Selective Ownership: Combining Object and Type Hierarchies for Flexible Sharing

Stephanie Balzer
School of Computer Science
Carnegie Mellon University
balzers@cs.cmu.edu

Thomas R. Gross
Department of Computer Science
ETH Zurich
thomas.gross@inf.ethz.ch

Peter Müller
Department of Computer Science
ETH Zurich
peter.mueller@inf.ethz.ch

Abstract

Most ownership systems enforce a tree topology on a program's heap. The tree topology facilitates many aspects of programming such as thread synchronization, memory management, and program verification. Ownership-based verification techniques leverage the tree topology of an ownership system (and hence the fact that there exists a single owner) to restore sound modular reasoning about invariants over owned objects. However, these techniques in general restrict sharing by limiting modifying access to an owned object to the object's owner and to other objects in that owner's ownership tree. In this paper, we introduce *selective ownership*, a less rigid form of ownership. The key idea is to structure the heap in two ways, by defining an order on a program's type declarations and by imposing ownership on *selected objects*. The order on type declarations results in a stratified program heap but permits shared, modifying access to instances further "down" in the heap topology. By superimposing object ownership on selected objects in the heap, programmers can carve out partial sub-trees in the heap topology where the objects are owned. We show how selective ownership enables the modular verification of invariants over heap topologies that subsume shared, modifiable sub-structures. Selective ownership has been elaborated for our programming language Rumer, a programming language with first-class relationships, which naturally give rise to an ordering on type declarations.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Class invariants

General Terms Languages, Verification

Keywords Selective ownership, Universe types, Ownership types, Visible-state verification techniques, First-class relationships

1. Introduction

Object-oriented programs typically produce graphs of highly-interconnected objects. These graphs bear little resemblance to the programs that produced them, complicating any reasoning about them. Ownership type systems [17–23, 37, 41] have been shown to ease program reasoning by imposing a tree structure on a program's heap. For instance, Ownership type systems have been successfully employed for program verification [30, 34, 37, 38], for guaranteeing thread safety [13], for memory management [14], and for enforcing architectural styles [2].

However, many common design patterns and programming idioms do not naturally produce a tree structure but a heap that subsumes owned and shared objects. For instance, the nodes of Java's linked list implementation are shared and manipulated by a list's head and all its iterators. Most ownership systems either disallow such implementations or provide weak guarantees. For example, the classical Ownership types [19] enforce the *owner-as-dominator* discipline [22, 24] and thus disallow direct access to the linked list both by the list's head and its iterators. Universe types [37], on the other hand, enforce the *owner-as-modifier* discipline [22, 24] and thus permit a limited form of sharing (restricting all non-owning accesses to read-only). But the verification technique built on top of Universe types restricts sharing by limiting modifying access to an owned object to the object's owner and to other objects in that owner's ownership tree.

This paper introduces *selective ownership*, a more flexible way of giving structure to a program heap. Selective ownership allows programmers to structure the heap in two ways: *(i)* by defining an order on a program's type declarations and *(ii)* by imposing ownership on selected objects. The order on type declarations results in a stratified program heap but permits shared, modifying access to instances further "down" in the heap topology. By superimposing object ownership on selected objects in the heap, programmers can carve out partial sub-trees in the heap topology where the objects are owned.

Like classical ownership, selective ownership may have several applications. In this paper, we describe how selective ownership can facilitate the modular verification of multi-object invariants in the presence of call-backs. Selective ownership-based verification leverages the type ordering for the prevention of transitive call-backs, and ownership for the declaration and verification of invariants on owned objects. We exemplify selective ownership in the context of Rumer [4, 5], a programming language we have designed to embody *first-class relationships* [1, 4, 6, 10, 33, 39, 42, 44]. In Rumer, first-class relationships naturally define an order on type declarations, making Rumer a natural fit as a host language for selective ownership.

This paper extends our earlier work [5], in which we introduce Rumer as well as a verification technique for Rumer. The described verification technique leverages the type order defined by relationship declarations (called Matryoshka Principle) as well as a relationship-specific encapsulation mechanism (called member interposition) for the modular verification of multi-object invariants. In [5], we further briefly touch on the idea to overlay the type order with ownership. This paper works out the details of this idea and

FOOL'12 October 22, 2012, Tucson, AZ, USA

also provides an abstract presentation, detached from relationship-based programming languages and Rumer, in particular.

The rest of this paper is structured as follows: Section 2 introduces the main idea of selective ownership, independently of a specific host language. Section 3 then exemplifies selective ownership in the context of Rumer. Section 4 briefly discusses some consequences of the co-existence of owned and shared objects. Section 5 summarizes related work, and Section 6 concludes the paper.

2. Selective ownership in a nutshell

This section introduces the core ideas of selective ownership. Even though we have developed selective ownership in the context of the Rumer programming language, the core ideas of selective ownership are of general applicability. We thus keep the presentation abstract in this section and defer the Rumer-specific aspects of selective ownership to Section 3. In the following, we first show how a program's heap can be structured using selective ownership and then sketch how the imposed structure facilitates the modular verification of object invariants.

2.1 Heap structure

Selective ownership provides programmers with a "mix and match" approach to giving structure to a program's heap. At its core, the approach relies on an ordering relation on a program's type declarations. This ordering relation gives some basic structure to the program heap. If additional structuring is required, then the resulting heap can further be shaped by superimposing ownership on selected type instances in the heap.

We illustrate the approach on Figure 1 (a) and Figure 1 (b). Figure 1 (a) displays a program heap that has been shaped by declaring a type order and Figure 1 (b) displays one that has been shaped by declaring both a type order and instance ownership. To keep the presentation abstract, the two figures are not geared towards a particular programming language. Instead, they use only the notions of a type, type instance, and type instance reference. Types are represented as dark grey boxes, which enclose the instances of the type (represented as light gray circles). Type instance references are represented as arrows with a split arrowhead. In a class-based, object-oriented setting, types correspond to classes, type instances to objects, and type instance references to object references. To increase readability, we may occasionally refer to "type instances" as "instances" or "objects" and to "type instance references" as "references".

Figure 1 (a) shows the result of imposing a *strict partial order* on a program's type declarations. This figure displays the ordering relation using bold arrows with filled arrowheads, which indicate the order's transitive reduction. To give structure to a program's heap, selective ownership confines the declaration of type instance references such that an instance o of type O can only declare a reference to an instance o' of type O' if the pair $O \mapsto O'$ is an element of the type order. As shown by Figure 1 (a), the enforcement of this requirement gives rise to a program heap that forms a *directed acyclic graph (DAG)*. Similarly to the Universe type system [20–23, 37], selective ownership leaves the declaration of read-only type instance references unconstrained. This relaxation guarantees that any modifications by means of assignments or side-effect-generating method invocations comply with a program's type order but allows for arbitrary reads or pure method invocations. For simplicity, Figure 1 (a) omits read-only references, but the reader can think of such references as occurring between arbitrary instances. A DAG topology is more permissive than the tree topology typically enforced by Ownership type systems [17–23, 37, 41] as it allows for shared, modifying access to instances further "down" in the heap topology. For example, the type instance "e1", in Figure 1 (a), can both be modified by the type instances "a2" and "c1", and the type instance "f3" by the type instances "c2", "b1", and "d1".

To shape a program's heap even further, a programmer can superimpose the type ordering with instance ownership. Figure 1 (b) shows the result of imposing ownership on *selected instances* in the program heap of Figure 1 (a). Ownership is displayed by a dotted arrow from the owner to the owned instance. In Figure 1 (b), ownership has been declared for three type instances: for the type instance "e1" with the owning type instance "c1", for the type instance "d1" with the owning type instance "b2", and for the type instance "d2" with the owning type instance "b3". The ownership declaration guarantees that only the owner can declare modifying type instance references to the owned instance and, hence, modify the owned type instance. As a result, the modifying references between the instances "a2" and "e1" and the instances "b3" and "d1", which are legal in Figure 1 (a), are illegal in Figure 1 (b) and are crossed out. The declaration of corresponding read-only references, however, would be admissible since selective ownership permits arbitrary read-only references. As opposed to the ownership enforced by Ownership type systems [17–23, 37, 41], the ownership enforced by selective ownership it not transitive. This property allows for more flexibility. For example, even though the type instances "d1" and "d2" are owned by different owners, they share modifying access to the type instance "g1".

2.2 Invariant verification technique

Invariants provide a foundation for verifying programs [26], and various object-oriented programming and specification languages [9, 28, 35] have adopted invariants for objects. An *object invariant* captures the properties of an object that the object exhibits in its consistent states. Object invariants are central to a wealth of object-oriented verification techniques [7, 8, 27, 30, 32, 36–38, 40, 43]. These techniques establish appropriate proof obligations to verify at compile-time that an object satisfies its invariant at designated program points at run-time. To be practical, those proof obligations must be *modular*, allowing classes to be verified independently from each other.

Modular, object-oriented verification techniques face two key challenges: multi-object invariants and call-backs [25, 29]. A *multi-object invariant* declared for an object o is an invariant that depends not only on the state of o but also on the state of any object p that o refers to. The verification of such invariants is difficult since the invariant may be violated not only through modifications of o but also by modifications of any of the objects p. A *call-back*, on the other hand, happens when a method $m()$ (possibly transitively) invokes a method $n()$ on $m()$'s current receiver object o_m. Call-backs may complicate the adoption of a visible-state semantics for invariants. A *visible-state semantics* [25, 38] expects the current receiver object to satisfy its invariant in the initial and final states of the executing method (the so-called *visible states*) but allows temporary violations in between those states. For instance, if $m()$ (possibly transitively) calls $n()$ while the invariant of o_m is temporarily broken, then o_m would not satisfy its invariant in the initial state of $n()$ when a call-back occurs.

Ownership-based verification techniques for object invariants [30, 34, 37, 38] leverage the tree topology of an ownership system to address the above mentioned challenges. In particular, they exploit *(i)* that invariants can depend only on fields of the invariant-declaring object or on fields of objects it owns, *(ii)* that modifications of an owned object's fields are initiated[1] by the object's owner ("owner-as-modifier discipline" [22, 24]), and *(iii)* that owned objects may invoke methods only on objects with the same owner

[1] Modifications of an owned object's field are initiated by the owner if there exists a stack frame on the call-stack with the owner as the current receiver.

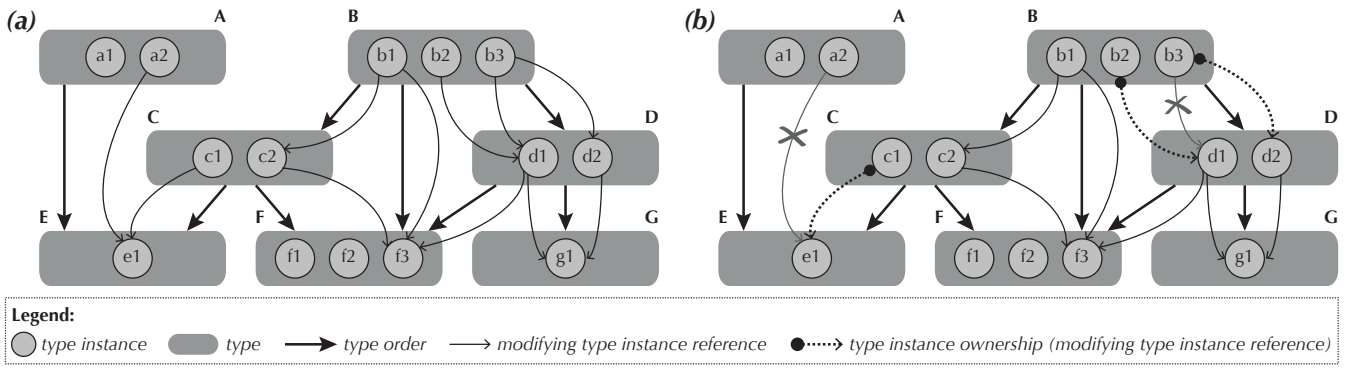

Figure 1. Abstract illustration of selective ownership: (a) valid program heap structure due to type order; (b) valid program heap structure due to type order *and* selective ownership. Illegal modifying type instance references from (a) are crossed out in (b).

or on objects they own. The first and the second property give an owner whose invariant relies on an owned object the chance to re-establish the invariant upon modifications of the owned object. The third property prevents call-backs into owners from objects within their ownership trees.

Next, we sketch how a visible-state verification technique can leverage both type ordering and object ownership to accommodate invariants over heap topologies that subsume shared, modifiable sub-structures. For simplicity, we assume that the underlying programming language allows assignments to a field f only of the form this.f = expr. In line with [38], we refer to this restriction as classical encapsulation. Our presentation draws from the visible-state verification technique we have developed for Rumer but generalizes its main ideas to fit the abstract description in this section. A detailed account of our verification technique, including its proof obligations as well as its soundness proof, can be found in [4].

A verification technique can leverage the type ordering entailed by selective ownership to prevent transitive call-backs. Since the type ordering forms a strict partial order, it is guaranteed to be *acyclic*. To prevent transitive call-backs, a verification technique simply needs to require that method invocations either propagate in the direction of a program's type ordering relation or that the caller and callee of an invocation are the same object. For example, under this restriction, the type instance "d1" in Figure 1 (a) is allowed to invoke methods only on itself or on any instances of the types "G" and "F", such as the referred-to instances "g1" and "f3".

Given the absence of transitive call-backs, the support of a visible-state semantics for single-object invariants is straightforward: a verification technique simply needs *(i)* to require invariants declared for a type instance to depend only on fields of that instance and *(ii)* to impose the proof obligation on a method to restore the invariant of the current receiver instance in the final state of the method as well as before any invocations on the current receiver instance. For example, assume that we declare an invariant for instances of type "D" in Figure 1 (a) and we consider the execution of the method $m()$ on the type instance "d1". The first requirement guarantees that $m()$ can violate at most the invariant of "d1" while it executes. The second requirement and the absence of transitive call-backs guarantee that any method invocation during $m()$'s execution encounters the callee instance in a consistent state.

The type ordering entailed by selective ownership is sufficient to prevent transitive call-backs, but not (generally[2]) sufficient to support multi-object invariants. For example, if the invariant of instance "b2" in Figure 1 (a) were allowed to depend on the state of instance "d1", a method with the receiver instance "d1" might violate the invariant of "b2". If calls to this method are not controlled by instance "b2", then the violation would go unnoticed during verification, making the verification technique unsound.

A way of accommodating multi-object invariants for a verification technique is to additionally impose ownership on the type instances on which a multi-object invariant depends. For example, assume that we declare an invariant for instances of type "B" in Figure 1 (b) such that the invariant not only depends on fields of the current instance but also on the fields of a referred-to instance of type "D". Modular reasoning about such an invariant can be restored by making the instances of type "B" become the owner of the referred-to "D" instance. Given the ownership, instances of type "B" are guaranteed that any modifications of the referred-to "D" instance are solely triggered by themselves, giving them a chance to re-establish the invariant accordingly. Furthermore, thanks to the absence of transitive call-backs, instances of type "B" are guaranteed that they are not re-entered via a method invocation from their owned instances (as the instances of type "B" might be in an inconsistent state at that moment).

So far, we have shown how a verification technique can leverage type ordering to prevent transitive call-backs and selective ownership to control modifications of objects and what objects an invariant depends on. Since the ownership enforced by selective ownership it not transitive, a verification technique based on selective ownership can accommodate even invariants over heap topologies that subsume shared, modifiable sub-structures. For example, in Figure 1 (b) instances "d1" and "d2" share and may modify instance "g1", although "d1" and "d2" have different owners. In Section 3, we show further how selective ownership can develop its full power in Rumer, a programming language supporting first-class relationships. Rumer complements the notion of an object or relationship instance with the one of an extent instance, that is, collections of instances. This combination allows for a more modular program design and facilitates the verification of invariants even in the presence of shared, modifiable sub-structures.

3. Selective ownership in Rumer

This section discusses how selective ownership can be incorporated into a programming language. We use as an example Rumer [4, 5], a programming language we have designed to embody *first-class relationships* [1, 4, 6, 10, 33, 39, 42, 44] and for which we have developed a *visible-state verification technique based on selective ownership* [4]. The section starts with a brief introduction to

[2] In earlier work [5], we describe a visible-state verification technique that leverages the type order defined by relationship declarations as well as an encapsulation mechanism called member interposition for the modular verification of multi-object invariants.

```
1  entity Node {
2    string info; // element field
3
4    void setInfo(string info) // element method
5    { this.info = info; }
6  }
7
8  relationship Parent
9    participants (Node child, Node parent) {
10
11   // extent method
12   extent void link(Node c, Node p) {
13     these.add(new Parent(c, p));
14   }
15 }
```

Figure 2. Simple Rumer program modeling node hierarchies.

Rumer; we introduce Rumer only as far as necessary to follow the presentation of selective ownership in this paper. An in-depth introduction to Rumer as well as a discussion of related work on first-class relationships is given in [4]. Then, based on a running example, we explain how programmers define type ordering and ownership in Rumer and we sketch the Rumer verification technique based on the example.

3.1 Introduction to Rumer

Relationship-based programming languages extend object-oriented languages with the abstraction of a *relationship* to encapsulate the relationships that naturally arise between instances of classes. In those languages, relationships are first-class citizens: relationships can be instantiated as well as declare their own fields and methods. Early research on first-class relationships was motivated by the observation that programmers are poorly served when trying to implement relationships in object-oriented programming languages. As those languages lack first-class support for relationships, relationships must be represented in terms of reference fields and auxiliary classes. This indirection leads to a distribution of relationship code across several classes, making the resulting program prone to error.

Figure 2 displays a simple Rumer program. The program models hierarchies between nodes. As illustrated by Figure 2, Rumer supports two kinds of programmer-definable types: *entities* and *relationships*. In the example, the entity `Node` and the relationship `Parent` are declared. Entities are similar to classes and abstract objects. Relationships, on the other hand, abstract the relationships between instances. To indicate the types of instances that a relationship instance relates, a relationship declaration includes a participants clause. According to its participants clause on line 9, relationship `Parent` relates instances of the entity `Node`. To disambiguate the position an instance takes in a relationship, identifiers are assigned to the type declarations in a participants clause, indicating the role an instance of the type plays in the relationship. In the example, a `Parent` relationship instance relates a `child` node to its corresponding `parent` node.

Figure 3 (a) provides a schematic illustration of the run-time instances that the Rumer program displayed in Figure 2 may produce. It represents entity and relationship instances as dark gray circles and light gray ellipses, respectively, and connects relationship instances to their participant instances by lines, which are labeled with the role identifiers of the relationship's participants clause. For later reference, we mark entity and relationship instances with numbers and letters, respectively. In comparison to the object graph that would be produced by a corresponding class-based object-oriented implementation of the program shown in Figure 2, the run-time structure displayed in Figure 3 (a) differs in the existence of explicit relationship instances and in the absence of references in nodes. Since relationships are bidirectional, Rumer facilitates access to the participating instances at either side of a relationship instance. As a result, the declaration of references in objects is no longer a prerequisite for navigating the object graph. The different nature of objects in Rumer is also the reason why we use the term "entity" for the type abstracting objects rather than the term "class".

A further distinguishing feature of Rumer is the support of *extents*. An extent denotes a programmer-instantiable collection of instances. The term originates from the ODMG (Object Data Management Group) object model [16], where an extent of a type denotes the set of all instances of that type. Existing relationship-based programming languages maintain a single extent instance for each relationship declaration. Rumer builds on Nelson et al.'s suggestion to support multiple extent instances for first-class relationships [39] and provides extent instantiation not only for relationships but also for entities. For example, the following line of code instantiates an extent of type `Parent` and assigns the resulting extent instance to the variable `parents`:

```
Extent<Parent> parents = new Extent<Parent>();
```

As indicated by the argument type provided in angle brackets, the extent instance `parents` comprises `Parent` relationship instances. To distinguish the instances residing in an extent instance from the extent instance itself, we use the term *element instance* to refer to the former. Thus, the variable `parents` denotes a `Parent` extent instance that comprises `Parent` element instances.

Extent instances are explicitly populated and depopulated by programmers, and the Rumer type system guarantees that element instances always inhabit exactly one extent instance. To facilitate retrieval of element instances from an extent instance, Rumer supports a rich range of side-effect free queries in the spirit of LINQ (.NET Language-Integrated Query) [11, 12]. For example, given a `Node` element instance `p`, the following query returns all transitive children of `p`:

```
parents.tClosure().select(x: x.parent == p).child
```

The built-in query operator `tClosure()` builds the transitive closure of the relation described by the `parents` extent instance, and the built-in query operator `select()` reduces the resulting relation to the subset containing only `Parent` element instances that have `p` as an immediate parent or ancestor. The role projection operator `child` then projects the resulting subset of `Parent` element instances onto all `Node` element instances that participate as a `child` in those `Parent` element instances.

Extent instances in Rumer are not plain collections, but rather they can be equipped by the programmer with customized fields and methods. As a result, a Rumer type declaration may comprise field and method declarations both for element instances and extent instances of the type. Extent fields and methods are denoted by the `extent` keyword. For instance, entity `Node` declares the element field `info` and the element method `setInfo()` on line 2 and line 4 in Figure 2, respectively, and relationship `Parent` declares the extent method `link()` on line 11. The element method `setInfo()` sets the `info` field of the `Node` element instance that is the current receiver of the method. The keyword `this` refers to the current receiver instance of an element method. The extent method `link()` connects the argument `Node` element instance `c` as a `child` to the argument `Node` element instance `p`. The method invokes the built-in addition operator `add()` (line 13) on the `Parent` extent instance that is the current receiver of the method. The keyword `these` refers to the current receiver instance of an extent method. The choice of the keyword reflects the fact that an extent instance may contain several element instances. The ad-

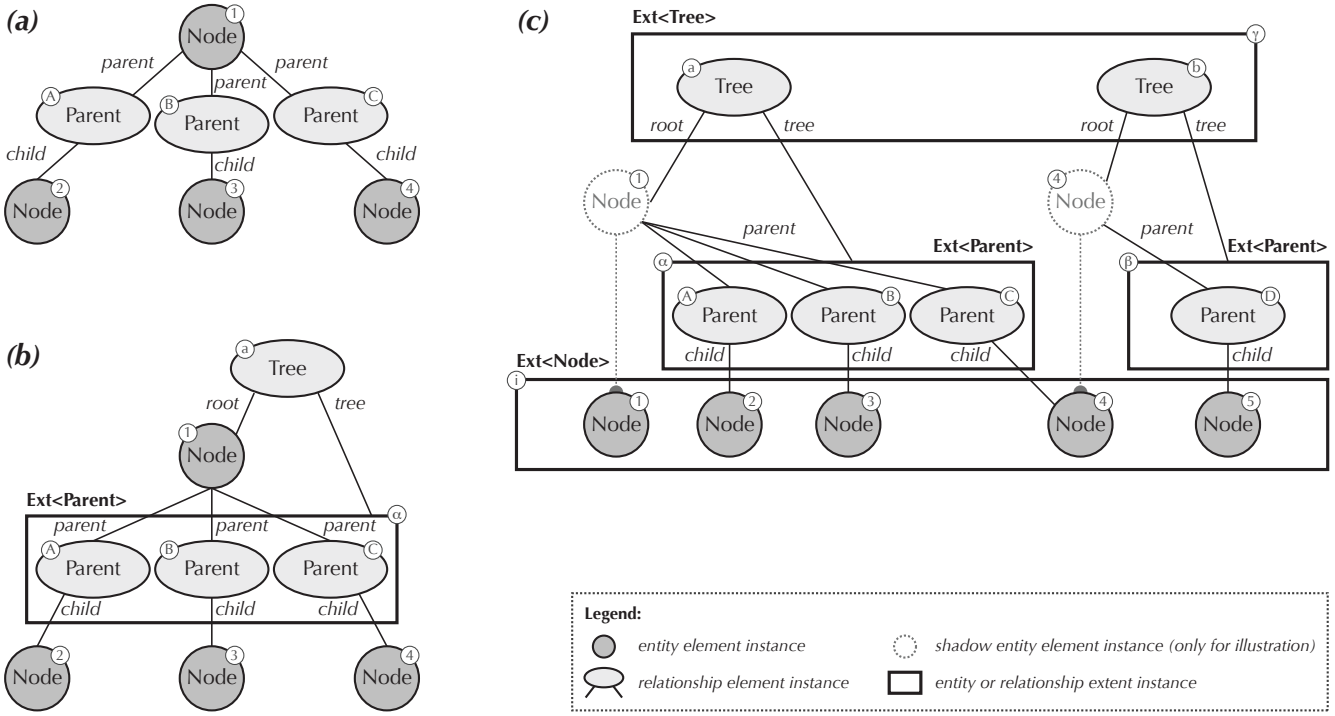

Figure 3. Schematic illustration of Rumer run-time instances: (a) several element instances of relationships declared in Figure 2; (b) one element instance instantiating relationship `Tree` declared in Figure 4; (c) complete heap for program declared in Figure 4 as well as its augmented version declared in Figure 6 that employs ownership.

dition operator instantiates a new `Parent` element instance that inhabits the current receiver extent instance.

3.2 Running example: tree

To illustrate the need for selective ownership, we extend the program shown in Figure 2 with a new relationship that allows us to model actual trees rather than node hierarchies. The new relationship `Tree` is shown in Figure 4. As indicated by its participants clause, a `Tree` element instance relates a `Node` element instance as its `root` to a `Parent` extent instance as its `tree`. In more abstract terms, a tree is thus represented by a tuple that has the root node of a tree as its left element and a relation describing the hierarchy between the tree's nodes as its right element. Whereas typical object-oriented implementations use the same abstraction to describe a tree as well as its children sub-trees, we chose to separate the two notions. As we will see (see Section 3.3 and Section 3.4), the chosen representation facilitates a concise formulation of the tree properties in terms of an invariant.

The chosen tree representation also becomes apparent in Figure 3 (b), which provides a schematic illustration of a `Tree` element instance. Figure 3 (b) uses the same graphical notations as Figure 3 (a) but complements them with extent instances. Extent instances are represented as rectangular boxes. Figure 3 (b) thus displays the `Parent` extent instance "α", which is the `tree` participant instance of the `Tree` element instance "a". The matching labels used in Figure 3 (a) and Figure 3 (b) further indicate that the `Parent` extent instance "α" displayed in Figure 3 (b) exactly subsumes the relationship element instances displayed in Figure 3 (a). To keep Figure 3 (b) simple, we have chosen a `Tree` element instance with only one layer. However, the relationship declaration in Figure 4 also supports multi-layered trees as well.

Figure 3 (c) finally shows the complete view of a run-time heap that may be produced by instantiating the declarations listed in Figure 2 and in Figure 4. This heap comprises the two `Tree` element instances "a" and "b". As indicated by the matching labels, the `Tree` element instance "a" exactly subsumes the run-time instances shown in Figure 3 (b). Since element instances are guaranteed to inhabit exactly one extent instance (see Section 3.1), Figure 3 (c) additionally displays the extent instances "i" and "γ" in which the `Node` element instances and `Tree` element instances, respectively, reside. To keep the graphical layout well-arranged, Figure 3 (c) makes use of "shadow" `Node` element instances. Those shadows are purely graphical copies of the instance they are connected to by a dotted line. For example, the `Node` element instance "4" is part of both displayed `Tree` element instances: it is a leaf node of the `Tree` element instance "a" and the root node of the `Tree` element instance "b". `Node` element instance "4" also nicely motivates the need for selective ownership as it represents a run-time instance further down in the heap topology that is shared among and modified by both trees.

Relationship `Tree` declares various methods for tree manipulations, such as appending one tree to another one. In the following, we briefly explain the implementations of these methods for the interested reader:

The extent method `createTree()` (line 4) instantiates a new `Tree` element instance that inhabits the current receiver extent instance referred to by `these`. In terms of Figure 3 (c), `these` denotes the `Tree` extent instance "γ". The newly created `Tree` element instance relates the argument `Node` element instance `r` as a `root` to an empty, newly created `Parent` extent instances as a `tree`.

The element method `appendTree()` (line 7) appends the `Tree` element instance `t` to the current receiver `Tree` element

```
1  relationship Tree participants (Node root, Extent<Parent> tree) {
2
3    // Instantiates a Tree element instance with 'root' r and a new empty 'tree' that inhabits these.
4    extent void createTree(Node r)
5    { these.add(new Tree(r, new Extent<Parent>())); }
6
7    void appendTree(Tree t, Node p) // Appends tree t to this as 'child' of p.
8    { this.appendNode(t.root, p); this.appendSubTree(t.tree, t.root); }
9
10   void appendSubTree(query Set<Parent> c, Node p) // Appends sub-tree c to this as 'child' of p.
11   { foreach (cp isElementOf c.select(x: x.parent == p)) {
12       this.appendNode(cp.child, cp.parent);
13       this.appendSubTree(c.select(x: x.child isElementOf
14        c.tClosure().select(y: y.parent == cp.child).child), cp.child); }}
15
16   void appendNode(Node c, Node p) // Appends node c to this as 'child' of p.
17   { this.tree.link(c, p); }
18 }
```

Figure 4. Relationship `Tree`. Extends program in Figure 2 to model trees.

instance as a `child` of the `Node` element instance `p`. Method `appendTree()` relies on the element methods `appendNode()` and `appendSubTree()`.

The element method `appendNode()` (line 16) appends the `Node` element instance `c` to the current receiver `Tree` element instance as a `child` of the `Node` element instance `p`. It relies on `Parent`'s extent method `link()`, which it invokes on the current receiver instance's `tree Parent` extent instance.

The element method `appendSubTree()` (line 10), finally, appends the sub-tree denoted by the set of `Parent` element instances `c` to the current receiver `Tree` element instance as a `child` of the `Node` element instance `p`. The method is implemented recursively to append the sub-tree in a depth-first traversal order. In each recursive step (line 12), one `Node` element instance of the sub-tree (denoted by `cp.child`) is appended to its corresponding `parent Node` element instance in the current receiver instance's `tree Parent` extent instance (denoted by `cp.parent`). Recursion stops whenever the sub-tree `c` denotes the empty set. This is the case as soon as a leaf node has been inserted in the preceding recursive invocation.

3.3 Declaration of type ordering

A glimpse at Figure 3 (c) reveals that first-class relationships naturally give rise to an ordering on type declarations. Building on this observation, Rumer allows programmers to structure a program's heap by declaring relationships. The ordering on entity and relationships, in particular, is defined by a relationship's participants clause. In terms of Figure 1 (a), a relationship's participants clause defines two type order arrows such that each arrow points from the relationship to one of the relationship's participant types. For the running example defined in Figure 2 and Figure 4, the transitive reduction of the resulting type order thus is:

$$\{\mathit{Tree} \mapsto \mathit{Parent}, \mathit{Tree} \mapsto \mathit{Node}, \mathit{Parent} \mapsto \mathit{Node}\}$$

To meet the criterion of a type order (see Section 2.1), the transitive closure of the above relation must form a strict partial order. This requirement, in particular, expects relationship declarations to be acyclic. It is noteworthy that this requirement does not prevent the implementation of recursive data structures in Rumer since the "links" between the structure's data are represented by relationship instances, as opposed to data references.

In earlier work [4, 5], we introduced a visible-state verification technique that leverages the type order defined by relationship declarations. As outlined in Section 2.2, our technique prevents transitive call-backs by requiring method invocations to either propagate in the direction of a program's relationship declarations or to dispatch on the current receiver instance. The built-in query operators in Rumer are not subject to this restriction as they are side-effect free.

To illustrate what kinds of invariants can be accommodated using the order prescribed by relationship declarations, we introduce an invariant to the running example. In particular, we declare an invariant for extent instances of relationship `Parent` to guarantee that a `Parent` extent instance indeed describes a hierarchy of nodes:

```
extent invariant // Parent's extent invariant
  these.isPartialFunction() &
  these.tClosure().isIrreflexive();
```

In addition to the built-in query operator `tClosure()` encountered earlier, the extent invariant uses the built-in query operators `isPartialFunction()` and `isIrreflexive()`. These operators evaluate to true if the set of relationship element instances on which the operator is invoked forms a partial function and irreflexive relation, respectively. The invariant thus guarantees that a `Parent` extent instance forms a forest of trees.

The extent invariant of `Parent` depends only on the state of the extent instance, but not on the states of the `Node` element instances since the links between the nodes are expressed as a relationship (as opposed to references stored in the `Node` element instances). Consequently, we can maintain the invariant without requiring the extent or the `Tree` relationship to own the nodes. The absence of ownership allows arbitrary instances to refer to and to modify the nodes - a setup that is not permitted in existing ownership-based verification techniques. Since Rumer enforces classical encapsulation (see Section 2.2) for its instances, the extent instance is the only instance that can write to its content. To maintain `Parent`'s extent invariant, our verification technique thus imposes the proof obligation on any of `Parent`'s extent methods to establish the current receiver's invariant in the final state of the method as well as before any invocations on the current receiver. An in-depth discussion of our verification technique, including a complete overview of its proof obligations as well as its soundness proof, can be found in [4].

3.4 Declaration of ownership

Thanks to relationship `Parent`'s extent invariant, relationship `Tree` is guaranteed that its element instances' `tree Parent` extent instances describe node hierarchies. However, as exempli-

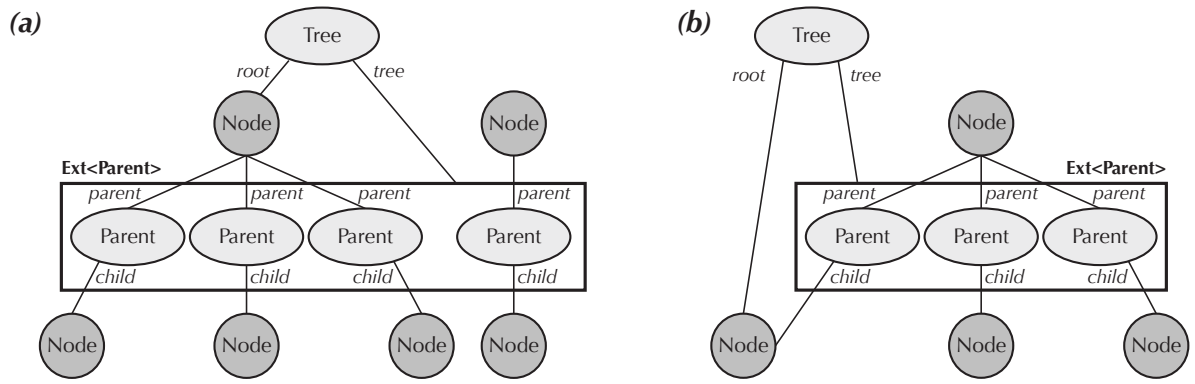

Figure 5. Possible instantiations of program declared in Figure 4. Both instantiations satisfy `Parent`'s extent invariant introduced in Section 3.3 but do not form proper trees: (a) the `Tree` element instance's `tree Parent` extent instance represents a forest of trees, (b) the `Tree` element instance's `root Node` element instance is not the topmost `Node` element instances of its `tree Parent` extent instance.

fied by Figure 5 (a) and Figure 5 (b), this invariant is not sufficient to guarantee that `Tree` element instances indeed form proper trees. For example, Figure 5 (a) displays a `Tree` element instance whose `tree Parent` extent instance represents a forest of trees, and Figure 5 (b) displays a `Tree` element instance whose `root Node` element instance is not the topmost `Node` element instance of its `tree Parent` extent instance.

To guarantee that `Tree` element instances actually form proper trees, we need to impose an appropriate invariant on element instances of relationship `Tree`. Intuitively, this invariant shall make sure, for any `Tree` element instance *t*, that *t*'s `root Node` element instance is the same as the topmost `Node` element instance in *t*'s `tree Parent` extent instance and, also, that there exists a topmost `Node` element instance in *t*'s `tree Parent` extent instance. This invariant can be expressed in Rumer as an element invariant on relationship `Tree` as follows:

```
invariant // Tree's element invariant
  !(this.root isElementOf this.tree.child) &
  (!this.tree.isEmpty() => this.root isElementOf
    this.tree.parent) &
  this.tree.tClosure().select(cp: cp.parent ==
    this.root).child == this.tree.child;
```

The first conjunct of the element invariant requires that a `Tree` element instance's `root Node` element instance never appears as a `child` in the relation described by the `Tree` element instance's `tree Parent` extent instance. The second conjunct requires that a `Tree` element instance's `root Node` element instance appears as a `parent` in the relation described by the `Tree` element instance's `tree Parent` extent instance, unless this relation is empty. The first and second conjunct thus rule out instantiations such as the one shown in Figure 5 (b). The third conjunct requires that a `Tree` element instance's `root Node` element instance is the transitive `parent` of all children nodes of the relation described by the `Tree` element instance's `tree Parent` extent instance. The third conjunct thus rules out instantiations such as the one shown in Figure 5 (a).

As opposed to `Parent`'s extent invariant, this invariant does not only depend on the state of the invariant's `Tree` element instance but also on the state of its instance's `tree Parent` extent instance. More specifically, the invariant depends on the content of its instance's `tree Parent` extent instance and can thus be violated by the addition or removal of any `Parent` element instances to or from the instance's `tree Parent` extent instance. As a result, the element invariant of relationship `Tree` cannot be accommodated by our verification technique solely by leveraging type ordering.

However, our verification technique can accommodate the element invariant of relationship `Tree` by superimposing the type order prescribed by relationship declarations with ownership. More specifically, we must make a `Tree` element instance the owner of its `tree Parent` extent instance. In terms of Figure 3 (c), the ownership will guarantee that the `Tree` element instances "a" and "b" are the owners of their `tree Parent` extent instances "α" and "β", respectively.

Figure 6 shows the result of superimposing ownership on the relationship `Tree` declared in Figure 4; the differences between the two versions are highlighted in Figure 6. The declarations of entity `Node` and relationship `Parent` are unaffected by the ownership declaration and are shown in Figure 2. As indicated by the participants clause of the new version of relationship `Tree`, we use type *modifiers* to express ownership of an instance relative to a current receiver instance. We distinguish the following selective ownership modifiers:

- **owned:** referred-to instance has the current receiver instance as its owner;
- **shared** (default modifier): referred-to instance does not have an owner;
- **readonly:** referred-to instance may or may not have an owner.

By annotating its `tree` participant type with the ownership modifier `owned`, relationship `Tree` guarantees that its element instances become the unique owner of their `tree Parent` extent instances. If a type declaration omits the ownership modifier, the default modifier `shared` is assumed. The use of type modifiers results in an ownership type system that is similarly lightweight as the Universe type system [20–23, 37]. However, unlike the ownership enforced by Universe types, the ownership enforced by selective ownership is not transitive. For example, the new version of relationship `Tree` in Figure 6 does neither affect the `Parent` element instances within an owned `Parent` extent instance nor the `Node` element instances related by those `Parent` element instances. As a result, the program heap depicted in Figure 3 (c) still amounts to a valid heap that can be produced by the new relationship declaration. The two `Tree` element instances "a" and "b" displayed in Figure 3 (c), in particular, are allowed to share and modify the `Node` element instance "4" while keeping their `tree Parent` extent instances "α" and "β", respectively, separate.

To guarantee that the two ways of structuring a program's heap offered by selective ownership — type order and instance ownership — nicely complement each other, the ownership relation must

```
1  // A Tree element instance owns its 'tree' Parent extent instance.
2  relationship Tree participants (Node root, owned Extent<Parent> tree) {
3
4    extent void createTree(Node r)
5      // New Tree element instance becomes owner of new Parent extent instance.
6    { these.add(new Tree(r, new owned Extent<Parent>())); }
7
8    void appendTree(Tree t, Node p)
9    { this.appendNode(t.root, p); this.appendSubTree(t.tree, t.root); }
10
11   void appendSubTree(query Set<Parent> c, Node p)
12   { foreach (cp isElementOf c.select(x: x.parent == p)) {
13       this.appendNode(cp.child, cp.parent);
14       this.appendSubTree(c.select(x: x.child isElementOf
15        c.tClosure().select(y: y.parent == cp.child).child), cp.child); }}
16
17   void appendNode(Node c, Node p)
18   { this.tree.link(c, p); } // Invocation admissible: this is owner of 'tree' Parent extent instance.
19 }
```

Figure 6. Relationship `Tree` augmented with ownership. Differences to version without ownership (see Figure 4) are highlighted.

be a subset of the type order. In Rumer, this requirement is met by allowing only relationships to impose ownership on instances of their participant types. Given this restriction, owners are guaranteed not to be re-entered (in a possibly inconsistent state) via method invocations from their owned instances.

Selective ownership modifiers constrain method invocations and thus strengthen the restrictions imposed on method invocations by the type order. In particular, selective ownership allows only owners to invoke methods on owned instances. For example, the invocation of method `link()` on the `Parent` extent instance referred-to by `this.tree` on line 18 in Figure 6 is admissible because the current receiver `Tree` element instance is the owner of its `tree Parent` extent instance. This regime guarantees that modifications of an owned instance's fields are initiated by the instance's owner, giving the owner a chance to re-establish the invariant upon modifications of the owned instance. Method invocations on shared instances, on the other hand, are not constrained by selective ownership. Furthermore, selective ownership permits the reading of fields of owned instances and the invocation of built-in query operators on owned instances. For example, the access `t.tree` on line 9 in Figure 6 is admissible because it is a read access. The selective ownership modifier `readonly`, lastly, forbids any method invocations on the referred-to instance but permits read accesses or invocations of built-in query operators.

4. Discussion

As exemplified by the program heap shown in Figure 3 (c), selective ownership enables a *hybrid* ownership scheme in which owned and shared instances coexist. To permit such a scheme, type declarations must be formulated so as to allow possible callers to obtain either owned or shared instances of the type. As a result, only the caller-site knows about the ownership of an instance and can thus guard the instance against prohibited modifications or accidental leaking by establishing appropriate selective ownership modifiers. The callee-site, on the other hand, does not know whether a particular instance is owned or shared and may thus perceive an owned instance as "supposedly shared". For example, relationship `Parent` in Figure 2 assumes the default selective ownership modifier `shared` for its extent instances and may thus accidentally leak a `Tree` element instance's `tree Parent` extent instance.

In [4] we establish appropriate well-formedness conditions on a Rumer program to prevent the accidental leaking of "supposedly shared" instances. In particular, we prove that, given those well-formedness conditions, "supposedly shared" instances cannot outlive the (possibly transitive) method executions within which they are produced. Furthermore, we prove a lemma that captures the effects of selective ownership on a program's call stack. The lemma is stated from the perspective of an owned instance and guarantees that any owned instance residing on the call stack is preceded by its owner. As a result, the lemma guarantees that any modifications of an owned instance's field are initiated by the instance's owner. However, due to the hybrid nature of the underlying ownership system, this property does not hold for all the instances in a program's heap, but only for those instances in a program's heap that are owned. To contrast the discipline emerging from selective ownership with the "owner-as-modifier" discipline of the Universe type system [22, 24], we refer to our discipline as the "owned-called-by-owner" discipline.

The currently enforced well-formedness conditions to prevent the accidental leaking of "supposedly shared" instances restrict the type of fields entities and relationships can declare. In particular, they prevent an entity or relationship from defining fields that "point to" type instances of which the entity or relationship is a (possibly transitive) participant. As part of future work, we would like to investigate less restrictive mechanisms to prevent the accidental leaking of "supposedly shared" instances.

5. Related work

In this section, we focus on related work on Ownership type systems and, in particular, on ownership-based verification techniques. An extensive discussion of related work on relationship-based programming languages is given in [4].

Work on "traditional" forms of ownership can broadly be categorized into work on ownership types [17–19, 41] and work on Universe types [20–23, 37]. In both ownership schemes, all objects are owned and have exactly one owning object. The two schemes, however, differ in their applied encapsulation discipline. Whereas ownership types typically enforce the *owner-as-dominator* discipline, Universe types typically enforce the *owner-as-modifier* discipline [22, 24]. The owner-as-dominator discipline requires all reference chains to an object to pass through the object's owner. The owner-as-modifier discipline, on the other hand, enforces a less stringent alias restriction and requires only modifications of an object to be initiated by the object's owner.

Whereas ownership types allow owned objects to establish back-references to their owners, Universe types permit such ref-

erences only if they are read-only. This restriction makes the Universe type system attractive for program verification since it forbids call-backs into owners. The amenability of Universe types and similar systems for program verification has been shown in [30, 37, 38]. The benefits of ownership for program verification have also been demonstrated in the context of Oval [34], a variant of an ownership-type-based language. As opposed to other ownership type systems [17–19, 41], Oval's types system enforces an owner-as-modifier discipline and employs effect annotations to deal with call-backs.

The presented verification technique is most closely related to the Universe-type-based verification technique [37, 38] since selective ownership is similarly lightweight as Universe types thanks to the use of ownership modifiers. However, selective ownership allows for unrestricted sharing of instances further "down" in the heap topology. This relaxation is due to the fact that selective ownership allows heap structure to be enforced either by type order alone or by type order combined with instance ownership. Selective ownership gives furthermore rise to a hybrid ownership scheme in which owned and shared instances coexist.

Our extent invariants are related to visibility-based invariants [30, 38] since they support certain multi-object invariants without requiring ownership. Extent invariants lead to simple proof obligations for extent methods, whereas visibility-based invariants require proof obligations that quantify over all objects that are possibly affected by a field update.

Leino et al. [31] use a programmer-declared type ordering to control class initialization and to verify static class invariants. In our system, the type ordering is used to prevent transitive call-backs. Instead of requiring explicit declarations, we infer the type ordering from the relationship declarations in a Rumer program.

Cameron et al. [15] propose a type system that supports multiple ownership. It enforces a DAG topology on the heap and, thus, permits several objects to own and modify owned objects. So far there is no verification technique that handles this expressiveness. In our approach, certain invariants can be stated as extent invariants rather than object invariants. Therefore, we can verify such invariants without requiring ownership.

In ownership domains [3], the objects owned by one owner can be grouped into several domains. A domain can be declared public; each client that may access an owner object may also own the objects in its public domain. For instance, a linked list structure may put all list nodes in a non-public domain and the list iterators in another, public domain. Each client with access to the list may then access its iterators, which in turn can be permitted to have access to the nodes. Therefore, ownership domains permit sharing of owned objects. Modular verification of invariants over such structures is difficult since owners do not have full control over the owned objects. For instance, it is unclear how to maintain the list's invariant when the nodes are modified via an iterator. In our approach, we can formulate structural properties of the list as an extent invariant and are thus able to verify such invariants without imposing ownership on the nodes.

6. Conclusions

This paper introduces selective ownership, a flexible "mix and match" approach to giving structure to a program's heap in two ways: by defining an ordering relation on a program's type declarations and by imposing ownership on selected instances. We evaluate selective ownership for the purpose of program verification. As compared to existing ownership-based verification techniques, the heap topology enforced by selective ownership does not amount to a tree, but to a DAG with partial sub-trees. This scheme permits the modular verification of ownership-based invariants without restricting access to instances further "down" in the heap topology.

Although selective ownership is not tied to a particular host language, it seems to develop its full power in a language that supports first-class relationships. We illustrate selective ownership in the context of Rumer, for which we have developed selective ownership. First-class relationships in Rumer give naturally rise to an ordering on type declarations. Furthermore, the availability of an elaborate set of type abstractions — entity versus relationship as well as element instance versus extent instance — allows for a practical modularization of programs that facilitates the expression of invariants over heap topologies that subsume shared, modifiable sub-structures.

Acknowledgments

We are grateful to Jonathan Aldrich and the anonymous reviewers for their valuable feedback on this paper. We also thank Sophia Drossopoulou, Alexander J. Summers, and James Noble for stimulating discussions on selective ownership. This research was supported in part by the Swiss National Science Foundation through grant PBEZP2_140051 and in part by the Army Research Office under Contract W911NF-09-1-0273.

References

[1] A. Albano, G. Ghelli, and R. Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *17th International Conference on Very Large Data Bases (VLDB'91)*, pages 565–575. Morgan Kaufmann Publishers Inc., 1991.

[2] J. Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, 2003.

[3] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *18th European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2004.

[4] S. Balzer. *Rumer: a Programming Language and Modular Verification Technique Based on Relationships*. PhD thesis, 19851, ETH Zurich, 2011.

[5] S. Balzer and T. R. Gross. Verifying multi-object invariants with relationships. In *25th European Conference on Object-Oriented Programming (ECOOP'11)*, volume 6813 of *Lecture Notes in Computer Science*, pages 358–382. Springer, 2011.

[6] S. Balzer, T. R. Gross, and P. Eugster. A relational model of object collaborations and its use in reasoning about relationships. In *21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 323–346. Springer, 2007.

[7] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *7th International Conference on Mathematics of Program Construction (MPC'04)*, Lecture Notes in Computer Science, pages 54–84. Springer, 2004.

[8] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6):27–56, 2004.

[9] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec$^\sharp$ programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[10] G. M. Bierman and A. Wren. First-class relationships in an object-oriented language. In *19th European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 262–286. Springer, 2005.

[11] G. M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: Formalizing proposed extensions to Spec$^\sharp$. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 479–498. ACM, 2007.

[12] D. Box and A. Hejlsberg. LINQ: .NET Language-Integrated Query. http://msdn2.microsoft.com/en-us/library/bb308959.aspx, February 2007.

[13] C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *17th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 211–230, New York, NY, USA, 2002. ACM.

[14] C. Boyapati, A. Salcianu, W. S. Beebee, and M. C. Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, pages 324–337. ACM, 2003.

[15] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 441–460. ACM, 2007.

[16] R. G. Cattell and D. K. Barry. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[17] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, October 2002.

[18] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *17th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 292–310. ACM, 2002.

[19] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *13th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 48–64. ACM, 1998.

[20] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. Universe Types for topology and encapsulation. In *6th International Symposium of Formal Methods for Components and Objects (FMCO'07)*, volume 5382 of *Lecture Notes in Computer Science*, pages 72–112. Springer, 2007.

[21] W. Dietl. *Universe Types Topology, Encapsulation, Genericity, and Tools*. PhD thesis, ETH Zurich, 2009. 18522.

[22] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, 2005.

[23] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer, 2007.

[24] W. Dietl, S. Drossopoulou, and P. Müller. Separating ownership topology and encapsulation with generic universe types. *ACM Trans. Program. Lang. Syst.*, 33:20:1–20:62, 2011.

[25] S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers. A unified framework for verification techniques for object invariants. In *22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *Lecture Notes in Computer Science*, pages 412–437. Springer, 2008.

[26] C. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[27] K. Huizing and R. Kuiper. Verification of object oriented programs using class invariants. In *3rd International Conference on Fundamental Approaches to Software Engineering (FASE'00)*, volume 1783 of *Lecture Notes in Computer Science*, pages 208–221. Springer, 2000.

[28] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev29, Iowa State University, 2006.

[29] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[30] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *18th European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer, 2004.

[31] K. R. M. Leino and P. Müller. Modular verification of static class invariants. In *International Symposium of Formal Methods Europe (FM'05)*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2005.

[32] K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science, pages 80–94. Springer, 2007.

[33] Y. D. Liu and S. F. Smith. Interaction-based programming with Classages. In *20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'05)*, pages 191–209. ACM, 2005.

[34] Y. Lu, J. Potter, and J. Xue. Validity invariants and effects. In *21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 202–226. Springer, 2007.

[35] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall Professional Technical Reference, 1997.

[36] R. Middelkoop, C. Huizing, R. Kuiper, and E. J. Luit. Invariants for non-hierarchical object structures. *Electronic Notes in Theoretical Computer Science*, 195:211–229, 2008.

[37] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.

[38] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62 (3):253–286, 2006.

[39] S. Nelson, D. J. Pearce, and J. Noble. First class relationships for OO languages. In *6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL'08)*, 2008.

[40] A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. Habilitation thesis, Technical University of Munich, January 1997.

[41] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *21th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'06)*, pages 311–324. ACM, 2006.

[42] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 466–481. ACM, 1987.

[43] A. J. Summers and S. Drossopoulou. Considerate reasoning and the Composite design pattern. In *11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2010)*, volume 5944 of *Lecture Notes in Computer Science*, pages 328–344. Springer, 2010.

[44] A. Wren. *Relationships for Object-oriented Programming Languages*. PhD thesis, University of Cambridge, November 2007.