge \mathsf{o.g}\ \mathbf{mod}\ 2 = 0$. This can be interpreted as the basic coupling R that relates h to h' just if there is some ℓ with $loctype\ \ell \leq \mathsf{A}$, h and h' have domain $\{\ell\}$, and $h\,\ell\,g = h'\,\ell\,g = 2 \times m$ for some integer $m \geq 0$. □

***Example* 7.7** The Observer examples show why we allow R to relate non-closed heaps. Consider the version in Fig. 2. Here Rep is Node, Own is Observable, and there is a client class Observer. Fig. 5 illustrates two instances of this simple data structure. Fig. 4 gives code for an alternative version which uses an extra node as

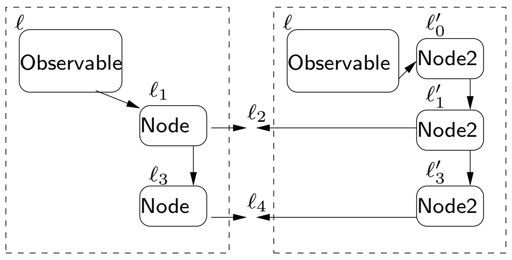

Fig. 7. Basic coupling example. Labels indicate locations as described in Example 7.7. Note dangling pointers ℓ_2 and ℓ_4 and sentinel node ℓ'_0.

sentinel for the list. The sentinel does not point to an Observer. Fig. 7 depicts a corresponding pair of heaps for the two alternatives, using arrows without destination objects to indicate dangling pointers. Upon initialization of an Observable, there are no installed Observers, so for the version of Fig. 2 we should have $\mathsf{fst} = nil$. But in the alternative version, this should correspond to snt holding the location of a Node with $\mathsf{ob} = nil$. This is established by the constructor in Fig. 4. An attempt at formalizing the correspondence is as follows:

$$(\mathsf{o.fst} = nil = \mathsf{o'.snt.nxt}) \vee (\mathsf{o.fst} \neq nil \neq \mathsf{o'.snt.nxt} \wedge \alpha(\mathsf{o.fst}) = \alpha'(\mathsf{o'.snt.nxt})$$

where α, α' are functions that yield the list of locations in the ob fields of successive nodes. But how should this formula be interpreted if, say, $\mathsf{o'.snt} = nil$ or there is sharing such as a chain with cyclic tail? Separation logic [Reynolds 2001] offers a precise way to formulate such definitions but its development is at an early stage. We simply sketch the coupling in terms of semantics: $R\,h\,h'$ iff either h and h' have the form

$$h = [\,\ell \mapsto [\mathsf{fst} \mapsto nil]]$$
$$h' = [\,\ell \mapsto [\mathsf{snt} \mapsto \ell'_0],\ \ell'_0 \mapsto [\mathsf{ob} \mapsto nil, \mathsf{nxt} \mapsto nil]]$$

or they have the form

$$h = [\,\ell \mapsto [\mathsf{fst} \mapsto \ell_1],\ \ell_1 \mapsto [\mathsf{ob} \mapsto \ell_2, \mathsf{nxt} \mapsto \ell_3],\ \ell_3 \mapsto [\mathsf{ob} \mapsto \ell_4, \mathsf{nxt} \mapsto \ldots], \ldots]$$
$$h' = [\,\ell \mapsto [\mathsf{snt} \mapsto \ell'_0],\ \ell'_0 \mapsto [\mathsf{ob} \mapsto nil, \mathsf{nxt} \mapsto \ell'_1],$$
$$\ell'_1 \mapsto [\mathsf{ob} \mapsto \ell_2, \mathsf{nxt} \mapsto \ell'_3],\ \ell'_3 \mapsto [\mathsf{ob} \mapsto \ell_4, \mathsf{nxt} \mapsto \ldots], \ldots]$$

for some locations ℓ in $locs(\mathsf{Observable})$, $\ell_1, \ell_3, \ldots$ in $locs(\mathsf{Node})$, $\ell'_0, \ell'_1, \ell'_3, \ldots$ in $locs(\mathsf{Node2})$, and $\ell_2, \ell_4, \ldots$ in $locs(\mathsf{Observer}{\downarrow})$.

Note that the owners are at the same location, ℓ, as are the referenced client objects at $\ell_2, \ell_4, \ldots$. No correspondence is required between locations $\ell_1, \ell_3, \ldots$ and $\ell'_0, \ell'_1, \ell'_3, \ldots$ of reps. $\square$

A basic coupling induces a relation $\mathcal{R}\,Heap$ on arbitrary heaps by requiring that they have confining partitions such that islands can be put in correspondence so that pairs are related by R. The formal definition uses the induced relation $\mathcal{R}\,(state\,C)$ for object states of non-rep classes $C \not\leq Own$, and this in turn is defined in terms of $\mathcal{R}\,C$ for non-rep classes $C \not\leq Own$. For uniformity, we give the definition of $\mathcal{R}$ for all θ, but forcing the case for $\theta = state\,Own$ to be false, as the compared states have different fields. Aside from the ramifications of heap confinement, the definition is induced in the standard way for logical relations.

***Definition* 7.8 (coupling relation, $\mathcal{R}\,\theta$)** In the context of a basic coupling with given relation R, we define for each θ a relation $\mathcal{R}\,\theta \subseteq [\![\theta]\!] \times [\![\theta]\!]'$ as follows.

For heaps h, h', we define $\mathcal{R}\ \mathit{Heap}\ h\ h'$ iff there are confining partitions of h, h', with the same number n of owner islands, such that

—$R\ (Oh_i * Rh_i)\ (Oh'_i * Rh'_i)$ for all i in $1..n$
—$\mathit{dom}(Ch) = \mathit{dom}(Ch')$
—$\mathcal{R}\ (\mathit{state}\ (\mathit{loctype}\ \ell))\ (h\ell)\ (h'\ell)$ for all $\ell \in \mathit{dom}(Ch)$

For other categories θ we define $\mathcal{R}\,\theta$ as follows.

$$
\begin{array}{ll}
\mathcal{R}\ \mathbf{bool}\ d\ d' & \Leftrightarrow\ d = d' \\
\mathcal{R}\ \mathbf{unit}\ d\ d' & \Leftrightarrow\ d = d' \\
\mathcal{R}\ C\ d\ d' & \Leftrightarrow\ d = d' \\
\mathcal{R}\ \Gamma\ \eta\ \eta' & \Leftrightarrow\ \forall x \in \mathit{dom}\,\Gamma \bullet \mathcal{R}\ (\Gamma x)\ (\eta x)\ (\eta' x) \\
\mathcal{R}\ (\mathit{state}\,C)\ s\ s' & \Leftrightarrow \\
\quad C \not\leq \mathit{Own} \wedge \forall f \in \mathit{dom}(\mathit{fields}\,C) \bullet \mathcal{R}\ (\mathit{type}(f,C))\ (s\ f)\ (s'\ f) & \\
\mathcal{R}\ (\theta_\bot)\ \alpha\ \alpha' & \Leftrightarrow\ (\alpha = \bot = \alpha') \vee (\alpha \neq \bot \neq \alpha' \wedge \mathcal{R}\ \theta\ \alpha\ \alpha') \\
\mathcal{R}\ (\mathit{Heap} \otimes \Gamma)\ (h,\eta)\ (h',\eta') & \Leftrightarrow\ \mathcal{R}\ \mathit{Heap}\ h\ h' \wedge \mathcal{R}\ \Gamma\ \eta\ \eta' \\
\mathcal{R}\ (\mathit{Heap} \otimes T)\ (h,d)\ (h',d') & \Leftrightarrow\ \mathcal{R}\ \mathit{Heap}\ h\ h' \wedge \mathcal{R}\ T\ d\ d' \\
\mathcal{R}\ (C, \overline{x}, \overline{T} \to T)\ d\ d' & \Leftrightarrow\ \forall (h,\eta) \in [\![\mathit{Heap} \otimes \Gamma]\!], (h',\eta') \in [\![\mathit{Heap} \otimes \Gamma]\!]' \bullet \\
\quad \mathcal{R}\ (\mathit{Heap} \otimes \Gamma)\ (h,\eta)\ (h',\eta') \wedge \mathit{conf}\ C\ (h,\eta) \wedge \mathit{conf}\ C\ (h',\eta') & \\
\quad \Rightarrow \mathcal{R}\ (\mathit{Heap} \otimes T)_\bot\ (d(h,\eta))\ (d'(h',\eta')) & \\
\quad \text{where } \Gamma = [\overline{x} \mapsto \overline{T}, \mathsf{self} \mapsto C] & \\
\mathcal{R}\ \mathit{MEnv}\ \mu\ \mu' & \Leftrightarrow\ \forall C, m\ \bullet (C \text{ is non-rep}) \wedge (\mathit{mtype}(m,C) \text{ is defined}) \\
\quad \Rightarrow \mathcal{R}\ (C, \mathit{pars}(m,C), \mathit{mtype}(m,C))\ (\mu\,C\,m)\ (\mu'\,C\,m) &
\end{array}
$$

The gist of the abstraction theorem is that if the methods of Own are related by $\mathcal{R}$ then all methods are. We can now express this conclusion as $\mathcal{R}\ \mathit{MEnv}\ [\![CT]\!]\ [\![CT']\!]'$. To express the antecedent, note that the relation applicable to a method m of Own is $\mathcal{R}\ (\mathit{Own}, \overline{x}, \overline{T} \to T)$ where $\mathit{mtype}(m, \mathit{Own}) = \overline{T} \to T$ and $\mathit{pars}(m, \mathit{Own}) = \overline{x}$. The definition of $\mathcal{R}\ (C, \overline{x}, \overline{T} \to T)$ quantifies over confined initial states but does not require confinement of outcomes.[19] The antecedent will also take into account that methods may be declared or inherited.

Although the definition is technically intricate, the core idea is the extension of a basic coupling, for a single owner instance, to a heap containing potentially many owners. This idea is given straightforward expression using heap partitions. By contrast, sharing of representations between owners would require a more complicated form of extension (see Sect. 12).

We aim to define per-instance simulations, and in particular the establishment of such a relation by a constructor of class Own on a single island. But to formulate this semantically we describe the constructor's action on a heap in which other

[19]One might think that $\mathcal{R}\ \mathit{Heap}$ could be defined in terms of admissible partitions without the assumption of confinement. But because partitions are not unique this leads to difficulties: a heap could be confined with respect to one partition but related with respect to another.

islands may be present. The reason is that there is not an easy way to connect a constructor's action on a small heap with its action on a larger one (see Footnote 15).

***Definition* 7.9 (simulation)** A simulation is a coupling $\mathcal{R}$ such that the following hold.

(1) (constructors of Own establish $\mathcal{R}$) For any $\ell \in locs(Own\!\downarrow)$, any h, h' with $\mathcal{R}\ Heap\ h\ h'$, and any μ, μ', let

$$\begin{aligned} h_1 &= [h \mid \ell \mapsto [\mathit{fields}(loctype\ \ell) \mapsto defaults]] \\ h'_1 &= [h' \mid \ell' \mapsto [\mathit{fields}(loctype\ \ell') \mapsto defaults']] \\ h_0 &= [\![\mathsf{self} : (loctype\ \ell) \vdash constr(loctype\ \ell) : \mathbf{con}]\!]\mu(h_1, [\mathsf{self} \mapsto \ell]) \\ h'_0 &= [\![\mathsf{self} : (loctype\ \ell) \vdash' constr(loctype\ \ell) : \mathbf{con}]\!]\mu'(h'_1, [\mathsf{self} \mapsto \ell]) \end{aligned}$$

Then $R\ h_0\ h'_0$.

(2) (methods of Own preserve $\mathcal{R}$) For any[20] confined μ, μ' such that $\mathcal{R}\ MEnv\ \mu\ \mu'$, the following conditions hold for every m with $mtype(m, Own)$ defined, where $\overline{x} = pars(m, Own)$ and $\overline{T} \to T = mtype(m, Own)$.

(a) $\mathcal{R}\ (Own, \overline{x}, \overline{T} \to T)\ ([\![M]\!]\mu)\ ([\![M']\!]'\mu')$
if m has declaration M in $CT(Own)$ and M' in $CT'(Own)$

(b) $\mathcal{R}\ (Own, \overline{x}, \overline{T} \to T)\ ([\![M]\!]\mu)\ (restr([\![M_B]\!]'\mu', Own))$
if m has declaration M in $CT(Own)$ and is inherited from B in $CT'(Own)$, with M_B the declaration of m in B

(c) $\mathcal{R}\ (Own, \overline{x}, \overline{T} \to T)\ (restr([\![M_B]\!]\mu, Own))\ ([\![M']\!]'\mu')$
if m has declaration M' in $CT'(Own)$ and is inherited from B in $CT(Own)$, with M_B the declaration of m in B

In the case where constructors in Own and its subclasses are **skip**, condition (1) simply says that the default values are related. Note that it also precludes aborting constructors, as R applies to heaps but not to $\bot$; this is convenient but not necessary.

The following properties are straightforward consequences of the definition.

Lemma 7.10 For all h, h' and all locations $\ell \notin locs(Rep\!\downarrow, Rep'\!\downarrow)$, if $\mathcal{R}\ Heap\ h\ h'$ then $\ell \in dom\, h \Leftrightarrow \ell \in dom\, h'$. $\square$

Lemma 7.11 $[\![T]\!] = [\![T]\!]'$ for all T, and $[\![\Gamma]\!] = [\![\Gamma]\!]'$ for all Γ. $\square$

Lemma 7.12 For any data type T, $\mathcal{R}\ T$ is the identity relation on $[\![T]\!]$ and $\mathcal{R}\ T_\bot$ is the identity relation on $[\![T_\bot]\!]$. $\square$

Lemma 7.13 If $\overline{U} \le \overline{T}$ and $\mathcal{R}\ \overline{U}\ \overline{d}\ \overline{d'}$ then $\mathcal{R}\ \overline{T}\ \overline{d}\ \overline{d'}$. $\square$

7.3 Restricting reps in owner subclasses

The preceding properties express a strong connection between locations for related heaps. To ensure that this connection is preserved by object construction, we shall assume the allocator is parametric. But it is not reasonable to require that related heaps have the same rep locations, so parametricity cannot be exploited for reps.

[20]In fact it suffices to consider μ_i, μ'_i in the approximation chains in the definition of $[\![CT]\!]$ (resp. $[\![CT']\!]'$).

As a result, the present form of simulation is not adequate for construction of reps in subclasses of Own, although such construction is allowed by confinement. The first abstraction theorem depends on an assumption expressed in the following terms.

***Definition* 7.14 (new rep in sub-owner)** We say CT *has a new rep in a sub-owner* if, for some $B \leq Rep$ or $B \leq Rep'$, an object construction **new** B occurs in some method declaration in a class $C < Own$.

If CT has no new reps in sub-owners then neither does a comparable CT' (and vice versa). The examples in Sects. 2 and 3 have no new reps in sub-owners; examples which do are given in Sect. 8.

In the rest of Sect. 7 we make the following assumption. It is used in the proof of Lemma 7.23 on which the first abstraction theorem depends. For the second abstraction theorem the second sentence of the assumption will be dropped.

Assumption 7.15 First, CT and CT' are confined class tables for which a simulation $\mathcal{R}$ is given. Second, CT has no new reps in sub-owners and the allocator is parametric in the sense of Def. 5.1.

7.4 Identity extension

A typical formulation of identity extension is that $\mathcal{R}\,T$ is the identity on any type T for which it is the identity on all primitive types b that occur in T. The reason is that no value of type b can occur in a value of type T if b does not occur in T —but this fails with extensible records and structural subtyping, and with procedures that may have global variables [Naumann 2002]. It can be made to work using name-based (declaration) subclassing [Cavalcanti and Naumann 2002]: in the context of a complete class table, one can consider the classes that have no attributes with subclasses in which b occurs. For our purposes here it is enough to deal with the heap.

In our language, $\mathcal{R}\ T$ is the identity for every data type T (Lemma 7.12), but that is only because the interesting data is in the heap —which is not typed at all.[21] In general, $[\![state\,Own]\!] \neq [\![state\,Own]\!]'$ and $\mathcal{R}(state\,Own)$ is not the identity. Related heaps can contain owner objects with different states that may point to completely different rep objects. But consider executing a method on an object o from whose fields no Own objects are reachable, i.e., Own objects are not part of the representation of o. The resulting heap may contain Own objects that were assigned to local variables, but if the method is confined then those objects are unreachable in the final state.

***Definition* 7.16 (garbage collection, Own-free)** For a set or list $\overline{d}$ of values, define the heap $gc(\overline{d}, h)$ to be the restriction of h to cells reachable from $\overline{d}$. For $(h, \eta) \in [\![Heap \otimes \Gamma]\!]$, define $collect(h, \eta) = (gc(rng\,\eta, h), \eta)$. Extend $collect$ to $[\![(Heap \otimes \Gamma)_\bot]\!]$ by $collect\,\bot = \bot$.

Say h is *Own-free* just if $dom\,h \cap locs(Own{\downarrow}) = \varnothing$ and η is *Own-free* just if $rng\,\eta \cap locs(Own{\downarrow}) = \varnothing$. $\square$

[21] Nor would we want to impose a typing system on the heap, as it would likely preclude unbounded data structures [Grossman et al. 2000].

Lemma 7.17 (identity extension) Suppose $\mathcal{R}\ (Heap \otimes \Gamma)\ (h, \eta)\ (h', \eta')$ and $\Gamma\,\mathsf{self}$ is non-rep. Let (h, η) and (h', η') be confined at $\Gamma\,\mathsf{self}$. If $collect(h, \eta)$ and $collect(h', \eta')$ are Own-free then $collect(h, \eta) = collect(h', \eta')$.

PROOF. In confined heaps, reps are only reachable from owners. Now the argument is a straightforward induction using the definition of $\mathcal{R}$. □

Lemma 7.18 For any $\mathcal{R}$ given by Def. 7.8 from a basic coupling, if $h \in [\![Heap]\!]$ is Own-free then $\mathcal{R}\ Heap\ h\ h$. If, in addition, $(h, \eta) \in [\![Heap \otimes \Gamma]\!]$ and $rng\,\eta$ is Own-free then $\mathcal{R}\ (Heap \otimes \Gamma)\ (h, \eta)\ (h, \eta)$.

PROOF. If h is Own-free and confined then it has no reps; its admissible partition is a single block, the clients. For such a heap it is immediate from the definition of $\mathcal{R}$ that $\mathcal{R}\ Heap\ h\ h$. If $rng\,\Gamma$ is Own-free then $\mathcal{R}\ \Gamma\ \eta\ \eta$ is also direct from the definition. □

7.5 Abstraction theorem

The main theorem says that if methods of Own preserve the coupling relation then so do all methods.[22] The proof depends on lemmas for constructors and commands. These are given following the theorem. The other main ingredient for the proof is the following connection between $\mathcal{R}$ and the semantics of inherited methods.

Lemma 7.19 Suppose C and all class names in $\overline{T}$ are non-rep, and $B < C$. If $\mathcal{R}\ (C, \overline{x}, \overline{T} \to T)\ d\ d'$ then $\mathcal{R}\ (B, \overline{x}, \overline{T} \to T)\ (restr(d, B))\ (restr(d', B))$ where $restr$ is the restriction to global states of B (see Def. 5.5).

PROOF. Straightforward, using Lemma 5.7(1). See Appendix. □

Theorem 7.20 (abstraction) If CT and CT' are confined and $\mathcal{R}$ is a simulation (as per Assumption 7.15), then $\mathcal{R}\ MEnv\ [\![CT]\!]\ [\![CT']\!]'$.

PROOF. We show that the relation holds for each step in the approximation chain in the semantics of class tables (see the definition of μ_i following Def. 5.5). That is, we show by induction on i that

$$\mathcal{R}\ MEnv\ \mu_i\ \mu'_i \quad \text{for every } i \in \mathbb{N}\ .$$

The result $\mathcal{R}\ MEnv\ [\![CT]\!]\ [\![CT']\!]'$ then follows by fixpoint induction, as $[\![CT]\!]$ and $[\![CT']\!]'$ are defined to be the fixpoints of these ascending chains. Admissibility of fixpoint induction is discussed preceding the proof of Theorem 6.17.

For the base case, we have $\mathcal{R}\ (C, pars(m, C), mtype(m, C))\ (\mu_0\, C\, m)\ (\mu'_0\, C\, m)$ for every m, C because $\lambda(h, \eta) \bullet \bot$ relates to itself.

For the induction step, suppose

$$\mathcal{R}\ MEnv\ \mu_i\ \mu'_i \quad . \tag{$*$}$$

We must show $\mathcal{R}\ MEnv\ \mu_{i+1}\ \mu'_{i+1}$, that is, for every non-rep C and every m with $mtype(m, C)$ defined:

[22]Readers familiar with Reynolds [1984] may expect that, as our language has fixpoints, the result only holds for couplings that are $\bot$-strict and join-complete. But our basic couplings have this property, trivially, because heaps are ordered by equality. The induced coupling is strict and join-complete by construction.

$$\mathcal{R}\ (C, \overline{x}, \overline{T} {\rightarrow} T)\ (\mu_{i+1}\, C\, m)\ (\mu'_{i+1}\, C\, m) \qquad (\dagger)$$

where $\overline{x} = \mathit{pars}(m, C)$ and $\overline{T} {\rightarrow} T = \mathit{mtype}(m, C)$. For arbitrary m we show (†) for all C with $\mathit{mtype}(m, C)$ defined, using a secondary induction on $\mathit{depth}(m, C)$. We have $\mathit{depth}'(m, C) = \mathit{depth}(m, C)$ from Lemma 7.3.

The base case is the unique C with $\mathit{depth}(m, C) = 0$; here m is declared in both $CT(C)$ and $CT'(C)$. We go by cases on C. If $C = \mathit{Own}$, we get (†) from the assumption that $\mathcal{R}$ is a simulation. In detail: Using (∗) and Def. 7.9(2a) we get

$$\mathcal{R}\ (\mathit{Own}, \overline{x}, \overline{T} {\rightarrow} T)\ ([\![M]\!]\mu_i)\ ([\![M']\!]'\mu'_i)\ ,$$

whence (†) by definition of μ_{i+1} and μ'_{i+1}. The other case is C a non-rep class different from Own. Then by Def. 7.1(1) of comparable class tables we have $CT(C) = CT'(C)$ and in particular both class tables have the same declaration

$$T\ m(\overline{T}\ \overline{x})\ \{S\}\ .$$

To show (†), suppose $\mathit{conf}\, C\, (h, \eta)$, $\mathit{conf}\, C\, (h', \eta')$, $\mathcal{R}\ \mathit{Heap}\ h\ h'$, and $\mathcal{R}\ \Gamma\ \eta\ \eta'$, where $\Gamma = \overline{x} : \overline{T}, \mathsf{self} : C$. Then by Lemma 7.23 below, using $\mathcal{R}\ \mathit{MEnv}\ \mu_i\ \mu'_i$, the results from S are related. That is, either $[\![\Gamma \vdash S]\!]\mu_i(h, \eta) = \bot = [\![\Gamma \vdash' S]\!]'\mu'_i(h', \eta')$ or neither is $\bot$. In the latter case, (h_0, η_0) is related to (h'_0, η'_0) where $(h_0, \eta_0) = [\![\Gamma \vdash S]\!]\mu_i(h, \eta)$ and $(h'_0, \eta'_0) = [\![\Gamma \vdash' S]\!]'\mu'_i(h', \eta')$. Then, by definition of $\mathcal{R}\,\Gamma$, $\mathcal{R}\ \Gamma\ \eta_0\ \eta'_0$ implies $\mathcal{R}\ T\ (\eta_0\, \mathsf{result})\ (\eta'_0\, \mathsf{result})$. Thus (†) holds by definition of μ_{i+1} and μ'_{i+1}. This concludes the base case of the secondary induction. The appeal to Lemma 7.23 depends on $\mathit{conf}\ \mu_i$ and $\mathit{conf}\ \mu'_i$ which holds by Assumption 7.15 and Theorem 6.17.

For the induction step, suppose $\mathit{depth}(m, C) > 0$. By induction on depth we have, by definition of depth,

$$\mathcal{R}\ (C, \overline{x}, \overline{T} {\rightarrow} T)\ (\mu_{i+1}\, (\mathit{super}\, C)\, m)\ (\mu'_{i+1}\, (\mathit{super}\, C)\, m)\ . \qquad (\ddagger)$$

If m is declared in both $CT(C)$ and $CT'(C)$ then the argument is the same as in the base case of the secondary induction. If m is inherited in both $CT(C)$ and $CT'(C)$ then (‡) follows from (†) because the semantics defines $\mu_{i+1}\, C\, m$ by restriction from $\mu'_{i+1}\, (\mathit{super}\, C)\, m$ and restriction preserves simulation. (This is Lemma 7.19, which is applicable because if $B > \mathit{Own}$ and m is inherited in Own from B then $\overline{T} \not\leq \mathit{Rep}$ and $\overline{T} \not\leq \mathit{Rep}'$ by confinement of CT, CT', Def. 6.9(4).) The remaining possibility is that m is declared in $CT(C)$ and inherited in $CT'(C)$ from some B (or the other way around). Then $C = \mathit{Own}$, by comparability of CT and CT'. Using Def. 7.9(2b) and (∗) we get

$$\mathcal{R}\ (\mathit{Own}, \overline{x}, \overline{T} {\rightarrow} T)\ ([\![M]\!]\mu_i)\ (\mathit{restr}([\![M_B]\!]\mu'_i, \mathit{Own}))$$

and thus (†) by definition of μ_{i+1} and μ'_{i+1}. □

Lemma 7.21 (establishment by constructors) Let μ and μ' be any method environments. Then the following holds for any non-rep class $C \neq \mathit{Own}$.

For all $(h, \ell) \in [\![\mathit{Heap} \otimes C]\!]$ and $(h', \ell') \in [\![\mathit{Heap} \otimes C]\!]$, if $\mathit{conf}\, C\, (h, \eta_1)$, $\mathit{conf}\, C\, (h', \eta'_1)$ and $\mathcal{R}\ (\mathit{Heap} \otimes C)\ (h, \ell)\ (h', \ell')$ then $\mathcal{R}\ \mathit{Heap}_\bot\ h_0\ h'_0$, where

$$\begin{array}{ll} \eta_1 = [\mathsf{self} \mapsto \ell] & h_0 = [\![\mathsf{self} : C \vdash \mathit{constr}\, C : \mathbf{con}]\!]\mu(h, \eta_1) \\ \eta'_1 = [\mathsf{self} \mapsto \ell'] & h'_0 = [\![\mathsf{self} : C \vdash' \mathit{constr}\, C : \mathbf{con}]\!]\mu'(h', \eta'_1) \end{array}$$

Proof. By well founded induction on C with respect to $\ll$. Suppose $\mathit{conf}\, C\,(h,\eta)$, $\mathit{conf}\, C\,(h',\eta')$, and $\mathcal{R}\ (\mathit{Heap}\otimes C)\ (h,\ell)\ (h',\ell')$. Let $h_1 = [\![\mathsf{self}: C \vdash S : \mathbf{con}]\!]\mu(h,\eta)$ be as in the semantics of S as a constructor, and similarly for h'_1. If $\mathit{super} C = \mathbf{Object}$ then $h_1 = h$ and thus $\mathcal{R}\ \mathit{Heap}\ h_1\ h'_1$ by hypothesis. Otherwise, $h_1 = [\![\mathsf{self}: \mathit{super} C \vdash \mathit{constr}(\mathit{super} C) : \mathbf{con}]\!]\mu(h,\eta)$ and we get $\mathcal{R}\ \mathit{Heap}\ h_1\ h'_1$ by the induction hypothesis noting that $\mathit{super} C \ll C$ by Lemma 4.9. It follows that $\mathcal{R}\ (\mathit{Heap}\otimes C)\ (h_1,\ell)\ (h'_1,\ell')$.

It remains to show $\mathcal{R}\ \mathit{Heap}_\perp\ h_0\ h'_0$, where h_0, h'_0 are obtained by applying the command semantics of $\mathit{constr} C$ to (h_1,ℓ) and (h'_1,ℓ). This holds because, taking S to be $\mathit{constr} C$ in the claim below, we get $\mathcal{R}\ (\mathit{Heap}\otimes C)_\perp\ (h_0,\eta_0)\ (h'_0,\eta'_0)$ and thus either both outcomes are $\perp$ or $\mathcal{R}\ \mathit{Heap}\ h_0\ h'_0$.

Claim: For all $\mathsf{self}: C \vdash S$ such that S has no method calls and every **new** B in S has $B \sqsubset C$, for all (h,η) and (h',η'), if $\mathit{conf}\, C\,(h,\eta)$, $\mathit{conf}\, C\,(h',\eta')$, and $\mathcal{R}\ (\mathit{Heap}\otimes\Gamma)\ (h,\eta)\ (h',\eta')$ then

$$\mathcal{R}\ (\mathit{Heap}\otimes\Gamma)_\perp\ ([\![\Gamma \vdash S]\!]\mu(h,\eta))\ ([\![\Gamma \vdash' S]\!]'\mu'(h',\eta'))\ .$$

Proof of the claim is by induction on the structure of S. Note that by hypothesis S has no method calls, so μ is not relevant. The argument is exactly the same as in the proof of Lemma 7.23 below, except for the case of **new**. In the proof of Lemma 7.23, the case of **new** appeals to the present Lemma for constructors. To prove the claim for this case, the argument is the same except for appealing to the main induction hypothesis; this is sound because the claim includes the hypothesis that if **new** B occurs in S then $B \sqsubset C$ and thus $B \ll C$ by Lemma 4.9. □

Lemma 7.22 (preservation by expressions) For any non-rep class $C \neq \mathit{Own}$ and any constituent expression $\Gamma \vdash e : T$ of a method declared in C, the following holds: For all $(h,\eta) \in [\![\mathit{Heap}\otimes\Gamma]\!]$ and $(h',\eta') \in [\![\mathit{Heap}\otimes\Gamma]\!]'$, if $\mathit{conf}\, C\,(h,\eta)$, $\mathit{conf}\, C\,(h',\eta')$, and $\mathcal{R}\ (\mathit{Heap}\otimes\Gamma)\ (h,\eta)\ (h',\eta')$ then

$$\mathcal{R}\ (T_\perp)\ ([\![\Gamma \vdash e : T]\!](h,\eta))\ ([\![\Gamma \vdash' e : T]\!]'(h',\eta'))\ .$$

Proof. By induction on the derivation of $\Gamma \vdash e : T$. For each case of e, we give an argument assuming that $\Gamma, C, T, \eta, \eta', h, h'$ satisfy the hypotheses of the Lemma.

Case $\Gamma \vdash (B)\ e : B$. Induction on e yields that $\mathcal{R}\ D_\perp\ \ell\ \ell'$ (or else both denotations of e are $\perp$). By confinement of e, as $C \neq \mathit{Own}$ and C is non-rep, we have $\ell \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ and $\ell' \notin \mathit{locs}(\mathit{Rep}'{\downarrow})$. Thus, $\ell' = \ell$ by Lemma 7.12. Hence either both semantics yield ℓ, whence $\mathcal{R}\ B_\perp\ \ell\ \ell$, or both yield $\perp$ and again $\mathcal{R}\ B_\perp\ \perp\ \perp$.

Case $\Gamma \vdash e$ **is** $B : \mathbf{bool}$. The argument is similar to that for type cast.

Case $\Gamma \vdash e.f : T$. By induction on e we have $\mathcal{R}\ C_\perp\ \ell\ \ell'$, hence $\ell = \ell'$ by Lemma 7.12. In the non-$\perp$ case, $\ell \neq \mathit{nil}$. By closure of the heaps, $\ell \in \mathit{dom}\, h$ and $\ell \in \mathit{dom}\, h'$. We consider cases on whether $C < \mathit{Own}$. Consider confining partitions $(\mathit{Ch} * \mathit{Oh}_1 * \mathit{Rh}_1 \ldots) = h$ and $(\mathit{Ch}' * \mathit{Oh}'_1 * \mathit{Rh}'_1 \ldots) = h'$ that have corresponding islands as in the definition of $\mathcal{R}\ \mathit{Heap}$. In the case $C < \mathit{Own}$, we have $\ell \in \mathit{locs}(\mathit{Own}{\downarrow})$ and hence ℓ in some $\mathit{dom}(\mathit{Oh}_i)$. From $\mathcal{R}\ \mathit{Heap}\ h\ h'$ we have

$$R\ (\mathit{Oh}_i * \mathit{Rh}_i)\ (\mathit{Oh}'_i * \mathit{Rh}'_i)$$

and thus $\ell \in \mathit{dom}(\mathit{Oh}'_i)$ by basic coupling Def. 7.4(1). Since $C \neq \mathit{Own}$, we know by visibility that f is not in the private fields $\overline{g}$ of Own. Thus, as $\mathit{type}(f, \mathit{loctype}\,\ell)) = T$, we have $\mathcal{R}\ T\ (h\ell f)\ (h'\ell f)$ by Def. 7.4(3) and Lemma 7.12.

Finally, in the case $C \nleq \mathit{Own}$ (recall that C is non-rep and $C \neq \mathit{Own}$ by hypothesis, we have $\ell \in \mathit{dom}(\mathit{Ch})$ and hence $\ell \in \mathit{dom}(\mathit{Ch}')$ by definition $\mathcal{R}\ \mathit{Heap}$. Hence $\mathcal{R}\ (\mathit{state}\,(\mathit{loctype}\,\ell))\ (h\ell)\ (h'\ell)$ and thus $\mathcal{R}\ T\ (h\ell f)\ (h'\ell f)$ by definition of $\mathcal{R}\ (\mathit{state}\,(\mathit{loctype}\,\ell))$.

The remaining cases are straightforward. See Appendix. □

Lemma 7.23 (preservation by commands) Suppose $\mathcal{R}$ is a simulation, and moreover μ and μ' are confined method environments such that $\mathcal{R}\ \mathit{MEnv}\ \mu\ \mu'$. Then the following holds for any non-rep class $C \neq \mathit{Own}$. For any constituent command $\Gamma \vdash S$ in a method declaration in $\mathit{CT}(C)$ and any (h, η) and (h', η'), if $\mathit{conf}\,C\,(h, \eta)$, $\mathit{conf}\,C\,(h', \eta')$, and $\mathcal{R}\ (\mathit{Heap} \otimes \Gamma)\ (h, \eta)\ (h', \eta')$ then

$$\mathcal{R}\ (\mathit{Heap} \otimes \Gamma)_\perp\ ([\![\Gamma \vdash S]\!]\mu(h, \eta))\ ([\![\Gamma \vdash' S]\!]'\mu'(h', \eta'))\ .$$

Proof. For any C, the proof is by structural induction on the derivation of $\Gamma \vdash S$.

Case $\Gamma \vdash x := e$. As CT is confined, constituent e of the assignment is confined. So by Lemma 7.22 we have $\mathcal{R}\ T_\perp\ d\ d'$. Hence, by $\mathcal{R}\ \Gamma\ \eta\ \eta'$ and definition of $\mathcal{R}\ \Gamma$, we have $\mathcal{R}\ \Gamma\ [\eta \mid x \mapsto d]\ [\eta' \mid x \mapsto d']$ whence the result.

Case $\Gamma \vdash e_1.f := e_2$. By Lemma 7.22 for e_1 we have $\mathcal{R}\ C\ \ell\ \ell'$, hence $\ell = \ell'$ by Lemma 7.12. By Lemma 7.22 for e_2 we have $\mathcal{R}\ U\ d\ d'$ and hence $\mathcal{R}\ T\ d\ d'$ by Lemma 7.13, where $(f : T) \in \mathit{dfields}\,C$ as in the typing rule. To conclude the argument it suffices to show

$$\mathcal{R}\ \mathit{Heap}\ [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]\ [h' \mid \ell \mapsto [h'\ell \mid f \mapsto d']]\quad . \tag{$*$}$$

Consider confining partitions $(\mathit{Ch} * \mathit{Oh}_1 * \mathit{Rh}_1 \ldots) = h$ and $(\mathit{Ch}' * \mathit{Oh}'_1 * \mathit{Rh}'_1 \ldots) = h'$ that correspond as in the definition of $\mathcal{R}\ \mathit{Heap}\ h\ h'$. We argue by cases on C.

—$C < \mathit{Own}$: Then $\mathit{loctype}\,\ell \leq C < \mathit{Own}$. From typing we have $e_1 : C$ and hence there is some i with $\{\ell\} = \mathit{dom}(\mathit{Oh}_i)$ and by $\mathcal{R}\ \mathit{Heap}\ h\ h'$ we get

$$R\ (\mathit{Oh}_i * \mathit{Rh}_i)\ (\mathit{Oh}'_i * \mathit{Rh}'_i)$$

and so $\{\ell\} = \mathit{dom}(\mathit{Oh}'_i)$. By typing and $C \neq \mathit{Own}$, field f is not in the private fields $\overline{g}$ of Own. So $(*)$ follows from $\mathcal{R}\ \mathit{Heap}\ h\ h'$ and $\mathcal{R}\ T\ d\ d'$.

—$C \nleq \mathit{Own}$: As C is non-rep, we have $\ell \in \mathit{dom}\,\mathit{Ch}$ and thus $\ell \in \mathit{dom}\,\mathit{Ch}'$ by hypothesis $\mathcal{R}\ \mathit{Heap}\ h\ h'$. Moreover, $\mathcal{R}\ (\mathit{state}\,(\mathit{loctype}\,\ell))\ (h\ell)\ (h'\ell)$ and so by $\mathcal{R}\ T\ d\ d'$ we get $\mathcal{R}\ (\mathit{state}\,(\mathit{loctype}\,\ell))\ [h\ell \mid f \mapsto d]\ [h'\ell \mid f \mapsto d']$. Hence $(*)$.

Case $\Gamma \vdash x := \mathbf{new}\ B$. By confinement of CT, this command is confined and hence the final states are confined: $\mathit{conf}\,C\,(h_0, \eta_0)$ and $\mathit{conf}\,C\,(h'_0, \eta'_0)$. We have $C \nleq \mathit{Rep}$ and $C \neq \mathit{Own}$. In the case $C \not< \mathit{Own}$ confinement of η_0 and η'_0 implies $\mathit{rng}\,\eta_0 \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \varnothing = \mathit{rng}\,\eta'_0 \cap \mathit{locs}(\mathit{Rep}'{\downarrow})$. So $\ell \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ and $\ell' \notin \mathit{locs}(\mathit{Rep}'{\downarrow})$, hence by typing B is non-rep. In the case $C < \mathit{Own}$, we have B non-rep by Assumption 7.15 (no reps in sub-owners). Either way, B is non-rep so Lemma 7.10 applies, to yield $\mathit{dom}\,h \cap \mathit{locs}\,B = \mathit{dom}\,h' \cap \mathit{locs}\,B$. Thus by parametricity of fresh

we have $\ell = \mathit{fresh}(B, h) = \mathit{fresh}(B, h') = \ell'$. So, by Lemma 7.12 and $\mathcal{R}\ \Gamma\ \eta\ \eta'$ we have $\mathcal{R}\ \Gamma\ \eta_0\ \eta'_0$.

It remains to show $\mathcal{R}\ \mathit{Heap}_\perp\ h_0\ h'_0$ in order to get the final result $\mathcal{R}\ (\mathit{Heap} \otimes \Gamma)_\perp\ (h_0, \eta_0)\ (h'_0, \eta'_0)$. We argue by cases on B.

—$B \not\leq \mathit{Own}$: Writing fields' for the fields given by CT', we have $\mathit{fields}\, B = \mathit{fields}'\, B$ and thus $\mathcal{R}\ (\mathit{state}\, B)\ [\mathit{fields}\, B \mapsto \mathit{defaults}]\ [\mathit{fields}'\, B \mapsto \mathit{defaults}]$. So, as B is non-rep and $B \neq \mathit{Own}$, we can add ℓ to Ch and Ch' to get partitions that witness $\mathcal{R}\ \mathit{Heap}\ h_1\ h'_1$. We also have $\mathit{conf}\, B\,(h_1, \eta_1)$ and $\mathit{conf}\, B\,(h'_1, \eta'_1)$ because $\mathit{conf}\, h$, $\mathit{conf}\, h'$, and defaults do not contain any locations. Now by Lemma 7.21 we get $\mathcal{R}\ \mathit{Heap}_\perp\ h_0\ h'_0$. Combining this with what was shown above we have $\mathcal{R}\ (\mathit{Heap} \otimes \Gamma)_\perp\ (h_0, \eta_0)\ (h'_0, \eta'_0)$.

—$B \leq \mathit{Own}$: By simulation Def. 7.9(1), we have $\mathcal{R}\ \mathit{Heap}_\perp\ h_0\ h'_0$.

Case $\Gamma \vdash x := e.m(\overline{e})$. By Lemma 7.22 for e we have $\mathcal{R}\ D_\perp\ \ell\ \ell'$, hence $\ell = \ell'$ by Lemma 7.12. Let $\eta_1 = [\mathsf{self} \mapsto \ell, \overline{x} \mapsto \overline{d}]$ and $\eta'_1 = [\mathsf{self} \mapsto \ell, \overline{x} \mapsto \overline{d}']$. By confinement of $x := e.m(\overline{e})$ (Def. 6.7) we have confined arguments, i.e., $\mathit{conf}\ (\mathit{loctype}\, \ell)\,(h, \eta_1)$ and $\mathit{conf}\ (\mathit{loctype}\, \ell)\,(h', \eta'_1)$. By Lemma 7.22 for $\overline{e}$ we have $\mathcal{R}\ \overline{U}_\perp\ \overline{d}\ \overline{d}'$ and hence $\mathcal{R}\ \overline{U}\ \overline{d}\ \overline{d}'$ as we are considering the non-$\perp$ case. Thus $\mathcal{R}\ [\overline{x} : \overline{T}, \mathsf{self} : \mathit{loctype}\, \ell]\ \eta_1\ \eta'_1$ by Lemma 7.13. From $\mathcal{R}\ \mathit{MEnv}\ \mu\ \mu'$ we get

$$\mathcal{R}\ (\mathit{loctype}\, \ell, \overline{x}, \overline{T} \to T)\ (\mu(\mathit{loctype}\, \ell)m)\ (\mu'(\mathit{loctype}\, \ell)m)$$

hence, as h, h', η_1, η'_1 are confined and related, $\mathcal{R}\ (\mathit{Heap} \otimes T)_\perp\ (h_1, d_1)\ (h'_1, d'_1)$, where $(h_1, d_1) = \mu(\mathit{loctype}\, \ell)m(h, \eta)$ and $(h'_1, d'_1) = \mu'(\mathit{loctype}\, \ell)m(h', \eta')$. Thus $\mathcal{R}\ T\ d_1\ d'_1$ and $\mathcal{R}\ \mathit{Heap}\ h_1\ h'_1$. It remains to show that the updated stores $[\eta \mid x \mapsto d_1]$ and $[\eta' \mid x \mapsto d'_1]$ are related for Γ. This follows from $\mathcal{R}\ T\ d_1\ d'_1$ and $T \leq \Gamma\, x$, using Lemma 7.13.

The remaining cases are similar. See Appendix. □

8. APPLICATIONS AND FURTHER EXAMPLES

In this section we use the abstraction theorem to show some program equivalences for the examples discussed in Sections 2 and 3. Then we discuss further variations on the observer pattern.

To establish the hypothesis of the abstraction theorem for the examples we use the couplings given as examples in Sect. 7.2. Both the theorem and these couplings are defined in terms of the semantics. To show that the couplings are simulations we argue directly in terms of the semantics. For practical purposes in program verification, the abstraction theorem would be expressed syntactically as a proof rule and rules for program constructs would be used to establish the simulation property [Reynolds 1981a; Jones 1986; Morgan and Gardiner 1990; de Roever and Engelhardt 1998]. Adequate proof rules for a language like ours remains an open challenge (see Sect. 12).

8.1 Program equivalence

We take *program* to mean a well formed class table CT together with a command $\Gamma \vdash S$. We consider the object states reachable from variables of Γ to be the inputs and outputs of the program. For example, if S is the body of method main in

Sect. 2.1 then Γ is self:Main and what can be reached is self and the string self.inout. We restrict attention to confined programs, meaning that CT and $\Gamma \vdash S$ are confined. Thus, by Theorem 6.17 the method environment $[\![CT]\!]$ is confined. To prove program equivalence using the abstraction theorem, we need to both introduce and eliminate a simulation. Elimination is by identity extension Lemma 7.17 and introduction is by the related Lemma 7.18. There is a small technicality: To establish the hypothesis of Lemma 7.23, we require, without loss of generality, that S occurs in some method of CT.

We compare programs only for class tables CT, CT' that are comparable in the sense of Def. 7.1, and with commands in the same context Γ. As commands denote functions on global states, the obvious notion of equivalence is that $[\![\Gamma \vdash S]\!]$ and $[\![\Gamma \vdash' S']\!]'$ are equal as functions. By Lemma 7.11, $[\![\Gamma]\!] = [\![\Gamma]\!]'$ for any Γ, but in general the semantic domains differ for owner object states which may have different private fields. A global state $(h, \eta) \in [\![Heap \otimes \Gamma]\!]$ for CT need not be an element of $[\![Heap \otimes \Gamma]\!]'$ for CT'. However, an Own-free heap in $[\![Heap]\!]$ is also an element of $[\![Heap]\!]'$. So we compare command meanings on the Own-free states, defined using *collect* from Def. 7.16.

***Definition* 8.1 (client program equivalence)** Suppose programs $CT, (\Gamma \vdash S)$ and $CT', (\Gamma \vdash' S')$ are such that CT, CT' are comparable and confined, and moreover S (resp. S') occurs in CT (resp. CT'). The programs are *equivalent* iff

$$collect([\![\Gamma \vdash S]\!]\hat{\mu}(h, \eta)) = collect([\![\Gamma \vdash' S']\!]'\hat{\mu}'(h, \eta))$$

for all confined and Own-free $(h, \eta) \in [\![Heap \otimes \Gamma]\!]$, where $\hat{\mu} = [\![CT]\!]$ and $\hat{\mu}' = [\![CT']\!]'$. □

If $\Gamma\,$self $\leq Own$ then η cannot be Own-free. The resulting vacuous quantification makes the definition equate all commands for such Γ. But we are only interested in using the definition for clients. Simulation is the relation of interest between owners.

The static analysis for confinement Sect. 11 can be used to show that each of the following examples is confined for the appropriate Own and Rep.

***Example* 8.2** Consider the command S comprising the body of method main of class Main in Sect. 2.1 and take $\Gamma = (\text{self}:\text{Main})$. As CT we take the declarations of Main, Bool, and the first version of OBool. For CT' we use the second version of OBool. Let Rep and Rep' be Bool and Own be OBool.

To show that $CT, (\Gamma \vdash S)$ is equivalent to $CT', (\Gamma \vdash S)$, recall the basic coupling of Example 7.5 and let $\mathcal{R}$ be the induced coupling. Let (h, η) be any confined state for Γ, noting that Main $\not\leq Own$ so η is Own-free. Let $\hat{\mu} = [\![CT]\!]$ and $\hat{\mu}' = [\![CT']\!]'$. To show

$$collect([\![\Gamma \vdash S]\!]\hat{\mu}(h, \eta)) = collect([\![\Gamma \vdash' S']\!]'\hat{\mu}'(h, \eta)) \quad , \qquad (*)$$

note first that $\mathcal{R}\ (Heap \otimes \Gamma)\ (h, \eta)\ (h, \eta)$ by Lemma 7.18. It is straightforward to show that $\mathcal{R}$ is established by the constructors and preserved by the methods of OBool; thus $\mathcal{R}$ is a simulation. The abstraction theorem yields $\mathcal{R}\ MEnv\ \hat{\mu}\ \hat{\mu}'$. This in turn justifies application of the preservation Lemma 7.23 to command S, as its context $Main$ is a non-rep class $\neq Own$. Thus the outcomes $[\![\Gamma \vdash S]\!]\hat{\mu}(h, \eta)$ and

$[\![\Gamma \vdash' S']\!]'\hat{\mu}'(h,\eta)$ are related by $\mathcal{R}$. By definition of $\mathcal{R}$, either both outcomes are $\bot$, in which case $(*)$ holds by definition of *collect*, or the outcomes are non-$\bot$ states (h_0,η_0) and (h'_0,η'_0) with $\mathcal{R}\ (\mathit{Heap}\otimes\Gamma)\ (h_0,\eta_0)\ (h'_0,\eta'_0)$. Note that h_0 and h'_0 each contains at least one owner, the one constructed in S. But $Main \not\leq Own$, so $\mathit{rng}\,\eta_0$ and $\mathit{rng}\,\eta'_0$ are Own-free. Moreover, the owners were reached only by variable z which is local in S; they are not reachable via fields of the objects $h_0(\eta\,\mathsf{self})$ or $h'_0(\eta'\,\mathsf{self})$. That is, both $\mathit{collect}(h_0,\eta_0)$ and $\mathit{collect}(h'_0,\eta'_0)$ are Own-free. Thus by identity extension Lemma 7.17 we have $\mathit{collect}(h_0,\eta_0) = \mathit{collect}(h'_0,\eta'_0)$ which concludes the proof of $(*)$. □

This proof depends on parametricity of the allocator, because that is needed for the abstraction theorem. The same argument will go through, however, for the second abstraction theorem in the sequel which drops parametricity of the allocator.

***Example* 8.3** Recall the Meyer-Sieber-O'Hearn example from Sect. 3.1, and in particular the command

$$\mathsf{C}\ \mathsf{y} := \textbf{new}\ \mathsf{C}\ \textbf{in}\ \mathsf{A}\ \mathsf{x} := \textbf{new}\ \mathsf{A}\ \textbf{in}\ \mathsf{x.callP(y)} \qquad (\ddagger)$$

Take $(\ddagger)$ to be the body of method main in

$$\textbf{class}\ \mathsf{Main}\ \textbf{extends Object}\ \{\ \textbf{unit}\ \mathsf{main}()\{\ \ldots\}\ \}$$

To be very precise we need to include a class

$$\textbf{class}\ \mathsf{Rep}\ \textbf{extends Object}\ \{\ \}$$

so we can take *Rep* and *Rep′* to be Rep which is not comparable to the classes C and A of interest. Let *Own* be A. Let *CT* consist of the declarations of A, Rep, Main, and an arbitrary class

$$\textbf{class}\ \mathsf{C}\ \textbf{extends Object}\ \{\ \textbf{unit}\ \mathsf{P(A\,z)}\{\ \ldots\}\ \ldots\}$$

such that methods of C satisfy the confinement conditions. Then CT and CT' are confined, because no reps are constructed or manipulated. We use the basic coupling of Example 7.6. To appeal to the abstraction theorem, we must argue that $\mathcal{R}$ is a simulation. The constructors are **skip** and the default value 0 for field g establishes the relation. Preservation by inc is straightforward because both versions have the same code and it makes no method calls. We give the details for preservation by callP. The relevant condition is Def. 7.9(2a). To show it for callP, suppose $i \geq 0$ and $\mathcal{R}\ \mathit{MEnv}\ \mu_i\ \mu'_i$. Note that μ_i and μ'_i are confined, by Theorem 6.17. Suppose that $\mathcal{R}\ (\mathit{Heap}\otimes y\!:\!C, \mathsf{self}\!:\!A)\ (h,\eta)\ (h',\eta')$ with $\mathit{conf}\ A\,(h,\eta)$ and $\mathit{conf}\ A\,(h',\eta')$. In both versions of callP, the body is a sequence and the first command is $\mathsf{y.P(self)}$. Let $\eta_1 = [z \mapsto \eta\,\mathsf{self}, \mathsf{self} \mapsto \eta\,y]$ and $\eta'_1 = [z \mapsto \eta'\,\mathsf{self}, \mathsf{self} \mapsto \eta'\,y]$ be the environments for semantics of this call. By definition of $\mathcal{R}$ we get $\mathcal{R}\ (\mathit{Heap}\otimes z\!:\!A, \mathsf{self}\!:\!C)\ (h,\eta_1)\ (h',\eta'_1)$. From the hypothesis $\mathit{conf}\ A\,(h,\eta)$ we get $\mathit{conf}\ C\,(h,\eta_1)$ and likewise $\mathit{conf}\ C\,(h',\eta'_1)$. Applying the hypothesis $\mathcal{R}\ \mathit{MEnv}\ \mu\ \mu'$ to these environments we get that either $\mu CP(h,\eta_1) = \bot = \mu CP(h,\eta_1)$ or neither are $\bot$ and $\mathcal{R}\ (\mathit{Heap}\otimes\textbf{unit})\ (h_0,\mathit{it})\ (h'_0,\mathit{it})$ where $(h_0,\mathit{it}) = \mu CP(h,\eta_1)$ and $(h'_0,\mathit{it}) = \mu' CP(h',\eta'_1)$. The call is desugared to an assignment of the result value to a local but the value is discarded for both versions, so the states following the calls are (h_0,η) and (h'_0,η') and we have $\mathcal{R}\ (\mathit{Heap}\otimes y\!:\!C, \mathsf{self}\!:\!A)\ (h_0,\eta)\ (h'_0,\eta')$. In these states we have $h_0\ell g = h'_0\ell g \wedge h_0\ell g\ \textbf{mod}\ 2 = 0$. So the command

if self.g **mod** $2 = 0$ **then abort else skip fi**

aborts, as does its counterpart which is simply **abort**. This concludes the argument that the bodies of callP are related.

Having established the antecedents of the abstraction theorem, we conclude that the command (‡) preserves $\mathcal{R}$. By semantics of the second version of A we know callP aborts, so both interpretations of (‡) abort. The programs are equivalent. □

This example is handled without using the identity extension Lemma 7.17, but that is only because the example uses abortion. In subsequent examples the proof needs all the steps of the one for Example 8.2. The steps are not spelled out in detail; only the interesting bits are highlighted.

Example **8.4** We consider the observer pattern, taking *Own* to be Observable. Let CT be given by the first version, Fig. 2, together with the client given in Fig. 3. Let CT' be given by the sentinel version of 4 together with Fig. 3. We consider equivalence for the command self:Main,ob:AnObserver $\vdash S$ where S is the body of Main.main. Because obl is local to S, no owners are reachable in the final state.

Taking Rep, Rep' to be Node,Node2, we use the coupling relation of Example 7.7. Clearly the constructors establish the relation. To show that method add preserves it, note that the bodies of these methods are both sequential compositions; both construct a new node and then set its ob field to the value passed as a parameter. The next step is to add it to the beginning of the list; the difference between the two versions is that self.snt.nxt is assigned in Fig. 4 whereas self.fst is assigned in Fig. 2. Both versions of add then invoke methods on the new node n. In practice one would argue in terms of the behavior of those methods. Note that they need not preserve the relation; it is just that their behavior is used to maintain the relation. To give a precise argument in terms of the semantics, we consider cases on i. For $i = 0$, both μ_i and μ'_i make every method abort, in which case the body of add aborts due to method calls. As the methods in class Node and class Node2 are not recursive, their semantics is already completely defined for $i = 1$, so for $i > 0$ the behavior of add is to insert nodes at the head of the list, maintaining the relation.

The remaining owner method is notifyAll. Again, the two versions are similar except for skipping over the sentinel node. To argue that the calls to getNext act correctly one considers cases as in the proof for add. For the calls to notify on the Observer objects, recall that by the relation, the related lists contain the same Observer pointers in the same order. The two versions thus make the same series of invocations of notify. Each of those calls preserves the relation by hypothesis $\mathcal{R}\ \mathit{MEnv}\ \mu_i\ \mu'_i$. □

The last step of the argument, concerning invocations of notify, is like reasoning about invocations of P in Example 8.3. This example has the additional complication of calls to objects within the owner island. The case distinction between $i = 0$ and $i > 0$ is needed because our argument is purely in semantic terms. In a practical proof system, one would reason only in terms of the actual semantics of the methods involved rather than its approximants.

Strictly speaking, use of Lemma 7.23 depends on desugaring the examples, and the desugarings Remark 4.1 do not include loops. We return to this issue in Sect. 9.

Example **8.5** Suppose we change the client of Fig. 3 to use the following.

```
class Node1 extends Object{ // rep for Observable
  Observer ob;
  Node1 nxt;
  unit setOb(Observer o){ self.ob := o }
  unit add(Node1 n){
     Observer o := n.ob; n.ob := self.ob; self.ob := o; n.nxt := self.nxt; self.nxt := n }
  unit notifyAll(){ self.ob.notify(); if self.nxt ≠ null then self.nxt.notifyAll() else skip fi } }
class Observable extends Object{ // owner
  Node1 fst;
  unit add(Observer ob){
     Node1 n := new Node1; n.setOb(ob); if self.fst = null then self.fst := n else self.fst.add(n) fi}
  unit notifyAll(){ if self.fst ≠ null then self.fst.notifyAll() else skip fi} }
```

Fig. 8. Version of the observer pattern in object-oriented style: nodes are active.

```
class AnObserver extends Observer { unit notify(){ skip } }
```

Then in Fig. 4 we can replace the body of Observable.notifyAll by **skip** and still have equivalence with the implementation of Fig. 2. What changes with respect to Example 8.4 is that the two implementations do not make the corresponding calls to notify. But because AnObserver.notify is **skip**, calling it has the same effect as not calling it; in particular, the relation is preserved.

The argument here is not modular: by contrast with the preceding example, here we reason directly in terms of the client code. □

8.2 Further variations on observer

Fig. 8 gives another implementation of Observable, using a singly linked list but with most of the work delegated to methods of Node1. Method add of class Node1 in the Figure is an example of class-based visibility: The private fields of object n are both assigned and read.

Unlike the example of Sect. 3.1, where method P is called once by callP, method Observable.notifyAll invokes notify on multiple objects —and multiple times if some of those are aliases. By sharing state, it is possible for multiple observers to detect the order in which they are notified. In our versions of Observable, method add maintains the set in last-in order. In Fig. 8, method add in Node1 shuffles pointers to maintain the last-in order.

A less awkward version, using a sentinel node, is given in Fig. 9.

The following example indicates the limits of what can be proved using the abstraction theorem. For this discussion, instead of treating loops as syntactic sugar we assume they are in the language. The semantic clause would use a fixpoint but this is separate from the fixpoint of the approximation chain used for method meanings. Thus for each $i > 0$ the full semantics of a loop is defined in μ_i.

***Example* 8.6** Consider the versions given by Figs. 2 and 8. The data structures are very similar; essentially the identity coupling can be used. (It is not literally the identity, because because Node and Node1 are distinct classes and thus the sets $[\![\mathsf{Node}]\!]$ and $[\![\mathsf{Node1}]\!]$ have no non-*nil* location in common. But that is just the reflection of a coding trick in our formalization of semantics.) The bodies of add and notifyAll in the two versions have significant differences in the calling graph, and in particular notifyAll in one version uses a loop whereas in the other it calls

```
class Node3 extends Object { // rep for Observable
  Node3 nxt;
  unit notif(){ skip }
  unit notifyAll(){ self.notif(); if self.nxt ≠ null then self.nxt.notifyAll() else skip fi }
  unit add(Observer ob){ NodeO n := new NodeO; n.setOb(ob); n.nxt := self.nxt; self.nxt := n } }
class NodeO extends Node3 { // rep subclass
  Observer ob;
  unit setOb(Observer o){ self.ob := o }
  unit notif(){ self.ob.notify() } }
class Observable extends Object{ // owner
  Node3 snt;
  con{ self.snt := new Node3 }
  unit add(Observer ob){ self.snt.add(ob) }
  unit notifyAll(){ self.snt.notifyAll() } }
```

Fig. 9. Sentinel in object-oriented style. In class Node3, method add constructs an object of the subclass NodeO and method notifyAll uses dynamic dispatch of notif.

a recursive method in Node1. To reason about these would require proving a loop invariant and verifying specifications for methods add and notifyAll in Node1. But for this one wants the final semantics of the program, not the approximate one given by μ_i and μ'_i. For given i, the semantics of notifyAll is only defined up to recursion depth i; for a list longer than that, the loop in Fig. 2 works correctly but the recursion in Fig. 8 aborts.

By contrast, equivalence between Figs. 4 and 8 can be shown by an argument similar to that in Example 8.4. They do not have the same method call graph, but the called methods are not recursive so one can argue by cases for $i = 0$ and $i > 0$.

If the loop in Fig. 2 is treated as syntactic sugar for a method call then the equivalence has a complicated proof in terms of corresponding unfoldings of the semantic approximations. But this is an accidental feature of the example. □

Example 8.6 might lead one to wonder whether there is a flaw in the definition of simulation. Instead of requiring that owner methods preserve the relation given any approximating and related environments μ_i, μ'_i, perhaps it should be enough to consider the final semantics $[\![CT]\!], [\![CT']\!]'$. But this is not a sufficiently strong induction hypothesis to prove the abstraction theorem. In fact the example reflects a limitation in most theories of simulation and logical relations: what can be shown equivalent are programs with the same structure in some sense; see Sect. 12.

***Example* 8.7** Equivalence between the versions given by Figs. 8 and 9 can be shown by an argument similar to that in Example 8.4. The basic coupling is like that of Example 7.7 with minor changes: Rep, Rep' are named Node1,Node3 and the sentinel is at location $\ell'_0 \in locs($Node3$)$ whereas the locations $\ell'_1, \ell'_3, \ldots$ following it are in $locs($NodeO$)$. The method call graphs are not identical for the two versions and dynamic dispatch is used in the second version for Node3.notif. But the differences involve non-recursive methods and it suffices, as in Example 8.4, to consider two cases for μ_i, μ'_i, namely $i = 0$ and $i > 0$. □

9. OWNER SUBCLASSING: THE PROTECTED INTERFACE

This section considers examples involving subclasses of the owner class. Rather than formalizing the "protected" construct of Java, we address the issues using a module construct. We augment the syntax to designate certain methods as having module scope, meaning that they cannot be called by clients. The confinement conditions for these methods are relaxed. The standard notion of "protected" scope can be obtained by putting all subclasse of *Own* in the module.

9.1 Owner subclassing and module scope

The code for notifyAll in Observable of Fig. 2 uses a loop. Here is an equivalent version using a tail recursive helper method doNotif.

```
unit notifyAll(){ doNotif(self.fst) }
unit doNotif(Node n){
    if n ≠ null then n.getOb().notify(); doNotif(n.getNext()) else skip fi }
```

In a language with nested method declarations, doNotif could be declared within notifyAll. Absent that, it could be given private scope, allowing its calls only in Observable. But the language of Sects. 4–8 has only public methods. To apply our abstraction theorem to the desugared version we would have to include a suitable implementation of doNotif in every version. This can be done for the examples in this paper, but it is awkward.

In Sect. 9.3 we add module-scoped methods to the language. These suffice for desugaring loops and for interactions between reps and owners. In the sequel we focus on their use in subclasses of *Own*.

Fig. 10 is a variation on the observer pattern in which class Observable has subclass ObservableAcc. For accounting purposes it keeps track of the number of times each observer has been notified. To this end, the rep class NodeAcc overrides method notifyAll of the client class Node4. Such examples have led to our treatment of owner subclasses: They are distinguished from clients in that their methods may manipulate reps, but unlike *Own* they cannot store reps in fields.

Method addn has been added to Observable, so that ObservableAcc can construct reps of the subtype NodeAcc and install although fst is a private field not visible in ObservableAcc. Method Observable.getFirst is also added for this purpose. But getFirst leaks a rep; it cannot be allowed in the public interface. One possibility is to treat getFirst and addn as visible only in subclasses of Observable. Instead, we give them module scope, meaning that calls to getFirst and addn are allowed in subclasses of both Observable and Node4.

Method add in class ObservableAcc constructs a rep, violating the condition "no reps in sub-owners" in Assumption 7.15. That assumption is needed for the first abstraction theorem because methods of an owner subclass are like clients in that they must preserve the induced relation. That means in particular that they manipulate related —i.e., equal— rep locations. (By contrast, methods of *Own* preserve the basic coupling which need not impose a correspondence on rep locations.) But if we compare two versions, one with sentinel node and one without, the parametricity condition for *fresh* will not apply and the new objects in ObservableAcc.add will be at different locations. The solution, given in Sect. 10, is to relax equality to bijection.

```
class Node4 extends Object { // rep for Observable
  Observer ob;
  Node4 nxt;
  unit setOb(Observer o){ self.ob := o }
  unit setNext(Node4 n){ self.nxt := n }
  Observer getOb(){ result := self.ob }
  Node4 getNext(){ result := self.nxt }
  Node4 getNextPri(){ result := self.nxt }
  unit notifyAll(){ self.ob.notify(); if self.nxt ≠ null then self.nxt.notifyAll() else skip fi } }
class NodeAcc extends Node4 {
  int notifs;
  unit notifyAll(){ self.notifs := self.notifs+1; super.notifyAll() }
  int notifications(Observer o){
    if o = self.getOb() then result := notifs
    else if self.getNext() ≠ null then result := (NodeAcc)(self.getNext()).notifications(o)
    else result := 0; fi }}
class ObservableSup extends Object { // superclass of owner; "abstract" class
  unit add(Observer ob){ abort }
  unit notifyAll(){ abort }
class Observable extends ObservableSup { // owner
  Node4 fst; // first node of list
  Node4 getFirst(){ result := self.fst } // module scope
  unit add(Observer ob){ Node4 n := new Node4; self.addn(ob,n) }
  unit addn(Observer ob, Node4 n){ n.setNext(self.fst); n.setOb(ob); self.fst := n } // module scope
  unit notifyAll(){ self.fst.notifyAll() }
class ObservableAcc extends Observable {
  unit add(Observer ob){ Node4 n := new NodeAcc; self.addn(ob,n) }
  int notifications(Observer ob){ result := ((NodeAcc)(self.getFirst())).notifications(ob) } }
```

Fig. 10. Version with owner and rep subclasses and super-call. The owner also has a superclass. The two versions of getNext in Node4 are needed for later examples.

This relaxation is needed anyway, to avoid unobservable distinctions. As an example, suppose we add to class Observable in Fig. 2 the following method:

```
String version(){ result := new String("vsn 0") }
```

Consider an alternative that is identical in every way except for the following:

```
String version(){ result := new String("trash"); result := new String("vsn 0") }
```

According to Def. 7.9, the induced relation for locations of type String is equality. But, even if the allocator is parametric, the locations returned by these two methods are not equal. (So condition (2a) fails in Def. 7.9 of simulation.) But they cannot be distinguished; this claim is justified by the generalized theory of Sect. 10, where the induced relation allows an arbitrary bijection between locations of client types like String. For this example, the bijection would be extended to relate the returned results from the two versions.

Returning to the example in Fig. 10, the interface betweeen Observable and its subclass ObservableAcc is awkwardly designed. An improvement is to use the factory pattern [Gamma et al. 1995] so that add itself can be inherited. In Fig. 11, we add method makeNode, which should have module scope, and remove addn.

To illustrate that owners may reference each other, let us add a method allNotifications which reports the number of times a given observer has been notified by

```
class Observable extends ObservableSup {
  Node4 fst;
  Node4 getFirst(){ result := self.fst } // module scope
  Node4 makeNode(){ result := new Node4 } // module scope
  unit add(Observer ob){ Node4 n := makeNode(); n.setNext(self.fst); n.setOb(ob); self.fst := n }
  unit notifyAll(){ self.fst.notifyAll() } }
class ObservableAcc extends Observable {
  Node4 makeNode(){ result := new NodeAcc } // module scope
  int notifications(Observer ob){ result := ((NodeAcc)(self.getFirst())).notifications(ob) } }
```

Fig. 11. Variation on Fig. 10 using factory pattern. Node4 and NodeAcc are as in Fig. 10.

```
class ObservableAccG extends ObservableAcc {
  ObservableAccG peer;
  con{ self.peer := self }
  unit joinGroup(ObservableAccG o){ // pre: self.peer=self and o.peer is cyclic list of length ≥ 1
    self.peer := o.peer; o.peer := self }
  int allNotifications(Observer ob){
    result := self.notifications(ob); ObservableAccG p := self.peer;
    while p ≠ self do result := result + p.notifications(ob); p := p.peer od } }
```

Fig. 12. Extension of Fig. 10 or Fig. 11 with grouped owners.

any observable in a group thereof. In the code of Fig. 12, groups are represented by cyclic lists. An ObservableAccG is initially in a singleton group; groups grow using method joinGroup.

These examples show subclasses of reps and owners. There is inheritance into the owner but not into the rep. Inheritance into reps is disallowed by our definition of confined class table, because to handle it requires a more sophisticated analysis to prevent leaks via self; a suitable analysis of "anonymous methods" is discussed in Sect. 12. Inheritance into owners also needs restriction; we have chosen a simple restriction that nonetheless allows the preceding examples.

Finally, let us consider an alternative version of Fig. 11 to illustrate the consequences of allowing the owner class, but not its subclasses, to differ in comparable class tables. In Fig. 11 the subclass ObservableAcc manipulates reps, both constructing a new NodeAcc and invoking method notifications declared in NodeAcc. Although an alternative version of Observable could use an entirely different type of nodes internally, it has to provide method getFirst with return type Node4. Because clients can manipulate objects of class ObservableAcc, methods of that class must preserve the relation and this only holds if methods they invoke preserve the relation. So coupling must be preserved not only by public methods of Observable but also by those module scope methods that are invoked in ObservableAcc. As a simple example, Fig. 13 gives an alternative that uses Node4 and differs from Fig. 11 only in using a sentinel node.

9.2 On behavioral subclassing

Behavioral subclassing [Liskov and Wing 1994] is very useful for reasoning about specific examples. However, as mentioned earlier, it is not required in general for representation independence. Client, rep, or owner subclasses may fail to exhibit

```
class Observable extends ObservableSup {
  Node4 snt;
  con{ snt := new Node4 }
  Node4 getFirst(){ result := self.snt.getNextPri() } // module scope
  Node4 makeNode(){ result := new Node4 } // module scope
  unit add(Observer ob){
    Node4 n := makeNode(); n.setNext(self.snt.getNextPri()); n.setOb(ob); self.snt.setNext(n) }
  unit notifyAll(){ self.snt.getNextPri().notifyAll() } }
```

Fig. 13. Variation on Fig. 11 using sentinel.

behavioral subclassing. To illustrate the point let us consider two revisions of Fig. 10, both of which violate behavioral subclassing. For the first example, we add an overriding declaration to NodeAcc:

```
Node4 getNext(){ abort }
```

This causes NodeAcc to fail to be a behavioral subclass of Node4 by most definitions. (It also prevents the intended functioning of the added method NodeAcc.notifications and its callers). Nonetheless, there is still a simulation between Figs. 11 and 13.[23] Making this true is the reason Fig. 13 uses getNextPri instead of getNext.

The second revision makes malicious use of a type test. We add nothing to NodeAcc, but rather revise Node4 as follows:

```
unit notifyAll(){ if self is NodeAcc then abort else self.ob.notify() fi;
                  if self.nxt ≠ null then self.nxt.notifyAll() else skip fi }
```

Method notifyAll in NodeAcc now fails to behave properly. In some sense, the revised Node4 is non-monotonic with respect to subclassing. Again, there is still a simulation between Figs. 11 and 13. Method notifyAll aborts for ObservableAcc objects in both versions.

9.3 Formalization of module-scoped methods

In Sect. 8 we saw the need for methods that are effectively private to *Own*, for desugaring loops, and also for methods in *Own* that cannot be called by clients but can be called in subclasses of *Own*. There is also a need for methods of owners and reps that can be called by each other but not by clients. For simplicity, we address these needs with a simple notion: *Own*, *Rep*, and their subclasses are considered to be inside a module, and methods may be designated as being visible only inside the module.

To avoid belaboring the formalization, we make no change to the concrete syntax. In particular, we do not formalize a module system, only a single module.

We assume that a class table designates the class names *Own* and *Rep* and is equipped with a predicate *mscope* with the interpretation that $mscope(m, C)$ means this method has package scope. The following changes are made to the definitions of preceding sections.

[23]Here we consider a class table comprised of Node4, NodeAcc, and ObservableSup from Fig. 10, along with the overriding declaration NodeAcc.getNext and also Observable and ObservableAcc from Fig. 11. The alternative class table is the same except for using Observable from Fig. 13.

(1) For a well formed class table, *mscope* must satify conditions that reflect what in practice would be achieved by declaring *Rep*, *Own*, and their subclasses inside the module. If $mscope(m, C)$ then
—$C \leq Own$ or $C \leq Rep$,
—$mtype(m, B)$ is undefined for $B > Own$ and $B > Rep$, and
—$B \leq C$ or $C \leq B$ implies $mscope(m, B)$.

(2) The typing rule for method call has an added restriction that module-scoped methods are visible only within the module:

$$\frac{\begin{array}{c}\Gamma \vdash e : D \quad mtype(m, D) = \overline{T} \to T \\ \Gamma \vdash \overline{e} : \overline{U} \quad \overline{U} \leq \overline{T} \quad x \neq \mathsf{self} \quad T \leq \Gamma\, x \\ mscope(m, D) \;\Rightarrow\; \Gamma\, \mathsf{self} \leq Own \vee \Gamma\, \mathsf{self} \leq Rep\end{array}}{\Gamma \rhd x := e.m(\overline{e})}$$

(3) For method environments, the confinement condition of Def. 6.5(1) is replaced by the following:
—$C \leq Own \wedge mscope(m, C) \;\Rightarrow\; conf\, C\, (h_0, \eta) \wedge h \trianglelefteq h_0 \wedge (d \in locs(Rep\!\downarrow) \Rightarrow d \in dom(Rh_j))$ for some confining partition and j with $\eta\, \mathsf{self} \in dom(Oh_j)$
—$C \not\leq Rep \wedge (C \not\leq Own \vee \neg mscope(m, C)) \;\Rightarrow\; conf\, C\, (h_0, \eta) \wedge h \trianglelefteq h_0 \wedge d \notin locs(Rep\!\downarrow)$

(4) For confinement of class tables, the restriction of Def. 6.9(3) is only applied to methods with $\neg(mscope(m, C))$.

(5) For simulation, Def. 10.10 in the sequel revises Def. 7.9(2) to require preservation of the relation only for public methods, that is, if $\neg(mscope(m, Own))$. But those module-scoped methods that are called in sub-owners must also preserve the relation.
To formalize this, we define $prot(m, C)$ just if $C \leq Own$, $mscope(m, Own)$, and there is a call to m in some subclass of Own.

(6) Comparable class tables must agree on the public and protected methods of Own. Def. 7.1(1) is extended to require that $mscope(m, C) = mscope'(m, C)$ for all $C \neq Own$. Moreover, if $mtype(m, Own)$ is defined then the following hold (and *mutatis mutandis* for $mtype'$):
— $\neg mscope(m, Own)$ implies $mtype'(m, Own) = mtype(m, Own)$ and moreover $\neg mscope'(m, Own)$, and
—$prot(m, Own)$ implies $mtype'(m, Own) = mtype(m, Own)$ and $mscope'(m, Own)$ (which in turn implies $prot'(m, Own)$).

***Example* 9.1** Method doNotif in Sect. 9.1 can be given module scope. It would not be called in owner subclasses, so it is not required to be present in a comparable class table. Method getFirst of Observable in Fig. 10 is called in subclass ObservableAcc, so *prot*(getFirst, Observable) holds and getFirst must be present in a comparable class table (and be simulated). □

Results of Sections 5 and 6 hold for the extended language; the only proof affected by the changes is that of Theorem 6.17 which says that $[\![CT]\!]$ is confined if CT is confined. The result holds for the revised definitions —the necessary revisions for the proof are as follows:

—In the base case of the induction on depth, the argument proving confinement of $\mu_{i+1} C m$ for the result value d goes by cases on C. The argument for the case $C \leq Own$ still holds for m with $\neg \mathit{mscope}(m, C)$. For the case $C \leq Own$ and $\mathit{mscope}(m, C)$, the revised definition requires the result value d to satisfy $d \in locs(Rep{\downarrow}) \Rightarrow d \in dom(Rh_j)$ for some confining partition and j with $\eta\,\mathsf{self} \in dom(Oh_j)$. This follows by definition from $\mathit{conf}\, C\, (h_0, \eta_0)$.

—In the step of the induction on depth, there is case analysis on C and B, proving claim $\mathit{conf}\, B\, (h, \eta)$ and confinement of the result value d. For the case $C \leq Own < B$, the argument still holds, noting that $\neg \mathit{mscope}(m, C)$ because in a well formed class table module-scoped methods do not occur outside owner and rep classes. For the cases $C < B \leq Own$ and $C < B \leq Rep$, the arguments still hold, noting that the restrictions on *mscope* ensure $\mathit{mscope}(m, B) = \mathit{mscope}(m, C)$ so the relevant conditions are the same.

10. SECOND ABSTRACTION THEOREM

This section improves the first abstraction theorem in two ways. First, the result applies to the language extended with modules (see Sect. 9.3). The module-scoped methods of the two versions of Own can be different unless they are used in subclasses of Own. The second improvement is that parametricity of the allocator is no longer required (cf. Sect. 7.3). To compare behaviors of two versions of a program we use a bijection between locations rather than equality. This can be seen as expressing that the language is parametric in locations, which would fail if the language had pointer arithmetic. As discussed in Sect. 9.1, allowing bijection handles the problem with new reps in sub-owners that necessitates Assumption 7.15. Moreover, it allows coarsening of the notion of equivalence for commands and method meanings so that, for example, the bodies of the two versions of method version in Sect. 9.1 are equivalent.

These extensions are enough to treat all the examples in Sect. 9.1 in addition to those of Sect. 8 (except Example 8.6, for reasons discussed there).

***Definition* 10.1 (typed bijection)** A *typed bijection* is finite bijective function σ from $Locs$ to $Locs$ such that $\sigma\,\ell = \ell'$ implies $\mathit{loctype}\,\ell = \mathit{loctype}\,\ell'$. $\square$

Throughout the section we let σ range over typed bijections and sometimes omit the word "typed". To express how bijections cut down to bijections on partition blocks, we use the notation $\sigma(X)$ for the direct image of X through σ.

***Definition* 10.2 (basic coupling)** Given comparable class tables, a basic coupling is a function G that assigns to each typed bijection a binary relation $G\,\sigma$ on heaps (not necessarily closed heaps) that satisfies the following. For any σ, h, h', if $G\,\sigma\,h\,h'$ then there are partitions $h = Oh * Rh$ and $h' = Oh' * Rh'$ and locations ℓ and ℓ' in $locs(Own{\downarrow})$ such that

(1) $\sigma\,\ell = \ell'$ and $\{\ell\} = dom\ Oh$ and $\{\ell'\} = dom\ Oh'$

(2) $dom(Rh) \subseteq locs(Rep{\downarrow})$ and $dom(Rh') \subseteq locs(Rep'{\downarrow})$

(3) $\mathcal{G}\ \sigma\ (type(f, \mathit{loctype}\,\ell))\ (h\ell f)\ (h'\ell' f)$ for all $(f : T) \in dom(\mathit{fields}(\mathit{loctype}\,\ell))$ with $f \notin \overline{g} = dom(\mathit{dfields}(Own))$ and $f \notin \overline{g}' = dom(\mathit{dfields}'(Own))$. $\square$

Item (3) uses the induced coupling $\mathcal{G}$ defined below; it is a harmless forward reference because the definition of $\mathcal{G}$ for data types does not depend on $\mathcal{G}$ (or G) for heaps. Note that we do not require $dom\,\sigma$ to include the reps, nor do we disallow that it includes some of them.

***Definition* 10.3 (coupling relation, $\mathcal{G}$)** In the context of a basic coupling with given relation G, and for each typed bijection σ, relations $\mathcal{G}\;\sigma\;\theta \subseteq [\![\theta]\!] \times [\![\theta]\!]'$ as follows. Note that in the case of method meanings and method environments there is no dependence on σ.

For heaps h, h', we define $\mathcal{G}\;\sigma\;Heap\;h\;h'$ iff there exist confining partitions of h, h', with the same number n of owner islands, such that

—$dom\,\sigma \subseteq dom\,h$ and $rng\,\sigma \subseteq dom\,h'$
—$G\;\sigma\;(Oh_i * Rh_i)\;(Oh'_i * Rh'_i)$ for all i in $1..n$
—$\sigma(dom(Ch)) = dom(Ch')$, i.e., σ restricts to a bijection between $dom(Ch)$ and $dom(Ch')$
—$\mathcal{G}\;\sigma\;(state\,(loctype\;\ell))\;(h\ell)\;(h'\ell')$ for all ℓ, ℓ' with $\ell \in dom(Ch)$ and $\sigma\,\ell\,\ell'$

For other categories θ we define $\mathcal{G}\;\sigma\;\theta$ as follows.

$$\begin{array}{ll}
\mathcal{G}\;\sigma\;\mathbf{bool}\;d\;d' & \Leftrightarrow\; d = d' \\
\mathcal{G}\;\sigma\;\mathbf{unit}\;d\;d' & \Leftrightarrow\; d = d' \\
\mathcal{G}\;\sigma\;C\;d\;d' & \Leftrightarrow\; \sigma\,d = d' \vee d = nil = d' \\
\mathcal{G}\;\sigma\;\Gamma\;\eta\;\eta' & \Leftrightarrow\; \forall x \in dom\,\Gamma \bullet \mathcal{G}\;\sigma\;(\Gamma x)\;(\eta x)\;(\eta' x) \\
\mathcal{G}\;\sigma\;(state\,C)\;s\;s' & \Leftrightarrow \\
\quad C \not\leq Own \wedge \forall f \in dom(fields\,C) \bullet \mathcal{G}\;\sigma\;(type(f,C))\;(s\;f)\;(s'\;f) & \\
\mathcal{G}\;\sigma\;(\theta_\bot)\;\alpha\;\alpha' & \Leftrightarrow\; (\alpha = \bot = \alpha') \vee (\alpha \neq \bot \neq \alpha' \wedge \mathcal{G}\;\sigma\;\theta\;\alpha\;\alpha') \\
\mathcal{G}\;\sigma\;(Heap \otimes \Gamma)\;(h,\eta)\;(h',\eta') & \Leftrightarrow\; \mathcal{G}\;\sigma\;Heap\;h\;h' \wedge \mathcal{G}\;\sigma\;\Gamma\;\eta\;\eta' \\
\mathcal{G}\;\sigma\;(Heap \otimes T)\;(h,d)\;(h',d') & \Leftrightarrow\; \mathcal{G}\;\sigma\;Heap\;h\;h' \wedge \mathcal{G}\;\sigma\;T\;d\;d' \\
\mathcal{G}\;(C,\overline{x},\overline{T}\rightarrow T)\;d\;d' & \Leftrightarrow\; \forall \sigma,\;(h,\eta) \in [\![Heap \otimes \Gamma]\!], (h',\eta') \in [\![Heap \otimes \Gamma]\!]' \bullet \\
\quad \mathcal{G}\;\sigma\;(Heap \otimes \Gamma)\;(h,\eta)\;(h',\eta') \wedge conf\;C\;(h,\eta) \wedge conf\;C\;(h',\eta') & \\
\quad \Rightarrow \exists \sigma_0 \supseteq \sigma \bullet \mathcal{G}\;\sigma_0\;(Heap \otimes T)_\bot\;(d(h,\eta))\;(d'(h',\eta')) & \\
\quad \text{where } \Gamma = [\overline{x} \mapsto \overline{T}, \mathsf{self} \mapsto C] & \\
\mathcal{G}\;MEnv\;\mu\;\mu' & \Leftrightarrow\; \forall C, m \bullet \\
\quad (\neg mscope(m,C) \vee prot(m,C)) \wedge (C \text{ is non-rep}) \wedge (mtype(m,C) \text{ is defined}) & \\
\quad \Rightarrow \mathcal{G}\;\;(C, pars(m,C), mtype(m,C))\;(\mu\,C\,m)\;(\mu'\,C\,m) \quad \square &
\end{array}$$

(Recall that *prot* is defined in (5) of Sect. 9.3.)

As an example, the body of makeNode in ObservableAcc (Fig. 11) returns a new rep. Consider a coupling with a version using a sentinel. Given a bijection σ and related heaps h, h', the location $\ell = fresh(\mathsf{Node4}, h)$ may be different from $\ell' = fresh(\mathsf{Node4}, h')$ even if *fresh* is parametric, because h' has extra reps, the sentinels. But σ can be extended with the pair (ℓ, ℓ').

The following facts are straightforward consequences of the definition. The first says that if h and h' are related by $\mathcal{G}$ at σ, then σ is a bijection between the domains of h and h' except for reps.

Lemma 10.4 For all σ, h, h' and all ℓ, ℓ' not in $locs(Rep{\downarrow}, Rep'{\downarrow})$, if $\mathcal{G}\ \sigma\ Heap\ h\ h'$ then $\sigma((dom\, h) \downharpoonleft (locs(Rep{\downarrow}, Rep'{\downarrow}))) = (dom\, h') \downharpoonleft (locs(Rep{\downarrow}, Rep'{\downarrow}))$. $\square$

Lemma 10.5 If $\overline{U} \leq \overline{T}$ and $\mathcal{G}\ \sigma\ \overline{U}\ \overline{d}\ \overline{d'}$ then $\mathcal{G}\ \sigma\ \overline{T}\ \overline{d}\ \overline{d'}$. $\square$

For equivalence of values and states, we define a family of relations indexed on categories θ. To streamline the notation, we say "$x \sim_\sigma x'$ in $[\![\theta]\!]$" here, and simply use the symbol $\sim_\sigma$ later.

***Definition* 10.6 (value equivalence)** For any σ, we define a relation $\sim_\sigma$ for data values, object states, heaps, and stores, as follows.

$$\begin{array}{lll}
\ell \sim_\sigma \ell' & \text{in } [\![C]\!] & \Leftrightarrow\ \sigma\,\ell = \ell' \vee \ell = nil = \ell' \\
d \sim_\sigma d' & \text{in } [\![T]\!] & \Leftrightarrow\ d = d' \quad \text{for primitive types } T \\
s \sim_\sigma s' & \text{in } [\![state\, C]\!] & \Leftrightarrow\ \forall f \in fields\, C \bullet s\, f \sim_\sigma s'\, f \\
\eta \sim_\sigma \eta' & \text{in } [\![\Gamma]\!] & \Leftrightarrow\ \forall x \in dom\, \Gamma \bullet \eta\, x \sim_\sigma \eta'\, x \\
h \sim_\sigma h' & \text{in } [\![Heap]\!] & \Leftrightarrow\ \sigma(dom\, h) = dom\, h' \wedge \forall \ell \in dom\, h \bullet h\,\ell \sim_\sigma h'(\sigma\,\ell) \\
(h,\eta) \sim_\sigma (h',\eta') & \text{in } [\![Heap \otimes \Gamma]\!] & \Leftrightarrow\ h \sim_\sigma h' \wedge \eta \sim_\sigma \eta' \\
d \sim_\sigma d' & \text{in } [\![\theta_\bot]\!] & \Leftrightarrow\ d = \bot = d' \vee (d \neq \bot \neq d' \wedge d \sim_\sigma d' \text{ in } [\![\theta]\!])
\end{array}$$

Lemma 10.7 (identity extension) Suppose $\mathcal{G}\ \sigma\ (Heap \otimes \Gamma)\ (h,\eta)\ (h',\eta')$ and $\Gamma\,\mathsf{self}$ is non-rep. Let (h,η) and (h',η') be confined at $\Gamma\,\mathsf{self}$. If both $collect(\eta,h)$ and $collect(\eta',h')$ are Own-free then $collect(\eta,h) \sim_\sigma collect(\eta',h')$. $\square$

The reader may care to check that in the case that σ is equality, the relations $\mathcal{G}\ \sigma\ \theta$ coincide with $\mathcal{R}\ \theta$ and $\sim_\sigma$ is just equality.

***Definition* 10.8 (client program equivalence)** Suppose programs $CT, (\Gamma \vdash S)$ and $CT', (\Gamma \vdash' S')$ are such that CT, CT' are comparable and confined, and moreover S (resp. S') occurs in CT (resp. CT'). The programs are equivalent iff for all confined, Own-free (h,η) and (h',η') in $[\![Heap \otimes \Gamma]\!]$ and all σ with $(h,\eta) \sim_\sigma (h',\eta')$, there is some $\sigma_0 \supseteq \sigma$ with

$$collect([\![\Gamma \vdash S]\!]\hat{\mu}(h,\eta)) \sim_{\sigma_0} collect([\![\Gamma \vdash' S']\!]'\hat{\mu}'(h',\eta'))\ ,$$

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

Lemma 10.9 Suppose B, C and all class names in $\overline{T}$ are non-rep and moreover $B < C$. If $\mathcal{G}\ \ (C, \overline{x}, \overline{T} {\to} T)\ d\ d'$ then $\mathcal{G}\ \ (B, \overline{x}, \overline{T} {\to} T)\ (restr(d,B))\ (restr(d',B))$ where $restr$ is the restriction to global states of B (see Def. 5.5). $\square$

As discussed in Sect. 9, the relation must be preserved not only by public methods but also by any module scope methods that are called by methods declared in subclasses of Own.

***Definition* 10.10 (simulation)** A simulation is a coupling relation $\mathcal{G}$ such that

(1) (constructors of Own establish $\mathcal{G}$) For any μ, μ', any ℓ, ℓ' in $locs(Own{\downarrow})$ with $\sigma\,\ell = \ell'$, and any h, h' with $\mathcal{G}\ \sigma\ Heap\ h\ h'$, let

$$\begin{array}{l}
h_1 = [h \mid \ell \mapsto [fields(loctype\,\ell) \mapsto defaults]] \\
h'_1 = [h' \mid \ell' \mapsto [fields'(loctype\,\ell') \mapsto defaults]] \\
h_0 = [\![\mathsf{self}: (loctype\,\ell) \vdash constr(loctype\,\ell) : \mathbf{con}]\!]\hat{\mu}(h_1, [\mathsf{self} \mapsto \ell]) \\
h'_0 = [\![\mathsf{self}: (loctype\,\ell') \vdash' constr(loctype\,\ell') : \mathbf{con}]\!]'\hat{\mu}'(h'_1, [\mathsf{self} \mapsto \ell'])
\end{array}$$

Then there is $\sigma_0 \supseteq \sigma$ such that $G\ \sigma\ h_0\ h'_0$.

(2) (methods of *Own* preserve $\mathcal{G}$) For any confined μ, μ' such that $\mathcal{G}\ \mathit{MEnv}\ \mu\ \mu'$, the following conditions hold for every m with $\mathit{mtype}(m, \mathit{Own})$ defined and $\neg \mathit{mscope}(m, \mathit{Own})$ or $\mathit{prot}(m, \mathit{Own})$, where $\overline{x} = \mathit{pars}(m, \mathit{Own})$ and $\overline{T} \to T = \mathit{mtype}(m, \mathit{Own})$.
   (a) $\mathcal{G}\ (\mathit{Own}, \overline{x}, \overline{T} \to T)\ (\llbracket M \rrbracket \mu)\ (\llbracket M' \rrbracket' \mu')$
   if m has declaration M in $\mathit{CT}(\mathit{Own})$ and M' in $\mathit{CT}'(\mathit{Own})$
   (b) $\mathcal{G}\ (\mathit{Own}, \overline{x}, \overline{T} \to T)\ (\llbracket M \rrbracket \mu)\ (\mathit{restr}(\llbracket M_B \rrbracket' \mu', \mathit{Own}))$
   if m has declaration M in $\mathit{CT}(\mathit{Own})$ and is inherited from B in $\mathit{CT}'(\mathit{Own})$, with M_B the declaration of m in B
   (c) $\mathcal{G}\ (\mathit{Own}, \overline{x}, \overline{T} \to T)\ (\mathit{restr}(\llbracket M_B \rrbracket \mu, \mathit{Own}))\ (\llbracket M' \rrbracket' \mu')$
   if m has declaration M' in $\mathit{CT}'(\mathit{Own})$ and is inherited from B in $\mathit{CT}(\mathit{Own})$, with M_B the declaration of m in B

Instead of Assumption 7.15 we need only the following.

Assumption 10.11 CT and CT' are confined class tables for which a (generalized) simulation $\mathcal{G}$ is given.

Theorem 10.12 (abstraction) $\mathcal{G}\ \mathit{MEnv}\ \llbracket \mathit{CT} \rrbracket\ \llbracket \mathit{CT}' \rrbracket'$.

The proof is essentially the same as the proof of Theorem 7.20. The definition of $\mathcal{G}\ \mathit{MEnv}$ requires the relation to be preserved by those module-scoped methods that are called by subowners, and this is ensured by Def. 10.10(2) of simulation. The lemmas used in the proof are as follows.

Lemma 10.13 (preservation by expressions) For any non-rep class $C \neq \mathit{Own}$ and any constituent expression $\Gamma \vdash e : T$ of a method declared in C, the following holds: For all σ and all $(h, \eta) \in \llbracket \mathit{Heap} \otimes \Gamma \rrbracket$ and $(h', \eta') \in \llbracket \mathit{Heap} \otimes \Gamma \rrbracket'$, if $\mathcal{G}\ \sigma\ (\mathit{Heap} \otimes \Gamma)\ (h, \eta)\ (h', \eta')$ then

$$\mathcal{G}\ \sigma\ (T_\perp)\ (\llbracket \Gamma \vdash e : T \rrbracket (h, \eta))\ (\llbracket \Gamma \vdash' e : T \rrbracket' (h', \eta'))\ .$$

Proof. The proof is very similar to the proof of Lemma 7.22 except in the case of field access.

For $\Gamma \vdash e.f : T$, the argument is as follows, for any σ. By induction on e we have $\mathcal{G}\ \sigma\ C_\perp\ \ell\ \ell'$. In the non-$\perp$ case, $\ell \neq \mathit{nil} \neq \ell'$ hence, by definition of $\mathcal{G}$, $\sigma\,\ell = \ell'$. By closure of the heaps, $\ell \in \mathit{dom}\,h$ and $\ell' \in \mathit{dom}\,h'$.

We consider cases on whether $C < \mathit{Own}$. Consider confining partitions $(\mathit{Ch} * \mathit{Oh}_1 * \mathit{Rh}_1 \ldots) = h$ and $(\mathit{Ch}' * \mathit{Oh}'_1 * \mathit{Rh}'_1 \ldots) = h'$ that have corresponding islands as in the definition of $\mathcal{R}\ \mathit{Heap}$. In the case $C < \mathit{Own}$, we have $\ell \in \mathit{locs}(\mathit{Own}{\downarrow})$ and hence ℓ in some $\mathit{dom}(\mathit{Oh}_i)$. From $\mathcal{G}\ \sigma\ \mathit{Heap}\ h\ h'$ we have

$$G\ \sigma\ (\mathit{Oh}_i * \mathit{Rh}_i)\ (\mathit{Oh}'_i * \mathit{Rh}'_i)$$

and thus $\ell' \in \mathit{dom}(\mathit{Oh}'_i)$ by basic coupling Def. 10.2(1) and bijectivity of σ. Since $C \neq \mathit{Own}$, we know by visibility that f is not in the private fields $\overline{g}$ of Own. Thus, as $\mathit{type}(f, \mathit{loctype}\ \ell)) = T$, we have $\mathcal{G}\ \sigma\ T\ (h \ell f)\ (h' \ell' f)$ by Def. 10.2(3).

In the case $C \not\leq \mathit{Own}$ we have $\ell \in \mathit{dom}(\mathit{Ch})$ and hence $\ell' \in \mathit{dom}(\mathit{Ch}')$ by $\sigma\,\ell = \ell'$ and definition $\mathcal{G}\ \mathit{Heap}$. Hence

$$\mathcal{G}\ \sigma\ (\mathit{state}\,(\mathit{loctype}\,\ell))\ (h\ell)\ (h'\ell')$$

and thus $\mathcal{G}\ \sigma\ T\ (h\ell f)\ (h'\ell' f)$ by definition of $\mathcal{G}\ (\mathit{state}\,(\mathit{loctype}\,\ell))$. Note that $\mathit{loctype}\,\ell = \mathit{loctype}\,\ell'$ because σ is a typed bijection. $\square$

Lemma 10.14 (preservation by commands) Suppose that μ and μ' are confined method environments and $\mathcal{G}\ \ \mathit{MEnv}\ \mu\ \mu'$. Then the following holds for any non-rep class $C \neq \mathit{Own}$. For any constituent command $\Gamma \vdash S$ in a method declaration in $CT(C)$, any σ, and any $(h, \eta) \in [\![\mathit{Heap} \otimes \Gamma]\!]$ and $(h', \eta') \in [\![\mathit{Heap} \otimes \Gamma]\!]'$, if $\mathit{conf}\,C\,(h, \eta)$, $\mathit{conf}\,C\,(h', \eta')$, and $\mathcal{G}\ \sigma\ (\mathit{Heap} \otimes \Gamma)\ (h, \eta)\ (h', \eta')$ then there is $\sigma_0 \supseteq \sigma$ such that

$$\mathcal{G}\ \sigma_0\ (\mathit{Heap} \otimes \Gamma)_\perp\ ([\![\Gamma \vdash S]\!]\mu(h, \eta))\ ([\![\Gamma \vdash' S]\!]'\mu'(h', \eta'))\ .$$

Proof. The proof is very similar to the proof of the corresponding Lemma 7.23 except in the cases of method call, field update, and most interestingly **new**. We no longer have the assumption of parametricity of the allocator, and we must consider construction of reps in sub-owners. We also need an analog to Lemma 7.21, saying that constructors establishes $\mathcal{R}$:

Little lemma: For all σ and all $(h, \ell) \in [\![\mathit{Heap} \otimes C]\!]$ and $(h', \ell') \in [\![\mathit{Heap} \otimes C]\!]$, if $\mathcal{G}\ \sigma\ \mathit{Heap}\ h\ h'$ and $\mathcal{G}\ \sigma\ C\ \ell\ \ell'$ then there is $\sigma_0 \supseteq \sigma$ such that $\mathcal{G}\ \sigma_0\ \mathit{Heap}\ h_0\ h'_0$ where

$$\begin{aligned} h_0 &= [\![\mathsf{self}: C \vdash \mathit{constr}\,C : \mathbf{con}]\!]\mu(h, [\mathsf{self} \mapsto \ell]) \\ h'_0 &= [\![\mathsf{self}: C \vdash' \mathit{constr}\,C : \mathbf{con}]\!]\mu'(h', [\mathsf{self} \mapsto \ell']) \end{aligned}$$

We omit the proof of the little lemma, which has the same structure as the proof of Lemma 7.21.

Case $\Gamma \vdash x := e.m(\overline{e})$. This goes through as before except for the case where $C < \mathit{Own}$. In that case, the called method may have module scope and this is why such methods (designated by *prot*) are included in the definition of $\mathcal{G}\ \mathit{MEnv}$.

Case $\Gamma \vdash e_1.f := e_2$. By Lemma 10.13 for e_1 we have $\mathcal{G}\ \sigma\ C\ \ell\ \ell'$, hence $\sigma\,\ell = \ell'$ definition of $\mathcal{G}$. By Lemma 10.13 for e_2 we have $\mathcal{G}\ \sigma\ U\ d\ d'$ and hence $\mathcal{G}\ \sigma\ T\ d\ d'$ by Lemma 10.5. To conclude the argument it suffices to show

$$\mathcal{G}\ \sigma\ \mathit{Heap}\ [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]\ [h' \mid \ell' \mapsto [h'\ell' \mid f \mapsto d']]\ . \qquad (*)$$

Consider confining partitions $(\mathit{Ch} * \mathit{Oh}_1 * \mathit{Rh}_1 \ldots) = h$ and $(\mathit{Ch}' * \mathit{Oh}'_1 * \mathit{Rh}'_1 \ldots) = h'$ that correspond as in the definition of $\mathcal{G}\ \sigma\ \mathit{Heap}\ h\ h'$. We argue by cases on C.

—$C < \mathit{Own}$: Then $\mathit{loctype}\,\ell \leq C < \mathit{Own}$. By $\sigma\,\ell = \ell'$ and $\mathcal{G}\ \sigma\ \mathit{Heap}\ h\ h'$, there is i such that $\{\ell\} = \mathit{dom}(\mathit{Oh}_i)$ and $\{\ell'\} = \mathit{dom}(\mathit{Oh}'_i)$ and

$$G\ \sigma\ (\mathit{Oh}_i * \mathit{Rh}_i)\ (\mathit{Oh}'_i * \mathit{Rh}'_i)\ .$$

By typing and $C \neq \mathit{Own}$, field f is not in the private fields $\overline{g}$ of Own. So $(*)$ follows from $\mathcal{G}\ \sigma\ \mathit{Heap}\ h\ h'$ and $\mathcal{G}\ \sigma\ T\ d\ d'$.

—$C \not\leq \mathit{Own}$: As C is non-rep, we have $\ell \in \mathit{dom}\,\mathit{Ch}$ and $\ell' \in \mathit{dom}\,\mathit{Ch}'$. Moreover, $\mathcal{G}\ \sigma\ (\mathit{state}\,(\mathit{loctype}\,\ell))\ (h\ell)\ (h'\ell')$ and so by $\mathcal{G}\ \sigma\ T\ d\ d'$ we get

$$\mathcal{G}\ \sigma\ (\mathit{state}\,(\mathit{loctype}\,\ell))\ [h\ell \mid f \mapsto d]\ [h'\ell' \mid f \mapsto d']\ .$$

Hence $(*)$.

Case $\Gamma \vdash x := \mathbf{new}\ B$. By confinement of CT, this command is confined and hence the final states are confined: $conf\, C\, (h_0, \eta_0)$ and $conf\, C\, (h'_0, \eta'_0)$. We have $C \not\leq Rep$ and $C \neq Own$. Let $\ell = fresh(B, h)$ and $\ell' = fresh(B, h')$. Define $\sigma_1 = \sigma \cup \{(\ell, \ell')\}$. This makes σ_1 bijective because ℓ, ℓ' are fresh and $\mathcal{G}\ \sigma\ Heap\ h\ h'$ implies, by definition, that $dom\,\sigma \subseteq dom\,h$ and $rng\,\sigma \subseteq dom\,h'$.

By $\mathcal{G}\ \sigma\ \Gamma\ \eta\ \eta'$ and definition of σ_1 we have $\mathcal{G}\ \sigma_1\ \Gamma\ \eta_0\ \eta'_0$. We proceed to show $\mathcal{G}\ \sigma_1\ Heap\ h_0\ h'_0$, by cases on B.

—$B \not\leq Own \wedge B \not\leq Rep$: We have $fields\,B = fields'\,B$ and thus

$$\mathcal{G}\ \sigma_1\ (state\ B)\ [fields\,B \mapsto defaults]\ [fields'\,B \mapsto defaults]\ .$$

So, as B is non-rep and $B \neq Own$, we can add ℓ to Ch and ℓ' to Ch' to get partitions that witness $\mathcal{G}\ \sigma_1\ Heap\ h_1\ h'_1$. Now the induction hypothesis yields some $\sigma_0 \supseteq \sigma_1$ such that $\mathcal{G}\ \sigma_0\ Heap\ h_0\ h'_0$. We obtain $\mathcal{G}\ \sigma_0\ \Gamma\ \eta_0\ \eta'_0$ from $\mathcal{G}\ \sigma_1\ \Gamma\ \eta_0\ \eta'_0$ because $\sigma_0 \supseteq \sigma_1$.

—$B \leq Own$: By basic coupling, Def. 10.2, we get σ_0 with $G\,\sigma_0\,h_2\,h'_2$. Moreover, h_2 and h'_2 are owner islands and the confining partitions for h, h' extend to ones for $h * h_2$. and $h' * h'_2$ with σ_0. Finally, by definition of $\mathcal{G}$ we get $\mathcal{G}\ \sigma_0\ Heap\ h_0\ h'_0$ as $h_0 = h * h_2$ and $h'_0 = h' * h'_2$.

—$B \leq Rep$: Here, $C \leq Own$ or $C \leq Rep$, as otherwise the command would not be confined. Let j be such that $\eta\,\mathsf{self} \in dom(Oh_j * Rh_j)$. Add ℓ to Rh_j and ℓ' to Rh'_j. This yields $\mathcal{G}\ \sigma_0\ Heap\ h_0\ h'_0$ with $h_0 = h * h_2$ and $h'_0 = h' * h'_2$. □

11. STATIC ANALYSIS

This section gives a syntax directed static analysis. The analysis checks a property called safety, which is shown to imply confinement.

The input is a well formed class table and designated class names Own and Rep. With one exception, only rep and owner code (including subclasses) is constrained. The exception is for **new**: a client cannot construct a new rep. For practical application, this can be ensured in a modular way: Rep and its subclasses would simply be declared with module scope.

The analysis is given for the language extended in Sect. 9.3 with module-scoped methods. For the original language, $mscope(m, C)$ can be taken to be false for all m and C.

***Definition* 11.1 (*safe*)** Class table CT is *safe* iff for every C and every m with $mtype(m, C) = \overline{T} \rightarrow T$ the following hold.

(1) If m is declared in C by $T\ m(\overline{T}\ \overline{x})\{S\}$ then $\overline{x} : \overline{T}, \mathsf{self} : C, \mathsf{result} : T \rhd S$ where $\rhd$ is the safety relation defined in the sequel.

(2) $\mathsf{self} : C \rhd constr\,C$, for all C

(3) If $C \leq Own$ and $\neg mscope(m, C)$ then $T \not\gneq Rep$.

(4) If m is inherited in Own from some $B > Own$ then $\overline{T} \not\gneq Rep$.

(5) No m is inherited in Rep from any $B > Rep$.

The safety relation $\rhd$ is defined by the following rules. There is no restriction on field declarations per se. A client can have a Rep type field, but can assign only **null** to it.

Safety for expressions

$$\Gamma \rhd x : \Gamma x \quad \Gamma \rhd \mathbf{null} : B \quad \Gamma \rhd \mathbf{unit} : \mathbf{unit} \quad \Gamma \rhd \mathbf{true} : \mathbf{bool} \quad \Gamma \rhd \mathbf{false} : \mathbf{bool}$$

$$\frac{C = (\Gamma\,\mathsf{self}) \quad \Gamma \rhd e : C \quad (f : T) \in \mathit{dfields}\, C \qquad C = \mathit{Own} \wedge e \neq \mathsf{self} \Rightarrow T \not\leq \mathit{Rep}}{\Gamma \rhd e.f : T}$$

$$\frac{\Gamma \rhd e_1 : T \quad \Gamma \rhd e_2 : T}{\Gamma \rhd e_1 = e_2 : \mathbf{bool}} \qquad \frac{\Gamma \rhd e : D \quad B \leq D}{\Gamma \rhd (B)\; e : B} \qquad \frac{\Gamma \rhd e : D \quad B \leq D}{\Gamma \rhd e \;\mathbf{is}\; B : \mathbf{bool}}$$

For expressions, the analysis imposes restrictions on field accesses and nothing else. If $e.f$ appears in the body of an owner method, then a *Rep* can be accessed only via the private fields of *Own*; this requires e to be self (instance-based visibility). If $e.f$ appears in a sub-owner, then the private fields of *Own* cannot be accessed, hence the result cannot be a *Rep*.

For commands, the rules impose restrictions on **new**, field update, and method call. The conditions on field update are analogous to those for field access. For an object construction $x := \mathbf{new}\ B$ in the body of a client method, it cannot create a new rep. And, if it appears in a subclass of *Rep*, it cannot create a new owner as this would break confinement of the heap.

For method call $x := e.m(\overline{e})$, the condition labelled (a) says that if m is a client method called from a subclass of *Own* or *Rep*, then m cannot be passed reps as parameters. Conditions (b) and (c) consider method calls from an owner class or its subclasses: (b) says that if m's type is comparable to *Own* then reps can be passed as parameters only if e is self. Finally, (c) says that if m's type is comparable to *Rep* then no owner, other than itself, can be passed as parameter —otherwise confinement will be violated.

Safety for commands

$$\frac{\begin{array}{c} C = (\Gamma\,\mathsf{self}) \quad B \neq \mathbf{Object} \\ x \neq \mathsf{self} \quad B \leq \Gamma x \\ C \not\leq \mathit{Rep} \wedge C \not\leq \mathit{Own} \;\Rightarrow\; B \not\leq \mathit{Rep} \\ C \leq \mathit{Rep} \;\Rightarrow\; B \not\leq \mathit{Own} \end{array}}{\Gamma \rhd x := \mathbf{new}\ B} \qquad \frac{\begin{array}{c} C = (\Gamma\,\mathsf{self}) \qquad (f : T) \in \mathit{dfields}\, C \\ \Gamma \rhd e_1 : C \quad \Gamma \rhd e_2 : U \quad U \leq T \\ C = \mathit{Own} \wedge e_1 \neq \mathsf{self} \;\Rightarrow\; U \not\lesseqgtr \mathit{Rep} \\ C < \mathit{Own} \;\Rightarrow\; U \not\lesseqgtr \mathit{Rep} \end{array}}{\Gamma \rhd e_1.f := e_2}$$

$$\frac{\begin{array}{l} \qquad\qquad \Gamma \rhd e : D \quad \mathit{mtype}(m, D) = \overline{T} \rightarrow T \quad T \leq \Gamma\, x \\ \qquad\qquad\qquad \Gamma \rhd \overline{e} : \overline{U} \quad \overline{U} \leq \overline{T} \quad x \neq \mathsf{self} \\ \qquad C = (\Gamma\,\mathsf{self}) \qquad \mathit{mscope}(m, D) \;\Rightarrow\; C \leq \mathit{Own} \vee C \leq \mathit{Rep} \\ (a) \quad (C \leq \mathit{Own} \vee C \leq \mathit{Rep}) \wedge (D \not\leq \mathit{Rep} \wedge (D \not\leq \mathit{Own} \vee \neg(\mathit{mscope}(m, D)))) \\ \qquad \Rightarrow \overline{T} \not\lesseqgtr \mathit{Rep} \wedge T \not\lesseqgtr \mathit{Own} \\ (b) \quad C \leq \mathit{Own} \;\Rightarrow\; D \not\lesseqgtr \mathit{Own} \vee (e = \mathsf{self}) \vee (\overline{T} \not\lesseqgtr \mathit{Rep} \wedge T \not\lesseqgtr \mathit{Rep}) \\ (c) \quad C \leq \mathit{Own} \;\Rightarrow\; D \not\lesseqgtr \mathit{Rep} \vee (\forall e_i \in \overline{e} \bullet (e_i = \mathsf{self}) \vee (T_i \not\lesseqgtr \mathit{Own})) \\ (d) \quad C \leq \mathit{Rep} \wedge D \leq \mathit{Own} \;\Rightarrow\; T \not\lesseqgtr \mathit{Own} \end{array}}{\Gamma \rhd x := e.m(\overline{e})}$$

$$\frac{\begin{array}{c} C = (\Gamma\,\mathsf{self}) \quad \mathit{mtype}(m, \mathit{super}\, C) = \overline{T} \rightarrow T \\ \Gamma \rhd \overline{e} : \overline{U} \quad \overline{U} \leq \overline{T} \quad x \neq \mathsf{self} \quad T \leq \Gamma\, x \end{array}}{\Gamma \rhd x := \mathbf{super}.m(\overline{e})}$$

$$\frac{x \neq \mathsf{self} \quad \Gamma \rhd e : T \quad T \leq \Gamma\, x}{\Gamma \rhd x := e} \qquad \frac{\Gamma \rhd S_1 \quad \Gamma \rhd S_2}{\Gamma \rhd S_1;\ S_2}$$

$$\frac{\Gamma \rhd e : \mathbf{bool} \quad \Gamma \rhd S_1 \quad \Gamma \rhd S_2}{\Gamma \rhd \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}} \qquad \frac{\Gamma \rhd e : U \quad U \leq T \quad (\Gamma, x : T) \rhd S}{\Gamma \rhd T\ x := e\ \mathbf{in}\ S}$$

Theorem 11.2 (soundness) If CT is safe then it is confined.

Proof. Items (3)–(5) in the definition of safety are the same as items (3)–(5) in the definition of confinement for class tables. For items (1) and (2), the confinement of method and constructor bodies follows from safety thereof, by Lemmas 11.3, 11.5, and 11.6 to follow. □

Lemma 11.3 (argument values confined) Suppose $\Gamma \vdash e : D$ and $\Gamma \vdash \overline{e} : \overline{U}$ are confined.

(1) If $\Gamma \rhd x := e.m(\overline{e})$ then $\Gamma \vdash x := e.m(\overline{e})$ has confined arguments.

(2) If $\Gamma \rhd x := \mathbf{super}.m(\overline{e})$ then $\Gamma \vdash x := \mathbf{super}.m(\overline{e})$ has confined arguments.

Proof. We give the argument for (1); the argument for (2) is similar (see Appendix).

As in Def. 6.8, let $C = (\Gamma\,\mathsf{self})$. Assume $\mathit{conf}\, C\,(h, \eta)$. Let $\ell = [\![\Gamma \vdash e : D]\!](h, \eta)$, let $\overline{d} = [\![\Gamma \vdash \overline{e} : \overline{U}]\!](h, \eta)$, and let $\eta_1 = [\mathsf{self} \mapsto \ell, \overline{x} \mapsto \overline{d}]$. Finally, let $\ell \neq \mathit{nil}, \ell \neq \perp$ and $\overline{d} \neq \perp$.

Because $\Gamma \rhd x := e.m(\overline{e})$ holds we can use conditions (a)–(d) in the analysis rule for method call. Now the proof proceeds by cases on caller's class C with subcases on callee's class $\mathit{loctype}\,\ell$. In each case we show $\mathit{conf}\,(\mathit{loctype}\,\ell)\,(h, \eta_1)$.

—$C \not\leq Rep \wedge C \not\leq Own$: By the hypothesis in lemma, because e and $\overline{e}$ are confined at C, we have $\ell \notin locs(Rep{\downarrow})$ and $d_i \notin locs(Rep{\downarrow})$ for all $d_i \in \overline{d}$. Thus $rng\,\eta_1 \cap locs(Rep{\downarrow}) = \varnothing$. Now we go by cases on $loctype\,\ell$ where $loctype\,\ell \leq D$. In case $loctype\,\ell \not\leq Rep \wedge (loctype\,\ell \not\leq Own \vee \neg mscope(m, loctype\,\ell))$ and case $loctype\,\ell \leq Own \wedge mscope(m, loctype\,\ell)$, the result follows because $rng\,\eta_1 \cap locs(Rep{\downarrow}) = \varnothing$. Finally, the case $loctype\,\ell \leq Rep$ is impossible because h is confined – hence Def. 6.2(1) applies.

—$C \leq Own$: Choose a confining partition and let j be such that $\eta\,\mathsf{self} \in dom(Oh_j)$. Because e and $\overline{e}$ are confined at C, we have $\ell \in locs(Rep{\downarrow}) \Rightarrow \ell \in dom(Rh_j)$ and $d_i \in locs(Rep{\downarrow}) \Rightarrow d_i \in dom(Rh_j)$ for all $d_i \in \overline{d}$. Now we go by cases on $loctype\,\ell$:

- —$loctype\,\ell \not\leq Rep \wedge (loctype\,\ell \not\leq Own \vee \neg mscope(m, loctype\,\ell))$: Hence $\ell \notin locs(Rep{\downarrow})$. We want $rng\,\eta_1 \cap locs(Rep{\downarrow}) = \varnothing$. We have $\ell \notin locs(Rep{\downarrow})$. Moreover, by the analysis condition (a), $\overline{T} \nleq Rep$. Thus $d_i \notin locs(Rep{\downarrow})$ for all $d_i \in \overline{d}$.
- —$loctype\,\ell \leq Own \wedge mscope(m, loctype\,\ell)$: Let $\ell \in dom(Oh_k)$ for some k. Because $loctype\,\ell \leq D$ we have, $D \leq Own \vee Own \leq D$. If $e = \mathsf{self}$ then $\ell = (\eta\,\mathsf{self})$ and $k = j$. Then $rng\,\eta_1 \cap locs(Rep{\downarrow}) = \overline{d} \cap locs(Rep{\downarrow}) \subseteq dom(Rh_j) = dom(Rh_k)$, by confinement of $\overline{e}$. Thus $conf\,(loctype\,\ell)\,(h, \eta_1)$ by Def. 6.4(2). If $e \neq \mathsf{self}$ then by condition (b) of the analysis, $\overline{T} \nleq Rep$. Hence $rng\,\eta_1 \cap locs(Rep{\downarrow}) = \varnothing$ proving $conf\,(loctype\,\ell)\,(h, \eta_1)$ by Def. 6.4(2).
- —$loctype\,\ell \leq Rep$: Hence $\ell \in dom(Rh_j)$ by confinement of e at C. As $loctype\,\ell \leq D$ we have, $D \leq Rep \vee Rep \leq D$. By Def. 6.4(3), to show $conf\,(loctype\,\ell)\,(h, \eta_1)$, we must show $rng\,\eta_1 \cap locs(Own{\downarrow}, Rep{\downarrow}) \subseteq dom(Oh_j * Rh_j)$. For any $d_i \in locs(Own{\downarrow})$, because $loctype\,d_i \leq T_i$, we have $T_i \leq Own \vee Own \leq T_i$. So by condition (c) of the analysis, $e_i = \mathsf{self}$, hence $d_i = (\eta\,\mathsf{self}) \in dom(Oh_j)$. For any $d_i \in locs(Rep{\downarrow})$ we have $d_i \in dom(Rh_j)$ by confinement of $\overline{e}$ at C.

—$C \leq Rep$: Choose a confining partition and let j be such that $\eta\,\mathsf{self} \in dom(Rh_j)$. Because e and $\overline{e}$ are confined at C, we have $\ell \in locs(Own{\downarrow}, Rep{\downarrow}) \Rightarrow \ell \in dom(Oh_j * Rh_j)$ and $d_i \in locs(Own{\downarrow}, Rep{\downarrow}) \Rightarrow d_i \in dom(Oh_j * Rh_j)$ for all $d_i \in \overline{d}$. Now we go by cases on $loctype\,\ell$.

- —$loctype\,\ell \not\leq Rep \wedge (loctype\,\ell \not\leq Own \vee \neg mscope(m, loctype\,\ell))$: We want $rng\,\eta_1 \cap locs(Rep{\downarrow}) = \varnothing$. We have $\ell \in locs(Rep{\downarrow})$. By the analysis condition (a), we have $\overline{T} \nleq Rep$. Hence $d_i \notin locs(Rep{\downarrow})$ for all $d_i \in \overline{d}$.
- —$loctype\,\ell \leq Own$: Hence $\ell \in dom(Oh_j)$. Now $rng\,\eta_1 \cap locs(Rep{\downarrow}) \subseteq dom(Rh_j)$ as required for $conf\,(loctype\,\ell)\,(h, \eta_1)$ by Def. 6.4(2).
- —$loctype\,\ell \leq Rep$: Hence $\ell \in dom(Rh_j)$. Now $rng\,\eta_1 \cap locs(Own{\downarrow}, Rep{\downarrow}) \subseteq dom(Oh_j * Rh_j)$ as required for $conf\,(loctype\,\ell)\,(h, \eta_1)$, by Def. 6.4(3). □

Lemma 11.4 (soundness for expressions) If $\Gamma \rhd e : T$ then $\Gamma \vdash e : T$ is confined.

PROOF. Let $C = (\Gamma\,\mathsf{self})$. Now we go by induction on $\Gamma \rhd e : T$. Assume $conf\,C\,(h, \eta)$ and $d = [\![\Gamma \vdash e : T]\!](h, \eta) \neq \bot$ for each case of e.

CASE $\Gamma \rhd e.f : T$. Then $d = h\ell f$. We consider cases on C.

—$C \not\leq Rep \wedge C \not\leq Own$: We must show $d \notin locs(Rep{\downarrow})$. Because $loctype\,\ell \leq C$, we have $\ell \notin locs(Own{\downarrow}, Rep{\downarrow})$. So ℓ is in the client part of a confining partition and by Def. 6.2(1) we have $d \notin locs(Rep{\downarrow})$.

—$C \leq Own$: Consider a confining partition and j such that $\eta\,\mathsf{self} \in dom(Oh_j)$. We must show $d \in locs(Rep{\downarrow}) \Rightarrow d \in dom(Rh_j)$. Assume $d \in locs(Rep{\downarrow})$. If $C = Own$, we have two subcases: If $e = \mathsf{self}$ we get $\ell = \eta\,\mathsf{self}$, so $i = j$ and $d \in dom(Rh_j)$; if $e \neq \mathsf{self}$ then by the analysis we get $T \not\leq Rep$ so $d \notin locs(Rep{\downarrow})$, falsifying the antecedent. This concludes the case $C = Own$. If $C < Own$ then $f \notin \overline{g}$. Hence $U \not\leq Rep$, so $d \notin locs(Rep{\downarrow})$, falsifying the antecedent.

—$C \leq Rep$: Consider a confining partition and j such that $\eta\,\mathsf{self} \in dom(Rh_j)$. We must show $d \in locs(Own{\downarrow}, Rep{\downarrow}) \Rightarrow d \in dom(Oh_j * Rh_j)$. Since $loctype\,\ell \leq C$, we have $\ell \in locs(Rep{\downarrow})$ and by induction on e we get $\ell \in dom(Rh_j)$ via Def. 6.6(3). Because h is confined we get $d \in dom(Oh_j * Rh_j)$ by Def. 6.2(4).

The remaining cases are similar; see Appendix. □

Lemma 11.5 (soundness for constructors) Suppose that $\mathsf{self} : C \rhd constrC$ for all C and let μ be arbitrary. Then the constructor semantics is confined in the following sense: For all (h, η) with $conf\,C\,(h, \eta)$ we have $conf\,h_0$ and $h \trianglelefteq h_0$ where $h_0 = [\![\mathsf{self} : C \rhd constr\,C : \mathbf{con}]\!]\mu(h, \eta) \neq \bot$.

Proof. By well founded induction on C using the order $\ll$ in an argument similar to that for Lemma 7.21. See Appendix. □

Lemma 11.6 (soundness for commands) If $\Gamma \rhd S$ then $\Gamma \vdash S$ is confined.

Proof. Let $C = (\Gamma\,\mathsf{self})$. Now we go by induction on $\Gamma \rhd S$ and by cases on C. Assume $conf\,C\,(h, \eta)$ and $conf\,\mu$ and $[\![\Gamma \vdash S]\!]\mu(h, \eta) \neq \bot$. Let $(h_0, \eta_0) = [\![\Gamma \vdash S]\!]\mu(h, \eta)$. In each case we must show h_0 is confined and $conf\,C\,(h_0, \eta_0)$.

Case $\Gamma \rhd e_1.f := e_2$. Here $\eta_0 = \eta$ and $h_0 = [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]$. Because $\Gamma \rhd e_1 : C$ and $\Gamma \rhd e_2 : U$, by Lemma 11.4, e_1 and e_2 are confined at C. We must first show that h_0 is confined and then show $conf\,C\,(h_0, \eta_0)$. By $conf\,C\,(h, \eta)$ we know there is a confining partition $h = Ch * \ldots$. We partition h_0 using the given partition for h. That is, the domain for each block, say Ch^0, is the same as the corresponding block for h, say Ch. We claim this partition is confining for h_0. It then follows by Def. 6.3 that $h \trianglelefteq h_0$. Then by Lemma 6.13, we get $conf\,C\,(h_0, \eta)$, hence $conf\,C\,(h_0, \eta_0)$. It remains to show the claim for which we need to show the conditions in Def. 6.2. We go by cases on C.

—$C \not\leq Rep \wedge C \not\leq Own$: Only condition (1) in Def. 6.2 can possibly be violated. By $conf\,C\,(h, \eta)$ we obtain $rng\,\eta \cap locs(Rep{\downarrow}) = \varnothing$. Because $loctype\,\ell \leq C$ we have $\ell \in dom(Ch^0)$. By confinement of e_2, $d \notin locs(Rep{\downarrow})$. Hence $Ch^0 \not\rightsquigarrow Rh^0_j$ for all j.

—$C \leq Own$: Let $\eta\,\mathsf{self} \in dom(Oh^0_i)$ for some i. Only conditions (2) and (3) in Def. 6.2 can possibly be violated. Because $loctype\,\ell \leq C$, $\ell \in dom(Oh^0_j)$ for some j. Because e_2 is confined at C we have, $d \in locs(Rep{\downarrow}) \Rightarrow d \in dom(Rh^0_i)$. We consider the case $C = Own$ and $e = \mathsf{self}$. Then $\ell = \eta\,\mathsf{self}$ and $i = j$, establishing condition (2). By typing, $f \in \overline{g}$. Hence $Oh_i \not\rightsquigarrow^{\overline{g}} Rh^0_i$ establishing condition (3). In the case $e \neq \mathsf{self}$, by the analysis we have $U \not\leq Rep$ thus establishing conditions (2) and (3).
Now we consider the case $C < Own$. By the analysis we have $U \not\leq Rep$ thus establishing conditions (2) and (3).

—$C \leq Rep$: Let $\eta\,\mathsf{self} \in dom(Rh^0_i)$ for some i. Only condition (4) in Def. 6.2 can possibly be violated. Because $loctype\,\ell \leq C$, $\ell \in locs(Rep{\downarrow})$. By confinement of e_1 at C, we have $\ell \in dom(Rh^0_i)$. And, by confinement of e_2 at C, we have $d \in locs(Own{\downarrow}, Rep{\downarrow}) \Rightarrow d \in dom(Oh^0_i * Rh^0_i)$. This establishes condition (4).

CASE $\Gamma \rhd x := e.m(\overline{e})$. Here $h_0 = h_1$ and $\eta_0 = [\eta \mid x \mapsto d_1]$. Because $\Gamma \rhd e : D$ and $\Gamma \rhd \overline{e} : \overline{U}$, by Lemma 11.4, e and $\overline{e}$ are confined at C. Hence by Lemma 11.3 we have $conf\,(loctype\,\ell)\,(h, \eta_1)$. Then by assumption $conf\,\mu$ we get $conf\,(loctype\,\ell)\,(h_0, \eta_1)$. Hence h_0 is confined. By $conf\,\mu$, we have $h \trianglelefteq h_0$. Hence by Lemma 6.13, $conf\,C\,(h_0, \eta)$. Thus to show $conf\,C\,(h_0, \eta_0)$, it suffices to show that the result d_1 is confined for C in h_0. We proceed by cases on C:

—$C \not\leq Rep \wedge C \not\leq Own$: We want $d_1 \notin locs(Rep{\downarrow})$. Now we go by cases on $loctype\,\ell$. By typing, $\neg mscope(m, D)$.
  —$loctype\,\ell \not\leq Rep \wedge (loctype\,\ell \not\leq Own \vee \neg mscope(m, loctype\,\ell))$: The result follows by $conf\,\mu$ because $loctype\,\ell \leq D$.
  —$loctype\,\ell \leq Own$: Then $\neg mscope(m, loctype\,\ell)$. Hence by safety of CT, $T \not\leq Rep$. Hence the result follows.
  —$loctype\,\ell \leq Rep$: This case is impossible because h is confined; Def. 6.2(1) applies.

—$C \leq Own$: Let $\eta\,\mathsf{self} \in dom(Oh_j)$ for some j in the confining partition of h. We want $d_1 \in locs(Rep{\downarrow})$ implies $d_1 \in dom(Rh^0_j)$. Because $x \neq \mathsf{self}$, we have $\eta_0\,\mathsf{self} = \eta\,\mathsf{self}$. Hence by $h \trianglelefteq h_0$, $\eta_0\,\mathsf{self} \in dom(Oh^0_j)$. Now we go by cases on $loctype\,\ell$, where $loctype\,\ell \leq D$.
  —$loctype\,\ell \not\leq Rep \wedge (loctype\,\ell \not\leq Own \vee \neg mscope(m, loctype\,\ell))$: Then by $conf\,\mu$ we have, $d_1 \notin locs(Rep{\downarrow})$. Hence the result follows.
  —$loctype\,\ell \leq Own \wedge mscope(m, loctype\,\ell)$: Let $\ell \in dom(Oh_k)$. Because $\eta_1\,\mathsf{self} = \ell$, by $conf\,\mu$ we have, $d_1 \in locs(Rep{\downarrow}) \Rightarrow d_1 \in dom(Rh^0_k)$. Because $loctype\,\ell \leq D$, we have $D \leq Own \vee Own \leq D$. Hence condition (b) of the analysis applies. We have two cases: Either $e = \mathsf{self}$; then $j = k$. So $d_1 \in locs(Rep{\downarrow}) \Rightarrow d_1 \in dom(Rh^0_j)$. Otherwise, $T \not\leq Rep$. So $d_1 \notin locs(Rep{\downarrow})$, and the result follows vacuously.
  —$loctype\,\ell \leq Rep$: Because e is confined at C, we have $\ell \in dom(Rh_j)$. Because $\eta_1\,\mathsf{self} = \ell$, by $conf\,\mu$ we have, $d_1 \in locs(Own{\downarrow}, Rep{\downarrow})$ implies $d_1 \in dom(Oh^0_j * Rh^0_j)$. Hence the result follows.

—$C \leq Rep$: Let $\eta\,\mathsf{self} \in dom(Rh_j)$. We want $d_1 \in locs(Own{\downarrow}, Rep{\downarrow})$ implies $d_1 \in dom(Oh^0_j * Rh^0_j)$. Because $x \neq \mathsf{self}$, we have $\eta_0\,\mathsf{self} = \eta\,\mathsf{self}$. Hence by $h \trianglelefteq h_0$, $\eta_0\,\mathsf{self} \in dom(Rh^0_j)$. Now we go by cases on $loctype\,\ell$, where $loctype\,\ell \leq D$.
  —$loctype\,\ell \not\leq Rep \wedge (loctype\,\ell \not\leq Own \vee \neg mscope(m, loctype\,\ell))$: Then by $conf\,\mu$ we have, $d_1 \notin locs(Rep{\downarrow})$. By the analysis condition (a), $T \not\leq Own$. Hence $d_1 \notin locs(Own{\downarrow}, Rep{\downarrow})$ and the result follows.
  —$loctype\,\ell \leq Own \wedge mscope(m, loctype\,\ell)$: By confinement of e at C, $\ell \in dom(Oh_j)$. By $conf\,\mu$, $d_1 \in locs(Rep{\downarrow})$ implies $d_1 \in dom(Rh^0_j)$. By the analysis condition (d), $T \not\leq Own$. Hence the result follows.
  —$loctype\,\ell \leq Rep$: Because e is confined at C, $\ell \in dom(Rh_j)$. By $conf\,\mu$ we have, $d_1 \in locs(Own{\downarrow}, Rep{\downarrow})$ implies $d_1 \in dom(Oh^0_j * Rh^0_j)$. Hence the result follows.

CASE $\Gamma \rhd x := \textbf{new}\ B$. First, we claim $\mathit{conf}\ B\,(h_1, \eta_1)$ and $h \trianglelefteq h_1$. Then by Lemma 11.5 we get $\mathit{conf}\ B\,(h_0, \eta_1)$ and $h_1 \trianglelefteq h_0$. So $h \trianglelefteq h_0$ and by Lemma 6.13 $\mathit{conf}\ C\,(h_0, \eta)$. To conclude, we argue that $\mathit{conf}\ C\,(h_0, [\eta \mid x \mapsto \ell])$ by cases on C.

—$C \not\leq \mathit{Own} \wedge C \not\leq \mathit{Rep}$: then $B \not\leq \mathit{Rep}$ so $\ell \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ by typing and hence $\mathit{conf}\ C\,(h_0, [\eta \mid x \mapsto \ell])$.

—$C \leq \mathit{Own}$: Let $h = \mathit{Ch} * (\mathit{Oh}_1 * \mathit{Rh}_1) \ldots (\mathit{Oh}_k * \mathit{Rh}_k)$ be a confining partition of h, and j such that $\eta\,\mathsf{self}$ be in $\mathit{dom}(\mathit{Oh}_j)$. If $B \not\leq \mathit{Rep}$ then $\mathit{conf}\ C\,(h_0, [\eta \mid x \mapsto \ell])$ by definition. If $B \leq \mathit{Rep}$ then we must show $\ell \in \mathit{dom}(\mathit{Rh}^0_j)$ where h_0 has confining extension $h_0 = \mathit{Ch}^0 * (\mathit{Oh}^0_1 * \mathit{Rh}^0_1) \ldots$. This is defined just as in the proof of Lemma 6.16, and we choose to put ℓ and the objects it constructs in Rh_j to obtain Rh^0_j.

—$C \leq \mathit{Rep}$: By the static analysis, $B \not\leq \mathit{Own}$. So $\mathit{Oh}^0_j = \mathit{Oh}_j$. Thus $\mathit{rng}\,[\eta \mid x \mapsto \ell] \cap \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Oh}_j * \mathit{Rh}^0_j)$. We choose to put ℓ and the objects it constructs in Rh_j to obtain Rh^0_j, which makes the inclusion hold.

It remains to prove the claims $\mathit{conf}\ B\,(h_1, \eta_1)$ and $h \trianglelefteq h_1$. In the semantic definition, $h_1 = [h \mid \ell \mapsto [\mathit{fields}\,B \mapsto \mathit{defaults}]]$ where $\ell = \mathit{fresh}(B, h)$. Define $\mathit{Bh} = [\ell \mapsto [\mathit{fields}\,B \mapsto \mathit{defaults}]]$ so $h_1 = h * \mathit{Bh}$. Let $\eta_1 = [\mathsf{self} \mapsto \ell]$. Next, we argue that $h \trianglelefteq h_1$ and $\mathit{conf}\ B\,(h_1, \eta_1)$. Because h is closed, ℓ is not in the range of any object state in h. To construct an extending partition it suffices to deal with the new object, as its addition cannot violate confinement of existing objects. We define the extension and argue by cases on B.

—$B \not\leq \mathit{Own} \wedge B \not\leq \mathit{Rep}$. For a confining partition of h_1 we extend that for h by defining $\mathit{Ch}^0 = \mathit{Ch} * \mathit{Bh}$ and using the given partition of owner islands. Because $\mathit{defaults}$ contains no locations, this is a confining partition and we have $\mathit{conf}\ B\,(h_1, \eta_1)$.

—$B \leq \mathit{Own}$. We extend the partition by adding an island $\mathit{Oh}^0_{k+1} * \mathit{Rh}^0_{k+1}$ with $\mathit{Oh}^0_{k+1} = \mathit{Bh}$ and $\mathit{Rh}^0_{k+1} = \varnothing$. This is a confining partition because $\mathit{defaults}$ has no locations and we have $\mathit{conf}\ B\,(h_1, \eta_1)$ because $\mathit{rng}\,\eta_1$ has no reps.

—$B \leq \mathit{Rep}$. Then, by the analysis we have $C \leq \mathit{Own}$ or $C \leq \mathit{Rep}$; moreover as $x \neq \mathsf{self}$, we have $\eta\ \mathsf{self} \neq \ell$, so $\eta\ \mathsf{self} \in \mathit{dom}(\mathit{Oh}_j * \mathit{Rh}_j)$ for some j. Then we can obtain a confining extension by adding Bh to Rh_j, as $\mathit{defaults}$ has no locations. As $\mathit{rng}\,\eta_1 = \{\ell\}$, we have $\mathit{conf}\ B\,(h_1, \eta_1)$ by definition.

This concludes the argument for $h \trianglelefteq h_1$ and $\mathit{conf}\ B\,(h_1, \eta_1)$.

The remaining cases are similar and can be found in the appendix. □

12. DISCUSSION AND RELATED WORK

Programmers draw pictures of pointers in heap-based data structures and often manage to get things right as far as the presence of pointers goes. For example, lists don't get disconnected. The absence of pointers is harder to picture and many bugs are due to unexpected aliasing. Expectations are raised through use of encapsulation constructs such as private fields and modules, but heap structure is not entirely manifested in language constructs. Simulation relations are often used for reasoning about abstractions and here too aliasing presents a challenge: Multiple instances of an abstraction may reference a shared client object or be shared by multiple clients —but client references to representation objects can violate encapsulation. Various notions of ownership confinement have been proposed for encapsulation of objects. We have formalized one and shown that clients are independent from confined representations. Independence is formalized by an abstraction theorem that licenses reasoning about equivalence of class implementations using simulation relations. Confinement is formalized by drawing boundaries that signify the absence of pointers.

12.1 Related work

Representation independence. The main proof technique for representation independence is so fundamental that it has appeared in many places, with a variety of names, e.g., simulation, logical relations, abstraction mappings, relational parametricity (e.g., [Plotkin 1973; Reynolds 1984; Lynch and Vaandrager 1995; de Roever and Engelhardt 1998]). Among the many uses of simulations are program transformations and justification of logics for reasoning about data abstraction and modification of encapsulated state.

Representation independence results are known for general transition systems [Milner 1971; Lynch and Vaandrager 1995], first order imperative languages [He et al. 1986; de Roever and Engelhardt 1998], higher order functional [Reynolds 1984; Mitchell 1986; 1991; 1996; Power and Robinson 2000] and higher order imperative languages [O'Hearn and Tennent 1995; Naumann 2002], and sequential object-oriented programs without heap allocation ([Cavalcanti and Naumann 2002] treats a language with class-based visibility and [Reddy 1998] treats one with instance-based visibility). As far as we know, our results are the first for shared references to mutable state, a ubiquitous feature in object-oriented and imperative programs. (The lacuna is mentioned in [Grossman et al. 2000].)

A widely held view seems to be that classical techniques based on denotational semantics and logical relations are inadequate in the face of the complex language features of interest. The combination of local state with higher order procedures makes it difficult to prove representation independence even for Algol, where procedures can be passed as arguments but not assigned to state variables [O'Hearn and Tennent 1995]. Objects exhibit similar features.

Difficulties with denotational semantics led to considerable advances using operational semantics [Gordon and Pitts 1998]. However, to get an adequate induction hypothesis for an abstraction theorem, parametricity needs to be imposed on the latent effects of procedure abstractions, either as a property to be proved or as an intrinsic feature of the semantic model [Reynolds 1981b; O'Hearn and Tennent 1995]. These conditions are most easily expressed in terms of a denotational model, but if procedures can be stored in the heap on which they act, difficult domain equations must be solved.[24] Recursive data types also lead to nontrivial domain equations. Even if solutions can be found, they may be quite complex structures that are difficult to understand and work with. Nevertheless, a modern treatment of recursive

[24]Recently Levy [2002] used functor categories to give a denotational model for a higher order language with pointers, but the model does not capture relational parametricity and the language has neither object-oriented features nor recursive types.

domain equations can allow one to make progress [Reus and Streicher 2002; Reus 2003].

For Idealized Algol, in which only integers can be stored in variables and there are no recursive types, Pitts [1997] formalizes logical relations using operational semantics. Equivalences like the Meyer-Sieber example in our Sect. 3.1 are proved.

One of the most relevant works using operational semantics is that of Grossman et al. [2000] where representation independence is approached using a dynamic notion of ownership by *principals* as in the security literature. To prove that clients are independent from the representation of an abstraction provided by a host program, a wrapper construct is used to tag code fragments with their owner (e.g., client or "host"), and to provide an opaque type for the client's view of the abstraction. This is a promising approach. However, the results so far only show "independence of evaluation" (reminiscent of nonintereference results in information flow security [Volpano et al. 1996; Abadi et al. 1999]) and do not provide a general notion of simulation. Although Grossman et al. [2000] offer their work as a simpler alternative to domain theoretic semantics, the technical treatment is somewhat intricate by the time the language is extended to include references, recursive and polymorphic types.

Except for parametric polymorphism, we treat all these features, as well as others such as subclassing, dynamic binding, type tests and casts. Although Java syntax seems less elegant than, say, lambda calculus, it has several features that ease the difficulties. Owing to name-based type equivalence and subtyping, and the binding of methods to objects via their class, we can use a denotational model with quite simple domains and fixpoint definitions in the manner of Strachey [2000] (cf. Sect. 3.1).

For applications in security and automated static checking, it is important to devise robust, comprehensible models that support not only the idealized languages of research studies but also the full languages used in practice. Denotational semantics has conceptual advantages, at least if the domains are simple enough to have a clear operational significance. However, we admit that our enthusiasm for the efficacy of denotational techniques has been tempered by the irritation of flushing out bugs in intricate definitions and induction hypotheses.

Our abstraction theorem and identity extension lemma can be used directly to prove equivalence of programs, where a program is a command in the context of a class table and designated class C. It would be reasonable to use a notion of equivalence based on field visibility: states would be equated if they are equal after hiding all fields except those visible in C. But this would beg the question whether hiding imposes encapsulation that is not intrinsic to the language. In this paper we use the finer equivalence on programs: for commands to be equivalent they must yield outcomes that are identical after garbage collection. Thus encapsulation is formulated in terms of private fields and confined reps but the identity extension lemma is expressed, in effect, in terms of local variable blocks (in the style of, e.g., He et al. [1986]).

Besides the "client interface" provided by public methods and analogous to the interfaces studied in previous work on representation independence, a class also has a "protected" interface to its subclasses. The combination of protected and public interfaces is complicated, but a thorough treatment of representation independence for object-oriented programs must take it into account. For reasoning about the protected interface, work on behavioral subclassing has used simulations to connect a class with its subclass [Liskov and Wing 1994; Leavens and Dhara 2000] but a formal connection has not been made with the use of simulations to connect alternative representations. The PhD thesis of Stata [1997] considers other aspects of the protected interface.

Confinement. Quite a few confinement disciplines have been proposed, by Hogg [1991], Almeida [1997],Vitek and Bokowski [2001], Clarke et al. [2001], Müller and Poetzsch-Heffter [2000b], Boyland [2001], Lea [2000], Aldrich et al. [2002], and Clarke [2001] (the latter has a more comprehensive recent survey). Most proposals have significant shortcomings; they disallow important design patterns or are not efficiently checkable. Although the aim is to achieve encapsulation and thereby support modular reasoning in one form or another, few proposals have been formally justified in these terms —none in terms of representation independence.

Several works justify a syntactic discipline by proving that it ensures a confinement invariant [Müller and Poetzsch-Heffter 2000b; Clarke 2001; Aldrich et al. 2002]. Others go further and show some form of modular reasoning principle, as we discuss in detail below. Existing justifications involve disparate techniques and objectives, so that it is quite hard to assess and compare confinement disciplines. One of our contributions is to show how standard semantic techniques can be used for such assessments.

The fact that type names are semantically relevant lets us use them to formulate in semantic terms a condition similar to the ownership confinement notions of Müller [2002], Clarke et al. [2001] and their predecessors [Hogg 1991; Almeida 1997]. Whereas several papers emphasize reachability via paths, our formulation of confinement emphasizes partitioning of heap objects and the one-step points-to relation. In this we were inspired by the work of Reynolds [2001] that shows the efficacy of reasoning about partition blocks that may have dangling pointers.

Reasoning on the assumption of confinement is a separate concern from enforcement or checking of confinement. Semantic considerations led us to a flexible, syntax-directed static analysis, but other analysis techniques such as model checking or theorem proving for (an approximation of) the semantic confinement property could be interesting.

It is interesting to note that we get a strong reasoning principle on the basis of ownership confinement alone, in a form that can be checked without program annotations. By contrast, other works use annotations and combine ownership with uniqueness and effects (e.g., read-only) [Clarke and Drossopoulou 2002; Aldrich et al. 2002; Müller 2002].

Confinement figures heavily in the verification logics of Müller and Poetzsch-Heffter [2000a] and in some work by the group of Nelson and Leino [Leino and Nelson 2002; Detlefs et al. 1998] where it is needed for sound reasoning about the "modifies clause" framing the scope of effects. Subsequent to the present work, Clarke and Drossopoulou [2002] state results on reasoning about effects, using a confinement discipline imposed using code annotations for confinement and effects. These works are concerned with delimiting the scope of effects, which is an important aspect of modular reasoning, but they do not address representation independence.

There has been much work on capturing encapsulation via visibility (lexical scope), using existential types and subsumption (see [Bruce et al. 1999; Bruce 2002; Pierce 2002] and references therein). None of these works addresses the problem of confinement; they are concerned with the complex typing issues for object oriented languages.

It is interesting to note that one of the main difficulties in designing safe and flexible type systems is due to the desire to eliminate or minimize the use of type testing and casting which are seen as loopholes that subvert type-based encapsulation. Indeed, parametric polymorphism has been much pursued as a means to cope with generic patterns that, in current practice, are usually coded using subsumption, casts, and type **Object** (a recent reference is the textbook by Bruce [2002]). Although parametric polymorphism has obvious merit, our results show that casts and type tests are themselves relationally parametric. It is behavioral subclassing which is at risk in some uses of casts and tests. This does not contradict [Reynolds 1984] because our language has a nominal type system [Pierce 2002]; it is the name of a type, not its set of values, that is involved with tests and casts.

Our aim is to deal with the rich languages currently in use, rather than to advance language design. It is challenging to formalize the syntax precisely yet perspicuously. Rather than devising our own idiosyncratic formalization, we adapted that of Igarashi et al. [2001]. The details differ, as our language includes imperative constructs and non-public scoping and their main concern is type soundness.

12.2 Future challenges

The language for which our results are given encompasses many important features of object oriented languages. Two major features are missing and will require substantial additional work: concurrency and parametric polymorphism. The interaction between parametric and subtyping polymorphism is non-trivial and there are a number of competing type systems. Some languages, e.g., C++, have parametric polymorphism but with significant limitations; for Java, parametric types are a late addition. We expect to extend our work to them in the future.

Ownership confinement is appropriate for reasoning about many designs in practice and we have shown through a series of examples that our notion is applicable to widely used designs such as the observer and factory patterns. Two important issues are beyond the reach of our work (and much of the previous work on confinement). The first is multiple ownership. A canonical example is a collection class with iterators. The reps for the collection are nodes of a data structure. The collection object mediates additions and deletions. To allow enumeration of elements of the collection it is common to use iterator objects which need access to the nodes of the data structure. Static analyses have been given that allow some form of multiple owners [Clarke 2001; Müller 2002; Aldrich et al. 2002]. Although our formalization of islands can be extended easily to encompass multiple owners, it is not as clear how to extend the notion of simulation in a useful way. Our result formalizes the notion that an owner instance provides an abstraction and this is easily expressed in terms of the class construct. The generalization can perhaps be expressed by grouping the related owners (e.g., the collection class and the iterator class) in a module, but this is left for future work.

The other challenging issue for confinement is ownership transfer. Consider a queue that owns objects representing tasks to be performed. For load balancing, tasks may be moved from one queue to another. In this case a task is owned by just one queue at a time and in a given state the system is confined according to the definition in this paper. A sequential program for transferring ownership from one queue might look as follows: q2.task := q1.task; q1.task := **null**. From a confined initial state this need not lead to a confined final state: there could be other references to task. But it does lead to a confined final state if q2.task is initially the only existing reference to the task. Unique references have been extensively studied so let us assume that a static analysis is given for uniqueness. Even with uniqueness, our theory fails to apply, for two reasons. The first reason is a small one: in the intermediate state two different owners reference the same task. This problem is well known and can be surmounted: It is easy to add to our language an atomic command with the effect of the above sequence [Minsky 1996] and to show, given uniqueness, that it is confined. For practical purposes one would use a static analysis to check that q1.task is a dead expression [Boyland 2001].

The second reason our theory does not apply is a technical one. To show that a method call is confined, we need that the caller's environment is confined in the final heap assuming it was confined in the initial one. We get this by using a condition stronger than confinement: from a confined state, a command or method yields a final heap that extends the initial one in the sense of Def. 6.3. All commands of our language yield heaps extended in this sense so all method meanings have this property. (See the proof of Theorem 6.17.) But, by definition of extension, $h \trianglelefteq h_0$ says that reps that exist in h have the same owners in h_0 as in h, disallowing ownership transfer.

For static analysis there are some more modest issues worthy of investigation. The simple conditions of Def. 6.9 ensure suitable confinement of the class table but they are unnecessarily strong. Methods inherited into rep classes are not risky if they do not leak self; such "anonymous methods" can be statically checked as shown by Vitek and Bokowski [2001] and Grothoff et al. [2001] in work on module-based confinement.[25] The conditions of our static analysis may also admit useful variations.

Having shown that simulation is sound one might proceed to study completeness. It is not the case that our confinement conditions are necessary in general for simulations to be preserved. A trivial simulation might depend on no confinement at all. Also, a rep could be leaked but not exploited by any client. One can see confinement as a kind of simulation which happens to be a rectangular predicate: h relates to h' just if h and h' are confined, independent of each other. This suggests folding the confinement condition into the simulation relation, an idea which is currently under study by Reddy and Yang for a Pascal-like language.[26]

[25]In fact the cited work is concerned with pragmatic aspects of the analysis and does not formalize a semantic property ensured by the analysis.

[26]Their aim is to explicate the semantic structure of languages involving heap storage. Their approach should lead to a lucid account on par with parametricity models for other languages [Reynolds 1984; 1981b; Reddy 1998]. They have defined a parametricity semantics for

For practical reasoning the benefits of treating confinement separately are clear: it accords with informal design practice, is amenable to static checking, and ensures soundness for a straightforward and modular notion of coupling.

The more practical question is how to express basic couplings and prove the simulation property for owner methods. To formalize the couplings for the observer examples one needs a formalism for inductive predicates on recursive data structures; separation logic appears promising for this purpose [Reynolds 2002].

As we discussed in conjunction with Example 8.6, representation independence licenses reasoning about equivalence of programs that are structurally similar [Banerjee et al. 2001; Riecke 1993]. This is quite adequate for uses of simulations such as static analyses and relating alternative interpretations for primitives, such as the lazy and eager access control implementations for Java [Banerjee and Naumann 2002a]. But for abstraction in program development, typically called data refinement, it is not uncommon to consider significantly different program structures and this calls for a full program logic in which something like the abstraction theorem appears as a proof rule. For first-order imperative languages, several proof systems have been given for reasoning about two versions of an abstraction [de Roever and Engelhardt 1998]. Typically, relations (especially "abstraction functions") are used to derive from one version the specification of the other version, which is then proved correct in a program logic. Logics for imperative object-oriented languages are at an early stage of development [Abadi and Leino 1997; Cavalcanti and Naumann 1999; Poetzsch-Heffter and Müller 1999; Huisman and Jacobs 2000; Huisman 2002; Reynolds 2002].

APPENDIX

A. ADDITIONAL PROOFS

Proof of Lemma 6.12

By cases on C and B. It suffices to consider $C < B$ and to deal with confinement of η in h.

—$C \leq \mathit{Rep}$. Then the hypothesis of the Lemma is falsified because $\eta\,\mathsf{self} \in \mathit{locs}(\mathit{Rep}{\downarrow})$.

—$C \not\leq \mathit{Own} \wedge C \not\leq \mathit{Rep}$. Then $B \not\leq \mathit{Own} \wedge B \not\leq \mathit{Rep}$, so $\mathit{conf}\, C\eta h \Leftrightarrow \mathit{conf}\, B\,(h,\eta)$ because both C and B are subject to condition (1) in Def. 6.4.

—$C < B \leq \mathit{Own}$. Again, both B and C are subject to the same condition, here (2) in Def. 6.4.

—$C \leq \mathit{Own} < B$. We have $\mathit{conf}\, B\,(h,\eta) \Rightarrow \mathit{conf}\, C\,(h,\eta)$ by implication between the consequents of (1) and (2) in Def. 6.4. The converse holds owing to hypothesis $\mathit{rng}\,\eta \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \varnothing$.

a Pascal-like language [Reddy and Yang 2002] in which heap cells are tuples of pointers and integers rather than objects with scoped fields. Several challenges remain to be addresssed, if this approach is to provide a foundation for reasoning about instance-based abstractions in Java-like languages using a practical confinement discipline. For example, nominal types and class-based visibility (which is not modelled by naive use of existential types).

Proof of Lemma 6.13

By cases on C. In the case $C \not\leq Own \wedge C \not\leq Rep$, we have $conf\, C\,(h,\eta) \Leftrightarrow conf\, C\,(h_0,\eta)$ because Def. 6.4(1) of $conf\, C$ is independent of the heap. For the cases $C \leq Own$ and $C \leq Rep$, we show $conf\, C\,(h_0,\eta)$ using $h \trianglelefteq h_0$. First, by definition of $\trianglelefteq$ we have $conf\, h_0$. To show that η is confined in h_0 for C, suppose

$$h = Ch * Oh_1 * Rh_1 * \ldots * Oh_k * Rh_k$$

is a confining partition of h. Let j be such that $rng\,\eta \cap locs(Own{\downarrow}, Rep{\downarrow}) \subseteq dom(Oh_j * Rh_j)$. Suppose, by $h \trianglelefteq h_0$, that this partition is extended by confining partition $h_0 = Ch^0 * Oh^0_1 * Rh^0_1 * \ldots$. In the case $C \leq Own$, we have $rng\,\eta \cap locs(Rep{\downarrow}) \subseteq dom(Rh_j) \subseteq dom(Rh^0_j)$, using $conf\, C\,(h,\eta)$ and the definition $\trianglelefteq$. The case $C \leq Rep$ is similar.

Additional cases for Lemma 6.16

Case $\Gamma \vdash x := e$. Here the heap is unchanged: $h_0 = h$ and the result holds by reflexivity of $\trianglelefteq$.

Case $\Gamma \vdash x := \mathbf{super}.m(\overline{e})$. The same argument as for method call $e.m$.

Case $\Gamma \vdash S_1;\ S_2$. Let $(\eta_1, h_1) = [\![\Gamma \vdash S_1]\!]\mu(h,\eta)$. By induction on S_1 we have $h \trianglelefteq h_1$. By confinement of S_1 we have $conf\, C\,(h_1,\eta_1)$. So we can use induction on S_2 to obtain $h_1 \trianglelefteq h_0$ and then $h \trianglelefteq h_0$ by transitivity of $\trianglelefteq$.

Case $\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}$. By induction on S_1 and S_2, using confinement of S_1 and S_2.

Case $\Gamma \vdash T\ x := e\ \mathbf{in}\ S$. Let $\eta_1 = [\eta \mid x \mapsto [\![\Gamma \vdash e : U]\!](h,\eta)]$. By $conf\, C\,(h,\eta)$ and confinement of e we have $conf\, C\,(h,\eta_1)$. Then by induction on S, using confinement of S, we get $h \trianglelefteq h_0$.

Proof of Lemma 7.3

By induction on depth. If $C \not\leq Own$ then the equality is direct from Def. 7.1(1). If $C \leq Own$ then it is possible that $CT(Own)$ declares m but $CT'(Own)$ does not (or vice versa). But in that case, by Def. 7.1(3) we have $mtype(m,C) = mtype'(m,C)$ so m must be declared in a superclass, whence $depth(m,C) = 1 + depth(m, super\,C) = 1 + depth'(m, super\,C) = depth'(m,C)$.

Proof of Lemma 7.19

Let $\Gamma_B = (\overline{x} : \overline{T}, \mathsf{self} : B)$ and $\Gamma_C = (\overline{x} : \overline{T}, \mathsf{self} : C)$. To show

$$\mathcal{R}\ (B, \overline{x}, \overline{T} \to T)\ (restr(d,B))\ (restr(d',B)) \qquad (*)$$

consider $(h,\eta) \in [\![Heap \otimes \Gamma_B]\!]$ and $(h',\eta') \in [\![Heap \otimes \Gamma_B]\!]'$ such that $conf\, B\,(h,\eta)$, $conf\, B\,(h',\eta')$, and $\mathcal{R}\ (Heap \otimes \Gamma_B)\ (h,\eta)\ (h',\eta')$. By definition of $restr$ we have $restr(d,B)(h,\eta) = d(h,\eta)$ and $restr(d',B)(h',\eta') = d'(h',\eta')$. So for $(*)$ it remains to show

$$\mathcal{R}\ (Heap \otimes T)_\perp\ (d(h,\eta))\ (d'(h',\eta')) \qquad (\dagger)$$

By Lemma 5.7(1) we have $(h,\eta) \in [\![Heap \otimes \Gamma_C]\!]$ and $(h',\eta') \in [\![Heap \otimes \Gamma_C]\!]'$. By hypothesis, C is non-rep so B is also non-rep. As $\overline{T}$ is non-rep, we have $rng\,\eta \cap$

$locs(Rep\!\downarrow) = \varnothing$ and $rng\,\eta' \cap locs(Rep'\!\downarrow) = \varnothing$. Thus Lemma 6.12 is applicable to η, η' and using hypothesis $B < C$ we obtain $conf\,C\,(h, \eta)$, and $conf\,C\,(h', \eta')$. Thus we have established the antecedents needed to use hypothesis $\mathcal{R}\ (C, \overline{x}, \overline{T} \to T)\ d\ d'$ to obtain (†).

Proof of Lemma 7.22

Case $\Gamma \vdash x : T$. Then $\mathcal{R}\ T_\bot\ (\eta x)\ (\eta' x)$ by $\mathcal{R}\ \Gamma\ \eta\ \eta'$, so the result follows by semantics of $x : T$.

Case $\Gamma \vdash \textbf{null} : B$. Then semantics is nil and $\mathcal{R}\ B_\bot\ nil\ nil$ by definition of $\mathcal{R}$.

Case $\Gamma \vdash \textbf{it} : \textbf{unit}$. Similar to **null**, as are the cases **true** and **false**.

Case $\Gamma \vdash e_1 = e_2 : \textbf{bool}$. Then, using identifiers from the semantic definition as usual, we consider cases on d_1. If $d_1 = \bot$ then $d_1' = \bot = d_1$ by induction on e_1 and definition of $\mathcal{R}\,T$. Hence, by semantics of $e_1 = e_2$, $[\![\Gamma \vdash e_1 = e_2 : \textbf{bool}]\!](h, \eta) = [\![\Gamma \vdash e_1 = e_2 : \textbf{bool}]\!]'(h', \eta')$ and thus

$$\mathcal{R}\ \textbf{bool}_\bot\ ([\![\Gamma \vdash e_1 = e_2 : \textbf{bool}]\!](h, \eta))\ ([\![\Gamma \vdash e_1 = e_2 : \textbf{bool}]\!]'(h', \eta')) \qquad (*)$$

The argument is symmetric for $d_2 = \bot$.

If none of d_1, d_1', d_2, d_2' are $\bot$ then, by induction on e_1 we have $\mathcal{R}\ (T_1)_\bot\ d_1\ d_1'$. Thus, by Lemma 7.12, $d_1 = d_1'$. Similarly, $d_2 = d_2'$. Hence $d_1 = d_2$ iff $d_1' = d_2'$, whence the result $(*)$ holds by semantics.

Additional cases for Lemma 7.23

Case $\Gamma \vdash x := \textbf{super}.m(\overline{e})$.

By $\mathcal{R}\,\Gamma\,\eta\,\eta'$ we have $\mathcal{R}\ C\ \ell\ \ell'$, hence $\ell = \ell'$ by Lemma 7.12. By $conf\,C\,(h, \eta)$ and $conf\,C\,(h', \eta')$ we have $\ell \notin locs(Rep\!\downarrow)$ and $\ell \notin locs(Rep'\!\downarrow)$. Let $\eta_1 = [\mathsf{self} \mapsto \ell, \overline{x} \mapsto \overline{d}]$ and $\eta_1' = [\mathsf{self} \mapsto \ell, \overline{x} \mapsto \overline{d}']$. By confinement of $x := \textbf{super}.m(\overline{e})$ (Def. 6.7) we have confined arguments, i.e., $conf\ (super C)\ (h, \eta_1)$ and $conf\ (super C)\ (h', \eta_1')$

By Lemma 7.22 for $\overline{e}$, and considering the non-$\bot$ case, we have $\mathcal{R}\ \overline{U}\ \overline{d}\ \overline{d}'$, whence, by Lemma 7.13, $\mathcal{R}\ \overline{T}\ \overline{d}\ \overline{d}'$. From $\mathcal{R}\ C\ \ell\ \ell'$ we get $\mathcal{R}\ (super C)\ \ell\ \ell'$ by Lemma 7.13, and thus $\mathcal{R}\ [\overline{x} : \overline{T}, this : super C]\ \eta_1\ \eta_1'$. From $\mathcal{R}\ MEnv\ \mu\ \mu'$ we get

$$\mathcal{R}\ (super C, mtype(m, super C))\ (\mu(super C)m)\ (\mu'(super C)m)$$

hence, as h, h', η_1, η_1' are confined and related, $\mathcal{R}\ (Heap \otimes T)\ (h_1, d_1)\ (h_1', d_1')$ where $(h_1, d_1) = \mu(super C)m(h, \eta)$ and $(h_1', d_1') = \mu'(super C)m(h', \eta')$. Thus $\mathcal{R}\ T\ d_1\ d_1'$ and $\mathcal{R}\ Heap\ h_1\ h_1'$. It remains to show that the updated stores $[\eta \mid x \mapsto d_1]$ and $[\eta' \mid x \mapsto d_1']$ are related. This follows from $\mathcal{R}\ T\ d_1\ d_1'$ and $T \le \Gamma x$ using Lemma 7.13.

Case $\Gamma \vdash S_1;\ S_2$.

As usual, we consider the non-$\bot$ case. By induction on S_1 we have $\mathcal{R}\ (Heap \otimes \Gamma)_\bot\ (h_1, \eta_1)\ (h_1', \eta_1')$. Moreover, as S_1 is a constituent of a method in CT and CT', by confinement of S_1 we have $conf\,C\,(h_1, \eta_1)$ and $conf\,C\,(h_1', \eta_1')$, so we can use induction on S_2 to obtain the result.

Case $\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}$. Similar to case of sequence, but also using Lemma 7.22 for e.

Case $\Gamma \vdash T\ x := e$ **in** S.

By Lemma 7.22 for e we have $\mathcal{R}\ U_\perp\ d\ d'$. If $d = \perp$, then $d' = \perp$ and both semantics yield $\perp$. Otherwise, we have $\mathcal{R}\ T\ d\ d'$ by the corollary to Lemma 7.12. Thus, from $\mathcal{R}\ \Gamma\ \eta\ \eta'$ we obtain $\mathcal{R}\ (\Gamma, x:T)\ \eta_1\ \eta_1'$ where $\eta_1 = [\eta \mid x \mapsto d]$ and $\eta_1' = [\eta' \mid x \mapsto d']$ as in the semantic definition. In order to use induction on S, we need to show $\mathit{conf}\, C\, (h, \eta_1)$ and $\mathit{conf}\, C\, (h', \eta_1')$. From condition (1) in Def. 6.9 of confinement for CT, e is confined. In the case $C \nleq \mathit{Own}$, confinement of e yields $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ and thus $\mathit{conf}\, C\, (h, \eta_1)$. In the case $C < \mathit{Own}$, confinement of e yields $d \in \mathit{locs}(\mathit{Rep}{\downarrow}) \Rightarrow d \in \mathit{dom}(\mathit{Rh}_j)$ for some partition and j with $\eta\,\mathsf{self} \in \mathit{dom}(\mathit{Oh}_j)$. This is the condition required for $\mathit{conf}\, C\, (h, \eta_1)$ in this case. Similarly, we get $\mathit{conf}\, C\, (h', \eta_1')$. Now, by induction on S we get that both semantics are $\perp$ or else the result states from S satisfy $\mathcal{R}\ (\mathit{Heap} \otimes \Gamma, x:T)_\perp\ (h_1, \eta_2)\ (h_1', \eta_2')$. In the latter case, $\mathcal{R}\ (\mathit{Heap} \otimes \Gamma)\ (h_1, (\eta_2 \downharpoonleft x))\ (h_1', (\eta_2' \downharpoonleft x))$ as required.

Proof of Lemma 11.3(2)

Again the proof proceeds by cases on C. In each case we show $\mathit{conf}\ (\mathit{super} C)\ (h, \eta_1)$, noting that $\ell = (\eta\,\mathsf{self})$.

—$C \nleq \mathit{Rep} \wedge C \nleq \mathit{Own}$: By confinement of η at C, we have $\ell \notin \mathit{locs}(\mathit{Rep}{\downarrow})$. Because $\overline{e}$ is confined at C, we have $d_i \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ for all $d_i \in \overline{d}$. Thus $\mathit{rng}\, \eta_1 \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \varnothing$. And, since $C < \mathit{super} C$, we have $\mathit{conf}\ (\mathit{super} C)\ (h, \eta_1)$ by Lemma 6.12 and Def. 6.4(1).

—$C \leq \mathit{Own}$: Choose a confining partition and let j be such that $\ell = (\eta\,\mathsf{self}) \in \mathit{dom}(\mathit{Oh}_j)$. Since $C < \mathit{super} C$ we have $\mathit{super} C \leq \mathit{Own}$ ($\mathit{Own} < \mathit{super} C$ is impossible by definition of $\mathit{super} C$). Because $\overline{e}$ is confined at C, we have $d_i \in \mathit{locs}(\mathit{Rep}{\downarrow}) \Rightarrow d_i \in \mathit{dom}(\mathit{Rh}_j)$ for all $d_i \in \overline{d}$. Thus $\mathit{rng}\, \eta_1 \cap \mathit{locs}(\mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Rh}_j)$ proving $\mathit{conf}\ (\mathit{super} C)\ (h, \eta_1)$ by Def. 6.4(2).

—$C \leq \mathit{Rep}$: Choose a confining partition and let j be such that $\ell = (\eta\,\mathsf{self}) \in \mathit{dom}(\mathit{Rh}_j)$. Since $C < \mathit{super} C$ we have $\mathit{super} C \leq \mathit{Rep}$ ($\mathit{Rep} < \mathit{super} C$ is impossible by definition of $\mathit{super} C$). Because $\overline{e}$ is confined at C, we have $d_i \in \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow}) \Rightarrow d_i \in \mathit{dom}(\mathit{Oh}_j * \mathit{Rh}_j)$ for all $d_i \in \overline{d}$. Thus $\mathit{rng}\, \eta_1 \cap \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Oh}_j * \mathit{Rh}_j)$ proving $\mathit{conf}\ (\mathit{super} C)\ (h, \eta_1)$ by Def. 6.4(3).

Additional cases for Lemma 11.4

Case $\Gamma \rhd x : \Gamma\, x$. Then $d = \eta\, x$. Confinement of x follows because the conditions for d are exactly the same as the conditions for η and η is confined.

Cases $\Gamma \rhd \mathbf{null} : B$, $\Gamma \rhd \mathbf{true} : \mathbf{bool}$, $\Gamma \rhd \mathbf{false} : \mathbf{bool}$, $\Gamma \rhd \mathbf{it} : \mathbf{unit}$, $\Gamma \rhd e\ \mathbf{is}\ B : \mathbf{bool}$. For **null** the result holds since $\mathit{nil} \notin \mathit{Loc}$ and for $\mathbf{true}, \mathbf{false}, \mathbf{it}, e\ \mathbf{is}\ B$ the result holds by Lemma 6.11.

Case $\Gamma \rhd (B)e : B$. Then $d = \ell$ and the result follows by induction on e for each subcase of C.

Proof of Lemma 11.5

First we show that h_1 is confined, where we have the following cases on $\mathit{super} C$:

—$\mathit{super} C = \mathbf{Object}$: then $h_1 = h$. So $\mathit{conf}\, h$ by hypothesis and $h \trianglelefteq h_1$ by reflexivity of $\trianglelefteq$.

—$\mathit{super}\,C <$ **Object**: as $\mathit{super}\,C \ll C$, we can appeal to induction for $\mathit{super}\,C$ to obtain $\mathit{conf}\,h_1$ and $h \trianglelefteq h_1$. Now by Lemma 6.13 we have $\mathit{conf}\,C\,(h_1, \eta)$.

It remains to show $\mathit{conf}\,h_0$ and $h \trianglelefteq h_0$. This is a consequence of a more general

Claim: For the given C, suppose $\mathsf{self}\!:\!C \vdash S$ is a command with no method calls and $\mathsf{self}\!:\!C \rhd S$. Moreover, suppose that for any **new** B that occurs in S we have $B \sqsubset C$. Then $\mathsf{self}\!:\!C \vdash S$ is confined.

Applying the claim to $\mathit{constr}\,C$, we get $\mathit{conf}\,h_0$. Then Lemma 6.14 applies, to yield $h_1 \trianglelefteq h_0$. So finally $h \trianglelefteq h_0$ by transitivity.

The proof of the claim is by structural induction on S. The argument is the same as the proof of Lemma 11.6, except that in the case of **new** that proof appeals to Lemma 11.5 whereas here we appeal to the induction hypothesis. This use of induction is sound because for any **new** B in the constructor, $B \sqsubset C$ and hence $B \ll C$.

Additional cases for Lemma 11.6

Case $\Gamma \rhd x := e$. Here $h_0 = h$, hence confinement of h_0 follows because $\mathit{conf}\,C\,(h, \eta)$. To show $\mathit{conf}\,C\,(h_0, \eta_0)$, we go by cases on C. First, as $\Gamma \rhd e\!:\!T$, by Lemma 11.4 we have e is confined. Choose a confining partition of h and let j be such that $\eta\,\mathsf{self} \in \mathit{dom}(\mathit{Oh}_j)$. As $x \neq \mathsf{self}$ we have $\eta\,\mathsf{self} = \eta_0\,\mathsf{self}$.

—$C \not\leq \mathit{Rep} \wedge C \not\leq \mathit{Own}$: We must show $\mathit{rng}\,\eta_0 \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \varnothing$, which follows because $d \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ and because $\mathit{rng}\,\eta \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \varnothing$ by $\mathit{conf}\,C\,(h, \eta)$.

—$C \leq \mathit{Own}$: We must show $\mathit{rng}\,\eta_0 \cap \mathit{locs}(\mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Rh}_j)$, which follows because $\{d\} \cap \mathit{locs}(\mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Rh}_j)$ by confinement of e at C and because $\mathit{rng}\,\eta \cap \mathit{locs}(\mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Rh}_j)$ by $\mathit{conf}\,C\,(h, \eta)$.

—$C \leq \mathit{Rep}$: We must show $\mathit{rng}\,\eta_0 \cap \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Oh}_j * \mathit{Rh}_j)$, which follows because $\{d\} \cap \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Oh}_j * \mathit{Rh}_j)$ by confinement of e at C and because $\mathit{rng}\,\eta \cap \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Oh}_j * \mathit{Rh}_j)$ by $\mathit{conf}\,C\,(h, \eta)$.

Case $\Gamma \rhd x := \mathbf{super}.m(\overline{e})$. Here $h_0 = h_1$ and $\eta_0 = [\eta \mid x \mapsto d_1]$. Because $\Gamma \rhd \overline{e} : \overline{U}$, by Lemma 11.4, $\overline{e}$ is confined at C. By Lemma 11.3 we have $\mathit{conf}\,(\mathit{super}\,C)\,(h, \eta_1)$. Then by assumption $\mathit{conf}\,\mu$ we get $\mathit{conf}\,(\mathit{super}\,C)\,(h_0, \eta_1)$. Hence h_0 is confined. To show $\mathit{conf}\,C\,(h_0, \eta_0)$, we go by cases on C. Recall that $\ell = \eta\,\mathsf{self}$, and, as $x \neq \mathsf{self}$, $\ell = \eta_0\,\mathsf{self}$.

—$C \not\leq \mathit{Rep} \wedge C \not\leq \mathit{Own}$: As $C < \mathit{super}\,C$ we have $\mathit{super}\,C \not\leq \mathit{Rep} \wedge \mathit{super}\,C \not\leq \mathit{Own}$. By $\mathit{conf}\,\mu$, $d_1 \notin \mathit{locs}(\mathit{Rep}{\downarrow})$. Hence $\mathit{rng}\,\eta_0 \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \varnothing$ by $\mathit{conf}\,C\,(h, \eta)$.

—$C \leq \mathit{Own}$: Let $\eta\,\mathsf{self} \in \mathit{dom}(\mathit{Oh}_j)$ for some j in the confining partition of h. As $C < \mathit{super}\,C$ we have either $\mathit{super}\,C \leq \mathit{Own}$ ($\mathit{Own} < \mathit{super}\,C$ is impossible by definition of super). By $\mathit{conf}\,\mu$, $d_1 \notin \mathit{locs}(\mathit{Rep}{\downarrow})$ and $h \trianglelefteq h_0$. Hence $\mathit{rng}\,\eta_0 \cap \mathit{locs}(\mathit{Rep}{\downarrow}) = \mathit{rng}\,\eta \cap \mathit{locs}(\mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Rh}_j)$ by $\mathit{conf}\,C\,(h, \eta)$. As $h \trianglelefteq h_0$, $\mathit{dom}(\mathit{Rh}_j) \subseteq \mathit{dom}(\mathit{Rh}_{0_j})$. That is, $\mathit{rng}\,\eta_0 \cap \mathit{locs}(\mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Rh}_{0_j})$.

—$C \leq \mathit{Rep}$: Because $\mathit{loctype}\,\ell \leq C$, let $\ell \in \mathit{dom}(\mathit{Rh}_j)$ for some j in the confining partition of h. As $C < \mathit{super}\,C$ we have either $\mathit{super}\,C \leq \mathit{Rep}$ ($\mathit{Rep} < \mathit{super}\,C$ is impossible by definition of super). By $\mathit{conf}\,\mu$, $d_1 \in \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow}) \Rightarrow d_1 \in \mathit{dom}(\mathit{Oh}_{0_j} * \mathit{Rh}_{0_j})$ and $h \trianglelefteq h_0$. Hence $\mathit{rng}\,\eta_0 \cap \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Oh}_{0_j} * \mathit{Rh}_{0_j})$ by $\mathit{conf}\,C\,(h, \eta)$ and Def. 6.3.

CASE $\Gamma \rhd S_1;\ S_2$. By induction on S_1, h_1 is confined and *conf* $C\,(h_1, \eta_1)$. Moreover, if S_1 is a method call, it has confined argument values. Now by induction on S_2, h_2 is confined and *conf* $C\,(h_2, \eta_2)$. And, if S_2 is a method call, it has confined argument values. Hence all method calls in $S_1;\ S_2$ have confined argument values.

CASE $\Gamma \rhd$ **if** e **then** S_1 **else** S_2 **fi**. By Lemma 6.11, e is confined at C. If $b = \mathit{true}$, result follows by induction on S_1 and if $b = \mathit{false}$, result follows by induction on S_2.

CASE $\Gamma \rhd T\ x := e$ **in** S. Because $\Gamma \rhd e : U$ we have by Lemma 11.4 that e is confined at C. And, because $x \neq \mathsf{self}$ and *conf* $C\,(h, \eta)$, we get *conf* $C\,(h, \eta_1)$. Since $\Gamma, x : T \rhd S$, by induction on S we have *conf* $C\,(h_1, \eta_2)$ and all method calls in S have confined argument values. Hence h_1 is confined and *conf* $C\,(h_1, \eta_2 \downharpoonleft x)$ and all method calls in $T\ x := e$ **in** S have confined argument values.

ACKNOWLEDGMENTS

Our work benefitted from discussions with a number of people. For particularly useful technical help and encouragement we thank Torben Amtoft, Steve Bloom, Paulo Borba, Sophia Drossopoulou, Nevin Heintze, Doug Lea, Peter Müller, Peter O'Hearn, Uday Reddy, David Schmidt, and Hongseok Yang. We are also grateful for very helpful feedback from the anonymous POPL and JACM referees.

REFERENCES

ABADI, M., BANERJEE, A., HEINTZE, N., AND RIECKE, J. G. 1999. A core calculus of dependency. In *ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 147–160.

ABADI, M. AND CARDELLI, L. 1996. *A Theory of Objects*. Springer-Verlag.

ABADI, M. AND LEINO, K. R. M. 1997. A logic of object-oriented programs. In *Theory and Practice of Software Development (TAPSOFT)*. Springer-Verlag. Expanded in DEC SRC report 161.

ALDRICH, J., KOSTADINOV, V., AND CHAMBERS, C. 2002. Alias annotations for program understanding. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. ACM Press.

ALMEIDA, P. S. 1997. Balloon types: Controlling sharing of state in data types. In *European Conference on Object Oriented Programming (ECOOP)*. Lecture Notes in Computer Science. Springer-Verlag, 32–59.

ARNOLD, K. AND GOSLING, J. 1998. *The Java Programming Language, second edition.* Addison-Wesley.

BANERJEE, A., HEINTZE, N., AND RIECKE, J. G. 2001. Design and correctness of program transformations based on control-flow analysis. In *Intl. Symp. on Theoretical Aspects of Computer Software (TACS)*. Lecture Notes in Computer Science. Springer-Verlag, 420–447.

BANERJEE, A. AND NAUMANN, D. A. 2002a. Representation independence, confinement and access control. In *ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 166–177.

BANERJEE, A. AND NAUMANN, D. A. 2002b. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 253–270.

BOYLAND, J. 2001. Alias burying: Unique variables without destructive reads. *Software Practice and Experience 31,* 6.

BRUCE, K. B. 2002. *Foundations of Object-Oriented Programming Languages: Types and Semantics.* MIT Press.

BRUCE, K. B., CARDELLI, L., AND PIERCE, B. C. 1999. Comparing object encodings. *Information and Computation 155,* 1/2, 108–133.

Cavalcanti, A. L. C. and Naumann, D. A. 1999. A weakest precondition semantics for an object-oriented language of refinement. In *FM'99 - Formal Methods, Volume II*. Number 1709 in Lecture Notes in Computer Science. Springer-Verlag, 1439–1459.

Cavalcanti, A. L. C. and Naumann, D. A. 2002. Forward simulation for data refinement of classes. In *Formal Methods Europe*. Lecture Notes in Computer Science, vol. 2391. Springer-Verlag, 471–490.

Clarke, D. 2001. Object ownership and containment. Ph.D. thesis, University of New South Wales, Australia.

Clarke, D. and Drossopoulou, S. 2002. Ownership, encapsulation and the disjointness of type and effect. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. ACM Press.

Clarke, D. G., Noble, J., and Potter, J. M. 2001. Simple ownership types for object containment. In *European Conference on Object Oriented Programming (ECOOP)*. Lecture Notes in Computer Science. Springer-Verlag.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms*. MIT Press.

Cousot, P. and Cousot, R. 1977. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices*. Vol. 12. ACM Press, 1–12.

Dahl, O.-J. and Nygaard, K. 1966. Simula: an Algol-based simulation language. *Communications of the ACM 9,* 9, 671–678.

Davey, B. and Priestley, H. 1990. *Introduction to Lattices and Order*. Cambridge University Press.

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

Detlefs, D. L., Leino, K. R. M., and Nelson, G. 1998. Wrestling with rep exposure. Research Rep. 156, DEC Systems Research Center.

Dhara, K. K. and Leavens, G. T. 1996. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*. IEEE Computer Society Press, 258–267.

Donahue, J. E. 1979. On the semantics of "data type". *SIAM Journal of Computing 8,* 4, 546–560.

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

German, S. M., Clarke, E. M., and Halpern, J. Y. 1989. Reasoning about procedures as parameters in the language L4. *Information and Computation 83*, 265–359.

Gong, L. 1999. *Inside Java 2 Platform Security*. Addison-Wesley.

Gordon, A. D. and Pitts, A. M., Eds. 1998. *Higher Order Operational Techniques in Semantics*. Cambridge University Press.

Grossman, D., Morrisett, G., and Zdancewic, S. 2000. Syntactic type abstraction. *ACM Trans. Program. Lang. Syst. 22,* 6.

Grothoff, C., Palsberg, J., and Vitek, J. 2001. Encapsulating objects with confined types. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. ACM Press.

Haynes, C. T. 1984. A theory of data type representation independence. In *International Symposium on Semantics of Data Types*, G. Kahn, D. B. MacQueen, and G. Plotkin, Eds. Lecture Notes in Computer Science, vol. 173. Springer-Verlag, 157–175.

He, J., Hoare, C. A. R., and Sanders, J. 1986. Data refinement refined (resumé). In *European Symposium on Programming*. Lecture Notes in Computer Science, vol. 213. Springer-Verlag.

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

Hogg, J. 1991. Islands: Aliasing protection in object-oriented languages. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. SIGPLAN Notices, vol. 26. ACM Press.

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

Huisman, M. 2002. Verification of Java's AbstractCollection class: A case study. Lecture Notes in Computer Science, vol. 2386. Springer-Verlag, 175–194.

Huisman, M. and Jacobs, B. 2000. Java program verification via a Hoare logic with abrupt termination. In *FASE 2000*. Lecture Notes in Computer Science. Springer-Verlag.

Igarashi, A., Pierce, B., and Wadler, P. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst. 23,* 3 (May), 396–459.

Jones, C. B. 1986. *Systematic software development using VDM.* International Series in Computer Science. Prentice-Hall.

Lea, D. 2000. *Concurrent Programming in Java*, Second ed. Addison-Wesley.

Leavens, G. T. and Dhara, K. K. 2000. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, Chapter 6, 113–135.

Leino, K. R. M. and Nelson, G. 2002. Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst. 24,* 5.

Levy, P. 2002. Possible world semantics for general storage in call-by-value. In *Computer Science Logic*. Number 2471 in Lecture Notes in Computer Science. Springer-Verlag.

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

Liskov, B. H. and Wing, J. M. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16,* 6.

Lynch, N. and Vaandrager, F. 1995. Forward and backward simulations part I: Untimed systems. *Information and Computation 121,* 2.

Meyer, A. R. and Sieber, K. 1988. Towards fully abstract semantics for local variables: Preliminary report. In *ACM Symposium on Principles of Programming Languages (POPL).* ACM Press, 191–203.

Milner, R. 1971. An algebraic definition of simulation between programs. In *Proceedings of Second Intl. Joint Conf. on Artificial Intelligence.* 481–489.

Minsky, N. H. 1996. Towards alias-free pointers. In *European Conference on Object Oriented Programming (ECOOP).* Lecture Notes in Computer Science, vol. 1098. 189–??

Mitchell, J. C. 1986. Representation independence and data abstraction. In *ACM Symposium on Principles of Programming Languages (POPL).* ACM Press, 263–276.

Mitchell, J. C. 1991. On the equivalence of data representations. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lifschitz, Ed. 305–330.

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

Moggi, E. 1991. Notions of computation and monads. *Information and Computation 93*, 55–92.

Morgan, C. and Gardiner, P. 1990. Data refinement by calculation. *Acta Inf. 27*, 481–503.

Müller, P. 2002. *Modular Specification and Verification of Object-Oriented programs.* Lecture Notes in Computer Science, vol. 2262. Springer-Verlag.

Müller, P. and Poetzsch-Heffter, A. 2000a. Modular specification and verification techniques for object-oriented software components. In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press.

Müller, P. and Poetzsch-Heffter, A. 2000b. A type system for controlling representation exposure in Java. In *ECOOP Workshop on Formal Techniques for Java Programs.* Technical Report 269, Fernuniversität Hagen.

Naumann, D. A. 2001. Predicate transformer semantics of a higher order imperative language with record subtyping. *Sci. Comput. Program. 41,* 1, 1–51.

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

O'Hearn, P. W. and Tennent, R. D. 1995. Parametricity and local variables. *Journal of the ACM 42,* 3, 658–709.

Olderog, E.-R. 1983. Hoare's logic for programs with procedures — what has been achieved? In *Proceedings, Logics of Programs*, E. Clarke and D. Kozen, Eds. Lecture Notes in Computer Science, vol. 164. Springer-Verlag.

Pierce, B. C. 2002. *Types and Programming Languages.* MIT Press.

Pitts, A. M. 1997. Reasoning about local variables with operationally-based logical relations. In *Algol-Like Languages*, P. W. O'Hearn and R. D. Tennent, Eds. Vol. 2. Birkhauser, Chapter 17, 173–193. Reprinted from *Proceedings Eleventh Annual IEEE Symposium on Logic in Computer Science*, Brunswick, NJ, July 1996, pp 152–163.

Plotkin, G. 1973. Lambda definability and logical relations. Tech. Rep. SAI-RM-4, University of Edinburgh, School of Artificial Intelligence.

Poetzsch-Heffter, A. and Müller, P. 1999. A programming logic for sequential Java. In *Programming Languages and Systems (ESOP)*, S. D. Swierstra, Ed. Lecture Notes in Computer Science, vol. 1576. Springer-Verlag, 162–176.

Power, A. J. and Robinson, E. P. 2000. Logical relations and data abstraction. In *Proceedings of Computer Science Logic (CSL)*, P. Clote and H. Schwichtenberg, Eds. Lecture Notes in Computer Science. Springer-Verlag, 497–511.

Reddy, U. S. 1998. Objects and classes in Algol-like languages. In *Fifth Intern. Workshop on Foundations of Object-oriented Languages.* Full version to appear in Information and Computation.

Reddy, U. S. and Yang, H. 2002. Correctness of data representations involving heap data structures. Presented at Pointerfest workshop, August, Queen Mary University of London.

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

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

Reynolds, J. C. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference.* ACM Press, 717–740.

Reynolds, J. C. 1974. Towards a theory of type structure. In *Colloques sur la Programmation.* Lecture Notes in Computer Science, vol. 19. 408–425.

Reynolds, J. C. 1978. User-defined types and procedural data structures as complementary approaches to data abstraction. In *Programming Methodology*, D. Gries, Ed. Springer-Verlag, 309–317.

Reynolds, J. C. 1981a. *The Craft of Programming.* Prentice-Hall.

Reynolds, J. C. 1981b. The essence of Algol. In *Algorithmic Languages*, J. W. de Bakker and J. C. van Vliet, Eds. North-Holland.

Reynolds, J. C. 1984. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, R. Mason, Ed. North-Holland, 513–523.

Reynolds, J. C. 2001. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*. Palgrave.

Reynolds, J. C. 2002. Separation logic: a logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science (LICS).* IEEE Computer Society Press.

Riecke, J. G. 1993. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science 3,* 4, 387–415.

Stata, R. 1997. Modularity in the presence of subclassing. Research Report 145, DEC SRC, 130 Lytton Avenue Palo Alto, CA 94301.

Strachey, C. 2000. Fundamental concepts in programming languages. *Higher Order and Symbolic Computation 13,* 1. Originally appeared in 1967 Lecture notes, International Summer School in Computer Programming, Copenhagen.

Vitek, J. and Bokowski, B. 2001. Confined types in Java. *Software Practice and Experience 31,* 6, 507–532.

Volpano, D., Smith, G., and Irvine, C. 1996. A sound type system for secure flow analysis. *Journal of Computer Security 4,* 3, 167–187.

Wallach, D., Appel, A., and Felten, E. 2000. SAFKASI: a security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology 9,* 4.

Received Month Year; revised Month Year; accepted Month Year