Ownership: transfer, sharing, and encapsulation

Anindya Banerjee* and David A. Naumann **

Kansas State University and Stevens Institute of Technology

Abstract. Ownership confinement expresses encapsulation in heap structures, in support of modular reasoning about effects, representation independence, and other properties. Most previous proposals for static enforcement of ownership confinement require annotations of whole programs using intricate type systems that are difficult to validate. Moreover, few proposals deal with transfer and sharing of ownership. We show how consideration of a reasoning objective suggests a particular pattern of ownership amenable to lightweight and modular imposition of confinement. We show how the pattern extends to handle transfer, at the cost of a heavier analysis.

1 Introduction

For scalability in reasoning about programs, be it informal reasoning by developers or automated reasoning embodied in static analysis tools, modularity is essential. Modularity is achieved using encapsulation mechanisms such as visibility modifiers for object fields, but cannot be achieved fully in object-oriented programs without control of sharing of objects in the heap —for which we use the general term *confinement*. This paper reports on work towards a theory for reasoning about representation independence (i.e., class equivalence) for sequential Java programs on the basis of a flexible form of *ownership confinement*. To handle ownership transfer, it turns out that we also need an additional confinement property which is similar to the recent proposal called external uniqueness [9].

To be useful, a confinement discipline should (a) ensure some invariant that facilitates modular reasoning about some property of interest; (b) be sufficiently flexible to be applicable and useful for an interesting class of programs; and (c) be amenable to efficient static checking. Let us consider some example invariants, loosely described. *Uniqueness* is often used to strongly delimit effects, e.g., to avoid the need for a lock [16]: the invariant is that the object referenced from a unique field is not accessible from anywhere else. Boyapati et al. [5] use a form of *ownership* as basis for a locking discipline: if a thread holds a lock on a certain object, it need not lock objects owned by that object. Strong forms of ownership have been proposed for various kinds of modular reasoning [14, 15, 10, 7, 23, 20, 1]. A typical ownership invariant is that an owned object is not accessible except via the object that owns it. Ownership is ubiquitous, as it is

* e-mail:`ab@cis.ksu.edu`; Supported by NSF grants CCR-0209205 and CCR-0296182.

** e-mail:`naumann@cs.stevens-tech.edu`; Supported by NSF grant CCR-0208984.

a natural embodiment of aggregation and the encapsulation of representations. Achieving such encapsulation is a key design objective, as it supports local reasoning about mutable state, in the form of frame specifications (the "modifies" clause, frame rules) [19, 17], equivalence between versions of a class [3, 2], and general non-interference assertions [8, 4].

In previous work [3], we formulated a semantic notion of ownership and proved that it is a sufficiently strong invariant to justify *representation independence*, i.e, modular reasoning about equivalence of class implementations using the standard notion of simulation [18, 13, 21, 11]. This notion of ownership has several restrictions that make it inflexible in ways similar to some of the earlier proposals [14, 15]. One restriction is that encapsulated representation objects may not have *outgoing references* to clients of the owner. In the full version [2] of [3], this restriction was lifted and a static analysis given for confinement. The analysis is modular in the sense that constraints are imposed only on the owner class and its representation class(es).

The present paper extends our previous work by overcoming two other restrictions that have also been challenging for other work on ownership: multiple ownership and ownership transfer, i.e., transfer of representations between owners. A leading example of multiple ownership is collection classes, where a collection object encapsulates nodes of a data structure but allows access to those nodes by iterators. An example of ownership transfer is a group of collections, such as task queues among which tasks are transfered for load balancing.

2 Representation independence and ownership

We consider a simple example of representation independence, to show the significance of ownership confinement for showing equivalence between two versions of a class. This leads us to focus on a particular pattern of heap encapsulation which is applicable to a wide class of programs and which admits a lightweight static analysis (i.e., syntax-directed and requiring no program annotations).

Consider class ListNode in Fig. 1. Its instances are used in class Fifo; the nodes reachable from an instance of Fifo comprise the representation thereof. We use Java-like notation; in particular, class types are implicitly reference types. Here and throughout we consider fields to be private; methods are public unless otherwise indicated.

Privacy of fields front and rear helps encapsulate the objects that are intended to comprise an internal data structure which should not be directly accessible to clients of the queue. Not all reachable objects are within the representation, however; the contained items are not.

One might wish to reduce the overhead of object construction and destruction by substituting this data structure with another that uses an array. Provided that the public interface does not change, this concern should be local to class Fifo. To ensure that this change of representation does not affect the behavior of clients, the standard reasoning (based on simulation [13]) involves showing that clients do not depend on the internal representation, but depend only on the behavior

```
class ListNode {
  Object item; ListNode nxt;
  Object getIt(){ return item; }
  ListNode getNxt(){ return nxt; }
  void setIt(Object o){ item := o; }
  void setNxt(ListNode n){ nxt := n; } }
class Fifo { // owner of the nodes reachable from its fields
  ListNode front, rear; // least, most recent
  void enq(Object o){ ListNode n := new ListNode();
    n.setIt(o); if (front==null){ front := n; rear := n; } else { rear.setNxt(n); rear := n; } }
  Object deq(){ ListNode n := front;
    if (front==rear){ front := null; rear := null; } else { front := front.getNxt(); }
    return n.getIt(); }
```

Fig. 1. Toy example of FIFO queue.

of Fifo observed through its public methods. It is for this that confinement is needed: Without restrictions on aliasing to enforce encapsulation, *representation exposure* can occur. A client can depend on the representation, and indeed clients can interfere with the behavior of the queue.

Representation exposures violate various confinement properties one can think of. One realistic example similar to the toy queue example is the Java Class Signers bug. As discussed by Vitek and Bokowski [23], the type of the leaked object in this example has package scope. They propose that such objects should not escape from the package and they give a static analysis for package confinement; of course the example violates their static checking rules.

Package confinement has several limitations. Although package confinement is an invariant that delimits the scope of effects, and in that way supports modular reasoning, it is a rather large-scale or coarse-grained property. In particular, to use it for reasoning about substitution of one class implementation by another, one would have to reason about all instances of the class at once. But a programmer changing the representation for Fifo likely thinks in terms of a single instance. This is typical for objects that provide a collection or other data abstraction. Also, package confinement does not help with the situation where some public class, e.g., HashTable, is used for encapsulated representations but also for many other purposes. Our previous work [3] also failed to handle this situation.

For reasoning about class equivalence, as in the queue example, it is natural to focus on a single instance of the class to be replaced. Various forms of *ownership confinement* have been proposed, in which each instance of class Fifo can be viewed as *owning* its internal representation, which is not shared with other instances. To formulate a general rule for using simulation to prove equivalence between one implementation of a class and another, it is enough to focus on

– an instance of the class, say *Own*, to be replaced (e.g., a Fifo object)

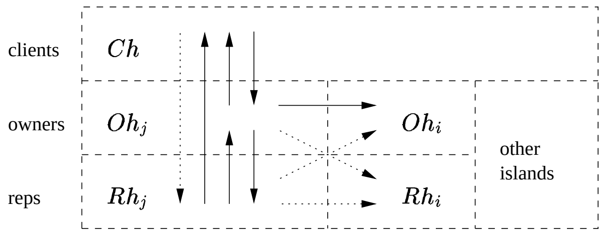

Fig. 2. Confinement scheme for island j with respect to another island i.Dashed boxes delimit partition blocks. No objects are shown. Solid lines indicate allowed references and dotted lines indicate prohibited ones. There is no restriction within blocks.

- the internal representation for that instance —the encapsulated objects, called *reps*, on which clients should not depend, and which may be replaced. We assume these objects have a common superclass *Rep* (the generalization to a covering set of superclasses is straightforward).
- Objects that are neither owners nor representations thereof —which we lump under the term *clients.*

Thus we arrive at the view of the heap depicted in Fig. 2. This view has a simple characterization in terms of types. The set of objects in the heap is partitioned[1] into the following blocks:

- The client block, *Ch*, containing any object with type C such that $C \not\leq Own$ and $C \not\leq Rep$.
- 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 $D \leq Rep$).[2]

The invariant on the heap (called *heap confinement*) that must be maintained can now be formulated as restrictions on the direct points-to relation (see Fig. 2). A heap is confined provided there is a partition (as above) such that

- Clients do not point to encapsulated reps.
- Islands are separated from each other:
  - Owners do not point to reps in a different island.
  - Reps do not point to reps or owners in a different island.
- The pointers from Oh_i to Rh_i are in private fields of Own. Methods defined in a subclass may manipulate reps but not store them in its fields; this allows reasoning about versions of the owner class independently from its subclasses. (See [2] for examples of how this fits with the factory pattern.)

For instance-oriented reasoning about substitution of an owner class, our previous work [2] gives an *abstraction theorem* that formally justifies such reasoning

[1] We allow the "partition" blocks to be empty.
[2] We use the symbol $*$ for union of disjoint heaps, as in Separation Logic [22].

for programs that preserve confinement. A key point is that the basis for defining simulations —the hypothesis of the proof rule for class equivalence— is a relation connecting a single island for each of the two representations. Hence in light of Figure 2, we can sharpen the reasoning that justifies replacement of one class implementation by another: (a) The reasoner defines a *basic coupling* relation for a single island: that is, how an island for one implementation corresponds to an island for another. To connect, e.g., two versions of class Fifo, the basic coupling relates a single island $Oh * Rh$ for the first version (so Rh contains the instances of ListNode for a single queue) with an island $Oh' * Rh'$ for the second version (where Rh' might contain an array or whatever other data structure is used). (b) Next, for each method of the class, the reasoner proves the simulation property by showing that the corresponding implementations preserve not the coupling itself but the induced relation on complete heaps. This accounts for the fact that client objects and other owners can be reached from the island, while supporting per-instance reasoning. The simulation property for class Fifo, for example, is proved in terms of methods of Fifo executing on a single instance of Fifo (though behavior of a method may depend on objects elsewhere in the heap, due to outgoing calls to client methods).

The present paper pursues this approach further by exploring richer notions of confinement. To conclude this section we emphasize two points. First, confinement is in the eye of the reasoner. To reason about the connection between two implementations of a class, what matters is encapsulation of the entire representation, regardless of whether it might be possible to further decompose that representation into other ownership relations.[3] (E.g., in a more object-oriented version, class ListNode could provide all the functionality of lists, via recursive calls to the tail, in which case one might view each node as owning the rest of the list. But for reasoning about Fifo, what matters is the behavior of operations on the list.)[4]

The second point is about static enforcement of confinement. Once we had found a semantic notion of confinement suitable for representation independence, we sought a static analysis, expecting that program annotations would be needed. But we were able to avoid that by making use of the program's own types in formulating confinement and confinement rules. In the present paper, we improve our formulation and show the approach to be surprisingly flexible. But to some extent we contradict this second point: to handle transfer of ownership we use annotations for a form of uniqueness.

[3] Thus we do not assume *hierarchical ownership* [7–9].

[4] As an analogy, in the proof rule for loops what matters is that the body preserves the invariant; internal structure of the loop body is not relevant. Another analogy is the frame rule in Hoare logic, i.e., the inference of $\{P \wedge R\}S\{Q \wedge R\}$ from $\{P\}S\{Q\}$ (for suitable R). What matters is the top-level connective $\wedge$; of course R may have structure but this is not relevant to the rule. In Separation Logic, the rule is extended to the heap using $*$ as top-level connective.

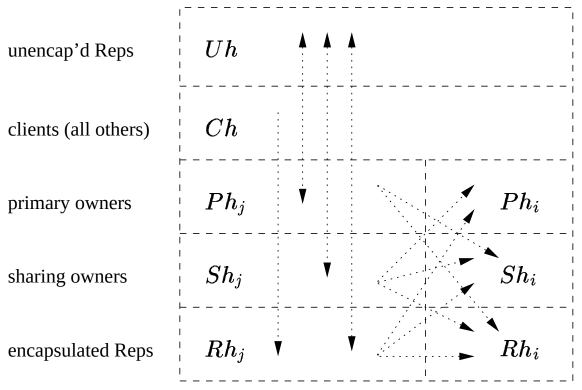

Fig. 3. Refined confinement scheme; dotted arrows indicate disallowed references.

3 Sharing ownership

A well known challenge for ownership systems is posed by collections with which iterators may be associated. As an example, let us extend class Fifo with method

```
Enumeration enum(){ return new ListEnum(front); }
```

where ListEnum is an iterator.

An iterator needs access to the representation, so it is not possible to reason about it entirely separately. For the kind of example considered here, it seems natural to consider a single *primary owner*, the collection itself (of type $\leq Own$) together with a number of *sharing owners*, e.g., instances of ListEnum (we designate their supertype as Osh). This suggests the partition scheme depicted in Fig. 3, where each primary owner is in an island containing its reps and sharing owners. (Fig. 3 also includes a block Uh that is discussed later.) The island is still the unit of encapsulation used for purposes of reasoning about equivalence, where now the equivalence is between implementations of a pair of classes Own, Osh (e.g., Fifo and ListEnum). Heap confinement can again be formulated as restrictions on the direct points-to relation (see Fig. 3). The distinction between Own and Osh is important for formulating an instance-based notion of representation-independence, in the sense that one instance of an abstraction — defined by a class— is compared with another —given by an alternative version of the class. The extended abstraction involves multiple objects, the iterators, but only those for a single primary owner.

We are not the first to treat sharing of ownership; see [19, 5, 8]. A closely related work is that by Boyapati, Liskov, and Shrira [6] which gives an ownership system that allows sharing and which is claimed to be strong enough to support modular reasoning. As they remark, most other proposals are either too permissive for sound local reasoning or too restrictive to handle iterators. However, while [6] states that type soundness can be proved by standard techniques

```
void transferTo(Fifo o){ o.pushR(self.pullR()); }
void transferFrom(Fifo o){ self.pushR(o.pullR()); }
void pushR(o-unique ListNode n) /*module scope*/ {
  if (front==null){ front := n; rear := n; } else { rear.setNxt(n); rear := n; } }
o-unique ListNode pullR() /*module scope*/ {
  ListNode n := front;
  if (front==rear){ front := null; rear := null; } else { front := front.getNxt(); }
  n.setNext(null); result := n; }
```

Fig. 4. Transfer of rep between owners.

and proves that their ownership types ensure a certain heap invariant, there is no formalization of modular reasoning much less a justification that the ownership invariant helps. Also, the achievement of [6] comes at the cost of requiring inner classes for iterators; this is a reasonable program structure but complex and challenging for formal reasoning.[5] Müller [19] handles a form of sharing and shows modular soundness for reasoning about modifies specifications; it is not clear how to adapt this result to representation independence.

All of the works cited above suffer from the shortcoming that the owner of an object is fixed. By contrast, our ownership invariant is a state predicate of the form: there is a partition of objects such that certain direct references do not exist. Our owners may traffic in reps.

4 Transfer of reps among owners

Owners may cooperate among themselves. For example, consider a number of queues of tasks, each serving its own processor. For purposes of load balancing, tasks from full queues can be transferred to empty ones. The public interface could be used to dequeue from one and enqueue to the other, but it could be difficult to do this and maintain task management information. Moreover, it incurs a performance penalty for object allocation and deallocation.

Figure 4 gives additional methods for class Fifo. Methods transferTo and transferFrom are for clients to use to transferring a rep from one Fifo instance to another. For example, the call q2.transferTo(q1) would be invoked to ask q2 to transfer one of its tasks to q1. [6] The other two methods are for use within the class, and are given package scope. Lea [16] identifies three forms of transfer via method calls: initiated by the recipient (e.g., pullR), initiated by the sender (e.g., pushR), and symmetric (exchange). Our example shows the first two, but exchange can be programmed similarly.

[5] Another point of concern is that downcasts are omitted from [6] and it appears that treating them requires runtime support; downcasts and subclassing pose non-trivial challenges for ownership systems and representation independence, which are addressed in our work and in some others (e.g., [8, 1]) by purely static means.

[6] Throughout the paper we omit both preconditions and error-checking code.

When is it safe to transfer a rep? As a first approximation we want *uniqueness* —the rep being transferred is referenced only by a unique pointer from the sending owner, say in island $Oh_j * Rh_j$, which is handed off to the receiving owner, in some island $Oh_i * Rh_i$. This ensures that putting the transferred rep in Rh_i does not create bad incoming pointers to Rh_i, such as might occur if the transferred rep had been referenced by an iterator in Oh_j. Outgoing pointers from the transferred rep to clients pose no problem. But if it had a pointer to the sending owner, or to an iterator in the sending island, the resulting state again violates confinement.

So we want a stronger property, *o-uniqueness*: a rep ℓ in $dom(Rh_j)$ may be transferred if the transferring owner in Ph_j has a unique pointer to ℓ. Moreover, consider the sub-island Rh'_j of Rh_j that is reachable from ℓ (note that ℓ is reachable from itself). We require that no locations of type $\leq Own$ or $\leq Osh$ be reachable from Rh'_j and that there be no pointers into Rh'_j from either Ph_j or Sh_j. In this case, the sub-island Rh'_j can be moved, *en masse*, to the destination island.

Although we have not completed the proofs, we are confident that these conditions suffice for a generalization of representation independence that is still based on a single-island simulation relation. The challenge is to prove, modularly, that after calling a method that may transfer reps, the caller's environment (which may contain reps) satisfies the confinement conditions. In [2] we achieve this because the effect of any command, and thus any method meaning, on a confining partition is to *extend* it: objects may be added, but they do not move between islands.

The other major challenge is to impose o-uniqueness. Whereas straightforward syntactic rules, expressed only in terms of program types, suffice to ensure the confinement invariant in the absence of transfer, such rules do not seem feasible for o-uniqueness. The reason is that we must delimit reachable reps from among other reps. It seems unlikely to achieve a useful analysis that is "lightweight" in the sense of not using annotations.

As in our previous work, our approach is to start from the ultimate objective: a reasoning principle. Both confinement and o-uniqueness are semantic properties that suffice for a useful reasoning principle, namely a form of representation independence that does not require global reasoning about the heap. Although we have a satisfactory lightweight analysis for confinement, we factor out and leave unsolved the problem of static checking for o-uniqueness. Both our representation independence result and the soundness of our static analysis for ownership confinement are proved on the assumption of o-uniqueness at points designated by explicit annotations (method parameters and returns).

5 Transfer of reps between clients and owners

Leino et al [12] point out that in many cases, representation objects need to be initialized outside the owner class [12, 8]. Resource management (such as memory managers) poses the problem of transferring objects not only from clients to encapsulating owners but also the reverse. Consider for example this manager of a toy resource Rsrc.

```
class Rsrc { Object it; Rsrc(Object i){ it := i; } }
class RsrcMgr { // owner
  Rsrc freeList;
  RsrcMgr(int n){
    for (int i := 0; i<n; i++){ Rsrc r := new Rsrc(freeList); freeList := r; } }
  o-unique Rsrc alloc(){
    Rsrc r := freeList; freeList := (Rsrc)freeList.it; r.it := null; return r; }
  void free(o-unique Rsrc r){ r.it := freeList; freeList := r; } }
```

To handle this, we refine our confinement scheme slightly, adding a single block of "unencapsulated reps", *Uh*, as depicted in Fig. 3. (The term is misleading and should be replaced; they are client objects that happen to have type $\leq$ *Rep* and are subject to the additional restriction that they may not be pointed to by encapsulated reps.) This refinement deals with another shortcoming with our lightweight formulation based on program types. The previous schemes preclude the use of library types like HashTable which might be used both for the representation of *Own* and for other purposes by clients. We no longer insist that all objects of class *Rep* are used as reps for *Own* or *Osh*; rather, reps not in use that way are sufficiently separate to admit static checking.

In the example above, alloc allocates a confined rep object and "returns" it to the client. In the heap, a transfer occurs – this is why alloc's return type is tagged **o-unique**. Following transfer, the object is no longer associated with the freeList but resides in *Uh*. As a result the owner no longer has access to it. Similarly, free disposes an unencapsulated rep object and returns it to the owner. In the heap, a transfer occurs – this is why free's parameter is tagged **o-unique**. Following transfer, the object is no longer associated with *Uh*, but resides in some Rh_i. As a result a client no longer has access to it.

6 (Scant) Discussion

We do not believe there is a single ownership discipline —much less static analysis technique— useful for all purposes. But we do believe the ownership pattern studied here is widely applicable. Once we complete a rigorous proof that it ensures strong encapsulation, a rigorous proof that our static analysis is sound for the confinement property, and a fair and thorough survey of related work, we plan to extend the formal setting to encompass safe multithreading.

Confinement disciplines can and should be formally justified in terms of the reasoning benefit: the precise confinement invariant and the reasoning consequences (such as validity of a proof rule). In our experience it is easy to write down plausible rules and definitions and much harder to get them right. Many reseachers including ourselves believe that confinement checking should be largely mechanized, so reasoners (be it the programmer or other tools) are freed to focus on other things; clearly such tools will be more useful if based on sound rules. Our work demonstrates that it is neither intractable nor entirely tedious to prove soundness. This is not to deny the importance, even the primacy, of gaining empirical experience using prototypes, but rather to say that positive empirical results for a confinement proposal should not be the end of the story.

A number of different disciplines have been proposed in recent years; while some are simply improvements on predecessors, others are incomparable in both the confinement invariant and its intended application. This suggests there is no single "right" notion of ownership, but rather a number of useful patterns of program structure and reasoning. Confinement is in the eye of the reasoner. Arguably, this implies that rather than global program annotations in support of a particular confinement discipline and checker, lightweight disciplines and checkers are needed.

Acknowledgement: Thanks to the anonymous referees for their feedback on an earlier draft.

References

1. Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA*, 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 *POPL*, pages 166–177, 2002.
4. Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *CSFW*, pages 253–270, 2002.
5. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.
6. Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, 2003.
7. David Clarke. *Object Ownership and Containment.* PhD thesis, University of New South Wales, Australia, 2001.
8. David Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, 2002.
9. David Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP*, 2003.
10. David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In *ECOOP*, 2001.
11. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge University Press, 1998.
12. D. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Technical Report 156, COMPAQ Systems Research Center, July 1998.
13. C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
14. J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, 1991.
15. John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.
16. Doug Lea. *Concurrent Programming in Java.* Addison-Wesley, second edition, 2000.
17. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5), 2002.
18. R. Milner. An algebraic definition of simulation between programs. In *Proceedings of Second Intl. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.
19. Peter Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262 of *Lecture Notes in Computer Science.* Springer-Verlag, 2002.
20. Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in Java. In *ECOOP Workshop on Formal Techniques for Java Programs.* Technical Report 269, Fernuniversität Hagen, 2000.
21. John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, 1984.
22. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.
23. Jan Vitek and Boris Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.