Ownership transfer and abstraction KSU CIS TR 2004-1 October, 2003

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

[1] Computing and Information Sciences, Kansas State University, USA
ab@cis.ksu.edu
[2] Computer Science, Stevens Institute of Technology, USA
naumann@cs.stevens-tech.edu

Abstract. Ownership confinement expresses encapsulation in heap structures, in support of modular reasoning about effects, representation independence, and other properties. This paper studies heap encapsulation from the perspective of substitutability for the class construct of Java-like languages and a particular form of confinement is justified by a representation independence result. A syntax-directed static analysis is specified and proved sound for checking confinement in the presence of ownership transfer.

1 Introduction

Contemporary programming languages offer well developed features in support of data abstraction, such as module-scoped fields and procedures/methods, enforced through strong typing. But few programmers have failed to discover that the encapsulation provided by standard constructs in contemporary object-oriented languages is easily breached by accidental or intentional sharing. In the last decade a number of disciplines have been proposed for expressing and enforcing encapsulation on heap structure, using both types and logics, to facilitate reasoning about properties of programs. Such properties include memory safety of low-level code [20], data races [17, 5], software architecture [1], modular reasoning about functional correctness of source programs [26, 9].

A fundamental issue is the encapsulation of data representations. Modular reasoning about "modifies" specifications can be achieved if representation objects (called *reps* in the sequel) are not accessible except via the abstraction's interface [18, 21]. Class substitutability, i.e., equivalence or refinement between programs using different versions of an abstract data type can be shown in terms of a single instance of the type, provided the reps for that instance are suitably encapsulated [2, 6].

Many alias controls involve a notion of object ownership. Despite impressive advances, none of the extant proposals for ownership are entirely satisfactory, for various reasons, e.g., failing to enforce a property sufficiently strong for soundness of the intended reasoning or imposing unnecessarily strong restrictions that conflict with common design patterns and data structure optimizations. We emphasize two shortcomings in particular. First, while there are attractive formulations in terms of typing, this requires substantial annotations and these annotations are subject to significantly different rules depending on what sort of reasoning is of interest. By contrast, ordinary program types, and some refinements such as non-null types [12], are useful for a variety of purposes.

A second major shortcoming is that the relation of ownership is fixed. For example, it has often been suggested that each node of a data structure "owns" its successor nodes, motivating a hierarchical notion of ownership. But if a node's owner cannot be changed then imperative reorganization of structure is quite restricted. Another class of examples involves the explicit need to change ownership, e.g., initialization of a rep by a client or moving a task from one queue to another. In some proposals, the connection between an object and its owner has an explicit, immutable runtime representation [6]. In others, static rules preclude change of ownership.

In this paper we focus on representation independence, which is a way to assess encapsulation in programs independent from particular forms of specification. The first contribution of this paper is to define a state predicate expressing a notion of ownership that is sufficiently strong for soundness of single-instance reasoning about equivalence of data representations in the presence of ownership transfer. The second contribution is to identify the isolation condition needed at transfer points and to give a sound static analysis for ownership confinement using rules that involve only ordinary program types.

In Section 2 we use examples to illustrate an ownership confinement property sufficient for modular reasoning about equivalence of class implementations. In Section 3 we consider further examples to illustrate transfer of reps between instances of an abstract data type as well as transfer between client and abstraction. Section 4 defines an illustrative sequential class-based language with unrestricted aliasing and recursive types. The semantics is sufficiently abstract and compositional to support proof of representation independence but it makes it difficult to formalize transferrable ownership. Section 5 formalizes ownership confinement and an isolation property akin to external uniqueness [10]. Section 6 shows that, modulo isolation, confinement can be statically enforced using constraints expressed in terms of program types without additional annotation and with sufficient flexibility to allow interesting design patterns (including all of the examples in Sections 2 and 3) beyond the reach of previous type systems. Section 7 sketches the representation independence theorem. Section 8 concludes. For lack of space, proofs are omitted. The appendix gives an illustrative case from the proof for soundness of the static analysis, along with key lemmas.

2 Representation independence and ownership

Consider the example program in Fig. 1. Class TaskQueue maintains a linked list of tasks and has a (simplistic) interface for executing tasks. Of course a sequence can also be represented using an array. The programmer used a private field in TaskQueue to reference the list, hoping to encapsulate the data structure and thus allow it to be replaced by code using a different representation, say an array of Task, without affecting client code. If the internal representation is indeed encapsulated, it should be sound to reason about the replacement in

```
class Task { // serves as interface
  unit exec(){ abort } }

class Node { // used by TaskQueue as rep
  Task ob;
  Node nxt; // next node in list
  unit setOb(Task x){ self.ob := x }        Task getOb(){ result := self.ob }
  unit setNxt(Node n){ self.nxt := n }        Node getNxt(){ result := self.nxt }
  unit exec(){ self.ob.exec() } }

class TaskQueue { // owner, maintains list of Tasks
  Node fst; // first node in list
  unit add(Task ob){
    Node n := new Node; n.setOb(ob); n.setNxt(self.fst); self.fst := n }
  unit execOneTaskVsn1(){
    Node n := "choose a task in the queue"; Task t := n.getOb(); t.exec() }
  unit execOneTaskVsn2(){
    Node n := "choose a task in the queue"; n.exec() } }
```

Fig. 1. Task queue. In our illustrative language, object types are implicitly references, fields are private to their class, and the final value of variable result is returned.

local terms: one describes the connection between the array and linked list data structures (a *coupling relation*) and verifies that relation is preserved by the two versions of each method of class TaskQueue (the *simulation* property).

If instances of TaskQueue were allowed to have lists sharing a common tail then the reasoner must resort to global reasoning about the entire heap and in particular all instances of the abstraction rather than a single exemplary one.[1] We aim for a discipline of confinement for reps that justifies a *representation independence* theorem [23–25, 15, 19]. It says: if the methods of two versions of TaskQueue have the simulation property for a single instance of TaskQueue (together with its reps), then simulation holds for all client programs and unboundedly many instances.

For programs in an object-oriented language, formulation of a useful theorem must take into account not just the interface for clients but also the interface to subclasses of the abstraction. Moreover, there can be outgoing calls to the client (as in the execOneTask methods) which can lead to calls back to the abstraction. The abstraction theorem in the sequel (Theorem 3, Section 7) handles these issues adequately, extending our previous work [2] to encompass ownership transfer.

We formalize encapsulation in two steps. The first step is to partition the heap in terms of the types of the objects of interest. Then restrictions are imposed on references between blocks of the partition. Suppose *Own* is the name of the class for which two versions are to be shown equivalent. Each version of *Own*

[1] If clients can gain direct access to reps, then the replacement is not likely to be possible at all unless all clients are known, in which case non-modular reasoning about those clients is necessary.

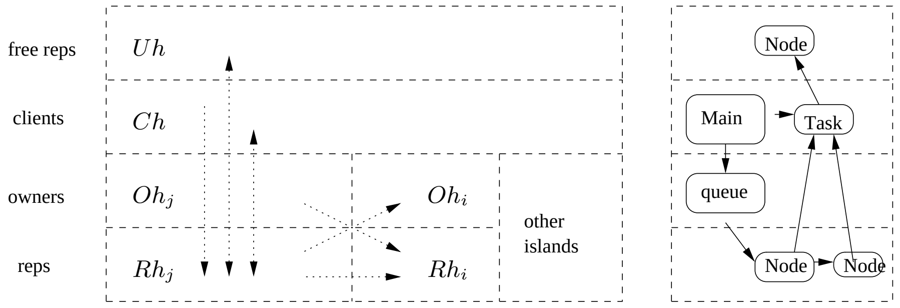

Fig. 2. Confinement scheme for island j with respect to another island i. Dashed boxes delimit partition blocks and dotted lines indicate prohibited references. The right side depicts object instances in a confined heap with a single island.

uses some representation objects. We assume that there is a class name *Rep* (resp. Rep') such that every representation object for one version of *Own* has type $\leq Rep$ (resp. $\leq Rep'$ for the other version). Typically, *Rep* and Rep' are library classes; we allow that one is a subclass of the other. Each version of *Own* encapsulates its reps, so it suffices to define encapsulation for *Rep*.

We write $C \lneq\gneq D$ for $C \not\leq D$ and $D \not\leq C$. Because $C \lneq\gneq D$ implies that classes C and D have no instances in common, we can define the partition in terms of types (under the assumption $Own \lneq\gneq Rep$), as depicted in Fig. 2. The set of objects in the heap is partitioned[2] into the following: (a) The client block, Ch, containing any object with type C such that $C \not\leq Own$ and $C \not\leq Rep$. (b) Some *islands* of the form $Oh_i * Rh_i$ where Oh_i consists of a single owner object (with type C, $C \leq Own$) and Rh_i consists of the encapsulated representations for that owner (each of which has some type $D \leq Rep$).[3] (c) A block, Uh, for objects of class $\leq Rep$ that are not encapsulated by an owner. The block Uh allows us to remedy a limitation of our previous work, where a library class could not be used both as the designated rep class and for other uses. In the sequel we also consider transfers of reps between clients and owners.

The second step in formalizing encapsulation is to restrict the direct points-to relation in terms of the partition. A heap is *confined* if there is a partition (as above) such that

- Clients do not point to encapsulated reps.
  In our notation for heaps, this is written $Ch \not\hookrightarrow Rh_j$, where $h \not\hookrightarrow h'$ says no location in $dom\, h'$ is in a field of an object of h.
- Islands are separated from free reps ($Oh_j * Rh_j \not\hookrightarrow Uh$ and $Uh \not\hookrightarrow Oh_j * Rh_j$).
- Islands are separated from each other: owners do not point to reps in a different island ($Oh_j \not\hookrightarrow Rh_k$) and encapsulated reps do not point to reps or owners in a different island ($Rh_j \not\hookrightarrow Oh_k * Rh_k$).

[2] We allow the "partition" blocks to be empty.
[3] We use the symbol $*$ for union of disjoint heaps, as in Separation Logic [26].

```
unit transferTo(TaskQueue o){ Node n := self.pull(); o.push(n) }
unit transferFrom(TaskQueue o){ Node n := o.pull(); self.push(n) }
unit push(isol Node n){ n.setNxt(self.fst); self.fst := n }
isol Node pull(){ result := self.fst; self.fst := result.getNext(); result.setNext(null) }
```

Fig. 3. Methods to be added to TaskQueue, possibly with module scope for push, pull.

– The pointers from Oh_i to Rh_i are in private fields of Own. The formal notation is $Oh_i \not\rightsquigarrow^{\bar{g}} Rh_i$ which weakens $\not\rightsquigarrow$ by excluding the private fields $\bar{g}$ of Own.

Methods defined in a subclass of Own may manipulate reps but not store them in its fields; this allows reasoning about versions of the owner class independently from its subclasses [2]. A subclass of TaskQueue could, for example, augment Node with fields to store accounting information; class TaskQueue would use a factory method [13] to construct nodes.

Our notion of confinement does not involve hierarchical ownership [8, 9]. It could well be that instances of Own are used as the representation of some object we have classified as a client, but this should have no relevance in modular reasoning about Own. Similarly, instances of Rep may themselves have associated representation objects, perhaps of an incomparable class Rep_1. But for reasoning about an instance of Own, say in island j, it is enough to lump all such objects together in block Rh_j. It could be convenient to refer to these objects using transitive ownership relations, but the hierarchical structure has no other significance so it is also possible to designate them using the straightforward generalization of our definitions to multiple classes $Rep, Rep_1, \ldots$.

3 Ownership transfer

For load balancing it might be useful to move a task from one TaskQueue, say q0, to another, say q1. Dequeuing from q0 and enqueuing to q1 would mean discarding a node and constructing a new one. Better performance might be achieved by transferring the existing node. In a more elaborate example, it might be necessary to retain the node in order to preserve accounting information about the task (perhaps visible only to a subclass of TaskQueue).

Consider the invocation q0.transferTo(q1) using a method defined in Fig. 3. The intended effect is to move the first node from the island of q0 to the island of q1. This code attempts to avoid several potential violations of confinement after the transfer. If transferTo referenced n after o.push(n), say to store n in a field of q0, then confinement would be violated because q0 would point to a node in q1's partition. Clearly we must consider confinement for local variables; requisite conditions can be derived by imagining that locals are like additional fields of self. In the case of n, we want it to be treated linearly in the sense that after its value is passed to push it is no longer used by transferTo. (This is signalled by tag **isol** as discussed in the sequel.) Sharing of a Task by different nodes within one queue or between queues is very different; such sharing is allowed without restriction.

```
class Rsrc { // rep for RsrcMgr
  Object it;
  Object getIt(){ result := self.it; }
  unit setIt(Object x){ self.it := x; } }
class RsrcMgr { // owner
  Rsrc freeList;
  isol Rsrc alloc(){
    result := self.freeList;
    Object o := result.getIt(); self.freeList := (Rsrc)o; result.setIt(null); }
  unit free(isol Rsrc r){ r.setIt(self.freeList); self.freeList := r; } }
class Client {
  RsrcMgr rm := . . .
    Rsrc r1, r2, r3; r1 := new Rsrc; r2 := new Rsrc;
    // r1 is isolated, r2 pinned
    rm.free(r1); r2 := rm.alloc(); r3 := r2;
    rm.free(r2); } } // disallowed because r2 not isolated
```

Fig. 4. Bidirectional transfer.

In this example, an encapsulated rep is transferred between two instances, q0 and q1 of the TaskQueue abstraction. Another form of transfer is between client and abstraction. Detlefs, Leino, and Nelson [11] describe an example where it is necessary for the client of an abstraction to initialize a rep, in order to maintain another abstraction. This is a transfer from client to instance of an encapsulating abstraction. Yet another example is in resource management, where there may be bidirectional transfer of encapsulated data between client and abstraction. Fig. 4 gives a toy example; the Rsrc can be used by clients to hold some object in a field, it, that is also used by the manager for chaining free objects.

The confinement scheme is adequate to ensure representation independence even in the presence of transfers. The challenge is to ensure that confinement is indeed maintained despite transfers. What is needed is that at the moment of transfer, the object explicitly transferred is the root of a block sufficiently separated that it can be merged with a different partition without violating confinement. (This is the purpose of the assignment result.setNxt(**null**) in the pull method in Fig. 3.) When this moment occurs is an interesting question. We simplify the answer by allowing transfers via method calls, so the transfer can be said to take place in the passing of arguments or results. An alternative is to allow transfers by direct field assignments. For private fields this would be allowed among owners though not between owner and client; but for a compositional formulation it would seem to require an atomic transfer construct which we prefer to avoid.

A key challenge is how to ensure that the block to be transferred is sufficiently separated. To meet this challenge, we can use simple but restrictive syntactic conditions (as in alias and ownership types), or we confront a need for general reachability predicates. On one hand, the confinement property delimits the program variables from which reachability needs to be considered, but on the other hand our confinement property requires, e.g., absence of pointers be-

```
unit push2(isol Node n) { n.getNxt().setNxt(fst); fst := n }
unit transferTwoTo(TaskQueue o1, TaskQueue o2) {
  Node n1, n2;
  n1 := fst; n2 := fst.getNxt().getNxt(); fst := n2.getNxt().getNxt());
  // n1 is first, n2 is third, fst is fifth
  n1.getNxt().setNxt(null); n2.getNxt().setNxt(null);
  // n1 and n2 isolated
  o1.push2(n1);
  // n1 is dead variable, n2 still isolated
  o2.push2(n2) }
```

Fig. 5. Methods to be added to TaskQueue for transfer of a two-element segment to each of two other TaskQueues. (Precondition: fst points to an acyclic chain of at least four nodes.)

tween reps in different islands, even reps that are hierarchically "components" of the reps directly manipulated by *Own*. This problem is one that we do not solve but rather specify. The specification in itself is a significant challenge in a denotational semantics where there is no explicit stack. Yet without a compositional semantics it is an open problem how to express and prove representation independence [14].

In our previous work [2], the key property which facilitates compositional reasoning is that the result heap h_0 from any command (and thus every method meaning) extends the initial heap h in this sense:

– For every confining partition of h there exists a confining partition of h_0 with possibly additional islands but each block of h extends, *without losing any objects*, to a corresponding block in the partition of h_0 (with possibly additional objects).

A program state consists of a heap h and a store η that gives values for the current command's parameters/variables. Confinement of a state, written $conf\ \Gamma\ (h, \eta)$, is defined so that the values in the range of η could not violate heap confinement if stored in fields of the object $\eta\mathsf{self}$. For example, if $\Gamma\,\mathsf{self} \leq Own$ then no reps in $rng\,\eta$ may be in other islands. The above notion of extension facilitates a compositional proof of preservation of confinement: Because confinement of a store η depends only on the domain of the heap, not states of objects, it follows easily that if $conf\ \Gamma\ (h, \eta)$ and h_0 extends h then $conf\ \Gamma\ (h_0, \eta)$.

Transfer invalidates this extension property. A transfer may move a rep between the free reps and an island or between islands. Figures 3 and 4 show simple cases where the transfer occurs as direct result of a method call. What makes the problem difficult is that the transfer may occur indirectly due to a chain of calls. Fig. 5 illustrates transfer of an isle consisting of two objects, forming an isolated pool. It also shows that a caller's store can usefully contain reps during a call to a method that may perform transfers. After the call o1.push2(n1), the store including n1 is not confined because n1 has been transferred from the island for self to the island for o1. This could be solved by using a destructive read of n1 as in some proposals for uniqueness, but Boyland has pointed out that it is enough to ensure that the alias is a dead variable at this point [7]. We formalize this by removing n1 from the state space after the call, so it is a syntax error to access n1 following a call like o1.push2(n1) that passes it by **isol**.

Fig. 4 shows a buggy version of transfer. Because of the assignment r3 := r2 in class Client, the method call rm.free(r2) results in an unconfined state because r3 retains a pointer to the transferred object. One way a programmer might choose to prevent such problems is to record in the state of the object r2 that it is not allowed to be removed, e.g., by setting a boolean lock field pin. We adopt this idea as a means of specifying a condition in the semantics of method call to prevent the bad example. In the correct example of transfer, (Fig. 5), immediately prior to the call o1.push2(n1), the semantics sets n2.pin to express that after the call the pinned object is still in the caller's island. In the buggy version of Fig. 4, r3 is pinned, as it is a rep still referenced in the caller's store. Now we can specify just in the semantics of the called method that the pinned object r3 moves to a new block, thus violating confinement.

The pinning mechanism is formulated as an auxiliary field in each rep, manipulated in an instrumented version of the semantics. Its only purpose is to be able to define the confinement condition in a modular way.

Our assumption about isolation for method call is expressed directly in terms of aliasing at the point of call: there can be no alias from the store of objects passed through designated isol parameters. Pinning lets us say that except for arguments passed as isol parameters, no objects are transferred —and to say this purely in terms of the semantics of the called method, without mention of the caller's store. (Much less the stack of callers' stores.)

4 Language

This section sketches the formalization of a language with class-based visibility, mutable local variables and fields, recursive types and methods, type tests and casts. We emphasize the typing system that accounts for how **isol** arguments are removed from the caller's state space. More leisurely and complete treatment of a formalization that is similar in all other regards can be found in [2]; it draws on [16]. The language is similar to the imperative core of Java/C#, and in particular the data types T range over primitives **bool** and **unit** and class names C.

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. Consider a declaration **class** C **extends** D $\{\ \bar{T}_1\ \bar{f};\ \bar{M}\ \}$. Let M be in the list $\bar{M}$ of method declarations, with $M = T\ m(\bar{T}_2\ \bar{x})\ \{S\}$. We record the typing information by defining $\mathit{mtype}(m, C) = \bar{T}_2 \rightarrow T$, the parameter names by $\mathit{pars}(m, C) = \bar{x}$. For the declared fields, we define $\mathit{dfields}\, C = (\bar{f} : \bar{T}_1)$. To include inherited fields, we define $\mathit{fields}\, C = \mathit{dfields}\, C \cup \mathit{fields}\, D$. The rules for field access and update enforce private visibility by using the *dfields* function. Note that the pin field is not included in *dfields* and therefore cannot be accessed in source code; it is only manipulated in the semantics.

A *typing context* Γ is a finite mapping from variable and parameter names to data types, such that $\mathsf{self} \in \mathit{dom}\, \Gamma$. Rather than formalizing the keyword **isol**

$$\frac{\begin{array}{c}\Gamma \vdash e : D \qquad \mathit{mtype}(m, D) = \bar{T} \to T \qquad \Gamma \vdash \bar{e} : \bar{U} \\ \bar{U} \leq \bar{T} \qquad x \neq \mathsf{self} \qquad T \leq \Gamma\, x \qquad F = \{\bar{e}_i \mid \mathit{isol\text{-}tag}(\bar{T}_i)\} \qquad F \subseteq \mathit{dom}(\Gamma \downharpoonleft \mathsf{self})\end{array}}{\Gamma \vdash x := e.m(\bar{e}) \searrow F}$$

$$\frac{\Gamma \vdash S_1 \searrow F \qquad (\Gamma \downharpoonleft F) \vdash S_2 \searrow F_2}{\Gamma \vdash S_1;\ S_2 \searrow F_2}$$

$$\frac{\begin{array}{c}\mathit{mtype}(m, C) \text{ is undefined or equals } \bar{T} \to T \qquad \bar{x} : \bar{T}, \mathsf{self} : C, \mathsf{result} : T \vdash S \searrow F \\ \mathit{pars}(m, C) \text{ is undefined or equals } \bar{x} \qquad \mathsf{self}, \mathsf{result} \notin F\end{array}}{C \vdash T\ m(\bar{T}\ \bar{x})\{S\}}$$

Table 1. Selected typing rules for commands and method declaration.

in syntax, we take the view that a data type T can be *tagged* or not, and this is tested by $\mathit{isol\text{-}tag}\,T$.

To account for variable `n1` being dead in Fig. 5, we type commands using the judgment form $\Gamma \vdash S \searrow F$ which means that the initial state space of the call is given by Γ but the final state space is $\Gamma \downharpoonleft F$ where F is a subset of $\mathit{dom}\,\Gamma$. Selected rules are in Table 1. For any parameter with the **isol** tag, the argument must be a local variable of the caller so that this variable can be explicitly included in F. The judgements for other command constructs manipulates F in a straightforward way.

A class table is *well formed* provided that each of its method declarations is well formed according to the rule above. Moreover, it must be *well tagged*: any occurrence of a type T such that $\mathit{isol\text{-}tag}\,T$ must be either the return type or parameter type for a method and also $T \leq \mathit{Rep}$. Note that the target of a call is not tagged. It is client and owner objects that can transfer reps among themselves. If the target of a call has type Rep, that call is not making a transfer so it does not make sense to tag it. What does need to be handled is that this call can lead to other calls that result in transfer of this rep.

The semantics is compositional: commands and methods denote state transfermers, defined as a limit of approximations. The state of a method in execution is comprised of a *heap* h, which is a finite partial function from locations to object states, and a *store* η, which assigns locations and primitive values to local variables and parameters. A command denotes a function from initial state to either a final state or the error value $\bot$. More precisely, the meaning is a function $[\![\Gamma \vdash S \searrow F]\!]$ in $[\![\mathit{MEnv}]\!] \to [\![\mathit{Heap} \otimes \Gamma]\!] \to [\![(\mathit{Heap} \otimes (\Gamma \downharpoonleft F))_\bot]\!]$. The semantic domain $[\![\mathit{MEnv}]\!]$ of method environments, maps each class C and method name m to an appropriate method meaning. The semantic domains and command semantics are straightforward — see Tables 5 and 6 in the appendix.

We only give the semantics for method call, as it is instrumented to manipulate the `pin` field. For any $P \subseteq \mathit{locs}(\mathit{Rep}{\downarrow})$ we define $\mathit{pin}\,P\,h$ to be h but with $h\,\ell\,\mathsf{pin} = \mathit{true}$ for every $\ell \in P$. Similarly, $\mathit{unpin}\,P\,h$ is h but with $h\,\ell\,\mathsf{pin} = \mathit{false}$ for every $\ell \in P$. The sematics, given in Table 2, pins the reps in the callers store, i.e., those which must not be transferred by the call; upon return, they

```
$[\![\Gamma \vdash x := e.m(\bar{e}) \searrow F]\!]\mu(h, \eta) =$
    let $\ell = [\![\Gamma \vdash e : D]\!](h, \eta)$ in
    if $\ell = nil$ then $\bot$ else
    let $\bar{x} = pars(m, D)$ in
    let $\bar{d} = [\![\Gamma \vdash \bar{e} : \bar{U}]\!](h, \eta)$ in
    let $\eta_1 = [\bar{x} \mapsto \bar{d}, \mathsf{self} \mapsto \ell]$ in
    let $P = \{\ell' \mid (\ell' = \ell \vee \ell' \in rng\,(\eta \lfloor F)) \wedge loctype\,\ell' \leq Rep \wedge \neg(h\,\ell'\,\mathsf{pin})\}$ in
    let $h_1 = pin\,P\,h$ in
    let $(h_2, d_1) = \mu(loctype\,\ell)m(h_1, \eta_1)$ in
    let $h_0 = unpin\,P\,h_2$ in $(h_0,\ ([\eta \mid x \mapsto d_1]) \lfloor F)$
```

Table 2. Method call semantics with auxiliary fields.

are unpinned unless they had already been pinned. None of the other semantic clauses manipulate the pin fields.

Define $pnd\,h$, the set of all locations pinned in h, as $pnd\,h = \{\ell \mid \ell \in dom\,h \wedge loctype\,\ell \leq Rep \wedge h\,\ell\,\mathsf{pin} = true\}$. We say h, h_0 are *equal up to pinning* if there's P with $h_0 = pin\,P\,h$ or $h_0 = unpin\,P\,h$.

Finally, as in [2], the semantics $[\![CT]\!]$ of a class table is defined as the least upper bound of an ascending chain of method environments. This is because a class table may have mutually recursive method declarations.

5 Semantic confinement, isolation, and pinning

We let ℓ range over $Locs \cup \{nil\}$ in the following. Reachability is defined inductively by $\ell \xrightarrow{h} \ell'$ iff $\ell \in dom\,h \wedge (\ell = \ell' \vee \exists \ell'' \in rng(h\,\ell) \bullet \ell'' \xrightarrow{h} \ell' \wedge \ell' \neq nil)$. We shall define predicate $isol\,h\,\ell$ to mean that the object referenced by ℓ does not have any sharing that prevents it from being transferred, and $isol\,\Gamma\,(h, \eta)$ to mean that any tagged variables in Γ-state (h, η) have this property. First we define the block of reps reached from a candidate for transfer. For any h and $\ell \in [\![Rep]\!]$, let $isle\,\ell\,h = \{\ell' \mid \ell \xrightarrow{h} \ell' \wedge loctype\,\ell' \leq Rep\}$.

Definition 1 (*isol*). *For heap h and $\ell \in [\![Rep]\!]$, define $isol\,h\,\ell$ iff the following hold.*

- *for all ℓ', if $\ell \xrightarrow{h} \ell'$ then $loctype\,\ell' \not\leq Own$ (that is, ℓ reaches no owner)*
- *for all $\ell' \in dom\,h$, if $rng(h\,\ell') \cap isle\,\ell\,h \neq \varnothing$ then $\ell' \in isle\,\ell\,h$ (that is, ℓ is isolated from the rest of the heap)*

For heap h and store $\eta \in [\![\Gamma]\!]$, define $isol\,\Gamma\,(h, \eta)$ iff the following hold for all $x \in dom\,\Gamma$ such that isol-tag$(\Gamma\,x)$.

- *$isol\,h\,(\eta\,x)$ (that is, $\eta\,x$ is isolated from owners and heap)*
- *$rng(\eta \lfloor x) \cap isle\,(\eta\,x)\,h = \varnothing$ (that is, $\eta\,x$ is isolated from the store)*

Note that in Figure 4, the call rm.free(r2) in class Client is disallowed because r2 and r3 are aliases. Thus the heap location referenced by r2 is not isolated from the store at the point of call.

Confinement for global state (h, η) for Γ, written $conf\ \Gamma\ (h, \eta)$, is defined by imposing conditions on η so that if η was taken to be an object then it would be allowed in the partition block containing $\eta\,\mathsf{self}$. Thus these conditions can be expressed in terms of the type $\Gamma\,\mathsf{self}$, except that in the case $\Gamma\,\mathsf{self} \leq Rep$ one must consider the two subcases where $\eta\,\mathsf{self}$ is in Uh and in some Rh_j. There is a similar definition for expressions. (See Def. 11 and 12 in the appendix.) The conditions are also similar to the conditions for method results, in Def. 3 below.

To express that confinement is preserved, we need to connect the initial and final heaps for methods, as a method may transfer some reps (possibly by calling many other methods). The following definition expresses that some set P of reps does not get transferred.

Definition 2 (confining P-extension, $\preceq^P$). *For any $P \subseteq locs(Rep{\downarrow})$ and any h with confining partition $h = Ch * Uh * Oh_1 * Rh_1 * \ldots * Oh_k * Rh_k$ $(k \geq 0)$, we define a* confining P-extension *of heap h_0 to be an admissible partition $h_0 = Ch^0 * Uh^0 \ldots$ that satisfies the following:*

- $n \geq k$
- $dom(Ch) \subseteq dom(Ch^0)$
- $dom(Uh) \cap P \subseteq dom(Uh^0)$
- $dom(Oh_j) \subseteq dom(Oh_j^0)$ *for all* $j \leq k$
- $dom(Rh_j) \cap P \subseteq dom(Rh_j^0)$ *for all* $j \leq k$

Define $h \preceq^P h_0$ iff h is confined and for any confining partition of h, there is a confining P-extension of h_0.

The following kind of confinement property of a transferrable location ℓ, given in terms of $isle\,h\,\ell$, can be proven. Essentially, if h has a confining partition and Vh is the subheap of h determined by $isol\,h\,\ell$, then for any rep block (Uh or one of the Rh_j) we get a confining partition by moving Vh to that block. Here we just consider the transfer to Uh.

Lemma 1. *Suppose $conf\ h$ and $isol\,h\,\ell$ for some $\ell \in dom\,h$. Consider a confining partition $h = Ch * Uh * \ldots$. If $\ell \in dom(Rh_j)$ then $Ch * Uh' * \ldots * Oh_j * Rh'_j * \ldots$ is a confining partition for h, where $Uh' = Uh * Vh$ and $Rh_j = Rh'_j * Vh$. Moreover, if $pnd(dom\,Vh) = \varnothing$, then the above partition is a $pnd\,h$-extension.*

A confined command is one that preserves confinement of global states. Because command meanings depend on method environments, confinement for method environment is formalized first. We need to ensure that a method call yields a heap that is an extension of the caller's heap and preserves all pinned reps in the caller's heap. Moreover, the result is confined for the target object.

Definition 3 ($iconf\,\mu$, isol-confined method, environment). *Let μ be a method environment, $mtype(m, C) = \bar{T} \to T$, $pars(m, C) = \bar{x}$, and $\Gamma = [\bar{x} : \bar{T}, \mathsf{self} : C]$. Then $\mu\,C\,m$ is* isol-confined *provided the following holds for all $(h, \eta) \in [\![Heap \otimes \Gamma]\!]$ with $conf\ \Gamma\ (h, \eta)$ and $isol\ \Gamma\ (h, \eta)$. If $(h_0, d) = \mu C m(h, \eta)$ then $h \preceq^{pnd\,h} h_0$ and moreover there exists a confining $(pnd\,h)$-extension, $h_0 = Ch^0 * Uh^0 * \ldots$, with*

1. *$pnd\, h_0 = pnd\, h$*
2. *$isol\text{-}tag\, T \Rightarrow isol\, h_0\, d$*
3. *$C \not\leq Rep \wedge C \not\leq Own$ and $d \in locs(Rep{\downarrow})$ implies $d \in dom(Uh^0)$*
4. *$C \leq Own$ and $d \in locs(Rep{\downarrow})$ and $\eta\, \mathsf{self} \in dom(Oh^0_j)$ implies $d \in dom(Rh^0_j)$.*
5. *$C \leq Rep$ and $\eta\, \mathsf{self} \in dom(Uh^0)$ and $d \in Locs$ implies $d \in dom(Ch^0 * Uh^0)$*
6. *$C \leq Rep$ and $\eta\, \mathsf{self} \in dom(Rh^0_j)$ and $d \in locs(Own{\downarrow}, Rep{\downarrow})$ implies $d \in dom(Oh^0_j * Rh^0_j)$.*

Method environment μ is isol-confined, written iconf μ, iff $\mu\, C\, m$ is isol-confined for all C, m.

Definition 4 (confined command). *Command $\Gamma \vdash S \searrow F$ is confined iff*

- *$iconf\ \mu \wedge conf\ \Gamma\, (h, \eta) \wedge (h_0, \eta_0) = [\![\Gamma \vdash S \searrow F]\!]\mu(h, \eta) \Rightarrow conf\ (\Gamma|F)\, (h_0, \eta_0)$, for any μ, and (h, η).*
- *if S is a method call then it has confined and isolated arguments (see below).*

To be able to reason compositionally about methods which can make "outgoing" calls to clients, we need to constrain the arguments they pass. *Confinement of arguments* means that the store η_1 passed in the semantics of method call $\Gamma \vdash x := e.m(\bar{e})$ is confined and isolated for the class of the callee e, whenever the call is made in an initial state (h, η) that is confined and isolated for the caller.

Definition 5 (confined arguments). *Consider a call $\Gamma \vdash x := e.m(\bar{e}) \searrow F$. The call has* confined arguments *provided the following holds. For any (h, η) with conf $\Gamma\, (h, \eta)$, let $U, D, \ell, \Gamma_1, \bar{d}, \eta_1$ be as in the semantics of method call. If $\ell \neq \bot$, $\ell \neq nil$, and $\bar{d} \neq \bot$, then conf $\Gamma_1\, (h, \eta_1)$.*

For simplicity in the following, we assume that the caller's environment Γ has disjoint domain from the parameter environment Γ_1 so we may form their union. We write *droptags* Γ for the environment where any isol-tags have been erased and self has been replaced by a fresh name.

Definition 6 (isolated arguments). *Consider a call $\Gamma \vdash x := e.m(\bar{e}) \searrow F$. The call has* isolated arguments *provided the following holds. For any (h, η) with conf $\Gamma\, (h, \eta)$, let $U, D, \ell, \Gamma_1, \bar{d}, \eta_1$ be as in the semantics of method call. If $\ell \neq \bot$, $\ell \neq nil$, and $\bar{d} \neq \bot$, then isol $((droptags\, \Gamma), \Gamma_1)\, (h, (\eta, \eta_1))$.*

Definition 7 (confined class table). *Well formed CT is* confined[4] *iff for every C and every m with $mtype(m, C) = \bar{T} \rightarrow T$ the following hold.*

1. *If m is declared in C by $T\ m(\bar{T}\ \bar{x})\{S\}$ then S and all its constituents are confined.*
2. *If m is inherited in Own then $\bar{T} \not\leq Rep$.*

[4] A purely semantic formulation would call class table CT confined just if $[\![CT]\!]$ is a confined method environment. But confinement of $[\![CT]\!]$ follows from confinement of method bodies and constructors and this is independent from the particulars of our static analysis. The proof of Theorem 1 is critical in that it holds together (and helped us debug) all the preceding definitions.

3. No m is inherited in Rep from any $B > Rep$.

The conditions on inheritance are as in [2].

We say μ has *isolated results* if, for all C and m such that $mtype(m, C) = \bar{T} \to T$ and $isol\text{-}tag\,T$, if $\mu C m(h, \eta) = (h_0, d)$ then $isol\ h_0\, d$.

Theorem 1. *If CT is confined and $[\![CT]\!]$ has isolated results then $iconf\ [\![CT]\!]$.*

The theorem uses the following crucial property of confinement of commands.

Lemma 2 (extension by commands). *Suppose $\Gamma \vdash S \searrow F$ is confined and all its constituents are confined. For any μ, h, η with $iconf\ \mu$ and $conf\ \Gamma\ (h, \eta)$, if $(h_0, \eta_0) = [\![\Gamma \vdash S \searrow F]\!]\mu(h, \eta)$ then $h \preceq^{pnd\,h} h_0$.*

6 Static analysis

This section gives a syntax directed static analysis. It checks a property called safety. Safety is shown to imply confinement. The input to the analysis is a well formed class table and designated class names Own and Rep.

Definition 8. ***(safe)*** *Well formed class table CT is* safe *iff for every C and every m with $mtype(m, C) = \bar{T} \to T$ the following hold.*

1. *If m is declared in C by $T\ m(\bar{T}\ \bar{x})\{S\}$ then $\bar{x} : \bar{T}, \mathsf{self} : C, \mathsf{result} : T \Vdash S \searrow F$ where $\Vdash$ is the safety relation defined in the sequel.*
2. *If m is inherited in Own from some $B > Own$ then $\bar{T} \not\leq Rep$.*
3. *No m is inherited in Rep from any $B > Rep$.*

The safety relation $\Vdash$ is defined by the rules appearing in Tables 3.[5] (The complete specification appears in the appendix, Tables 7,8). For expressions, the analysis imposes restrictions on field access and nothing else. If $e.f$ occurs 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. For commands, we use a judgement $\Gamma \Vdash S \searrow F$ interpreted as follows: S is safe in the context Γ provided that no variables in "freed set" F are accessed subsequently.

For lack of space, we only explain conditions (b) and (d) in the method call case. Condition (b) considers method calls from an owner class or its subclasses: if the target's type is comparable to Own, then reps can be passed as parameters if e is self. In all other cases, for a rep to be passed as parameter or returned as result, it must be isolated in the heap. Then the caller can transfer the isolated rep to the callee's partition. Likewise, isolation for the result rep allows the callee to transfer it to the caller's partition. Method calls, o1.push2(n1) and o2.push2(n2) in Fig. 5 satisfy (b). Condition (d) says that if m is an owner method called by a client, then a rep may be passed as parameter or returned as result, only if this rep is isolated in the heap. For parameter passing, the rep must be isolated in the unconfined partition of the heap and it can be transferred to the confined reps in the owner's partition. The reverse transfer happens for reps returned as result. Method calls rm.free(r1) and r2 := rm.alloc() in Fig. 4 satisfy (d).

[5] The rules must be read in concert with the typing rules for expressions and commands. Because CT is well formed, all expressions and commands appearing in method bodies are well-typed. We elide typing information except when necessary.

$$\frac{\Gamma \Vdash e : C \qquad (f : T) \in \mathit{dfields}\, C \qquad \begin{array}{c} C = (\Gamma\, \mathsf{self}) \\ C = \mathit{Own} \wedge e \neq \mathsf{self} \Rightarrow T \not\lessgtr \mathit{Rep} \end{array}}{\Gamma \Vdash e.f : T}$$

$$\frac{\begin{array}{c} \Gamma \Vdash e : D \qquad \mathit{mtype}(m, D) = \bar{T} \rightarrow T \qquad \Gamma \Vdash \bar{e} : \bar{U} \qquad C = (\Gamma\, \mathsf{self}) \\ (a)\ (C \leq \mathit{Own} \vee C \leq \mathit{Rep}) \wedge (D \not\leq \mathit{Rep} \wedge D \not\leq \mathit{Own}) \Rightarrow (\forall U \in (T, \bar{T}) \bullet U \not\lessgtr \mathit{Rep} \vee \mathit{isol\text{-}tag}\, U) \\ (b)\ C \leq \mathit{Own} \Rightarrow D \not\lessgtr \mathit{Own} \vee (e = \mathsf{self}) \vee (\forall U \in (T, \bar{T}) \bullet U \not\lessgtr \mathit{Rep} \vee \mathit{isol\text{-}tag}\, U) \\ (c)\ C \leq \mathit{Own} \Rightarrow D \not\lessgtr \mathit{Rep} \vee (\forall e_i \in \bar{e} \bullet e_i = \mathsf{self} \vee T_i \not\lessgtr \mathit{Own}) \\ (d)\ C \not\leq \mathit{Own} \wedge C \not\leq \mathit{Rep} \Rightarrow D \not\lessgtr \mathit{Own} \vee (\forall U \in (T, \bar{T}) \bullet U \not\lessgtr \mathit{Rep} \vee \mathit{isol\text{-}tag}\, U) \\ (e)\ C \leq \mathit{Rep} \wedge (D \leq \mathit{Own} \vee D \not\leq \mathit{Rep}) \Rightarrow T \not\lessgtr \mathit{Own} \\ (f)\ C \not\leq \mathit{Own} \wedge C \not\leq \mathit{Rep} \Rightarrow D \not\lessgtr \mathit{Rep} \vee \bar{T} \not\lessgtr \mathit{Own} \end{array}}{\Gamma \Vdash x := e.m(\bar{e}) \searrow F}$$

Table 3. Safety for selected expressions and commands.

Lemma 3 (argument values confined). *Suppose $\Gamma \vdash e : D$ and $\Gamma \vdash \bar{e} : \bar{U}$ are confined at Γ self. If $\Gamma \vdash x := e.m(\bar{e}) \searrow F$ has isolated arguments and $\Gamma \Vdash x := e.m(\bar{e}) \searrow F$ then m has confined arguments.*

Lemma 4 (safe commands). *If $\Gamma \Vdash S \searrow F$ and all method calls in S have isolated arguments then $\Gamma \vdash S \searrow F$ is confined.*

We show a proof subcase in the Appendix.

Theorem 2 (safety). *If CT is safe and all method calls in CT have isolated arguments then CT is confined.*

As a corollary of the safety theorem we obtain that if CT is safe, all method calls in CT have isolated arguments, and $[\![CT]\!]$ has isolated results then $[\![CT]\!]$ is isol-confined.

In summary, we note that confinement is an invariant and imposes constraints throughout a program. Isolation is a state predicate that is only needed at call boundaries, and only for calls that perform transfers. These are the exception rather than the rule. We focus on confinement and have given a static analysis that is sound relative to isolation. In order to specify isolation, it is necessary to use annotations on method types, but no other annotations are needed.

As presented, the static analysis is less modular than the one in our previous work [2]. In that work, only rep and owner code (including subclasses) is constrained except for **new**: a client cannot construct a new rep. On the contrary, in this work we allow clients to have access to free reps and pass them to owner methods. We impose constraints on clients calling owners and reps and no longer constrain the case for **new**. While this is done for flexibility, for practical deployment, one could impose constraints directly on signatures of classes *Own* and *Rep*. For *Own* methods, this would mean annotating *Rep* parameters and *Rep* results with tag **isol**, and for *Rep* methods it would mean ensuring that there are no *Own* parameters. Thus the analysis can be made modular and one can avoid directly checking client code.

7 The abstraction theorem

Representation-independence involves comparing two class tables, CT, CT', that differ only in that (a) they declare different fields and method bodies for class Own, and (b) the designated rep classes may be different. The theorem requires that CT be confined with respect to Rep and CT' confined with respect to Rep'. Simulation for global states is induced from the *basic coupling* that would be defined by a programmer to describe the correspondence between a single instance of Own for each of the two implementations. A basic coupling is thus a predicate on a pair $(Oh * Rh,\ Oh' * Rh')$ of islands where Rh (resp. Rh') has objects $\leq Rep$ (resp. $\leq Rep'$). A basic coupling induces a relation on heaps h, h' that requires corresponding pairs of islands to be pointwise related by the basic coupling. Although h is partitioned with respect to Rep and h' to Rep', the free reps can be treated as clients for purpose of coupling. Another complication is the use of a bijection on locations in order to allow the two versions to have different allocation behavior without making assumptions about the allocator. Yet another complication is the possibility that one version of Own can override a method that is inherited in the other version. The technical details are omitted for lack of space; we just sketch the highlights. Aside from the differences due to the presence of free reps, the definitions and proofs are similar to those in [2]. In particular, isolation and heap extension ($\preceq^P$) play no role in the representation independence proof.

The induced coupling relation depends on a partial bijection, σ, on locations.[6] Locations visible to clients, i.e., objects in Ch, Uh, and Oh_js, are related by σ. If (h, h') is in the induced coupling relation, islands are related in accord with the given coupling. For client-visible locations $(\ell, \ell') \in \sigma$, the objects $h\,\ell$ and $h'\,\ell'$ have related fields. Similarly for stores. Note in particular that free reps in h and h' must correspond. For example, if $loctype\,\ell \leq Rep$ and ℓ is a free rep in h then there must be $(\ell, \ell') \in \sigma$ and the states of $h\,\ell$ and $h'\,\ell'$ must correspond.

The role of confinement is to restrict the quantification in the logical-relations condition for command and method meanings. The simulation property for commands is that if $(h, \eta), (h', \eta')$ are confined and coupled then the corresponding final states are coupled. Because command semantics depends on method environments, the precise definition also assumes simulation for the method environments. Simulation for (μ, μ') means that corresponding method meanings preserve coupling —again, only for initial states that are confined.

Theorem 3 (Abstraction). *Suppose that the induced coupling relation is preserved by every method (declared or inherited) in* Own. *Then simulation holds for* $[\![CT]\!], [\![CT']\!]$.

To prove program equivalence using the theorem, one uses an identity extension lemma that says coupled states are equal from the point of view of clients.

[6] In [3] we avoided the complication of a non-identity partial bijection on locations by imposing a parametricity condition on the allocator: the location chosen for an object of given type C depends only on the currently allocated objects of type C. In the present context this is inadequate because a client can allocate an object of type Rep and states h, h' may differ in the number of allocated Rep's.

Here equality is modulo the bijection on locations, and the client perspective can be formalized by garbage-collecting owners or by hiding private fields of owners.

8 Discussion

One could say that transfer liberates an object from its owner, but the title of this paper alludes to two other ideas. One is that confinement is supposed to liberate reasoners from global reasoning. In the present case, we show how to reason about a collection class in terms of a single instance while reaching a conclusion that applies to unboundedly many instances and unrestricted sharing among clients. The other idea is that confinement deserves to be liberated from the predominant type-theoretic view, to allow thorough exploration of other parts of the design space. To impose confinement through the use of a type system requires the global imposition of that type system and it requires the types to be sufficiently flexible and general to handle all the design and reasoning patterns of interest. Even for the single reasoning problem of substitutability, there are a number of important patterns that are beyond the reach of current type systems. In this paper we consider replacement of a single class but for practical purposes it is important to consider several classes that share representation, e.g., a collection and an unbounded number of iterators, or direct access to reps from subclasses. Such patterns are all amenable to the approach elaborated in this paper — but a single "most general" pattern would be clumsy at best. In unfinished work, we have also shown that our approach can be extended to a language with C#-style parametric classes and methods (which avoids the problem that `Object` is comparable to every class). We use ordinary program types with little or no additional annotation to express an invariant on heap structure directly pertinent to the pattern of interest. Our slogan is: "What do you want to confine today?"

The closest related work on ownership transfer is that by Clarke and Wrigstad [10] on external uniqueness. They use ownership types [8] and consequently, a hierarchical ownership model, to specify uniqueness of pointers. External uniqueness allows an object to have several aliases within the abstraction but a unique alias from outside. This facilitates transfer of ownership of entire aggregates between abstractions. They give a static analysis for external uniqueness but do not prove the analysis correct. Assuming correctness, we could adapt their static analysis to give a static analysis for isolation at the cost of significant annotation of the program. Even so, it is unclear how the analysis will handle isolation of `n1` and `n2` in the method `transferTwoTo` in Fig. 5. The issue is that `n2` is a pending rep in the caller's store when the transfer of `n1` occurs and `n2` cannot be moved during the transfer.

Boyapati *et al.* [6] also use ownership types for object encapsulation and local reasoning about program correctness. Their ingenious observation is that in Java there is a fixed association between an instance of an inner class and its outer class instance. In terms of ownership types, every inner class instance has a specific owner, namely, its outer class instance. This approach allows them to express the iterator pattern: the inner class can be used to implement an iterator for the outer class. Local reasoning is supported because the outer class and inner class instances can be reasoned about together as a module. Unfortunately, because of the fixed ownership association, this model cannot handle ownership transfer. Clarke and Drossopoulou [9] use annotations (e.g., read-only) and ownership types to get reasoning principles about aliasing and disjointness of effects. These papers thus show the use of ownership types for data abstraction – but they do not consider ownership transfer.

Barnett *et al.* [4] recently proposed a means of modular checking of object invariants and "modifies" specifications without alias confinement: an auxiliary variable tracks whether an object is owned and rules ensure that there is at most one owner while allowing transfer of ownership. We are currently investigating confinement and data abstraction in this setting.

O'Hearn et. al. [22] use Separation Logic to reason about the heap. (In fact, our work responds to O'Hearn's challenge [Dagstuhl, March 2003] to handle malloc and free.) The language considered is a low-level imperative language with parameterless procedures. The key result is the soundness of the hypothetical frame rule which implies soundness of local reasoning for procedures. Their theory handles single instances of abstractions; to cope with object-oriented languages it has to be extended to handle dynamic creation of instances. It would be a challenging exercise to devise a sound reasoning framework for encapsulation and ownership transfer for an object-oriented language as considered in our paper.

References

1. Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. ACM Press, 2002.
2. Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. Revised and extended from [3]; submitted., 2002.
3. Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 166–177. ACM Press, 2002.
4. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. In *ECOOP Workshop on Formal Techniques for Java-like Programs*, 2003.
5. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2002.
6. Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2003.
7. John Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6), 2001.
8. David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Australia, 2001.
9. David Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. ACM Press, 2002.

10. David Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *European Conference on Object Oriented Programming (ECOOP)*, 2003.
11. D. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Technical Report 156, COMPAQ Systems Research Center, July 1998.
12. Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2003.
13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
14. Daniel Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems*, 22(6), 2000.
15. J. He, C. A. R. Hoare, and J.W. Sanders. Data refinement refined (resumé). In *European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.
16. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–459, May 2001.
17. Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, second edition, 2000.
18. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5), 2002.
19. John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
20. G. Morrisett, F. Smith, and D. Walker. Alias types. In *ESOP*, 2000.
21. Peter Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
22. Peter O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2004.
23. Gordon Plotkin. Lambda definability and logical relations. Technical Report SAI-RM-4, University of Edinburgh, School of Artificial Intelligence, 1973.
24. John C. Reynolds. Towards a theory of type structure. In *Colloques sur la Programmation, LNCS 19*, pages 408–425, 1974.
25. John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, 1984.
26. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 2002.

Appendix for convenience of referees

Syntax and semantics

Table 4 completes the definition of syntax.

The semantic domains are defined in Table 5. For locations, we assume that a countable set *Loc* is given, along with a distinguished entity *nil* not in *Loc* and a function $\mathit{loctype} : \mathit{Loc} \to \mathit{ClassNames}$. The semantic definitions and results are given for an arbitrary *allocator*, i.e., a location-valued function *fresh* such that $\mathit{loctype}(\mathit{fresh}(C, h)) = C$ and $\mathit{fresh}(C, h) \notin \mathit{dom}\, h$, for all C, h.

Table 6 gives the semantics for commands other than method call.

$$\Gamma \vdash x : \Gamma x \qquad\qquad \Gamma \vdash \mathbf{true} : \mathbf{bool} \qquad\qquad \Gamma \vdash \mathbf{null} : B$$

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

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

$$\frac{\Gamma \vdash e : T \qquad T \leq \Gamma\, x \qquad x \neq \mathsf{self}}{\Gamma \vdash x := e} \qquad\qquad \frac{B \leq \Gamma x \qquad x \neq \mathsf{self} \qquad B \neq \mathsf{Object}}{\Gamma \vdash x := \mathbf{new}\ B}$$

$$\frac{\Gamma \vdash e_1 : (\Gamma\, \mathsf{self}) \qquad (f : T) \in \mathit{dfields}(\Gamma\, \mathsf{self}) \qquad \Gamma \vdash e_2 : U \qquad U \leq T}{\Gamma \vdash e_1.f := e_2}$$

$$\frac{\Gamma \vdash e : \mathbf{bool} \qquad \Gamma \vdash S_1 \searrow F_1 \qquad \Gamma \vdash S_2 \searrow F_2}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi} \searrow F_1 \cap F_2}$$

$$\frac{\Gamma \vdash e : U \qquad U \leq T \qquad x \neq \mathsf{self} \qquad (\Gamma, x : T) \vdash S \searrow F}{\Gamma \vdash T\ x := e\ \mathbf{in}\ S \searrow F - \{x\}}$$

Table 4. Typing rules for expressions and commands not in Table 1.

Definition 9 (admissible partition). *An* admissible partition *of heap h is a set of pairwise disjoint heaps $Ch, Uh, Oh_1, Rh_1, \ldots, Oh_k, Rh_k$, for $k \geq 0$, with*

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

such that for all i $(1 \leq i \leq k)$

- *$dom\ Oh_i \subseteq locs(Own{\downarrow})$ and $size(dom\ Oh_i) = 1$* *(owner blocks)*
- *$dom\ Rh_i \subseteq locs(Rep{\downarrow})$* *(confined rep blocks)*
- *$dom\ Ch \cap locs(Own{\downarrow}, Rep{\downarrow}) = \varnothing$* *(client block)*
- *$dom\ Uh \subseteq locs(Rep{\downarrow})$* *(free Reps)*

Definition 10 (confined heap, confining partition). *A heap h* is confined*, written $conf\, h$, iff h has a confining partition. A* confining partition *is an admissible partition such that for all j, k with $j \neq k$ we have*

1. *$Ch \not\leadsto Rh_j$* *(clients do not point to confined reps)*
2. *$Oh_j \not\leadsto Rh_i$* *(owners do not share reps)*
3. *$Uh \not\leadsto Oh_j * Rh_j$* *(free reps do not point to islands)*
4. *$Oh_j * Rh_j \not\leadsto Uh$* *(islands do not point to free Reps)*
5. *$Oh_j \not\leadsto^{\bar{g}} Rh_j$ where $\bar{g} = dom(dfields(Own))$* *(reps are private to Own)*
6. *$Rh_j \not\leadsto Oh_k * Rh_k$* *(reps are confined to their islands)*

Definition 11 (confined store, global state). *Let h be a confined heap and η be a store in $[\![\Gamma]\!]$. We say η is* confined in h for Γ *iff*

$$
\begin{array}{ll}
[\![\mathbf{bool}]\!] = \{\mathit{true}, \mathit{false}\} \qquad [\![\mathbf{unit}]\!] = \{\mathit{it}\} & \\
[\![C]\!] & = \{\mathit{nil}\} \cup \{\ell \mid \ell \in \mathit{Loc} \wedge \mathit{loctype}\,\ell \leq C\} \\
[\![\Gamma]\!] & = \{\eta \mid \mathit{dom}\,\eta = \mathit{dom}\,\Gamma \wedge \eta\,\mathsf{self} \neq \mathit{nil} \wedge \forall x \in \mathit{dom}\,\eta \bullet \eta\,x \in [\![\Gamma\,x]\!]\} \\
[\![\mathit{state}\,C]\!] & = \mathsf{if}\ C \not\leq \mathit{Rep} \\
 & \quad \mathsf{then}\ \{s \mid \mathit{dom}\,s = \mathit{dom}(\mathit{fields}\,C) \wedge \forall (f:T) \in \mathit{fields}\,C \bullet s\,f \in [\![T]\!]\,\} \\
 & \quad \mathsf{else}\ \{s \mid \mathit{dom}\,s = \{\mathsf{pin}\} \cup \mathit{dom}(\mathit{fields}\,C) \wedge s(\mathsf{pin}) \in [\![\mathbf{bool}]\!] \\
 & \qquad\qquad \wedge \forall (f:T) \in \mathit{fields}\,C \bullet s\,f \in [\![T]\!]\,\} \\
[\![\mathit{Heap}]\!] & = \{h \mid \mathit{dom}\,h \subseteq_{\mathit{fin}} \mathit{Loc} \wedge \forall \ell \in \mathit{dom}\,h \bullet h\ell \in [\![\mathit{state}(\mathit{loctype}\ \ell)]\!] \wedge \mathit{rng}(h\,\ell) \cap \mathit{Loc} \subseteq \mathit{dom}\,h\} \\
[\![\mathit{Heap} \otimes \Gamma]\!] & = \{(h, \eta) \mid h \in [\![\mathit{Heap}]\!] \wedge \eta \in [\![\Gamma]\!] \wedge \mathit{rng}\,\eta \cap \mathit{Loc} \subseteq \mathit{dom}\,h\} \\
[\![\mathit{Heap} \otimes T]\!] & = \{(h, d) \mid h \in [\![\mathit{Heap}]\!] \wedge d \in [\![T]\!] \wedge (d \in \mathit{Loc} \Rightarrow d \in \mathit{dom}\,h)\} \\
[\![C,\, \bar{x},\, \bar{T} \to T]\!] & = [\![\mathit{Heap} \otimes (\bar{x} : \bar{T}, \mathsf{self} : C)]\!] \to [\![(\mathit{Heap} \otimes T)_\perp]\!] \\
[\![\mathit{MEnv}]\!] & = \{\mu \mid \forall C, m \bullet \mu C m \text{ is defined iff } \mathit{mtype}(m, C) \text{ is defined,} \\
 & \qquad \text{and } \mu C m \in [\![C, \mathit{pars}(m, C), \mathit{mtype}(m, C)]\!] \text{ if } \mu C m \text{ defined }\}
\end{array}
$$

Table 5. Semantic domains.

1. $(\Gamma\,\mathsf{self}) \not\leq \mathit{Rep} \wedge (\Gamma\,\mathsf{self}) \not\leq \mathit{Own}$ *implies that there is some confining partition such that* $\mathit{rng}\,\eta \cap \mathit{locs}(\mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Uh})$.
2. *If* $(\Gamma\,\mathsf{self}) \leq \mathit{Own}$ *then* $\mathit{rng}\,\eta \cap \mathit{locs}(\mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Rh}_j)$ *for some confining partition and* j *with* $\eta\,\mathsf{self} \in \mathit{dom}(\mathit{Oh}_j)$.
3. *If* $(\Gamma\,\mathsf{self}) \leq \mathit{Rep}$ *then for any confining partition* $\eta\,\mathsf{self} \in \mathit{dom}(\mathit{Uh}) \Rightarrow \mathit{rng}\,\eta \cap \mathit{Locs} \subseteq \mathit{dom}(\mathit{Ch} * \mathit{Uh})$ *and* $\eta\,\mathsf{self} \in \mathit{dom}(\mathit{Rh}_j) \Rightarrow \mathit{rng}\,\eta \cap \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow}) \subseteq \mathit{dom}(\mathit{Oh}_j * \mathit{Rh}_j)$.

A global state (h, η) *is* confined *for* Γ, *written* $\mathit{conf}\,\Gamma\,(h, \eta)$, *iff* h *is confined and* η *is confined in* h *for* Γ.

Lemma 5. *If* $P \subseteq Q$ *and* $h \preceq^Q h_0$ *then* $h \preceq^P h_0$.

Lemma 6. *If* h *and* h' *(resp.* h_0 *and* h'_0*) are equal up to pinning then, for any* P, $h \preceq^P h_0$ *iff* $h' \preceq^P h'_0$.

Lemma 7. *If* h, h_0 *are equal up to pinning then* $\mathit{conf}\,\Gamma\,(h, \eta)$ *iff* $\mathit{conf}\,\Gamma\,(h_0, \eta)$, *for any* P.

Lemma 8. *If* h *and* h_0 *are equal up to pinning then* $\mathit{isol}\,\Gamma\,(h, \eta)$ *iff* $\mathit{isol}\,\Gamma\,(h_0, \eta)$.

Lemma 9. *If* $\mathit{conf}\,\Gamma\,(h, \eta)$ *and* $\mathit{rng}\,\eta \cap \mathit{locs}(\mathit{Rep}{\downarrow}) \subseteq P$ *and* $h \preceq^P h_0$ *then* $\mathit{conf}\,\Gamma\,(h_0, \eta)$.

Lemma 10. *If* $\mathit{iconf}\,\mu$ *and* $[\![\Gamma \vdash S \searrow F]\!]\mu(h, \eta) = (h_0, \eta_0)$ *then* $\mathit{pnd}\,h = \mathit{pnd}\,h_0$.

Definition 12. ***(confined expression)*** *Expression* $\Gamma \vdash e : T$ *is confined iff for any* (h, η), *if* $\mathit{conf}\,\Gamma\,(h, \eta)$, *and* $[\![\Gamma \vdash e : T]\!](h, \eta) \neq \perp$ *then the following hold, where* $d = [\![\Gamma \vdash e : T]\!](h, \eta)$.

1. $(\Gamma\,\mathsf{self}) \not\leq \mathit{Rep} \wedge (\Gamma\,\mathsf{self}) \not\leq \mathit{Own}$ *implies that there is some confining partition such that* $d \in \mathit{locs}(\mathit{Rep}{\downarrow}) \Rightarrow d \in \mathit{dom}(\mathit{Uh})$.

$[\![\Gamma \vdash x := e]\!]\mu(h,\eta) =$
$\quad \mathsf{let}\ d = [\![\Gamma \vdash e : T]\!](h,\eta)\ \mathsf{in}\ (h, [\eta \mid x \mapsto d])$
$[\![\Gamma \vdash e_1.f := e_2]\!]\mu(h,\eta) =$
$\quad \mathsf{let}\ \ell = [\![\Gamma \vdash e_1 : (\Gamma\,\mathsf{self})]\!](h,\eta)\ \mathsf{in}$
$\quad \mathsf{if}\ \ell = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else}$
$\quad \mathsf{let}\ d = [\![\Gamma \vdash e_2 : U]\!](h,\eta)\ \mathsf{in}\ ([h \mid \ell \mapsto [h\ell \mid f \mapsto d]], \eta)$
$[\![\Gamma \vdash x := \mathbf{new}\ B]\!]\mu(h,\eta) =$
$\quad \mathsf{let}\ \ell = \mathit{fresh}(B,h)\ \mathsf{in}$
$\quad \mathsf{let}\ h_0 = [h \mid \ell \mapsto [\mathit{fields}\,B \mapsto \mathit{defaults}]]\ \mathsf{in}\ (h_0, [\eta \mid x \mapsto \ell])$
$[\![\Gamma \vdash S_1;\ S_2 \searrow F_2]\!]\mu(h,\eta) =$
$\quad \mathsf{let}\ (h_1,\eta_1) = [\![\Gamma \vdash S_1 \searrow F]\!]\mu(h,\eta)\ \mathsf{in}\ [\![(\Gamma \downharpoonleft F) \vdash S_2 \searrow F_2]\!]\mu(h_1,\eta_1)$
$[\![\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}]\!]\mu(h,\eta) =$
$\quad \mathsf{let}\ b = [\![\Gamma \vdash e : \mathbf{bool}]\!](h,\eta)\ \mathsf{in}$
$\quad \mathsf{let}\ (h_0,\eta_0) = \mathsf{if}\ b\ \mathsf{then}\ [\![\Gamma \vdash S_1 \searrow F_1]\!]\mu(h,\eta)\ \mathsf{else}\ [\![\Gamma \vdash S_2 \searrow F_2]\!]\mu(h,\eta)\ \mathsf{in}$
$\quad (h_0, \eta_0 \downharpoonleft (F_1 \cap F_2))$
$[\![\Gamma \vdash T\ x := e\ \mathbf{in}\ S]\!]\mu(h,\eta) =$
$\quad \mathsf{let}\ d = [\![\Gamma \vdash e : U]\!](h,\eta)\ \mathsf{in}$
$\quad \mathsf{let}\ \eta_1 = [\eta \mid x \mapsto d]\ \mathsf{in}$
$\quad \mathsf{let}\ (h_1,\eta_2) = [\![(\Gamma, x : T) \vdash S \searrow F]\!]\mu(h,\eta_1)\ \mathsf{in}\ (h_1, (\eta_2 \downharpoonleft x \downharpoonleft F))$

Table 6. Semantics of commands other than method call.

$$\Gamma \Vdash x : \Gamma x \qquad \Gamma \Vdash \mathbf{null} : B \qquad \Gamma \Vdash \mathbf{true} : \mathbf{bool} \qquad \Gamma \Vdash \mathbf{false} : \mathbf{bool}$$

$$\frac{\Gamma \Vdash e_1 : T_1 \qquad \Gamma \Vdash e_2 : T_2}{\Gamma \Vdash e_1 = e_2 : \mathbf{bool}} \qquad \frac{\Gamma \Vdash e : D}{\Gamma \Vdash (B)\ e : B} \qquad \frac{\Gamma \Vdash e : D}{\Gamma \Vdash e\ \mathbf{is}\ B : \mathbf{bool}}$$

Table 7. Safety for expressions other than field access

2. $(\Gamma\,\mathsf{self}) \leq \mathit{Own} \Rightarrow (d \in \mathit{locs}(\mathit{Rep}{\downarrow}) \Rightarrow d \in \mathit{dom}(\mathit{Rh}_j))$ *for some confining partition and* j *with* $\eta\,\mathsf{self} \in \mathit{dom}(\mathit{Oh}_j)$
3. *If* $(\Gamma\,\mathsf{self}) \leq \mathit{Rep}$ *then for any confining partition*
   $\eta\,\mathsf{self} \in \mathit{dom}(\mathit{Uh}) \wedge d \in \mathit{Locs} \Rightarrow d \in \mathit{dom}(\mathit{Ch} * \mathit{Uh})$
   and $\eta\,\mathsf{self} \in \mathit{dom}(\mathit{Rh}_j) \wedge d \in \mathit{locs}(\mathit{Own}{\downarrow}, \mathit{Rep}{\downarrow}) \Rightarrow d \in \mathit{dom}(\mathit{Oh}_j * \mathit{Rh}_j)$

Static Analysis

Lemma 11 (safe expressions). *If* $\Gamma \Vdash e : T$ *then* $\Gamma \vdash e : T$ *is confined.*

Lemma 12 (safe commands). *If* $\Gamma \Vdash S \searrow F$ *and all method calls in* S *have isolated arguments then* $\Gamma \vdash S \searrow F$ *is confined.*

Proof. Let $C = (\Gamma\,\mathsf{self})$. Now we go by induction on $\Gamma \Vdash S \searrow F$ and by cases on C. Assume $\mathit{conf}\,\Gamma\,(h,\eta)$ and $\mathit{iconf}\,\mu$ and let $(h_0,\eta_0) = [\![\Gamma \vdash S]\!]\mu(h,\eta)$. In each case we must show $\mathit{conf}\,(\Gamma \downharpoonleft F)\,(h_0,\eta_0)$. We show the subcase of the method call case that involves constraint (d).

$$\frac{C \leq \mathit{Rep} \;\Rightarrow\; B \not\leq \mathit{Own}}{\Gamma \Vdash x := \mathbf{new}\ B}$$

$$\frac{\Gamma \Vdash e_2 : U \qquad \begin{array}{c} C = (\Gamma\,\mathsf{self}) \qquad \Gamma \Vdash e_1 : C \\ C = \mathit{Own} \wedge e_1 \neq \mathsf{self} \;\Rightarrow\; U \not\lesseqgtr \mathit{Rep} \end{array} \qquad C < \mathit{Own} \;\Rightarrow\; U \not\lesseqgtr \mathit{Rep}}{\Gamma \Vdash e_1.f := e_2}$$

$$\frac{\Gamma \Vdash e : T}{\Gamma \Vdash x := e} \qquad\qquad \frac{\Gamma \Vdash S_1 \searrow F_1 \qquad (\Gamma \downharpoonleft F_1) \Vdash S_2 \searrow F}{\Gamma \Vdash S_1;\ S_2 \searrow F}$$

$$\frac{\Gamma \Vdash e : \mathbf{bool} \qquad \Gamma \Vdash S_1 \searrow F_1 \qquad \Gamma \Vdash S_2 \searrow F_2}{\Gamma \Vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi} \searrow (F_1 \cap F_2)}$$

$$\frac{\Gamma \Vdash e : U \qquad (\Gamma, x : T) \Vdash S \searrow F}{\Gamma \Vdash T\ x := e\ \mathbf{in}\ S \searrow (F - \{x\})}$$

Table 8. Safety for commands other than method call

As in the semantics of method call, let $\eta_1 = [\bar{x} \mapsto \bar{d}, \mathsf{self} \mapsto \ell]$, let $P = \{\ell' \mid (\ell' \in \mathit{rng}\,(\eta \downharpoonleft F) \vee \ell' = \ell) \wedge (\mathit{loctype}\,\ell' \leq \mathit{Rep}) \wedge \neg(h\,\ell'\,\mathsf{pin})\}$, let $h_1 = \mathit{pin}\,P\,h$, let $(h_2, d_1) = \mu(\mathit{loctype}\,\ell)m(h_1, \eta_1)$, and let $h_0 = \mathit{unpin}\,P\,h_2$. Note that $\eta_0 = ([\eta \mid x \mapsto d_1] \downharpoonleft F$. Let $\Gamma_1 = [\bar{x} : \bar{U}, \mathsf{self} : (\mathit{loctype}\,\ell)]$. Because m has isolated arguments, by Lemma 3, $\mathit{conf}\,\Gamma_1\,(h, \eta_1)$. Then by Lemma 7, $\mathit{conf}\,\Gamma_1\,(h_1, \eta_1)$.

Let $Q = \{\ell' \mid (\ell' \in \mathit{rng}\,(\eta \downharpoonleft F) \vee \ell' = \ell) \wedge (\mathit{loctype}\,\ell' \leq \mathit{Rep}) \wedge h\,\ell'\,\mathsf{pin}\}$. Then $\mathit{pnd}(\mathit{pin}\,P\,h) = Q \cup \mathit{pnd}\,h$. Because $h_1 = \mathit{pin}\,P\,h$, we get $\mathit{rng}(\eta \downharpoonleft F) \cap \mathit{locs}(\mathit{Rep}{\downarrow}) \subseteq Q$. By Def. 3, we obtain $h_1 \preceq^{\mathit{pnd}(\mathit{pin}\,P\,h)} h_2$, so by Lemma 5, $h_1 \preceq^{Q} h_2$ and $h_1 \preceq^{\mathit{pnd}\,h} h_2$. Because $h_0 = \mathit{unpin}\,P\,h_2$, we get by Lemma 6 that $h \preceq^{Q} h_0$ and $h \preceq^{\mathit{pnd}\,h} h_0$. Then, by Lemma 9, and assumption $\mathit{conf}\,\Gamma\,(h, \eta)$ we get $\mathit{conf}\,(\Gamma \downharpoonleft F)\,(h_0, (\eta \downharpoonleft F))$.

It remains to deal with $d_1 = \eta_0\,x$, for which we go by cases on C.

Case $C \not\leq \mathit{Own} \wedge C \not\leq \mathit{Rep}$: We want some confining partition of h_0 such that $d_1 \in \mathit{locs}(\mathit{Rep}{\downarrow})$ implies d_1 appears in the free reps of the partition. We now proceed by cases on the callee's class $\mathit{loctype}\,\ell$. We show the proof of one subcase.

Subcase $\mathit{loctype}\,\ell \leq \mathit{Own}$: Because $\Gamma \Vdash e : D$, by Lemma 11, $\Gamma \vdash e : D$ is confined for C. Let $\ell \in \mathit{dom}(\mathit{Oh}_j)$. Because $h \preceq^{\mathit{pnd}\,h} h_0$, $\ell \in \mathit{dom}(\mathit{Oh}^0_j)$. By Def. 3 condition 4, if $d_1 \in \mathit{locs}(\mathit{Rep}{\downarrow})$, then $d_1 \in \mathit{dom}(\mathit{Rh}^0_j)$. Case (d) of the analysis applies. If $T \not\lesseqgtr \mathit{Rep}$, then the result follows vacuously. Otherwise, $\mathit{isol\text{-}tag}\,T$. Then, by condition (2) of definition 3, $\mathit{isol}\,h_2\,d_1$. By Lemma 8, $\mathit{isol}\,h_0\,d_1$. Let Vh^0 be the subheap of h_0 determined by $\mathit{isol}\,h_0\,d_1$. By Lemma 1, we can transfer Vh^0 to the free reps. Thus $h_0 = \mathit{Ch}^0 * \mathit{Uh}'^0 * \ldots * \mathit{Oh}^0_j * \mathit{Rh}'^0_j * \ldots$, with $\mathit{Uh}'^0 = \mathit{Uh}^0 * \mathit{Vh}^0$ and $\mathit{Rh}^0_j = \mathit{Rh}'^0_j * \mathit{Vh}^0$, yields a confining partition of h_0. Thus d_1 appears in the free reps of the partition. Moreover, by Def. 3, $\mathit{pnd}\,h = \mathit{pnd}\,h_0$. Hence nothing in Vh^0 is pinned, so $h \preceq^{\mathit{pnd}\,h} h_0$ holds.

Abstraction theorem

Definition 13 (typed bijection). *A* typed bijection *is finite bijective function σ from Locs to Locs such that $\sigma\,\ell = \ell'$ implies $loctype\,\ell = loctype\,\ell'$.* $\square$

Definition 14 (basic coupling). *A* basic coupling *is a function R that assigns to each typed bijection a binary relation $R\,\sigma$ on heaps (not necessarily closed heaps) that satisfies the following. For any σ, h, h', if $R\;\sigma\;h\;h'$ 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. $\mathcal{R}\;\sigma\;(type(f, loctype\,\ell))\;(h\ell f)\;(h'\ell' f)$ *for all* $(f : T) \in dom(fields(loctype\,\ell))$ *with* $f \notin \bar{g} = dom(dfields(Own))$ *and* $f \notin \bar{g}' = dom(dfields'(Own))$. $\square$

Item (3) uses the induced coupling $\mathcal{R}$ defined below; it is a harmless forward reference because the definition of $\mathcal{R}$ for data types does not depend on $\mathcal{R}$ (or R) 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 15 (coupling relation, $\mathcal{R}\,\sigma\,\theta$). *Given basic coupling R, we define for each σ and semantic category θ a relation $\mathcal{R}\,\sigma\,\theta \subseteq [\![\theta]\!] \times [\![\theta]\!]'$ as follows.*

For heaps h, h', we define $\mathcal{R}$ Heap h h' iff there are confining partitions $h = Ch * Uh * Oh_1 * Rh_1 \dots Oh_n * Rh_n$ and $h' = Ch' * Uh' * Oh'_1 * Rh'_1 \dots Oh'_n * Rh'_n$ such that

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

For other categories θ we define $\mathcal{R}\,\theta$ in Table 9.

$$
\begin{array}{ll}
\mathcal{R}\ \sigma\ \mathbf{bool}\ d\ d' & \Leftrightarrow d = d' \\
\mathcal{R}\ \sigma\ \mathbf{unit}\ d\ d' & \Leftrightarrow d = d' \\
\mathcal{R}\ \sigma\ C\ d\ d' & \Leftrightarrow \sigma\, d = d' \vee d = \mathit{nil} = d' \\
\mathcal{R}\ \sigma\ \Gamma\ \eta\ \eta' & \Leftrightarrow \forall x \in \mathit{dom}\, \Gamma \bullet \mathcal{R}\ \sigma\ (\Gamma x)\ (\eta x)\ (\eta' x) \\
\mathcal{R}\ \sigma\ (\mathit{state}\, C)\ s\ s' & \Leftrightarrow \\
\quad C \not\leq \mathit{Own} \wedge \forall f \in \mathit{dom}(\mathit{fields}\, C) \bullet \mathcal{R}\ \sigma\ (\mathit{type}(f, C))\ (s\ f)\ (s'\ f) & \\
\mathcal{R}\ \sigma\ (\theta_\perp)\ \alpha\ \alpha' & \Leftrightarrow (\alpha = \perp = \alpha') \vee (\alpha \neq \perp \neq \alpha' \wedge \mathcal{R}\ \sigma\ \theta\ \alpha\ \alpha') \\
\mathcal{R}\ \sigma\ (\mathit{Heap} \otimes \Gamma)\ (h, \eta)\ (h', \eta') & \Leftrightarrow \mathcal{R}\ \sigma\ \mathit{Heap}\ h\ h' \wedge \mathcal{R}\ \sigma\ \Gamma\ \eta\ \eta' \\
\mathcal{R}\ \sigma\ (\mathit{Heap} \otimes T)\ (h, d)\ (h', d') & \Leftrightarrow \mathcal{R}\ \sigma\ \mathit{Heap}\ h\ h' \wedge \mathcal{R}\ \sigma\ T\ d\ d' \\
\mathcal{R}\ (C, \bar{x}, \bar{T} \rightarrow T)\ d\ d' & \Leftrightarrow \forall \sigma,\ (h, \eta) \in [\![\mathit{Heap} \otimes \Gamma]\!], (h', \eta') \in [\![\mathit{Heap} \otimes \Gamma]\!]' \bullet \\
\quad \mathcal{R}\ \sigma\ (\mathit{Heap} \otimes \Gamma)\ (h, \eta)\ (h', \eta') \wedge \mathit{conf}\, C\, (h, \eta) \wedge \mathit{conf}\, C\, (h', \eta') \wedge \mathit{freeRep}(C, h, h', \eta, \eta') & \\
\quad \Rightarrow \exists \sigma_0 \supseteq \sigma \bullet \mathcal{R}\ \sigma_0\ (\mathit{Heap} \otimes T)_\perp\ (d(h, \eta))\ (d'(h', \eta')) & \\
\quad \text{where } \Gamma = [\bar{x} \mapsto \bar{T}, \mathsf{self} \mapsto C] &
\end{array}
$$

Finally, $\mathcal{R}\ \mathit{MEnv}\ \mu\ \mu'$ iff for all C, m with $\mathit{pars}(m, C) = \bar{x}$ and $\mathit{mtype}(m, C) = \bar{T} \rightarrow T$, we have $\mathcal{R}\ (C, \bar{x}, (\bar{T} \rightarrow T))\ (\mu C m)\ (\mu' C m)$.

Table 9. The induced coupling relation, where we use $\mathit{freeRep}(C, h, h', \eta, \eta')$ iff either $C \leq \mathit{Rep} \wedge C \leq \mathit{Rep}'$ or there are confining partitions satisfying the conditions for $\mathcal{R}\, \sigma\, \mathit{Heap}$ and moreover $\eta\, \mathsf{self} \in \mathit{dom}(Uh)$ and $\eta'\, \mathsf{self} \in \mathit{dom}(Uh')$ (the last two conditions are equivalent in the contexts where $\mathit{freeRep}$ is used).