Proceedings of the nth International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming (IWACO)

In conjunction with ECOOP 2007, Berlin, July 30th–August 3rd

Tobias Wrigstad (editor)

DSV Report series No. 07-010
ISSN 1101-8526
ISRN SU-KTH/DSV/R–07/10–SE

Organisers

Dave Clarke	(CWI)
Sophia Drossopoulou	(Imperial College)
James Noble	(Victoria University of Wellington)
Tobias Wrigstad	(Stockholm University)

Program Committee

Jonathan Aldrich	(Carnegie Mellon University)
Chandrasekhar Boyapati	(University of Michigan)
Dave Clarke	(CWI)
Sophia Drossopoulou	(Imperial College)
Rustan Leino	(Microsoft Research)
Peter Müller	(ETH Zurich)
James Noble	(Victoria University of Wellington)
Peter O'Hearn	(Queen Mary, University of London)
Alex Potanin	(Victoria University of Wellington)
Jan Vitek	(Purdue University)
Tobias Wrigstad (chair)	(Stockholm University)

Additional Reviewers

Werner Dietl, Johan Östlund, and Arsenii Rudich

Contents

Schedule

09:00–10:00	Invited talk by Vijay Saraswat
10:00–10:30	Session 0
	Primitive Associations (5 min) Erik Ernst
	Maintaining Invariants Through Object Coupling Mechanisms (5 min) Eric Kerfoot and Steve McKeever
	Class Invariants: The end of the road? (10 min) Matthew Parkinson
10:30–11:00	Coffee break
11:00–12:30	Session 1
	Annotations for (more) Precise Points-to Analysis (20 min) Mike Barnett, Manuel Fähndrich, Diego Garbervetsky and Francesco Logozzo
	Ownership, Uniqueness and Immutability (20 min) Johan Östlund, Tobias Wrigstad, Dave Clarke and Beatrice Åkerblom
	Ownership Meets Java (10 min) Christo Fogelberg, Alex Potanin and James Noble
	2007 State of the Universe Address (10 min) Werner Dietl and Peter Müller
12:30–14:00	Lunch
14:00–15:30	Session 2
	Iterators can be Independent "from" Their Collections (20 min) John Boyland, William Retert and Yang Zhao
	Simple and Flexible Stack Types (20 min) Frances Perry, Chris Hawblitzel and Juan Chen
	See the Pet in the Beast: How to Limit Effects of Aliasing (10 min) Franz Puntigam
	Using ownership types to support library aliasing boundaries (10 min) Luke Wagner, Jaakko Järvi and Bjarne Stroustrup
15:30–16:00	Coffee break

16:00–17:30	Session 3
	Runtime Universe Type Inference (20 min) Werner Dietl and Peter Müller
	Compile-Time Views of Execution Structure Based on Ownership (20 min) Marwan Abi-Antoun and Jonathan Aldrich
	Ownership Domains in the Real World (20 min) Marwan Abi-Antoun and Jonathan Aldrich
	Formalizing Composite State Encapsulation (5 min) Adrian Fiech and Ulf Schuenemann

Primitive Associations

Erik Ernst
University of Aarhus, Denmark
eernst@daimi.au.dk

Abstract

This position paper presents a very simple mechanism, *primitive associations*, and argues that this mechanism is worth careful consideration in connection with the kind of support for program correctness that grows out of mechanisms for ownership, controlled aliasing, sharing, escape analysis, and so on.

Categories and Subject Descriptors D.3 - PROGRAMMING LANGUAGES [*D.3.3 - Language Constructs and Features*]: Data types and structures

Keywords Ownership, confinement, alias control, primitive associations, inverse fields, path-restricted features.

1. Primitive Associations

Almost all object-oriented programming languages support a notion of references. A reference provides access to a specific object, and type systems are often mainly focused on specifying which (kinds of) objects are reachable from a given object. However, normally only little is known about the set of references referring *to* a given object—which we will designate as *incoming references*. Linear types [13], ownership types [9, 4, 3, 1, 11], escape analysis [10, 2], and other kinds of mechanisms and analyses help in establishing invariants or knowledge about these incoming references, and this may simplify reasoning about program correctness, especially because the sources of changes to objects and object graphs are simpler. However, we believe that it is useful to complement these techniques with a dynamic mechanism, namely *primitive associations*, because it is useful, simple, flexible, and understandable.

We define primitive associations to mean bidirectional references, i.e., a pair of references in two objects that refer to each other, see Fig. 1. Changes to these references must be restricted by the language semantics to enforce this invariant at all times: if A is an object and A`.f` is a field in A that is part of a primitive association, then either A`.f` is `NULL` or it refers to an object B such that B has a field B`.g` which is the other half of that primitive association, and B`.g` refers to the object A. Hence, the language must support statically decidable pairing of fields, and the run-time manipulation of fields which take part in a primitive association must occur atomically.

Given that the language semantics enforces this invariant, it is known for any given object A having a primitive association to another object B that no other object B' (respectively A') is in the same relation to A (resp. B). This may be interpreted as an ownership relation—that A owns B, or vice versa.

However, this ownership relation differs from more traditional ownerships by being more dynamic, because it may be changed by assignment. Other ownership related mechanisms would specify an owner via type declarations or type arguments and fix it at creation time for each owned object, thus disallowing the change of owner during the life-time of the owned object.

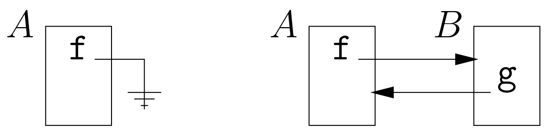


Figure 1. A primitive association is either `NULL` or cyclic

On the other hand, the dynamic flexibility of ownership by primitive associations provides fewer guaranteed properties at runtime. E.g., an inconsistency arises if the primitive association is modified during the execution of some operation which is only permitted for owners.

Primitive associations are closely related to *parent-child attributes* or *inverse fields* in JavaFX [12], because they also involve bidirectional references with language support for simultaneous updating. However, in this context we are interested in the ability to help managing uniqueness relationships rather than maintaining problem domain related constraints.

Note that it is easy to build associations of different arity than 1–1 based on primitive associations; for example, an array of length k may be used as an intermediate object to model a 1–k association.

2. Derived Correctness Properties

The main idea behind ownership is that it is easier to reason about the correctness of a program when ownership related invariants can be used to show that other invariants are maintained. The ownership related invariants are generally concerned with the exclusion of a (large) class of possible incoming pointers.

Consider for instance a `List` data structure which uses a number of `ListCell` objects to represent a linking structure and keep a reference to each of the contained objects. Now, invariants about the structure of each `List` object, including its `ListCell`s, is easier to reason about if each `ListCell` is owned by one particular `List` object, and access to list cells is thereby restricted to come from the owner list or the list cells themselves. Conventional ownership mechanisms are well suited for this type of purpose; they associate each owned object (e.g., each `ListCell`) with an owner (a `List`) at creation time, and never change this binding during the lifetime of the owned object.

However, it is not always convenient to bind each owned object to one particular owner for its entire life-time. For example, it may be useful to move owned objects from one “owning context” to another. The main benefit of using primitive associations for ownership management is exactly this dynamic flexibility of being able to change owner during the lifetime of the owned object.

This property, however, creates challenges for exploiting ownership, i.e., to derive other correctness properties, because it gets harder to maintain a complex invariant that expresses a structural relation in the object graph of owned and owning objects when an assignment to a primitive association may suddenly change the owner. However, for the simple relationship that only involves the two objects directly connected by a primitive association, there is a potential for reconciling these to opposing forces.

The concept required to express this is that of a *path-restricted feature*, i.e., a feature of an object that is only accessible via a specified path. Consider the pseudo-code example in Box 1 below:

```
class Person {
  private Wallet wlt <-> owner; // pr.ass.
  int pay(int value) {
    if (wlt.has(value)) {
      wlt.take(value); return value;
    } else {
      // error handling
    }
  }
}
class Wallet {
  private Person owner <-> wlt; // pr.ass.
  private int contents;
  public bool has(int value) {
    return (contents>=value);
  }
  restricted(wlt) void take(int value) {
    contents -= value;
  }
}
```

Box 1

In this example, the instances of the classes `Person` and `Wallet` are connected by a primitive association whose ends are named `wlt` and `owner`. In class `Wallet` there is a method `take` which is path-restricted by `wlt`. This means that an invocation of `take` is only allowed if it is on the form `wlt.take(...)` where `wlt` is the opposite end of a primitive association that connects a `Person` and this `Wallet`. The effect is that only the `owner` is allowed to call this method. Note that this differs from traditional ownership in that the person may choose to transfer the wallet to some other person.

3. Integrating Primitive Associations into gbeta

Primitive associations and the corresponding mechanism of path-restricted features are currently being implemented in the language gbeta [5, 8], where they complement a more traditional notion of ownership which is expressed using family polymorphism [6] and invisible mixins [7].

Family polymorphism includes a restricted form of dependent types: Classes are features of objects and thus two nested classes `Outer` and `Inner` give rise to a unbounded set of distinct types at runtime, because each instance of `Outer` contains its own, distinct class corresponding to the declaration named `Inner`. An invisible mixin is a mixin which is guaranteed to have a zero effect on the type of any class that it is added to—in other words, an invisible mixin can only add implementation, not interface. A consequence of this is that no code outside the mixin can refer to its declared features. Note that the notion of invisible mixins is in fact built on the notion of path restriction, because most of the characteristics of an invisible mixin are specified in terms of restrictions on paths.

Putting the two together, traditional ownership can be expressed by declaring owned classes in an invisible mixin. This is now complemented with the ability for owned object structures to include temporary ownership based on primitive associations and path restrictions.

It is our impression so far that this combination of life-time ownership and temporary ownership makes it easier to express practical program designs and still have a better basis for reasoning about the possible run-time object structures than that which is offered through traditional ownership or traditional unrestricted (un-owned) objects.

4. Conclusion

This position paper presented some preliminary thoughts about the usefulness of the very simple construct of primitive associations (aka inverse fields), used to express a dynamic kind of ownership. The notion of path-restricted features was created as a consequence of this analysis, as a special case of earlier work on so-called invisible mixins. We believe that this combination of mechanisms provides a simple and useful complement to traditional ownership mechanisms.

Acknowledgments

The IWACO reviewers provided some very helpful comments on this work.

References

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *Proceedings ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2004. ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings.

[2] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings POPL '98*, pages 25–37. ACM SIGACT and SIGPLAN, ACM Press, 1998.

[3] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02)*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 292–310, New York, November 4–8 2002. ACM Press.

[4] David Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering; University; of New South Wales, Australia, July 12 2001.

[5] Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, DEVISE, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.

[6] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP'01*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[7] Erik Ernst. Reconciling virtual classes with genericity. In *Proceedings JMLC'06*, LNCS 4228, pages 57–72, Oxford, UK, September 2006. Springer-Verlag.

[8] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proceedings POPL'06*, pages 270–282, Charleston, SC, USA, 2006. ACM.

[9] James Noble, John Potter, David Holmes, and Jan Vitek. Flexible alias protection. In *Proceedings of ECOOP'98*, Brussels, Belgium, July 20 - 24 1998.

[10] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–127, 1992.

[11] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic java. In Peri L. Tarr and William R. Cook, editors, *Proceedings OOPSLA*, pages 311–324. ACM, 2006.

[12] Inc. Sun Microsystems. Javafx script – an overview. `http://www.sun.com/software/javafx/script/`, July 2007.

[13] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

Maintaining Invariants Through Object Coupling Mechanisms

Eric Kerfoot Steve McKeever
Oxford University Computing Laboratory
{eric.kerfoot, steve.mckeever}@comlab.ox.ac.uk

Abstract

Object invariants are critical components to the specification of object-oriented systems, which define valid states for objects and how they may be interrelated. A complex problem is created when an invariant relies on objects that are externally aliased and modified, since the invariant's class cannot ensure that modification to these objects preserves the invariant. This paper informally introduces a method of coupling objects called the Colleague Technique, which creates strong relationships between objects whose invariants rely on one another and defines additional conditions to ensure these invariants. The technique builds on the classical technique by providing a method of ensuring object-dependent invariants are maintained by the operations of an object-oriented system. We demonstrate our technique using the Java programming language and the JML specification language.

1. Introduction

An object's invariant is a predicate stating conditions for its members which defines the valid states of the object. This predicate is expected to be maintained by the object's operations if their contracts are met, and by clients if members are modified directly. This leads to the expectation that well-formed conditions would ensure this soundness property in the Design-by-Contract (DbC) technique [14]. However sources of unsoundness are present even with well-defined conditions, one primary cause being the situation where an invariant relies on an object for its condition that is aliased outside the invariant's object. The implication of this situation is that this dependee object could be modified by another in a valid way, but which may still invalidate the invariant that depends upon it.

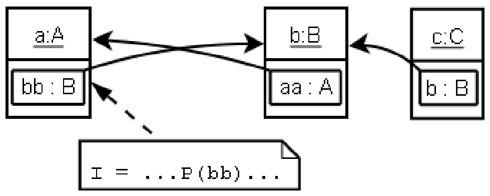


Figure 1. The Indirect Invariant Effect

To illustrate the example, consider Figure 1 that illustrates the object *a* whose invariant depends on object *b*. If *b* were modified by object *c* this could invalidate *a*'s invariant without violating any of *a*'s method contracts. The problem was first identified in [14] where it is described as the Indirect Invariant Effect. Invariants that experience the effect are dependent on instances of another object type, which are said to be vulnerable to such an invariant.

Although the problem is simply stated, it is found in many common design patterns and idioms in object-oriented systems where it can be a significant source of error. In these situations the assumption of soundness, which is that an invariant will be satisfied if method contracts are met, no longer holds. This is a result of the fact that objects can be modified in ways that satisfy their contracts, and so also their invariants, but break the invariants of other objects that depend on them.

For example, a set of iterators depend on the collection over which they iterate for their invariant conditions, such that if the collection were to have too many objects removed, an iterator may refer to a position in the collection that no longer exists. The problem also occurs in self-referential classes whose invariants rely on instances of themselves. An example is a person class with a spouse attribute and an invariant which states that the spouse's spouse must be the current `this` object. In this case it may occur that one spouse is assigned a new spouse and so breaks the old spouse's invariant. Any additional invariant that such a class may have would rely on the assumption of marriage being an exclusive bidirectional relationship between two objects.

The effect is addressed in [11] which presents a solution as an axiomatic verification methodology. The verification methodology is stated in terms of Hoare logic and concludes with a scheme of proof obligations for invariants and method conditions. The added obligations are complex and cumbersome, requiring a degree of global reasoning. This is a result of the need to globally verify that all vulnerable objects do not at any point invalidate invariants that depend on them.

Another solution to the problem of objects being vulnerable to an invariant is found in confined type and object ownership models [2, 4, 5, 10], where the type system prevents objects from being aliased outside of their creator object. Such a method may result in runtime systems organized into hierarchical series of references with upper level objects owning those below. An object's invariant may only depend on objects that are owned, which are safe from third part modification, thus preventing the Indirect Invariant Effect. This condition on invariants and the required confined/owned properties can be statically checked, such as in the Universe type system [7, 15] that encodes object ownership as special reference types.

What ownership requires is that the invariant of an object can only rely on owned objects, which only the invariant's object may reference and directly modify. This ensures that an object's invariant cannot be broken when objects it depends on are modified, since the conditions of the object's methods ensure that any modification is always valid.

The hierarchical nature of object organization that ownership creates has certain limitations in how objects may be related. For example, straight-forward ownership disallows iterators whose invariants depend on data structures that they do not own, recursive data structures such as linked lists, or recursive types like the person class where a person cannot own its spouse. Different ownership techniques address these issues, such as the visibility technique [16] that weakens the ownership requirements at the cost of greater proof obligations, but which again add to the complexity of verifying correctness.

The solution [3] used in the Boogie methodology is quite similar to the proposed solution in this paper. Using special language constructs, objects can relate themselves to "friend" objects that share responsibility for their friend's invariant. This builds on the Boogie methodology described in [17] that partially addresses the issue with a form of ownership. However this methodology relies on these specialized constructs, additional auxiliary variables, and specialized assertion statements, thus is more difficult to apply in a more general DbC approach. A more preferable approach would define a method that can be used with existing DbC analysis and verification approaches.

The root problem with the Indirect Invariant Effect is that invariants reliant on other objects create dependency relationships that are weakly represented, and so a method of defining these relationships more concretely would lead to a solution. Our Colleague Technique addresses this dependency problem by providing a mechanism of coupling objects whose invariants rely on one another, and defining additional invariant conditions which ensure that no modification to either object invalidates the other's invariant. The disadvantage of this method is reduced software reuse that's a consequence of close coupling, however this is outweighed by the ability to soundly predicate invariants on external objects. What the technique does not provide is an encapsulation mechanism, which can be provided using an ownership methodology that prevents the internal representation of an object from being externally aliased.

Collegiality is defined as an additional technique that is used with classical DbC methods, such that if a specification is correct classically then applying the Colleague Technique will result in a correct specification. This resulting specification will also use invariants and conditions as defined in the classical technique, and so allows existing analysis, verification, and code generation techniques to be applied in conjunction with collegiality. The technique is described using Java [9] and JML [12] as the specification language which it extends with a new annotation. Thus existing tools and analysis techniques developed for JML can be used in conjunction with the technique.

This section has discussed the Indirect Invariant Effect and its consequences. The remainder of this paper will discuss the Colleague Technique as a solution to this problem. Firstly, the technique will be defined as an additional concept to classical DbC constructs. Object types that are suitable as colleague types must meet certain requirements that are discussed next. This is followed by a description of how additional invariant conditions are formulated which protect invariants from being invalidated by operations on dependee objects. Finally the technique will be applied to the Iterator and Person examples discussed in this section.

2. The Colleague Technique

The previous section has outlined the Indirect Invariant Effect problem and how it introduces unsoundness in the classical invariant technique. This section will describe the Colleague Technique and how it creates strong relationships between objects whose invariants rely on one another. These relationships are used to define additional conditions on the invariants of both objects such that modification to one will not invalidate the invariant of the other.

2.1 Definition

The relationship between objects used by the Colleague Technique is defined by stating that each object type has an attribute that refers to an instance of the other type, or is a set of such references. These two attributes are declared as being each other's colleague, and the types they reside in as colleague types. The Colleague Technique is thus an additional specification concept with specific semantic requirements that solves the problem of predicating invariants on other objects.

The Indirect Invariant Effect is caused when an invariant is allowed to rely on any arbitrary object. With the Colleague Technique an invariant may rely on an object if it is referred to by one of the object's colleague attributes. This restriction limits which objects an invariant may rely on, and adds the knowledge to the specification of a colleague type that its instances may be relied upon by the invariants of other objects. With this knowledge the invariant of a colleague type can be augmented with additional conditions that prevent modifications which would invalidate their colleague's invariant.

Definition *Two object types* `A` *and* `B` *are colleague types if* `A` *has an attribute* `bb` *which is a single or a set of* `B` *references, and* `B` *has an attribute* `aa` *which is a single or a set of* `A` *references, and* `aa` *is defined as being collegial with* `A.bb` *and* `bb` *as collegial with* `B.aa`*. The invariants of* `A` *and* `B` *may only rely on objects referred to in these attributes, owned objects, and primitive values.*

An instance of `A`*,* `a`*, and one of* `B`*,* `b`*, are collegial if a reference to* `B` *is stored in* `a.bb` *and a reference to* `a` *is stored in* `b.aa`*. This is illustrated in Figure 2. Both* `a` *and* `b` *are responsible for maintaining that the relationship holds by ensuring the cross referencing and removing references when the relationship is established between them or when either object is removed from the system.*

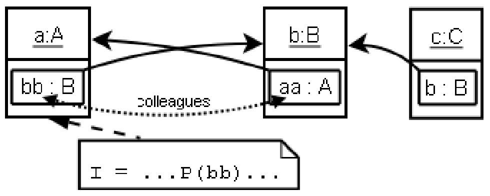


Figure 2. The Collegial Solution

This concept of explicit relationships between the attributes of two object types is similar to that used by the object-based Booster specification language [6]. Booster is descendent from Z [18] and the B Method [1], but is a domain-specific language that targets database systems. The explicit relationships are used as a means of maintaining associations between data, and through the use of Weakest Precondition [8] methods to auto-generate conditions that ensure the relationship.

This bidirectional binding between objects creates ad-hoc contexts [5] that are similar to ownership contexts, except that there is no owner/owned relationship, but a partnership between objects. If a collegial attribute is a reference value then that object may have only one colleague of that type, and if it is a set then it can have multiple colleagues.

The purpose of having this bidirectional relationship is so that an assumption about responsibility can be made by both collegial partners. If an object has a reference to a colleague object, then it can be assumed that the colleague will reference it as well. Thus an object's specification can assume that these colleagues will not allow any modification to themselves that breaks the invariant conditions that depend on them.

Figure 2 illustrates how a depends on b since its invariant includes the predicate $P(bb)$ and bb stores a reference to b. The specification for type A can safely define such an invariant since the assumption exists that b's attribute aa will alias a, and that B's invariant will include a condition on aa that prevents modifications that would invalidate P. Thus the bidirectional property of the relationship is critical to the technique, and so certain requirements and facilities must be present to ensure that the relationship is created and broken correctly. These requirements are discussed in the next section.

2.2 Requirements

The colleague relationship imposes requirements on the invariants and methods of object types that must be met for them to be used as colleague types. These requirements are necessary for the technique to correctly guarantee invariants:

- A colleague object type can only have one collegial attribute for any colleague object type. For example, an object type *A* cannot define two attributes to both be collegial with attributes of object type *B*, even if the attributes are different. Otherwise an object type can be defined to be collegial with any number of other object types.

  This restriction prevents situations where a contradiction may arise between two conjuncts of an object's invariant. If an object type *A* were allowed to have two attributes that were collegial with two attributes of *B*, then instances of *A* and *B* could become double collegial through these two means of association. It would then be possible for invariant conditions to be placed on the collegial attributes that would be contradictory if they were asserted for the same object. This also eliminates the possibility of circular dependency of invariants between two objects.

- Each invariant is responsible in maintaining its colleagues' invariant, and so colleagues must be defined in a way such that they and parts of their invariants are visible to one another. This may require that the invariants be declared as publicly visible and only rely on public members.

  This is needed since an invariant of an object will be augmented with added conditions that reflect the invariants of colleagues. If the colleague objects' invariants were not public, that is they were not publicly visible or relied on non-public members, then these added conditions could not be formulated or would be required to access non-public members.

- Both invariants may only rely on objects that are colleagues of their objects or `this`.

- They may also only rely on the members of these objects – but not members of members – which do not evaluate to regular reference types nor depend on regular reference types for their values. This restricts how complex invariants can be, but any degree of complexity can be created by classes providing appropriate methods that return useful information. Such methods, for example, may calculate values that can be used in specifications that requires other objects.

  The purpose of this restriction is to limit which objects are dependees of an invariant. If an invariant were allowed to state a condition dependent on regular objects, even if they were referenced by attributes of a colleague object, then this would reintroduce the Indirect Invariant Effect. This prevents the situation where, if an invariant includes a predicate of the form "$P(this.x.y.foo())$" where the value of *foo* is of interest, both objects *x* and *y* become dependees. In this situation *x* will not have sufficient invariant conditions to prevent invalidating modifications to *y*, since *y* is accessed directly and not through a method of *x*, which would be able to perform an invariant check which would prevent invalidating modifications to *y*.

- If a method is used in an invariant, this requires that the method's return value depend only on colleague objects, primitive values, or other methods of same object that are similarly restricted. These methods must also be pure, that is they are side-effect free.

This technique has limitations in that each colleague must be pre-defined to be part of a collegial relationship, as opposed to ownership where any arbitrary object can be owned by another. Greater coupling between objects reduces reusability, however invariants that rely on other objects already create this coupling, which the Colleague Technique formalizes.

Collegiality provides a method to closely couple two object types, whose instances may be aliased in different parts of a system, in a way that wouldn't be permitted with ownership. From the perspective of a software module, it allows an object that relies on the module's internal state to be passed over the module's public interface boundary to the client. This object is used to provide some functionality of the module, but since it relies on the internal structure it must be defined in a way that does not adversely affect this state but also allows it to be aliased by the modules's client objects. The Colleague Technique aims to provide a method of specifying such objects so that this can be achieved.

Constructing and breaking the collegial relationship is important since the cross referencing must be maintained. If one object was collegial with another, then it relies on that colleague object to alias it and so prevent operations that would break its invariant. If the relationship between two objects was malformed in that it became unidirectional, then the assumption about invariant responsibility breaks down.

What this implies is that creating and breaking the relationship are specific operations that the code of an object type should not be responsible for. Although the technique can be defined purely as a specification, it is helpful to describe these operations in terms of helper methods that define the criteria for determining when two objects are collegial and managing the relationship between colleagues:

- To access collegial references, an accessor is defined for each collegial attribute that returns the reference value if the type is a singleton or an iterator if the type is a set. This accessor is called '`getY()`' for an attribute named `Y`, eg. an attribute named '`foo`' is accessed by '`getfoo()`'.

- A colleague type must have a boolean-returning method '`isAssociated`' for every type *X* that it is collegial with, which takes a reference of type *X* and determines if it is an object that is a colleague of the current object.

- Determining if an association relationship can be formed is performed by a method called '`isAssociable`' that accepts the colleague candidate as an argument.

- A colleague type must have a void-returning method '`associate`' for every type *X* that it is collegial with. This method takes as the single argument a reference of type *X* which it adds to the collegial attribute. The method '`associate`' is then called on the argument object, passing `this` as the argument.

- A fourth method for a collegial type called '`disassociate`' is defined for every collegial type *X* which has the corollary effect of disengaging two objects from a collegial relationship, by assigning `null` to singleton types or removing the given reference from the collegial set type.

Thus a set of requirements are defined that an object type must meet so as to be useable as a colleague type, and a set of helper methods are described which are essential to the operation of the technique. These methods need not be concrete but may be abstract methods in a specification, however if colleague relationships need to be concrete in the implementation of the system then these methods would need to be as well. The next section will build on the collegial relationship and discuss how this is used to construct new invariant conditions that ensure invariant soundness.

2.3 Invariant Conditions

The purpose of entering two objects into a collegial relationship is to allow one or both to predicate their invariants on the other, such that each object's specification has the information to ensure the object's methods do not violate the others invariant. This is achieved by adding extra conditions to an object's invariant that ensure the properties its colleagues require of it.

These extra conditions are derived from the part of the object's invariant predicated on the colleague attribute, which are then expressed in terms of the colleague object itself. Taking a condition placed on a member of a colleague attribute and replacing the name of the attribute with `this` restates the condition from the perspective of the colleague object itself. This new condition, which states the same property but from the perspective of the other colleague object, can then be used as the needed additional condition.

Given the object types A and B from the above discussion and their respective invariants I_A and I_B, the part of I_A predicated on `bb` is denoted by P which must be in a form where every member access must explicitly begin with '`this.`' (called normal form in this context):

$I_A = ...P(bb)...$	– if `bb` is a singleton type
$I_A = ... \forall\, i : bb \mid S \bullet P(i)...$	– if `bb` is a set type, given S
$I_A = ... \exists\, i : bb \mid S \bullet P(i)...$	– if `bb` is a set type, given S

These three forms of the invariant for A relate members of `bb`, either attributes or values returned from pure method calls, to members of A or constant values. P is stated in the perspective from A to B, and so to reverse the perspective and produce an invariant for B, the roles of `this` and `bb` must be reversed. This takes P and produces a mirror P_m stated in terms of `aa`.

If `aa` is a singleton attribute then there are two forms of the mirror P_m predicate:

$$P_m(aa) == P[this, this.aa/this.bb, this]$$

– if `bb` is a singleton

$$P_m(aa) == S[this, this.aa/i, this] \Rightarrow P[this, this.aa/i, this]$$

– if `bb` is a set

If `aa` is a set attribute then the two forms are quantified over the elements of the set:

$$P_m(aa) == \forall\, i : this.aa \bullet P[this, i/this.bb, this]$$

– if `bb` is a singleton

$$P_m(aa) == \forall\, i : this.aa \mid S[this, i/i, this] \bullet P[this, i/i, this]$$

– if `bb` is a set

Therefore the invariant of B has the additional requirement of maintaining the predicate $P_m(aa)$:

$$I_B = ... \wedge P_m(aa)$$

The predicate P states relationships between the members of the classes A and B, and P_m states the same relationships but from the perspective of the other colleague type. This has the effect of swapping collegial references with `this` wherever they occur in P and reverses the direction of the predicate.

If, for example, P represented the expression '`this.bb.m<10`', then the mirror P_m would equal '`this.m<10`', which would ensure that the required property of `m` would be maintained. For a more complex example take P to represent '`this.bb.m==this.n()`' for some method `n`, then the mirror P_m is '`this.m==this.aa.n()`'.

There is another possible original form of the invariant other than the three given above. If bb is a singleton which may be set to `null` (that is it is nullable in JML terms) then the P predicate would be false when this occurs, thus an implication relation is used to guard against this possibility:

$$I_A = ...bb \neq \texttt{null} \Rightarrow P(bb)...$$

The mirror invariant of this form is derived by taking P and applying the above transformation. If aa is a set type then this P_m becomes the resulting invariant, but if it is a singleton that is nullable then a guard implication is used in this instance as well:

$$I_B = ...aa \neq \texttt{null} \Rightarrow P_m(aa)...$$

In the presence of inheritance where an object type can inherit or implement a colleague type, it is not difficult to see that behaviour subtyping [13] is necessary for the technique to work. If this were not the case then an object type may inherit from a collegial type and not be responsible for the inherited mirror invariant, thus even if it remains internally consistent the invariants of dependent objects may be invalidated.

2.4 Results

The Colleague Technique as described is used to make explicit the relationships created by invariant dependencies. The purpose in doing so is to develop a means of preventing the Indirect Invariant Effect from allowing invariants to become invalidated without contractual violations. The additional conditions that are added to the invariants of colleague types achieve this, and are dependent upon the fact that only their colleague types will depend on them for their invariants.

To understand how the technique provides this guarantee, consider the conditions that the classical DbC technique places on a method call. The precondition and invariant of an object must hold before a method begins, and since the mirror invariant must also be asserted here then the object is guaranteed to be in a state that does not break the invariant of another. When a method exits, the postcondition and invariant is asserted which again makes the valid state guarantee. By encoding the reciprocal responsibilities that collegial objects have to one another as invariant conditions, the Colleague Technique uses existing DbC methodologies to safeguard object-dependent invariants. Thus the Colleague Technique does not require additional proof obligations in addition to those used in a pre-existing verification methodology.

An implementation, in Java and using JML, of the iterator problem discussed previously demonstrates how the technique prevents an instance of unsoundness in the classical DbC approach. An additional "collegial" annotation is used to declare those attributes that are collegial with which other object type, and with what attribute.

Figure 3 lists the code for this example. It states that the attribute `iterators` of *List* is collegial with *List* or *ListIterator*. The invariant of *ListIterator* that relies on the instance of *List* it iterators over states that its size must not be less than what it was when the iterator was instantiated:

```
this.list.size()>=this.last
```

To ensure that this does not happen, `List` must have an invariant that ensures its size is never less than the `last` attribute of any associated iterators:

```
(\forall ListIterator i; this.iterators.contains(i);
                 this.size()>=i.last)
```

Since the `associate` method constructs the relationship correctly, this additional invariant prevents the removal of enough elements from an instance of *List* to break an associated iterator's invariant. To allow the removal of elements again from a *List* instance, it would be required to disassociate collegial iterators, which occurs when they are no longer needed and are removed from the system. Thus collegiality forces coordination between iterators and collections, which is implicitly required by the fact that iterators are dependent on their collection's state.

This invariant was generated using a prototype Java tool that has been successfully used with this example and the Person example in described below. The tool analyzes the invariants of input Java classes, generates mirror invariants using the methodology outlined in this paper, and outputs the classes again with the mirror invariants and helper methods added. The resulting classes can be compiled into standard Java using the Common JML Tools[1], which adds runtime assertion checks to the compiled bytecode. The resultant classes have been analyzed through testing and successfully provide runtime checks that prevent the Indirect Invariant Effect.

This tool demonstrates how the described technique can be used in conjunction with existing DbC techniques to close the unsoundness gap created by object-dependent invariants. With only the additional `collegial` annotation augmenting standard JML, the tool produces resulting code that has only standard JML annotations and standard Java code, such that other tools that analyze and transform JML-annotated Java code can be subsequently used. A more sophisticated tool may be able to identify attributes of classes that need to be collegial without the additional `collegial` annotation, thus without adding significantly new specification constructs or methodologies that other solutions require, the Colleague Technique effectively addresses the problem of object-dependent invariants and can be employed with a relatively simple code-generating tool.

The second discussed example involved self-referential types, such as the spouse example in Figure 4. The invariant of the class requires that one's spouse be married to one's self. The method by which `associate` operates, which ensures the cross-referencing of colleague objects, would guarantee that this invariant would always be true if the `spouse` parameter was collegial as the code defines. Invariants that state conditions on members of colleagues can also be used in this instance, but would still require additional conditions stating the same property for the local attributes.

3. Conclusion

This paper has described the Colleague Technique, and its associated ownership technique, that is stated as a solution for the Indirect Invariant Effect. The effect is a critical problem with the classical DbC invariant technique since many common design patterns and programming idioms rely on the aliasing of objects within a system.

This technique defines a method of correctly constructing a relationship between objects whose invariants depend on one another, and how additional conditions ensure that operations on either will not invalidate the other's invariant. This discussion has been done in terms of concrete Java methods and attributes, however the technique can be defined in terms of abstract model variables entirely in some cases and without the concrete helper methods. Either as a concrete or abstract component of a specification, the purpose of the technique is to make explicit the relationship between objects that are created when an invariant relies on other objects for its conditions. What the technique does not provide is a method of guaranteed encapsulation, which is best accomplished using a lightweight method of ownership.

The net result of this technique is to close the unsoundness gap created in the classical DbC technique cause by invariants

[1] http://sourceforge.net/projects/jmlspecs/

```
class List {
    private /*@ spec_public @*/
            ArrayList items = new ArrayList();
    private /*@ collegial ListIterator.list; @*/
            Set iterators = new LinkedHashSet();

    // The mirror invariant derived from ListIterator
    //@ invariant (\forall ListIterator i;
    //@   this.iterators.contains(i); this.size()>=i.last);

    //@ requires o != null;
    //@ ensures this.items.contains(o);
    public void add(Object o) { this.items.add(o); }


    //@ requires i>=0 && i<this.size();
    //@ ensures \result == this.items.get(i);
    public /*@ pure @*/ Object get(int i)
            { return this.items.get(i); }

    //@ ensures: \result == this.items.length();
    public /*@ pure @*/ int size()
            { return this.items.length(); }

    //@ requires i >= 0 && i < this.size();
    //@ ensures !this.items.contains(
    //@         \old(this.items.get(i)));
    public void remove(int i) { this.items.remove(i); }

    public ListIterator iterator()
        { return new ListIterator(this);}
}

class ListIterator {
    private /*@ nullable collegial List.iterators; @*/
            List list;
    private /*@ spec_public @*/ int position=0, last;

    //@ invariant this.position<=this.last;

    // The invariant dependent on the colleague object
    //@ invariant this.list!=null ==>
    //@         this.list.size()>=this.last;

    //@ requires this.isAssociable(l);
    //@ ensures this.isAssociated(l);
    public ListIterator(List l){ this.associate(l);
            this.last=l.size(); }

    //@ requires this.list != null;
    //@ ensures \result == this.position<this.last;
    public /*@ pure @*/ boolean hasNext()
            { return position<=last; }

    //@ requires this.hasNext();
    //@ ensures this.position==\old(this.position)+1;
    //@ ensures \result==this.list.get(\old(this.position));
    public Object next(){ this.position++;
            return this.list.get(this.position-1); }

    protected void finalize()
            { this.disassociate(this.list); }
}
```

Figure 3. Iterator Collegial Example

```
class Person {
    private /*@ nullable collegial Person.spouse @*/
            Person spouse;

    //@ invariant this.spouse != null ==>
    //@     this.spouse.spouse == this;
    ...
}
```

Figure 4. Marriage Example

relying on externally aliased objects. Allowing invariants to be predicated on objects is an important component when specifying complex layered object structures, and so a method that ensures the soundness of the technique, i.e. if the conditions of operations are met then the invariants will remain valid, contributes significantly to the correctness and applicability of this formal method to real-world complex software engineering challenges.

This paper has informally defined and discussed the Colleague Technique by examining the problem and the proposed solution. Future work with the technique will elaborate on how it can be integrated with JML and be used as an abstract or concrete specification technique. Proving the property that colleague specifications derived from correct specifications are themselves correct, and that it does solve the problem of soundness with object-dependent invariants, is also part of future research with the technique. Through induction on the method of creating mirror invariants from original invariants, the proof must show that the mirror invariants are well-formed, well-typed, and do not represent new restrictions on the system. To prove that the technique does solve the soundness problem, it will be necessary to formally express an invariant's dependence on objects, and use this to demonstrate that the technique correctly guards against invalidating modifications. The development of tool support is also planned, whose objective is to analyze and possibly prove the correctness of programs that use the technique.

References

[1] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1992.

[2] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32, 1997.

[3] Mike Barnett and David Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer, 2004.

[4] Boris Bokowski and Jan Vitek. Confined types. Technical report, 1999.

[5] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, October 18–22 1998. ACM Press.

[6] Jim Davies, Charles Crichton, Edward Crichton, David Neilson, and Ib Holm Sørensen. Formality, evolution, and model-driven software engineering. In *Proceedings of the Brazilian Symposium on Formal Methods (SBMF 2004)*, volume 130 of *Electronic Notes in Theoretical Computer Science*, pages 39–55, May 2005.

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

[8] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[9] James Gosling et al. *The Java Language Specification*. GOTOP Information Inc.

[10] John Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285, New York, NY, USA, 1991. ACM Press.

[11] Kees Huizing and Ruurd Kuiper. Verification of object oriented programs using class invariants. In *FASE '00: Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering*, pages 208–221, London, UK, 2000. Springer-Verlag.

[12] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[13] B. Liskov and J. Wing. Behavioral subtyping using invariants and constraints, 1999.

[14] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[15] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[16] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. Technical Report 424, ETH Zurich, March 2005.

[17] K. Rustan, M. Leino, and P. uller. Object invariants in dynamic contexts, 2004.

[18] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, 2 edition, 1992.

Class Invariants: The end of the road?
(Position Paper)

Matthew Parkinson
University of Cambridge
Matthew.Parkinson@cl.cam.ac.uk

Introduction Since Hoare's seminal paper on data abstraction [5], the class invariant has been the foundation for verifying object-oriented programs. Experience has shown that there are two complications in scaling class invariants to real programs: (1) invariants need to depend on multiple objects; and (2) invariants need to be temporarily broken owing to call-backs. There are several proposals in the literature, which extend class invariants to partially address these two problems. The time seems to be right to pose the following (deliberately provocative!) question: "Is the class invariant the correct foundation for verifying object-oriented programs?"

The basic unit of a Java program is a class, but interesting programs use more than one class. They are decomposed into aggregate[1] structures, containing many inter-related classes collaborating in some function of the system. The aggregate structure is the key concept in any object-oriented program. Hence our verification method needs to describe invariants of these aggregate structures.

The class invariant can only reason about a single object. Using ownership based methodologies [1–4, 7, 8] we can extend the class invariant to some aggregate structures, and allow an object's invariant to depend on objects it owns completely. However, in more complex examples the ownership is less clear cut. Consider two collaborating classes: neither owns the other and each has an invariant depending on the state of the other. Any update to one object will potentially invalidate the invariant of the other object. So how can we update this co-dependent structure? Ideas such as peer invariants [6], friends and update guards [3], and history properties [7], have been used to extend the idea of a class invariant, so that it can depend, soundly, on other objects. But is the complexity of these proposals a sign that the class invariant is not the correct foundation?

Our position is to take a step back and consider a more general foundation. Our approach uses predicates [9, 10] to simply specify the properties of aggregate structures. A class invariant is then just a particular (useful!) predicate.

Subject/Observer The subject/observer pattern, given in Figure 1, exhibits many of the difficulties in reasoning with class invariants. We would like to specify an invariant for the Observer that **this**.sub.val = **this**.cache. However, this invariant does not always hold, because there is a time between when update is called on a Subject, and notify is called on the Observer where the invariant is not satisfied.

Instead of trying to write a property of the individual classes, let us consider a property of the aggregate structure. A single Subject object will have many Observer objects. We expect that, if we update the Subject object, then all the Observer objects will be notified and their status suitably updated. The aggregate structure

```
class Subject {                         class Observer {
  List obs; int val;                      Subject sub;
  Subject()                               int cache;
  { obs = new List(); }                   Observer(Subject s) {
  void register(Observer o)                 this.sub = s;
  { this.obs.add(o);                        s.register(this);
    o.notify(); }                         }
  void update(int n) {                    void notify()
    this.val = n;                         { this.cache = s.get(); }
    foreach(Observer o:obs)               int val()
      o.notify();                         { return cache; }
  }                                     }
  int get() { return this.val; }
}
```

Figure 1. Source code for subject/observer pattern

can be specified with the following predicate definition:

$$SubObs(s, O, v) \overset{\text{def}}{=} Sub(s, O, v) \wedge \forall o \in O. Obs(o, s, v)$$

Here s is the Subject, O is a set (list) of Observers and v is the current value of the Subject. In the definition, $Sub(s, O, v)$ represents a Subject object s, that has O Observers, and current value, v; and $Obs(o, s, v)$ represents an Observer object, with Subject s that had value v last time it was notified. The ownership properties are captured directly by using separation logic (see [9, 10] for more details).[2] We give the definitions of the predicates in Figure 2.

The $SubObs$ predicate can be seen as the invariant of the aggregate structure. Accordingly, it should hold on the entry and exit of every public method of the aggregate (this is just a generalization of a class invariant). So the "aggregate invariant" should hold on the entry and exit of the two constructors, the update method of the Subject, and the val method of the Observer. The other methods (register, notify and get) are internal to the aggregate structure. We present the specifications of the methods and constructors in Figure 2.

When verifying the Subject methods, we use the definition of $SubObs$ and Sub predicates, and verifying the Observer methods we can use both $SubObs$ and Obs definitions. Hence, the Subject is independent of the Observer, and vice-versa, but they are both dependent on the aggregate structure to which they belong, hence our reasoning remains modular.

We present an example verification of the constructor of the Observer:

$\{SubObs(\mathsf{s}, O, v) * \mathsf{this.sub}{\mapsto}\_ * \mathsf{this.cache}{\mapsto}\_\}$

[1] Here we mean *aggregate* in its most general sense to capture additionally the UML meanings of *association* and *composition*.

[2] *Separation logic in a footnote.* Separation logic is an extension to Hoare logic that allows reasoning about heap data-structures. It has two new connectives: $P * Q$ means the state can be split into two disjoint parts, one satisfying P and the other Q; $x.f \mapsto y$ means the object x has a field f containing y; and $\circledast_{i \in \{i_1,\ldots,i_n\}}.P(i)$ means $P(i_1) * \ldots * P(i_n)$.

Predicates
$SubObs(s,O,v)$
$\overset{\text{def}}{=}\ Sub(s,O,v) * \circledast_{o\in O}\, Obs(o,s,v)$
$Sub(s,O,v)$
$\overset{\text{def}}{=}\ \exists l.\ s.\mathsf{val}\mapsto v * s.\mathsf{obs}\mapsto l * list(l,O)$
$Obs(o,s,v)$
$\overset{\text{def}}{=}\ o.\mathsf{cache}\mapsto v * o.\mathsf{sub}\mapsto s$

Method	Pre-condition	Post-condition
s=Subject()	emp	$SubObs(\mathsf{s},\emptyset,\_)$
s.register(o)	$Sub(\mathsf{s},O,v) * Obs(\mathsf{o},\mathsf{s},\_)$	$Sub(\mathsf{s},\mathsf{o}::O,v) * Obs(\mathsf{o},\mathsf{s},v)$
s.update(n)	$SubObs(\mathsf{s},O,v)$	$SubObs(\mathsf{s},O,\mathsf{n})$
ret=s.get()	$Sub(\mathsf{s},O,v)$	$Sub(\mathsf{s},O,v) \wedge \mathsf{ret}=v$
o=Observer(s)	$SubObs(\mathsf{s},O,v)$	$SubObs(\mathsf{s},\mathsf{o}::O,v)$
o.notify()	$Sub(s,O,v) * Obs(\mathsf{o},s,\_)$	$Sub(s,O,v) * Obs(\mathsf{o},s,v)$
ret=o.val()	$SubObs(s,O,v) \wedge \mathsf{o}\in O$	$SubObs(s,O,v) \wedge \mathsf{ret}=v$

where $o \in (o' :: O') \overset{\text{def}}{=} o = o' \vee o \in O'$ and $o \in \emptyset \overset{\text{def}}{=} false$

Figure 2. Specification of subject/observer pattern

$\quad$ **this**.sub = s;
$\{SubObs(\mathsf{s},O,v) * \mathsf{this.sub}\mapsto\mathsf{s} * \mathsf{this.cache}\mapsto\_\}$
$\{SubObs(\mathsf{s},O,v) * Obs(\mathsf{this},\mathsf{s},\_)\}$
$\{Sub(\mathsf{s},O,v) * (\circledast_{o\in O}\, Obs(o,\mathsf{s},v)) * Obs(\mathsf{this},\mathsf{s},\_)\}$
$\quad$ s.register(**this**);
$\{Sub(\mathsf{s},\mathsf{this}::O,v) * (\circledast_{o\in O}\, Obs(o,\mathsf{s},v)) * Obs(\mathsf{this},\mathsf{s},v)\}$
$\{Sub(\mathsf{s},\mathsf{this}::O,v) * (\circledast_{o\in(\mathsf{this}::O)}\, Obs(o,\mathsf{s},v))\}$
$\{SubObs(\mathsf{s},\mathsf{this}::O,v)\}$

Interestingly, the Observer's constructor causes problems for class invariant based verification, because it calls another class's method, which in turn calls back into the Observer. This complicated calling pattern is forbidden in the class invariant approach, and requires additional machinery [1]. Simply by using predicates over aggregates we avoid such constraints.

Conclusion We have demonstrated a straightforward proof of the subject/observer pattern. We have not invented new methodology or ownership types. We have simply considered a property of an aggregate structure. These properties, we claim, are the key to verifying object-oriented programs, and should not be shoehorned into class invariants. Class invariants have taken us a long way, but properties of aggregate structures should now form the foundation of verification.

Acknowledgments We thank Gavin Bierman, Sophia Drossopoulou, and Peter O'Hearn for encouraging me to write this position paper.

References

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

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec$^\sharp$ programming system: An overview. In *Proceedings of CASSIS*, pages 49–69, 2005.

[3] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, pages 54–84, 2004.

[4] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 2005.

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

[6] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.

[7] K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *Proceedings of ESOP*, LNCS, 2007.

[8] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[9] M. J. Parkinson. *Local Reasoning for Java*. PhD thesis, Computer Laboratory, University of Cambridge, 2005. UCAM-CL-TR-654.

[10] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.

Annotations for (more) Precise Points-to Analysis

Mike Barnett Manuel Fähndrich
Francesco Logozzo
Microsoft Research
{mbarnett, maf, logozzo}@microsoft.com

Diego Garbervetsky
Departamento de Computación, FCEyN, UBA
diegog@dc.uba.ar

Abstract

We extend an existing points-to analysis for Java in two ways. First, we fully support .NET which has structs and parameter passing by reference. Second, we increase the precision for calls to *non-analyzable* methods. A method is non-analyzable when its code is not available either because it is abstract (an interface method or an abstract class method), it is virtual and the callee cannot be statically resolved, or because it is implemented in native code (as opposed to managed bytecode). For such methods, we introduce extensions that model potentially affected heap locations. We also propose an annotation language that permits a modular analysis without losing too much precision. Our annotation language allows concise specification of points-to and read/write effects. Our analysis infers points-to and read/effect information from available code and also checks code against its annotation, when the latter is provided.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Object-oriented programming, static analysis, points-to analysis, effects analysis

Keywords object-oriented, points-to analysis

1. Introduction

Object-oriented languages, as C# or Java, strongly rely on the manipulation (read/write) of dynamically allocated objects. As a consequence, static analysis tools for these languages need to compute some heap abstraction. Here, we focus our attention on a static analysis for determining the side-effects of statements and methods.

Side effect information can be used for program analysis, specification, verification and optimization. If it is known that a method m has no side-effects, then during the analysis of a caller, m can be handled in a purely functional way. Furthermore, m can be used in assertions and specifications [13, 5]. Side effect-free methods enable several optimizations such as caching the computed results and automatic parallelization.

Analysis of side-effects in mainstream OO languages is not simple as (i) different variables or fields may refer to the same memory location (aliasing); (ii) the relationship between objects can be very complex (shape); (iii) the number of objects can be unbounded (scalability); and (iv) it can be difficult or impossible to statically determine the control flow because of dynamic binding or because not all the code is not available at analysis time, e.g., when analyzing a class library or programs that use native code.

We extend an existing points-to and effect analysis presented by Salcianu et al. [22] to infer read and write effects for code targetting the .NET Common Language Runtime (CLR) [11]. The CLR is the common infrastructure for languages such as C#, Visual Basic, Managed C++, etc. Unlike Java, the CLR adds support for struct types and parameter passing by reference via managed pointers, i.e., garbage collector controlled pointers. For each method in the application we compute a summary describing a read/write effects and a points-to graph that approximates the state of the heap at the method's exit point.

The more important extension is the inclusion of additional support for *non-analyzable* calls. We can analyze programs that have calls to non-statically resolvable calls such as interface calls, virtual calls, and native calls while being less pessimistic than Salcianu's analysis. We define a concise yet expressive specification language to describe points-to and read/write effects for a method. The method annotations are used (i) as summaries, to analyze code involving calls to non-analyzable methods; (ii) to enable modular analysis, i.e., when analyzing a method n that invokes a method m, we (a) use the annotation $\mathcal{A}(m)$ in the analysis of the body of n and (b) we check m against its specification $\mathcal{A}(m)$; (iii) as documentation and contracts to impose restrictions on eventual implementations [18]. This allows our analysis to work even without computing a precise call graph.

In this work we apply our analysis primarily for checking *method purity* but it can be used for any other analysis that requires aliasing information and/or conservative read/write effect information. Purity is informally understood to mean that a method has no effect on the state. Formally, however, there are different levels of purity [6]. Our analysis computes weak purity, i.e., it infers weak purity and it checks whether a method annotated as being weakly pure lives up to its contract. A *weakly pure* method does not mutate any object that was allocated prior to the beginning of the method's execution. Because a weakly-pure method can return newly allocated objects and since object equality can be observed by clients, there may be further restrictions on weakly-pure methods in order to use them in specifications [10].

The main contributions of the paper are:

- An interprocedural read/write effect inference technique, built on the top of the points-to analysis, for the .NET memory model that relaxes the *closed world* assumption.
- A new set of annotations for representing points-to and effect information in a modular fashion. The annotations are considered valid for interprocedural analysis when the methods are called, and verified when the implementations of the methods are analyzed.
- An implementation integrated into the Spec# compiler [23] to infer and verify method purity and for checking the admissibility of specifications in the Boogie methodology [5].

1.1 The Problem

Consider the following simple, but realistic example. Figure 1 contains a method written by a programmer to copy a list of inte-

```
List<int> Copy(IEnumerable<int> src)
{
  List<int> l = new List<int>();
  foreach (int x in src)
    l.Add(x);
  return l;
}
```

Figure 1. A simple use of an iterator in C#.

```
List<int> Copy(IEnumerable<int> src)
{
  List<int> l = new List<int>();
  IEnumerator<int> iter =
    src.GetEnumerator();
  while (iter.MoveNext()){
    int x = iter.get_Current();
    l.Add(x);
  }
  return l;
}
```

Figure 2. "Desugared" version of the iterator example.

gers. In C#, the *foreach* is syntactic "sugar" which the compiler expands ("desugars") into the code shown in Figure 2. (Programmers are also able to directly write the de-sugared version.) The desugared version shows that there is one method call from the interface $IEnumerable\langle T\rangle$ and two from the interface $IEnumerator\langle T\rangle$. In addition, the constructor for the type $List\langle T\rangle$ is called, as is its Add method.

A points-to analysis produces the set of memory locations that are read and written by $Copy$. That information can then be used to determine if $Copy$ is (weakly) pure. It clearly mutates the list that it creates and returns, but that list is created after entry into the method and the original collection from which the integers are drawn is unchanged. Thus, we desire an analysis that is precise enough to recognize its purity.

Salcianu's analysis would not be able to analyze the calls to the interface methods. It would make the conservative approximation that the parameter src could escape to any location in memory and that the method has a (potential) write effect on all accessible locations, such as all static variables. This precludes $Copy$ from being pure and, perhaps more importantly, pollutes the analysis of any method that calls it because those effects then become the effects of the caller.

We have created a specification language for concisely describing the points-to graph and read/write effects of a method. The design of such a language is subject to common engineering tradeoffs: it should be precise enough to enable the recognition of common programming idioms while at the same time be concise enough for programmers to use in everyday practice.

We add annotations written in the language to method signatures. At call sites, we trust the annotation of the called method; annotations are then verified when analyzing a method implementation. Annotations are inherited: they must be respected in every subtype by overriding methods. We use the set of annotations to model non-analyzable calls with better precision than previously possible while still computing a conservative points-to graph and read and write effects of the callee. The annotations describe an approximation of the read and write effects of the method.

1.2 Paper structure

First, we review the essential ideas from Salcianu's analysis in Section 2 and present our extensions to deal with .NET memory model and non-analyzable calls. Section 3 presents our annotations and the extensions to Salcianu's analysis needed to process the points-to graphs they represent. Our preliminary experimental results appear in Section 4. Some related work is reviewed in Section 5 and our conclusions are presented in Section 6.

2. Salcianu's Analysis

Salcianu et al. [22] created an analysis for Java programs that performs an intra-procedural analysis of each method to obtain a method summary that models the result of the analysis at the end of the method's execution. We briefly review their analysis.

Their analysis relies on having a precise precomputed call graph for the entire application. Methods are traversed in a bottom up fashion, using already computed method summaries at each call site. To deal with recursion, a fixpoint computation operates over every strongly-connected component (i.e., group of mutually recursive methods). When a method invokes another method, the current state of the caller and the method summary for the callee are unified to represent the caller's state after the call.

The intra-procedural analysis is a forward analysis that computes a points-to graph (PTG) which over-approximates the heap accesses made by a method m during all its possible executions. Given a method m and a program location pc, a points-to graph P_m^{pc} is a triple $\langle I, O, L\rangle$, where I is the set of inside edges, O the set of outside edges and L the mapping from locals to nodes [1]. The nodes of the graph represent heap objects; there are basically three different types of nodes. *Inside nodes* represent objects created by m, while *parameter nodes* represent the value of an object passed as an argument to m. *Load nodes* are used as placeholders for unknown objects or addresses. A load node represents elements read from outside m.

Relations between objects are represented using two kind of edges: *inside* edges model references created inside the body of m and *outside* edges model heap references read from objects reachable from outside m, e.g., through parameters or static fields.

When the statement at the program point pc is a method call, op, the analysis uses a summary of the callee P_{callee}—a PTG representing the callee effect on the heap—and computes an inter-procedural mapping $\mu_m^{pc} :: \mathsf{Node} \mapsto \mathcal{P}(\mathsf{Node})$. It relates every node $n \in nodes(\mathsf{P}_{callee})$ in the callee to a set of existing or fresh nodes in the caller $(nodes(\mathsf{P}_m^{pc}) \cup nodes(\mathsf{P}_{op}))$ and is used to bind the callee's nodes to the caller's by relating formals with actual parameters and also to try to match callee's outside egdes (reads) with caller's inside egdes (writes).

For each program point within m, the analysis also records the locations that are written to the heap. The summary of a method represents the abstract state at the method's exit point in term of its parameters. It contains all reachable nodes from the (original) parameter nodes.

2.1 Extensions for the .NET Memory Model

We extend this analysis to support features of the .NET platform not present in Java: parameter passing by reference and struct types. Struct types have *value* semantics; they encompass both the primitive types like integers and booleans as well as user-defined record types. To accommodate both references and structs, we add

[1] The set of nodes is implicitly described by the two sets of edges and the local variables map. Salcianu's analysis also has one more element, E, the escaping node set. Instead, we represent an escaping node by connecting it to a special node that represent the global scope.

a new level of dereference using *address nodes*. In this model, every variable or field is represented by an address node. In the case of objects (or primitive types) the address node then refers to the object itself. A struct value is represented directly by its address. To access an object we first get a reference to an address node and then follow that to the value. In the case of structs we directly consider the address as the starting offset of the struct. Thus, an address node for an object has outgoing edges labeled with the "contents-of" symbol "*", while an address node for a struct value has one outgoing edge for each field of the struct: the labels are the field names.

This distinction is used in the assignment of objects and structs. For objects, we just copy the value pointed to by the address node, and for structs we also copy all the values pointed to by its fields. Figure 3 shows the representation of object and struct values and how the assignment of struct values is done. Address nodes are depicted as ovals, values as boxes.

In [4] we formally present the concrete and abstract semantics of the extended model. Basically we support the statements that operate on managed pointers. For instance the statement that loads an address `a = &b` assigns to `a` the address of `b`. If the type of `b` is a struct type `a` will contain a reference to it. Thus, `a` can be used as if it were an object. The pair of statements indirect load, `a = *b`, and indirect store, `*a = b`, allows indirect access to values and are typically used to implement parameter passing by reference. We also keep track of read effects by registering every field reference (load operation).

Figure 4 shows a simple method and three points-to graphs at different control points in the method. All of the addresses in the figure refer to objects. One node models all globally accessible objects. The graph on the left shows the points-to graph as it exists at the entry point of the method. The middle graph shows the effect of executing the body of the method: the points-to graph is shown at the exit point of the method. Finally, the right graph is the summary points-to graph for the method. It represents the method's behavior from a caller's point of view. Notice that the initial value of the parameter a has been restored since a caller would not be able to detect that it is re-assigned within the method. The summary for the method is a triple made up of a points-to graph that approximates the state of the heap, a write set $\mathcal{W}$, and a read set $\mathcal{R}$.

2.2 Extensions for Non-analyzable Methods

Salcianu's analysis computes a conservative approximation of the heap accesses and write effects made by a method. A call to a non-analyzable method causes all arguments to escape the caller and also to cause a write effect on a global location [22].

For a more precise model of non-analyzable calls, we generate summary nodes for non-analyzable methods. A load node (in particular, a parameter node) is a placeholder for unknown objects that may be resolved in the caller's context. In the case of analyzable calls, at binding time the analysis tries to match every load node with nodes in the caller. A match is produced when there is a path starting from a callee parameter that unifies with a path in the caller. That means that a read or write made on a callee's load node corresponds to a read or write in the caller. As reads and writes in the callee are represented by edges in the points-to graph, those edges must be translated to the caller.

Non-analyzable calls may have an effect on every node reachable from the parameters. That means that, unlike analyzable calls, some effects might not be translated directly to the caller points-to graph as it may not have enough context information to do the binding. For instance, a non-analyzable callee $m2$ may modify $p1.f1.f2.f3$ to point to another parameter $p2$ and a caller m that performs the method call $m2(a1, a2)$ may have points-to information only about $a1.f1$. As we don't know "a priori" the effect of $m2$ it would be unsound to consider only an effect over $a1.f1$ in the caller. We need some mechanism to update $a1$ when more information becomes available (e.g., when binding m with its caller).

2.2.1 Omega Nodes

We introduce a new kind of node, an ω node, to model the set of reachable nodes from that node. At binding time, instead of mapping a load (or parameter) node with the corresponding node in the caller, ω nodes are mapped to every node reachable from the corresponding starting node in the caller. For instance, an ω node for a parameter in the callee will be mapped to every node reachable from the corresponding caller argument.

Figure 5 shows an example of how ω nodes are mapped to caller nodes during the inter-procedural binding. Suppose that somehow we know the non-analyzable method call creates a reference from some object reachable from $p1$ to some object reachable from $p2$. Since we don't know which fields are used on the access path, we use a new edge label, $?$, that represents any field. At binding time we know that from $a1$ we can reach $IN1$ and $IN2$. Thus, we must add a reference from both nodes to the nodes reachable from $a2$.

We want to distinguish between a node being merely reachable from it being writable (e.g., an iterator may access a collection for reading but not for writing). For this purpose, we introduce a variant of ω nodes: ωC nodes. The C stands for *confined*, a concept borrowed from the Spec# ownership system [2]. These nodes have the same meaning as ω nodes for binding a callee to a caller, but they represent only nodes reachable from the caller through fields it *owns*. Ownership is specified on the class definition: a field f marked as being an *owning* field in class T means that an object o of type T owns the object pointed to by its f field, $o.f$ (if any).

To model potential read or writes we use $?$ edges to mean that the method may generate a reference using an unknown field for any object reachable from the object(s) represented by the source node to the object(s) represented by the target node. As we want a conservative approximation of the callee's effect, we only generally introduce inside edges in non-analyzable methods because they do not disappear when bound with the caller's edges. We use another wildcard edge label $\$$, that includes only a subset of the labels denoted by $?$. $\$$ denotes only non-owned fields and allows distinguishing between references to objects that can be written by a method, from references that can only be reached for reading (see Section 3 in particular the $WriteConfined$ attribute). This is the distinction that allows the use of impure methods while retaining guarantees that some objects are not written. For the worst case scenario we connect every parameter ω node of the non-analyzable method to other parameter nodes and to themselves using edges labeled as $?$ to indicate potential references created between objects reachable from the parameters. Section 3 presents our annotation language that helps eliminate some of these edges.

2.2.2 Interprocedural binding

To deal with the new nodes and edge labels, we adapt the inter-procedural mapping μ. Recall that μ is a mapping from nodes in the callee to nodes in the callee and the caller. Thus, for every ω node $n_{pc}^{L\ \omega}$ we compute the closure of $\mu(n_{pc}^{L\ \omega})$ by adding the set of reachable nodes from $\mu(n_{pc}^{L\ \omega})$ to itself.

When computing the set of reachable nodes matching an ωC node we consider only paths that pass through owned fields[2] and $?$ edges. Note that we reject paths that contain $\$$ edges.

Finally, we convert any load nodes, n_{pc}^{L}, contained in the set $\mu(n_{pc}^{L\ \omega})$ to ω nodes. This is because these nodes could be resolved when more context is available, at which point we still need to apply the effect of the non-analyzable call to those nodes. For

[2] We mean "owned fields" as defined in the Boogie methodology [2].

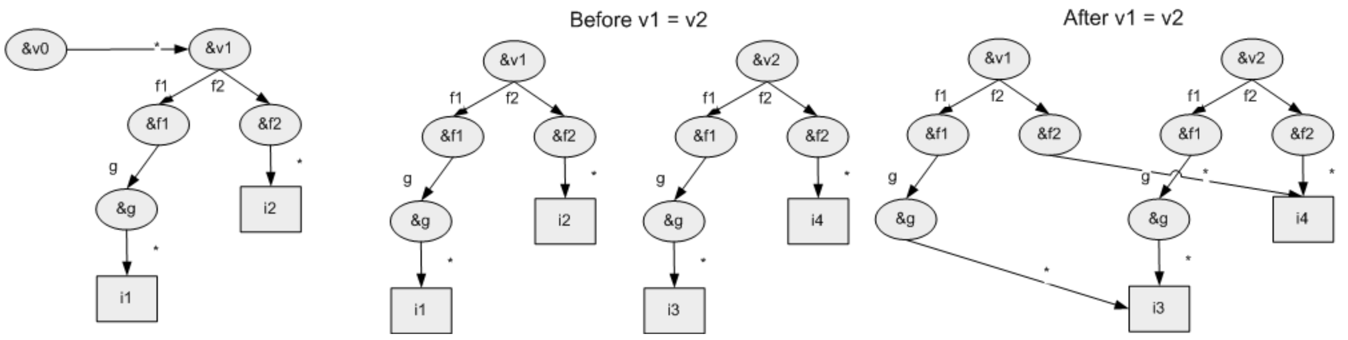


Figure 3. Modeling objects and structs. On the left v_0 is the address of v_1, which is a value of a struct type with two fields $f1$ and $f2$. (v_0 can be thought of as an object, e.g., if the struct is passed to a method that takes an object as a parameter then v_1 would be a *boxed* value.) The type of $f1$ is also a struct type with one field g which is of an object type. The type of $f2$ is an object type. The center and right figures show an assignment of two variables of struct type.

instance in Figure 5, before the binding all nodes reachable from $a1$ are inside nodes. Those nodes do not change at binding time as they were created by the caller itself and are not place holders for unknown objects. Thus, no more context is necessary to solve the binding between $a1$ and $p1$. However, $a2$ can reach the load node $L4$ meaning that more context might be necessary to resolve nodes reachable from $a2$. That is why we convert $L4$ to an ω node. Full details on the modified computation for the inter-procedural mapping μ is in [4].

We also modify the operation that models field dereference to support the $?$ and $\$$ edges. It considers those edges as "wild cards" allowing every field dereference to follow those edges.

3. Annotations

Table 1 summarizes our annotation language. The annotations provide concise information about points-to and effect information and allows us to mitigate the effect of non-analyzable calls. Annotating a method as pure is the same as marking each parameter as not being writable (unless it is an out parameter). A method annotated as being write-confined is shorthand for marking every parameter as write-confined. Obviously not all combinations of the attributes are allowed. For example, it would be contradictory to label a method as being both pure and as writing globals.

The full details for mapping the attributes into points-to and write effect information are found in [4]. Basically their impact is to a) remove $?$ edges, b) replace ω nodes by inside nodes, and c) avoid registering write effects over parameters or the global scope.

We explain the effect of the annotations using some of the methods in our running example. Figure 7 presents the full list of annotations. The $GetEnumerator$ method returns an object that is modified later on in $Copy$. Notice that the loop would never terminate unless $iter.MoveNext$ returns false at some point. So either the loop never executes or else some state somewhere must change so that a different value can be returned. If the state change involves global objects, then $Copy$ is not pure so let us assume that the change is to the object $iter$ itself. As long as that object was allocated by $GetEnumerator$, changes to it would not violate the weak purity of $Copy$. We expect $GetEnumerator$ to return a fresh object: the iterator. At the same time, it is likely that the returned iterator has a reference to the collection. We need a way to distinguish the write effects in $MoveNext$ so that we do not conclude that it modifies the collection.

Figure 6 shows the points-to graph for $GetEnumerator$. It corresponds to the following annotations.

- The return value is annotated as $Fresh$. This generates the inside node for the return value instead of an ω node.
- The receiver ($this$ variable) is annotated as $Escapes$ which means that the points-to graph must introduce edges from the nodes reachable from outside (in this case the return value) to the receiver. Note that we do not annotate it as $Capture$. This is why the edge between the return value and the collection is labeled as $\$$ which means that the receiver is reachable from outside but only for reading. A $Capture$ annotation would generate a $?$ edge. There are no edges starting from the ω node pointed by $\&this$ because of the default annotation for the receiver as $Write(false)$.
- The method is annotated as not accessing globals. This means that there is no global node (and so no write or read effects on the global state).

We believe these are reasonable constraints on the behavior of $GetEnumerator$. The points-to graph for $MoveNext$ is also shown in Figure 6. It corresponds to these annotations:

- The method is annotated as $WriteConfined$, which means that it can only mutate objects it owns. This is represented using an ωC node for the receiver. Note how this is implemented. The parameter node has two edges. The edge labeled as $?$ which leads back to the reciever means that the method can perform any write to nodes in its ownership cone. The other edge labeled as $\$$ leads to a separate ω node. That means that objects reachable using not-owned fields can be read but not modified. Thus, edges labeled as $ do not need to be considered when computing write effects for the method.

```
class List<T> {
  [GlobalAccess(false)]
  public List<T>();
  [GlobalAccess(false)]
  public void Add(T t);
  ...
}
interface IEnumerable<T>{
  [return: Fresh]
  [Escapes(true)] // receiver spec
  [GlobalAccess(false)]
  IEnumerator<T> GetEnumerator();
}
interface IEnumerator<T> {
  [WriteConfined] bool MoveNext();
  T Current { [GlobalAccess(false)] [Pure] get; }
  [WriteConfined] void Reset();
}
```

Figure 7. The methods needed for analyzing $Copy$ along with their annotations.

```
void m(A a){
  a = this;
  D d = new D();
  a.f = d;
}
```

$\mathcal{W}(\mathtt{m}) = \{\langle \mathtt{PLN(this)}, f \rangle\}$
$\mathcal{R}(\mathtt{m}) = \{\}$
$Write(m) = \{\mathtt{this.f}\}$
$Read(m) = \{\}$

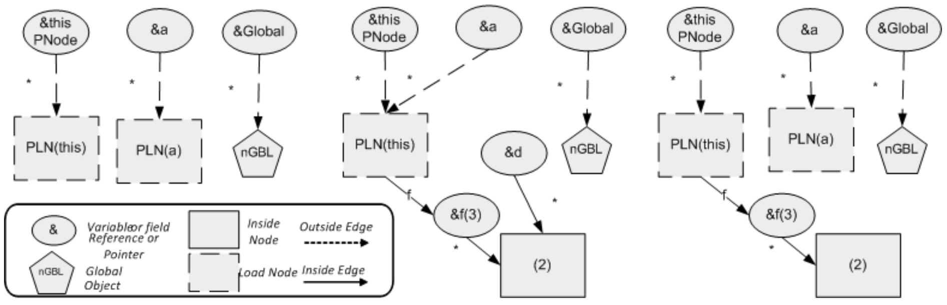


Figure 4. An example method, three points-to graphs for the beginning, end, and summary of the method and the read and write sets for the method. The latter is expressed both as the sets of nodes (`PLN` means *parameter load node*) and as the access paths.

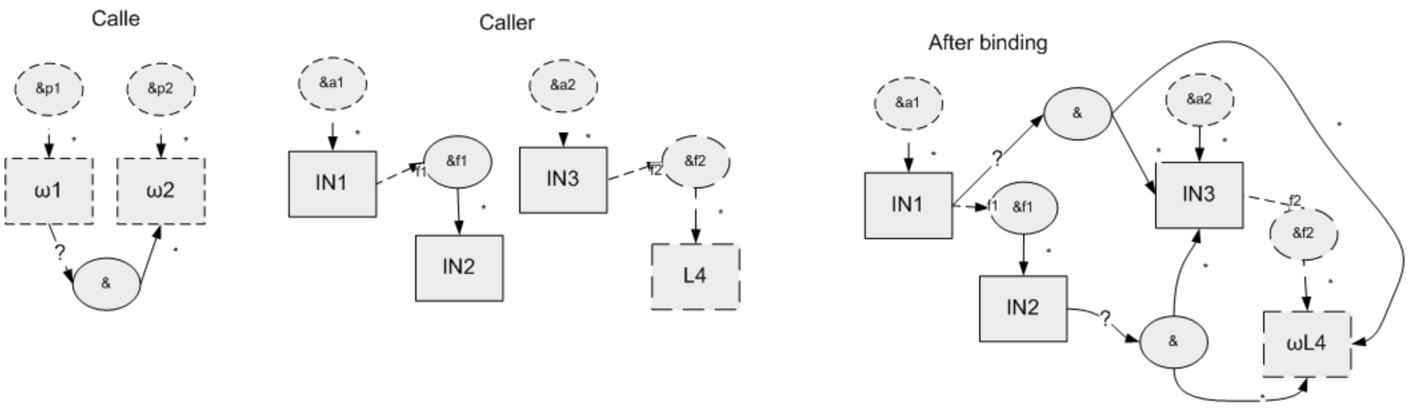


Figure 5. Effect of omega nodes in the inter-procedural mapping

Attribute Name	*Target*	*Default*	*Meaning*
Fresh	out Parameter	False	The returned value is a newly created object.
Read	Parameter	True	The content can be transitively read.
Write	Parameter	False	The content can be transitively mutated.
WriteConfined	Parameter	False	The content can transitively mutate only captured objects.
Escape(bool)	Parameter	False	Will any object reachable from the parameter be reachable from another object in addition to the caller’s argument
Capture(bool)	Parameter	False	Will some caller object own the escaping-parameter’s objects ?
GlobalRead(bool)	Method	True	Does the method read a global?
GlobalWrite(bool)	Method	True	Does the method write a global?
GlobalAcccess(bool)	Method	True	Does the method read or write a global?
Pure	Method	False	The method can not mutate any object from its prestate except for out parameters
WriteConfined	Method	False	The method mutates only objects owned by the parameters (captured).

Table 1. The set of attributes used to summarize the points-to graph and the read and write sets. The attributes $Fresh$ and $Escape$ also are allowed on the “return value” of the method since we model that as an extra (out) parameter. In C#, attributes on return values are specified at the method level with an explicit target, e.g., `[return:Fresh]`.

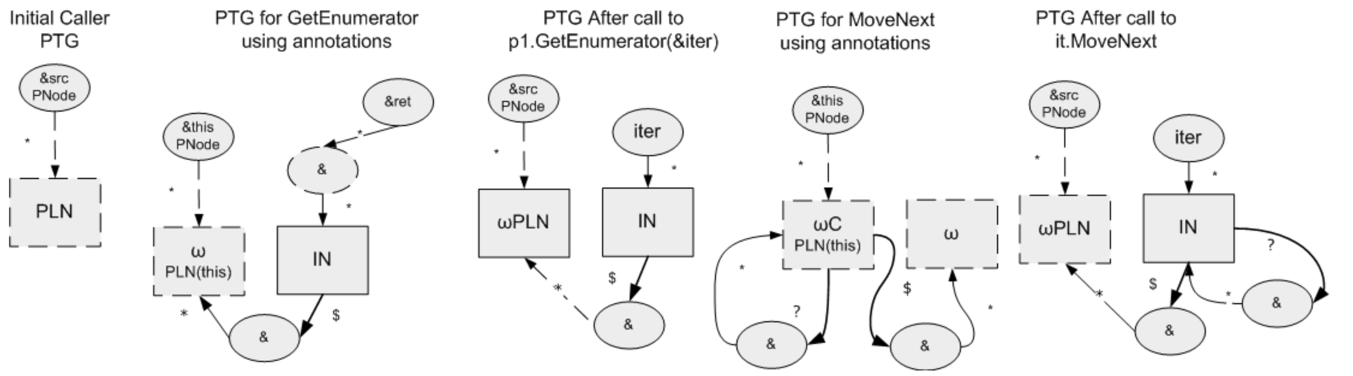

Initial Caller PTG
PTG for GetEnumerator using annotations
PTG After call to p1.GetEnumerator(&iter)
PTG for MoveNext using annotations
PTG After call to it.MoveNext
&src PNode
PLN
&this PNode
&ret
ω PLN(this)
IN
iter
ωPLN
ωC PLN(this)
ω
&
$
?
*

Figure 6. The evolution of $Copy$'s points-to graph after calling `src.GetEnumerator` and `iter.MoveNext`. We use the special field \$ to indicate that src is reachable from $iter$ but $iter$ is able to mutate objects only using fields that $iter$'s class owns. For simplicity we do not show the evolution of the newly created objects pointed to by the list l.

4. Experimental Results

Our implementation is integrated into the Spec# compiler pipeline and can also be run as a stand alone application. We analyze Boogie [3], a program verification tool for the Spec# language [2]. Boogie is itself written in Spec# and so already has some annotations. In this case we use our tool to verify methods annotated as pure. We analyzed the eight application modules using three different approaches. *Intra-procedural:* We analyze each method body independently. In the presence of method calls we use any annotations provided by the callee. *Inter-procedural (bottom up with fixpoint):* This is a whole program analysis. We compute a partial call graph and analyze methods in a bottom up fashion in order to have the callee precomputed before any calls to that method. To deal with recursive calls we perform a fixpoint computation over the strongly connected graph of mutually recursive calls. *Inter-procedural (top down with depth 3):* Again, a whole program analysis with inline simulation. For every method we analyze call chains to a maximum length of three.

Table 2 contains the results for the three kinds of analysis. We show only modules that contain purity annotations. The intra-procedural analysis is only slightly less precise than the other two analyses. Furthermore, when using annotations with intra-procedural analysis, the precision is substantially better than a full inter-procedural analysis without annotations. For this application we don't find a big difference between the two inter-procedural analyses. This is because most of the methods are not recursive.

One interesting thing is that we found that many of the methods declared pure in Boogie were not actually pure. Some are observationally pure, but others either record some logging information in static fields, or else were just incorrectly annotated as being pure.

5. Related work

Our analysis is a direct extension of the points-to and effect analysis by Salcianu et al. [22]. We add support for a more complex memory model (managed pointers and structs) and provide a different approach for dealing with non-analyzable methods. Instead of assuming that every argument escapes and the method writes the global scope, we try to bound the effect of unknown callees using annotations. Using their analysis it is difficult to decide that a method is pure when it calls a non-analyzable method (e.g., the iterator example). One alternative is to generate by hand all the information about the callee (points-to and effects) but it has to be done for every implementation of an interface or abstract class. Our annotation language simplifies that task and allows us to verify the annotations when code becomes available.

Type and effect systems have been proposed by Lucassen et al. [17] for mostly functional languages. There has been a significant amount of work in specification and checking of effect information relying on user annotations. Clarke and Drossopoulou use ownership types [9] while Leino et al. use data groups [16]. In [14], an effect system using annotations is proposed: it allows effects to be specified on a field or set of fields (regions). It also has a notion of "unshared" fields that corresponds to our ownership system. Using a purely intra-procedural analysis, they verify methods against their annotations. However, it seems that it doesn't compute points-to-information. Compared to their approach, our annotation language is less precise, but still allows enough information about escaping and captured parameters. JML [15] and Spec# [2] are specification languages that allow specification of write effects. One of the aims of our technique is to assist the Spec# compiler in the verification and inference of the read and write effects. We use the purity analysis to check whether a method can be used in specifications. Javari [24] uses a type system to specify and enforce read-only parameters and fields. To cope with caches in real applications, Javari allows the programmer to declare mutable fields; such fields can be mutated even when they belong to a read-only object. Our technique computes weak purity so mutation of prestate objects are not allowed in methods. To automatically deal with caching writes, it is necessary to infer observationally pure methods [6].

Points-to information has also been used to infer side effects [21, 19, 8, 7]. Our analysis, as well as Salcianu's analysis [22], is able to distinguish between objects allocated by the method and objects in the prestate. This enables us to compute weak purity instead of only strong purity. In more recent work, Cherem and Rugina [7] present a new inter-procedural analysis that generates method signatures that give information about effects and escaping information. It allows control of the heap depth visibility and field branching, which permits a tradeoff between precision and scalability. Our analysis also computes method summaries containing read and write effect information that are comparable with the signatures computed by their analysis but our technique is able to deal with non-analyzable library methods with a concise set of annotations that can be checked when code is available. AliasJava [1] is an annotation language and a verification engine to describe aliasing and escape information in Featherweight Java. Our work also uses annotations to deal with escape, aliasing and some ownership information but also some minimal description about read and write effects in order to compensate for information lacking at non-analyzable calls. Hua et al. [20] proposed a technique to compute points-to and effect information in the presence of dynamic loading. Instead of relying on annotations, they only compute information for elements that may not be affected by dynamic loading and warn about the others.

6. Conclusions and Future Work

We have implemented an extension to Salcianu's analysis [22] that works on the complete .NET intermediate language CIL. The extensions involve several non-trivial details that enable it to deal

Project	#Meths	DP	Using Annotations						Without Annotations					
			Intra	%	Inter 3	%	IF	%	Intra	%	Inter 3	%	IF	%
AbsInt	348	66	66	100%	66	100%	66	100%	51	77%	51	77%	51	77%
AIFramework	15063	3514	2702	77%	2725	77%	2730	78%	1631	46%	1688	48%	1688	48%
Graph	97	20	14	70%	14	70%	14	70%	10	50%	10	50%	10	50%
Core	9628	1326	1164	88%	1224	92%	1224	92%	709	53%	729	55%	729	55%
ByteCodeTrans	5564	984	781	79%	845	86%	863	88%	255	26%	297	30%	297	30%
VCGeneration	2050	187	171	91%	171	91%	171	91%	155	83%	155	83%	155	83%
Compiler Plugin	55	12	10	83%	10	83%	10	83%	8	66%	8	66%	8	66%

Table 2. Results for Boogie showing the number of methods annotated as pure that were verified as pure by our analysis. The "DP" (declared pure) column lists the number of methods in each module that were annotated as pure. The column labeled "Intra" shows the number of methods verified using the intra-procedural analysis, "Inter 3" the inter-procedural top-down analysis limited to a call-chain depth of three, and "IF" is the full bottom-up inter-procedural analysis.

with call-by-reference parameters, structs, and other features of the .NET platform. Our model provides a simple operational semantics for a useful part of CIL. Full details are presented in an accompanying technical report [4].

We have extended the previous analysis by including ω-nodes that model entire unknown sub-graphs. Together with our annotation language, this allows treatment of otherwise non-analyzable calls without losing too much precision.

The abstraction aspect of ω-nodes also holds the promise to improve the scalability of the analysis by enabling points-to graphs to be abstracted further than possible in the original analysis by Salcianu.

We believe our annotation system strikes the proper balance between precision and conciseness. The annotations are specifications that are useful not only for the analysis itself, but represent information programmers need to use an API effectively. Our technique needs to be very conservative when dealing with load nodes. We are planning to improve it by recomputing the set of egdes ($?$, $\$$, ω) when new nodes become available. We also plan to leverage type information to avoid aliasing between incompatible nodes.

Our annotation language appears to be general, but it was designed with our purity analysis in mind. It is possible to create a different set of annotations; our approach would work given a mapping from the set of annotations into points-to graphs. It is also possible to imagine the annotations being elements of the abstract domain themselves, instead of using a separate annotation laguage. Besides usability concerns for real programmers, it could make the verification of a method against its specification more difficult: our annotation language is intentionally simple enough to make the verification easy to perform.

One problematic aspect of the system is the necessity to introduce an ownership system. The concept of ownership certainly exists in real code, but the right formalization is not fully agreed upon. There are several different ownership systems in the literature and we believe the meaning of our annotations would work for any of them. For now, we have connected our annotations to the Spec# ownership system.

By relaxing the closed-world requirements so that we do not need full programs, we hope to enable the use of our system within real programming practice. In the future we hope to present results from some real-world case studies.

There are other uses for a points-to and effect analysis besides method (weak) purity. In addition to using it for checking forms of observational purity, we have adapted the analysis for studying *method re-entrancy* [12]. It is also possible to use it for inferring and checking method *modifies clauses*.

Acknowledgements

We would like to thank the anonymous reviewers for their comments.

References

[1] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 311–330, New York, NY, USA, 2002. ACM Press.

[2] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[3] Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006.

[4] Mike Barnett, Manuel Fandrich, Diego Garbervetsky, and Francesco Logozzo. A read and write effects analysis for C#. Technical Report MSR-TR-2007-xx, Microsoft Research, April 2007. Forthcoming.

[5] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[6] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC 2004*, LNCS, pages 54–84. Springer, July 2004.

[7] Sigmund Cherem and Radu Rugina. A practical escape and effect analysis for building lightweight method summaries. In *CC 2007: 16th International Conference on Compiler Construction*, Braga, Portugal, March 2007.

[8] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1993. ACM Press.

[9] Dave G. Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *ACM SIGPLAN Notices*, 37(11):292–310, November 2002.

[10] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, June 2006.

[11] ECMA. Standard ECMA-335, Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/-ecma-335.htm, Ecma International, 2006.

[12] Manuel Fändrich, Diego Garbervetsky, and Wolfram Schulte. A reentrancy analysis for object oriented programs. In *FTfJP 2007:*

ECOOP Workshop on Formal Techiques for Java-like Programs, July 2007.

[13] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[14] Aaron Greenhouse and John Boyland. An object-oriented effects system. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *LNCS*, pages 205–229. Springer-Verlag, New York, June 1999.

[15] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[16] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. *SIGPLAN Notices*, 37(5):246–257, May 2002.

[17] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, New York, NY, USA, 1988. ACM Press.

[18] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.

[19] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[20] Phung Hua Nguyen and Jingling Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *ACSC '05: Proceedings of the Twenty-eighth Australasian conference on Computer Science*, pages 9–18, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[21] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 20–36, London, UK, 2001. Springer-Verlag.

[22] Alexandru Salcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*, January 2005.

[23] http://research.microsoft.com/specsharp/.

[24] Matthew S. Tschantz and Michael D. Ernst. Javari: adding reference immutability to Java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2005. ACM Press.

Ownership, Uniqueness and Immutability

Johan Östlund, Tobias Wrigstad
Royal Institute of Technology, Sweden
{johano, tobias}@dsv.su.se

Dave Clarke
CWI, Amsterdam, The Netherlands
dave@cwi.nl

Beatrice Åkerblom
Royal Institute of Technology, Sweden
beatrice@dsv.su.se

Abstract

Programming in an object-oriented language demands a fine balance between high degrees of expressiveness and control. At one level, we need to permit objects to interact freely to achieve our implementation goals. At a higher level, we need to enforce architectural constraints so that the system can be understood by new developers and can evolve as requirements change. To resolve this tension, numerous explorers have ventured out into the vast landscape of type systems expressing ownership and behavioural restrictions such as immutability. (Many have never returned.) This work in progress reports on our consolidation of the resulting discoveries into a single programming language. Our language, Joe$_3$, imposes little additional syntactic overhead, yet can encode powerful patterns such as fractional permissions, and the reference modes of Flexible Alias Protection.

1. Introduction

Recent years have seen a number of proposals put forward to add more structure to object-oriented programming languages, for example, via ownership types [12], or to increase the amount of control over objects by limiting how they can be accessed by other objects, via notions such as read-only or immutability. Immutability spans the following spectrum: *Class immutability* ensures that all instances of a class are immutable, for example, Java's String class; *object immutability* ensures that some instances of a class are immutable, though other instances may remain mutable; and *read-only*—or *reference immutability*—prevents modifications of an object via certain references, without precluding the co-existence of normal and read-only references to the same object.

Immutable objects help avoid aliasing problems and data races in multi-threaded programs [4, 18], and also enhance program understanding, as read-only or immutable annotations are verified to hold at compile-time [31]. According to Zibin et al. [35], immutability (including read-only references) can be used for modelling, verification, compile- and run-time optimisations, refactoring, test input generation, regression oracle creation, invariant detection, specification mining and program comprehension. Read-only references have been used in proposals to strengthen object encapsulation and manage aliasing. Kniesel and Theisen [21] use read-only references to allow and to manage side-effects due to aliasing. Noble, Vitek and Potter [27] introduce an *arg* reference mode to allow aggregates to rely only on immutable parts of external objects. Hogg's Islands [19] and Müller and Poetzsch-Heffter's Universes [24] use read-references to allow temporary representation exposure in a safe fashion.

1.1 Our Contributions

The programming language, Joe$_3$, we propose in this paper offers ownership and uniqueness to control the alias structure of object graphs, and lightweight effects and a mode system to encode various notions of immutability. It is a relatively straightforward extension of Clarke and Wrigstad's *external uniqueness* proposal (Joline) [14, 32] (without inheritance), and the syntactic overhead due to additional annotations is surprisingly small given the expressiveness of the language. Not only can we encode the three forms of immutability mentioned above, but we can encode something akin to the *arg* mode from Flexible Alias Protection [27], Fractional Permissions [7], and the context-based immutability of Universes [24], all the while preserving the owners-as-dominators encapsulation invariant. Furthermore, as our system is based on ownership types, we can distinguish between outgoing aliases to external, non-rep objects and aliases to internal objects and allow modification of the former (but not the latter) through a read-only reference.

Our system is closest in spirit to SafeJava [4], but we allow access modes on all owner parameters of a class, read-only references and an interplay between borrowing and immutable objects that can encode fractional permissions.

1.2 Why We Could Add Read-Only To Java (Almost)

In his paper "Why We Shouldn't Add Read-Only To Java (Yet)" [8], John Boyland criticises existing proposals for handling read-only references on the following points:

1. Read-only arguments can be silently captured when passed to methods;
2. A read-only annotation cannot express whether
   (a) the referenced *object* is immutable, and hence the reference can be safely stored;
   (b) a read-only reference is unique and thus immutable, as no aliases exist which could be used to mutate the object;
   (c) mutable aliases of a read-only reference can exist, implying that the referenced object should be cloned before used, to prevent it being modified underfoot resulting in *observational exposure*.[1]

Joe$_3$ addresses all of these problems. First, Joe$_3$ supports owner-polymorphic methods, which can express that a method does not capture one or all of its arguments. Second, we decorate owners with modes that govern how the objects owned by that owner will be treated in a context. Together with auxiliary constructs inherited from Joline, the modes can express immutability both in terms of 2.a) and 2.b), and read-only which permits the existence of mutable aliases (2.c). Moreover, Joe$_3$ supports fractional permissions—converting a mutable unique reference into several immutable references for a certain context. This allows safe representation exposure without the risk for observational exposure (2.c).

Joe$_3$ allows class, object and reference immutability. Unique references, borrowing and owner-polymorphic methods allow us to

[1] Observational exposure occurs when changes to state are observed through a read-only reference.

simulate fractional permissions and staged, external initialisation of immutable objects through auxiliary methods. As we base modification rights on owners (in the spirit of Joe_1's effects system), we achieve what we call *context-based* immutability, which is essentially the same kind of read-only found in Müller and Poetzsch-Heffter's Universes [24].

Joe_3 allows both read-only references and true immutables in the same language. This provides the safety desired by Boyland, but also allows coding patterns which do rely on observing changes in an object. Apart from the fact that we do not yet consider inheritance, which we believe to be a straightforward extension, we conclude that we could indeed add read-only to Java, now[2].

Outline Section 2 introduces the Joe_3 language through a set of motivating examples—different nestings of mutable and immutable objects, context-based immutability, immutable objects, and staged construction of immutables. Section 3 gives a brief formal account of Joe_3. Section 4 outlines a few simple but important extensions—immutable classes and Greenhouse and Boyland style regions [17]—describes how they further enhance the system and discusses how to encode the modes of Flexible Alias Protection [27]. Section 5 surveys related work not covered above. Section 6 contains an outlook for the future, and Section 7 concludes.

2. Meet Joe_3

In this section we describe Joe_3 with the help of a couple of motivating examples. Joe_3 is a class-based, object-oriented programming language with deep ownership, owner-polymorphic methods, ownership transfer through external uniqueness, an effects (revocation) system and a simple mode system which decorates owners with permissions to indicate how references with the annotated owners can be used. Beyond the carefully designed combination of features, the annotation of owners with modes is the main novelty in Joe_3. The modes indicate that a reference may be read or written (`+`) or only read (`-`), or that the reference is immutable (`*`). Read and immutable annotations on an owner in the class header represent a promise that the code in the class body will not change objects owned by that owner. The key to preserving and respecting immutability and read-only in Joe_3 is a simple effects system, rooted in ownership types, and inspired by Clarke and Drossopoulou's Joe_1 [11]. Classes, and hence objects, have rights to read or modify objects belonging to certain owners; only a minor extension to the type system of Clarke and Wrigstad's Joline [14, 32] is required to ensure that these rights are not violated.

The syntax of Joe_3 (shown in Figure 5) should be understandable to a reader with insight into ownership types and Java-like languages. Classes are parameterised with owners related to each other by an inside/outside nesting relation. An owner is a permission to reference the representation of another object. Class headers have this form:

```
class List<data outside owner> { ... }
```

Each class has at least two owner parameters, `this` and `owner`, which represent the representation of the current object and the representation of the owner of the current object, respectively. In the example above, the `List` class has an additional permission to reference objects owned by `data`, which is nested outside `owner`. Types are formed by instantating the owner parameters, `this:List<owner>`. An object with this type belongs to the representation of the current object and has the right to reference objects owned by `owner`. There are two nesting relations between owners, inside and outside. They exist in two forms each, one reflexive (`inside`/`outside`) and one non-reflexive (`strictly-inside`/`strictly-outside`). Thus, going back to our list example, a type `this:List<this>` denotes a list object belonging to the current representation, storing objects in the current representation.

A more detailed introduction is given in Section 3. Apart from ownership types, the key ingredients in Joe_3 are the following:

- (externally) unique types (written `unique[p]:Object`), a special *borrowing* construct for temporarily treating a unique type non-uniquely, and *owner casts* for converting unique references permanently into normal references.
- modes on owners—mutable '`+`', read-only '`-`', and immutable '`*`'. These appear on every owner parameter of a class and owner polymorphic methods, though not on types.
- an effects revocation clause on methods which states which owners will not be modified in a method. An object's default set of rights is derived from the modes on the owner parameters in the class declaration. An additional example of a use of `revoke` is found at the end of Section 4.2.

Annotating owners at the level of classes (that is, for all instances) rather than types (for each reference) is a trade-off. Rather than permitting distinctions to be made using modes on a per reference basis, we admit only per class granularity. Some potential expressiveness is lost, though the syntax of types does not need to be extended. Nonetheless, the effects revocation clauses regain some expressiveness that per reference modes would give. Another virtue of using per class rather than per reference modes is that we avoid some covariance problems found in other proposals (see related work) as what you can do with a reference depends on the context and is not a property of the reference. Furthermore, our proposal is statically checkable in a modular fashion. We also need no run-time representation of the modes.

2.1 Motivating Examples

The following examples illustrate the range of constraints that can be expressed in Joe_3.

2.1.1 A Mutable List With Immutable Contents

The code in Figure 1 shows parts of an implementation of a list class. The owner parameter `data` is decorated with the mode read-only (denoted '`-`'), indicating that the list will never cause write effects to objects owned by `data`.

The owner of the list is called `owner` and is implicitly declared. The method `getFirst()` is annotated with `revoke owner`, which means that the method will not modify the object or its transitive state. This means the same as if `owner-` and `this-` would have appeared in the class head. This allows the method to be called in objects where the list owner is read-only.

This list class can be instantiated in four different ways, depending on the access rights to the owners in the type held by the current context:

- both the list and its data objects are immutable, which only allows `getFirst()` to be invoked, and its resulting object is immutable;
- both are mutable, which imposes no additional restrictions;

[2] While the syntactic price of our proposal is no doubt steep when added to Java, adding it to a language with ownership types and uniqueness, such as Joline, is virtually for free. On a side-note, the authors believe that if ownership types is ever to make it into mainstream languages, simple but powerful extensions, such as external uniqueness, read-only references and immutability will be crucial in convincing programmers of the virtues of the added complexity.

```
class Link<data- strictly-outside owner> {
  data:Object obj = null;
  owner:Link<data> next = null;
}

class List<data- strictly-outside owner> {
  this:Link<data> first = null;

  void addFirst(data:Object obj) {
    this:Link<data> tmp = new this:Link<data>();
    tmp.obj = obj;
    tmp.next = this.first;
    this.first = tmp;
  }

  void filter(data:Object obj) {
    this:Link<data> tmp = this.first;
    if (tmp == null) return;
    while (tmp.next != null)
      if (tmp.next.obj == obj)
        tmp.next = tmp.next.next;
      else
        tmp = tmp.next;
    if (this.first != null && this.first.obj == obj)
      this.first = this.first.next;
  }

  data:Object getFirst() revoke owner {
    return this.first.obj;
  }
}
```

Figure 1. Fragment of a list class. As the `data` owner parameter is declared read-only (via '`-`') in the class header, no method in `List` may modify an object owned by `data`. Observe that the syntactic overhead is minimal for an ownership types system.

- the list is mutable but the data objects are not, which imposes no additional restrictions, though `getFirst()` returns a read-only reference; and
- the data objects are mutable, but the list not, which only allows `getFirst()` to be invoked, though the resulting object is mutable.

The last form is interesting and relies on the fact that we can specify, thanks to ownership types, that the data objects are not part of the representation of the list. Most existing proposals for read-only references (*e.g.*, Islands [19], JAC [20, 21], ModeJava [28, 29], Javari [31], and IGJ [35]) cannot express this constraint in a satisfactory way, as these proposals cannot distinguish between an object's outside and inside.

2.1.2 Context-Based Read-Only

As shown in Figure 2, different clients of the list can have different views of the same list at the same time. The class `Reader` does not have permission to mutate the list, but has no restrictions on mutating the list elements. Dually, the `Writer` class can mutate the list but not its elements.

As owner modes only reflect what a class is allowed to do to objects with a certain owner, `Writer` can add data objects to the list that are read-only to itself and the list, but writable by `Example` and `Reader`. This is a powerful and flexible idea. For example, `Example` can pass the list to `Writer` to filter out certain objects in the list. `Writer` can then consume or change the list, or copy its contents to another list, *but not modify them*. `Writer` can then return the list to `Example`, without `Example` losing its right to modify the objects obtained from the returned list. This is similar

```
class Writer<o+ outside owner, data- strictly-outside o> {
  void mutateList(o:List<data> list) {
    list.addFirst(new data:Object());
  }
}

class Reader<o- outside owner, data+ strictly-outside o> {
  void mutateElements(o:List<data> list) {
    list.elementAt(0).mutate();
  }
}

class Example {
  void example() {
    this:List<world> list = new this:List<world>();
    this:Writer<this, world> w =
                          new this:Writer<this, world>();
    this:Reader<this, world> r =
                          new this:Reader<this, world>();
    w.mutateList(list);
    r.mutateElements(list);
  }
}
```

Figure 2. Different objects can have different views of the same list at the same time. `r` can modify the elements of `list` but not the `list` itself, `w` can modify the `list` object, but not the list's contents, and instances of `Example` can modify both the list and its contents.

to the context-based read-only in Universes-based systems [24, 26]. In contrast, however, we do not allow representation exposure via read-only references.

2.1.3 Borrowing Blocks and Owner-polymorphic Methods

Before moving on to the last two examples, we need to introduce borrowing blocks and owner-polymorphic methods [13, 32, 10], which make it easier to program using unique references and ownership. (The interaction between unique references, borrowing, and owner-polymorphic methods has been studied thoroughly by Clarke and Wrigstad [14, 32].) A borrowing block has the following syntax:

$$\texttt{borrow}\ \mathit{lval}\ \texttt{as}\ \alpha\, x\ \texttt{in}\ \{\ s\ \}$$

The borrowing operation destructively reads a unique reference from an l-value (*lval*) to a non-unique, stack-local variable (x) for the scope of the borrowing block (s). The block also introduces a fresh block-local owner that becomes the new owner of the borrowed value. Every type of every variable or field that stores an alias to the borrowed value must have this owner in its type. Clearly, this is not the case for any pre-existing field or variable. Owner-polymorphic methods (see below) allow granting permissions to reference the borrowed value for the duration of a method call. This is the only way in which references to borrowed values can be exported to outside a borrowing block. As all method calls in the borrowing block must have returned when the block exits, clearly no residual aliasing can exist. Thus, when the borrowing block exits, the borrowed value can be reinstated and is once again unique.

Due to the strong encapsulation of external uniqueness, borrowing borrows an entire unique aggregate in one single hit and makes it stack-local.

An owner-polymorphic method is simply a method which takes owners as parameters. The methods `m1` and `m2` in `Client` in Figure 3 are examples of such. Owner-polymorphic methods can be seen as accepting stack-local permissions to reference (and possibly mutate) objects that it otherwise may not be allowed to reference. Owner parameters (p`*` and p`-` in the methods in Figure 3) of

```
class Client {
  <p* inside world> void m1(p:Object obj) {
    obj.mutate(); // Error
    obj.toString(); // Ok
    // assign to field is not possible
  }

  <p- inside world> void m2(p:Object obj) {
    obj.mutate(); // Error
    obj.toString(); // Ok
  }
}

class Fractional<o+ outside owner> {
  unique[this]:Object obj = new this:Object();

  void example(o:Client c)  {
    borrow obj as p*:tmp in {   // **
      c.m1(tmp);                // ***
      c.m2(tmp);                // ****
    }
  }
}
```

Figure 3. An implementation of fractional permissions using borrowing and unique references.

owner-polymorphic methods are not in the scope at the class level. Thus, method arguments with such a parameter in its type cannot be captured within the method body (—it is *borrowed* [6]).

2.1.4 Immutability

The example in Figure 2 shows that a read-only reference to an object does not preclude the existence of mutable references to the same object elsewhere in the system. This allows observational exposure—for good and evil.

The immutability annotation '*' imposes all the restrictions a read-only type has, but it also guarantees that no aliases with write permission exist in the system. Our simple way of creating an immutable object is to move a *mutable* unique reference into a variable with immutable type, just as in SafeJava [4].

This allows us to encode fractional permissions and to do staged construction of immutables, both discussed below.

2.1.5 Fractional Permissions

The example in Figure 3 shows an implementation of Fractional Permissions. We can use Joline's borrowing construct to *temporarily* move a mutable unique reference into an immutable variable (line **), freely alias the reference (while preserving read-only) (lines *** and ****), and then implicitly move the reference back into the unique variable again and make it mutable. This is essentially Boyland's Fractional Permissions [7]. As stated above, both the owner-polymorphic methods and the borrowing block guarantee not to capture the reference. A borrowed reference can be aliased any number of times in any context to which it has been exported, without the need to keep track of "split permissions" [7] as we know for sure that all permissions to alias the pointer are invalidated when the borrowing block exits. The price of this convenience is that the conversion from mutable to immutable and back again must be done in the same place.

Interestingly, `m1` and `m2` are equally safe to call from `example`. Both methods have revoked their right to cause write effects to objects owned by `p`, indicated by the `*` and `-` annotations on `p`, respectively. The difference between the two methods is that the first method knows that `obj` will not change under foot (making it safe to, for example, use `obj` as a key in a hash table), whereas the second method cannot make such an assumption.

```
class Client<p* outside owner, data+ strictly-outside p> {
  void method() {
    this:Factory<p, data> f = new this:Factory<p, data>();
    p:List<data> immutable = f.createList();
  }
}

class Factory<p* inside world, data+ strictly-outside p> {
  p:List<data> createList() {
    unique[p]:List<data> list = new p:List<data>();
    borrow list as temp+ l in { // 2nd stage of construct.
      l.add(new data:Object());
    }
    return list--;   // unique reference returned
  }
}
```

Figure 4. Staged construction of an immutable list

2.1.6 Initialisation of Immutable Objects

An issue with immutable objects is that even such objects need to mutate in their construction phase. Unless caution is taken the constructor might leak a reference to `this` (by passing `this` to a method) or mutate other immutable objects of the same class. The standard solution to this problem in related proposals is to limit the construction phase to the constructor [31, 35, 18]. Continuing initialisation by calling auxiliary methods *after* the constructor returns is simply not possible. Joe$_3$, on the other hand, permits *staged construction*, as we demonstrate in Figure 4. In this example a client uses a factory to create an immutable list. The factory creates a unique list and populates it. The list is then destructively read and returned to the caller as an immutable.

3. A Formal Definition of Joe$_3$

In this section, we formally present the static semantics of Joe$_3$, and argue how it guarantees immutability and read-only.

3.1 Joe$_3$'s Static Semantics

We now describe Joe$_3$'s type system, which can be seen as a simplification of Joline's [14, 32] extended with effects annotations and modes on owners. To simplify the formal account, we omit inheritance and constructors. Furthermore, following Joline, we rely on destructive reads to preserve uniqueness and require that movement is performed using an explicit operation.

The abstract syntax of Joe$_3$ is shown in Figure 5. For simplicity, we assume that names of fields, method and classes are unique. c, m, f, x are metavariables ranging over names of classes, methods, fields and local variables, respectively. q and p are names of owners.

Types have the syntax $p\, c\langle \overline{p} \rangle$. We sometimes write $p\, c\langle \sigma \rangle$ for some type where σ is a map from the names of the owner parameters in the declaration of a class c to the actual owners used in the type. In code, a type's owner is connected to the class name with a ':' to make the type one syntactic unit.

Unique types have the syntax $\texttt{unique}_p\, c\langle \overline{p} \rangle$. The keyword `unique` specifies that the owner of an object is really the field or variable that contains the only (external) reference to it in the system. The owner annotation on the unique type is called the *movement bound*. Movement bounds govern the maximal outwards movement of a unique, so as to preserve the owners-as-dominators property. In code, movement bounds are denoted `unique[p]`. For details, see Wrigstad [32].

In systems with ownership types, an owner is a permission to reference objects with that owner. Classes, such as the canonical list example, can be parameterised with owners to enable them to

$$
\begin{array}{lll}
P & ::= \overline{C} & \text{(program)}\\
C & ::= \texttt{class}\ c\langle\overline{\alpha\,\mathsf{R}\,p}\rangle\ \texttt{\{}\ \overline{fd}\ \overline{md}\ \texttt{\}} & \text{(class)}\\
fd & ::= t\ f := e; & \text{(field)}\\
md & ::= \langle\overline{\alpha\,\mathsf{R}\,p}\rangle\ t\ m(\overline{t\ x})\ \texttt{revoke}E\ \texttt{\{}\ s; \texttt{return}\ e\ \texttt{\}} & \text{(method)}\\
e & ::= lval \mid lval\texttt{--} \mid e.m(\overline{e}) \mid \texttt{new}\ p\ c\langle\sigma\rangle \mid \texttt{null} & \text{(expr.)}\\
s & ::= lval := e \mid t\ x := e \mid s;s \mid e & \text{(statement)}\\
 & \quad \mid \texttt{borrow}\ lval\ \texttt{as}\ \alpha\ x\ \texttt{in}\ \texttt{\{}\ s\ \texttt{\}} & \\
lval & ::= x \mid e.f & \text{(l-value)}\\
\mathsf{R} & ::= \prec^* \mid \succ^* \mid \prec^+ \mid \succ^+ & \text{(nesting relation)}\\
t & ::= p\ c\langle\overline{p}\rangle \mid \texttt{unique}_p\ c\langle\overline{p}\rangle & \text{(type)}\\
E & ::= \epsilon \mid E, p & \text{(write right revocation clause)}\\
\Gamma & ::= \epsilon \mid \Gamma, x : t \mid \Gamma, \alpha\,\mathsf{R}\,p & \text{(environment)}\\
\sigma & ::= \overline{\alpha \mapsto p} & \text{(owner substitution)}\\
\alpha & ::= p\texttt{-} \mid p\texttt{+} \mid p\texttt{*} & \text{(owner param.)}
\end{array}
$$

Figure 5. Abstract syntax of Joe$_3$. In the code examples, owner nesting relations (R) are written as `inside` ($\prec^*$), or `strictly-inside` ($\prec^+$), etc. for clarity.

be given permission to access external objects. For example, the list class has an owner parameter for the (external) data objects of the list. In Joe$_3$ the owner parameters of a class or owner-polymorphic method also carry information about what effects *the current context may cause on* the objects having the owner in question. For example, if p`-` (p is read-only) appears in some context c, this means that c may reference objects owned by p, but not modify them directly. We refer to the part of an owner that controls its modification rights as the *mode*.

In contrast with related effect systems (*e.g.*, [17, 11]), we use effect annotations on methods to show what is *not* affected by the method—essentially *temporarily revoking* rights to change. For example, `getFirst()` in the list in Figure 1 does not modify the list object and is thus declared using a `revoke` clause thus:

```
data:Object getFirst() revoke owner { ... }
```

This will force the method body to type-check in an environment where `owner` (and `this`) are read-only.

Notation Given σ, a map from (annotated) owner parameters to actual owners, let σ^p mean $\sigma \uplus \{\texttt{owner+} \mapsto p\}$. For the type `this:List<owner>`, $\sigma = \{\texttt{owner+} \mapsto \texttt{this}, \texttt{data-} \mapsto \texttt{owner}\}$. We write $\sigma(p\,c\langle\overline{p}\rangle)$ to mean $\sigma(p)\ c\langle\sigma(\overline{p})\rangle$. For simplicity, we sometimes completely disregard modes and allow $\sigma(p)$. On the other hand, σ° denotes a mode preserving variant of σ *s.t.* if $q\texttt{+} \mapsto p \in \sigma$, then $q\texttt{+} \mapsto p\texttt{+}$ in σ°.

Let $\mathsf{md}(\alpha)$ and $\mathsf{nm}(\alpha)$ return the mode and owner name of α, respectively. For example, if $\alpha = p\texttt{+}$, then $\mathsf{md}(\alpha) = \texttt{+}$ and $\mathsf{nm}(\alpha) = p$.

CT is a class table computed from a program P. It maps class names to type information for fields and methods in the class body. $\mathsf{CT}(c)(f) = t$ means that field f in class c has type t. $\mathsf{CT}(c)(m) = \forall\,\overline{\alpha\,\mathsf{R}\,q}.\,\overline{t} \rightarrow t; E$ means that method m in class c have formal owner-parameters declared $\overline{\alpha\,\mathsf{R}\,q}$, formal parameter types $\overline{t}$, return type t and revoked rights E.

Predicate $\mathsf{isunique}(t)$ is true iff t is a unique type. $\texttt{owner}(t)$ returns the owner of a type, and $\mathsf{owners}(t)$ returns the owner names used in a type or a method type. Thus, $\texttt{owner}(p\,c\langle\overline{p}\rangle) = p$ and $\mathsf{owners}(p\,c\langle\overline{p}\rangle) = \{p\} \cup \overline{p}$.

E_c denotes the set of owners to which class c has *write* permission. For example, the list class in Figure 1 has $E_{\texttt{List}} = \{\texttt{owner}\}$, whereas the writer class in Figure 2 has $E_{\texttt{Writer}} = \{\texttt{owner}, \texttt{o}\}$. E_c is defined thus:

$$
E_c = \begin{cases} \{p \mid p\texttt{+} \in \overline{\alpha}\} \cup \{\texttt{owner}\} & \text{if } \texttt{class}\ c\langle\overline{\alpha\,\mathsf{R}\,\_}\rangle\ \texttt{\{}\ \_\ \texttt{\}} \in P \\ \bot & \text{otherwise} \end{cases}
$$

$\Gamma \vdash C$	Good class
$\Gamma \vdash fd$	Good field
$\Gamma \vdash md$	Good method
$\Gamma \vdash s; \Gamma'$	Statement s is wf under Γ and produces Γ'
$\Gamma \vdash e : t$	Expression e has type t under Γ
$\Gamma \vdash t$	Good type
$\Gamma \vdash E$	Good write right revocation clause
$\Gamma \vdash \alpha\,\mathsf{R}\,p$	Owner parameter α is R-related to p in Γ
$\Gamma \vdash \alpha\ perm$	Good owner parameter α
$\Gamma \vdash p$	Good owner
$\Gamma \vdash \diamond$	The environment Γ is well-formed

Table 1. Judgments in the Joe$_3$ formalisation.

$E \setminus E'$ denotes set difference. The judgments in the type system are summarised in Table 1.

Good Class

$$
\textsc{(class)}\quad
\frac{\begin{array}{c}\Gamma = \texttt{owner+} \prec^* \texttt{world}, \texttt{this+} \prec^+ \texttt{owner}, \overline{\alpha\,\mathsf{R}\,p}, \texttt{this} : t \\ t = \texttt{owner}\ c\langle\overline{\mathsf{nm}(\alpha)}\rangle \quad \Gamma \vdash \texttt{owner+} \prec^* \overline{\mathsf{nm}(\alpha)} \quad \Gamma \vdash \overline{fd} \quad \Gamma \vdash \overline{md}\end{array}}{\vdash \texttt{class}\ c\langle\overline{\alpha\,\mathsf{R}\,p}\rangle\ \texttt{\{}\ \overline{fd}\ \overline{md}\ \texttt{\}}}
$$

A class is well-formed if all its owner parameters are outside `owner`. This makes sure that a class can only be given permission to reference external objects and is key to preserving the owners-as-dominators property of deep ownership systems [10]. The environment Γ is constructed from the owners in the class header, their nesting relations and modes, plus `owner+` and `this+` giving an object the right to modify itself. Thus, class-wide read/write permissions are encoded in Γ, and must be respected by field declarations and methods.

Good Field, Good Method The function $\Gamma\ \mathsf{revoke}\ E$ is a key player in our system—it revokes the write rights mentioned in E, by converting them to read rights in Γ. It also makes sure that `this` is not writable whenever `owner` is not. For example, given $E = \{p\}$, we have $p\texttt{+} \notin \mathrm{dom}(\Gamma\ \mathsf{revoke}\ E)$, so if $\Gamma\ \mathsf{revoke}\ E \vdash s; \Gamma'$, s does not write to objects owned by p.

$$
\begin{array}{rcl}
\epsilon\ \mathsf{revoke}\ E & = & \epsilon \\
(\Gamma, x : t)\ \mathsf{revoke}\ E & = & (\Gamma\ \mathsf{revoke}\ E), x : t \\
(\Gamma, \alpha\,\mathsf{R}\,p)\ \mathsf{revoke}\ E & = & (\Gamma\ \mathsf{revoke}\ E), (\alpha\,\mathsf{R}\,p\ \mathsf{revoke}\ E) \\
(\alpha\,\mathsf{R}\,p)\ \mathsf{revoke}\ E & = & (\alpha\ \mathsf{revoke}\ E)\ \mathsf{R}\,p \\
p\texttt{-}\ \mathsf{revoke}\ E & = & p\texttt{-} \\
p\texttt{+}\ \mathsf{revoke}\ E & = & p\texttt{-},\ \text{if } p \in E \text{ else } p\texttt{+} \\
\texttt{this+}\ \mathsf{revoke}\ E & = & \texttt{this-},\ \text{if } \texttt{owner} \in E \text{ else } \texttt{this+} \\
p\texttt{*}\ \mathsf{revoke}\ E & = & p\texttt{*}
\end{array}
$$

$$
\textsc{(field)}\quad \frac{\Gamma \vdash e : t}{\Gamma \vdash t\ f := e}
\qquad
\textsc{(method)}\quad \frac{\begin{array}{c}\Gamma' = \Gamma, \overline{\alpha\,\mathsf{R}\,p} \quad \Gamma' \vdash E \\ (\Gamma'\ \mathsf{revoke}\ E), \overline{x : t} \vdash s; \Gamma'' \quad \Gamma'' \vdash e : t\end{array}}{\Gamma \vdash \langle\overline{\alpha\,\mathsf{R}\,p}\rangle\ t\ m(\overline{t\ x})\ \texttt{revoke}\ E\ \texttt{\{}\ s; \texttt{return}\ e\ \texttt{\}}}
$$

A field declaration is well-formed if its initialising expression has the appropriate type. The rules for good method is a little more complex: any additional owner parameters in the method header are added to Γ, with modes and nesting. Furthermore, the effect clause must be valid: *i.e.*, you can only revoke rights that you own.

Expressions The expression rules pretty much follow those of Joline extended to cater for effects.

$$\frac{\Gamma \vdash_{\mathsf{lv}} \mathit{lval} : t \qquad \neg\mathsf{isunique}(t)}{\Gamma \vdash \mathit{lval} : t}\ (\text{EXPR-LVAL}) \qquad \frac{\Gamma \vdash_{\mathsf{lv}} \mathit{lval} : t \quad \mathsf{isunique}(t) \qquad \mathit{lval} \equiv e.f \Rightarrow \Gamma \vdash e : p\, c\langle\sigma\rangle \land \Gamma \vdash p\texttt{+}\ \mathit{perm}}{\Gamma \vdash \mathit{lval}\texttt{--} : t}\ (\text{EXPR-LVAL-DREAD})$$

Destructively reading a field in an object owned by some owner p requires that p+ is in the environment.

$$\frac{x : t \in \Gamma}{\Gamma \vdash_{\mathsf{lv}} x : t}\ (\text{EXPR-VAR}) \qquad \frac{\Gamma \vdash e : p\, c\langle\sigma\rangle \quad \mathsf{CT}(c)(f) = t \qquad \texttt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \texttt{this}}{\Gamma \vdash_{\mathsf{lv}} e.f : \sigma^p(t)}\ (\text{EXPR-FIELD})$$

Judgements of the form $\Gamma \vdash_{\mathsf{lv}}$ *lval* : t deal with l-values.

In Joline, owner arguments to owner-polymorphic methods must be passed in explicitly. Here, we assume the existence of an inference algorithm to bind the names of the owner parameters to the actual arguments at the call site. This is σ_a in the rule.

$$\frac{\begin{array}{c}\Gamma \vdash e : p\, c\langle\sigma\rangle \quad \mathsf{CT}(c)(m) = \forall\, \overline{\alpha\, \mathsf{R}\, p}.\, \overline{t} \rightarrow t; E \quad \sigma' = \sigma^p \uplus \sigma_a \\ \Gamma \vdash \sigma'(\overline{\alpha\, \mathsf{R}\, p}) \quad \Gamma \vdash \sigma'^{\circ}(\overline{\alpha})\ \mathit{perm} \quad \Gamma \vdash \overline{e} : \sigma'(\overline{t}) \quad \Gamma \vdash \sigma'(t) \\ \Gamma \vdash \sigma'(E_c \backslash E) \quad \texttt{this} \in \mathsf{owners}(\mathsf{CT}(c)(m)) \Rightarrow e \equiv \texttt{this}\end{array}}{\Gamma \vdash e.m(\overline{e}) : t}\ (\text{EXPR-INVOKE})$$

By the first clause of (EXPR-INVOKE), method invocations are not allowed on unique types. The third clause creates a substitution from the type of the receiver (σ^p) and the implicit mapping from owner parameter to actual owner (σ_a). $\Gamma \vdash \sigma'^{\circ}(\overline{\alpha})$ *perm* makes sure that owner parameters that are writable and immutable are instantiated with writable or immutable owners respectively. Clauses six and seven ensure that the argument expressions have the correct types and that the return type is valid. Clause eight checks that the method's effects are valid in in the current context, and clause nine makes sure that any method with `this` in its type (return types, argument types or owners in the owner parameters declaration) can only be invoked with `this` as receiver—this is the standard static visibility constraint of ownership types [12].

$$\frac{\Gamma \vdash t}{\Gamma \vdash \texttt{null} : t}\ (\text{EXPR-NULL}) \qquad \frac{\Gamma \vdash p\, c\langle\overline{p}\rangle}{\Gamma \vdash \texttt{new}\ p\, c\langle\overline{p}\rangle : \texttt{unique}_p\ c\langle\overline{p}\rangle}\ (\text{EXPR-NEW})$$

By (EXPR-NULL), `null` can have any well-formed type. By (EXPR-NEW), object creation results in unique objects. (Without constructors, it is obviously the case that the returned reference is unique—see Wrigstad's dissertation [32] for an explanation why adding constructors is not a problem.)

Good Statements

$$\frac{x \neq \texttt{this} \quad x : t \in \Gamma \quad \Gamma \vdash e : t}{\Gamma \vdash x := e; \Gamma}\ (\text{STAT-LOCAL-ASGN}) \qquad \frac{\Gamma \vdash e : p\, c\langle\sigma\rangle \quad \mathsf{CT}(c)(f) = t \quad \Gamma \vdash e' : \sigma^p(t) \quad \Gamma \vdash p\texttt{+}\ \mathit{perm} \quad \texttt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \texttt{this}}{\Gamma \vdash e.f := e'; \Gamma}\ (\text{STAT-FIELD-ASGN})$$

In contrast to local variable update, assigning to a field requires write permission to the object containing the field.

$$\frac{\begin{array}{c}\mathit{lval} \equiv e.f \Rightarrow \Gamma \vdash e : q\, c'\langle\_\rangle \land \Gamma \vdash q\texttt{+}\ \mathit{perm} \\ \Gamma \vdash \mathit{lval} : \texttt{unique}_p\ c\langle\sigma\rangle \quad \Gamma, \alpha \prec^{+} p, x : \mathsf{nm}(\alpha)\, c\langle\sigma\rangle \vdash s; \Gamma\end{array}}{\Gamma \vdash \texttt{borrow}\ \mathit{lval}\ \texttt{as}\ \alpha\, x\ \texttt{in}\ \{\ s\ \}; \Gamma}\ (\text{STAT-BORROW})$$

In our system, unique references must be borrowed before they can be used as receivers of method calls or field accesses. The borrowing operation moves the unique object from the source l-value to a stack-local variable temporarily and introduces a fresh owner ordered strictly inside the unique object's movement bound. The new owner is annotated with a read/write permission which must be respected by the body of the borrowing block. As the owner of the borrowed unique goes out of scope when the borrowing block exits, all fields or variables with types that can refer to the borrowed object become inaccessible. Thus, the borrowed value can be reinstated and is once again unique. As borrowing temporarily nullifies the borrowed l-value, the same requirements as (EXPR-DREAD) applies with respect to modifying the owner of the l-value.

$$\frac{\Gamma \vdash s; \Gamma' \quad \Gamma' \vdash s'; \Gamma''}{\Gamma \vdash s; s'; \Gamma''}\ (\text{STAT-SEQUENCE}) \qquad \frac{\Gamma \vdash e : t \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash t\ x := e; \Gamma, x : t}\ (\text{STAT-DECL})$$

Statements can be chained together in the obvious fashion. Local variable declaration and initialisation is straightforward.

Good Effects Clause

$$\frac{\forall p \in E.\, \Gamma \vdash p\texttt{+}\ \mathit{perm}}{\Gamma \vdash E}\ (\text{GOOD-EFFECT})$$

An effects clause is well-formed if it only revokes write permissions in the current environment.

Good Environment

$$\frac{}{\epsilon \vdash \diamond}\ (\text{GOOD-EMPTY}) \qquad \frac{\Gamma \vdash q \quad p \notin \mathrm{dom}(\Gamma) \quad \dagger \in \{\texttt{+}, \texttt{-}, \texttt{*}\}}{\Gamma, p\dagger\, \mathsf{R}\, q \vdash \diamond}\ (\text{GOOD-R}) \qquad \frac{\Gamma \vdash t \quad x \notin \mathit{dom}(\Gamma)}{\Gamma, x : t \vdash \diamond}\ (\text{GOOD-VARTYPE})$$

The rules for good environment require that owner variables are related to some owner already present in the environment or `world`, and that added variable bindings have types that are well-formed under the preceding environment.

Good Permissions and Good Owner By (WORLD), `world` is a good owner and is always writable. By (GOOD-α), a permission is good if it is in the environment. By (GOOD-p-), a read mode of objects owned by some owner p is good if p with any permission is a good permission—write or immutable implies read.

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{world+}\ \mathit{perm}}\ (\text{WORLD}) \qquad \frac{\Gamma \vdash \diamond \quad \alpha \in \mathit{dom}(\Gamma)}{\Gamma \vdash \alpha\ \mathit{perm}}\ (\text{GOOD-}\alpha)$$

$$\frac{\Gamma \vdash p\dagger\ \mathit{perm} \quad \dagger \in \{\texttt{+}, \texttt{*}\}}{\Gamma \vdash p\texttt{-}\ \mathit{perm}}\ (\text{GOOD-}p\text{-}) \qquad \frac{\Gamma \vdash \alpha\ \mathit{perm}}{\Gamma \vdash \mathsf{nm}(\alpha)}\ (\text{GOOD-OWNER})$$

Good Nesting We can easily define judgements $\Gamma \vdash p \prec^* q$ and $\Gamma \vdash p \prec^+ q$ as the reflexive transitive closure and the transitive closure, respectively, of the relation generated from each $\alpha\, \mathsf{R}\, p \in \Gamma$, where $\mathsf{R} \in \{\prec^*, \prec^+\}$ or $\mathsf{R}^{-1} \in \{\prec^*, \prec^+\}$, combined with $p \prec^*$ `world` for all p.

Good Type

$$\frac{\begin{array}{c}q\texttt{*} \in \overline{\alpha} \Rightarrow \sigma^{\circ}(q\texttt{*}) = p\texttt{*},\ \text{for some } p \\ \texttt{class}\ c\langle\overline{\alpha\, \mathsf{R}\, p}\rangle\ \{\ \ldots\ \} \in P \\ \Gamma \vdash \sigma^p(\overline{\alpha\, \mathsf{R}\, p}) \quad \Gamma \vdash \sigma^{p\circ}(\overline{\alpha})\ \mathit{perm}\end{array}}{\Gamma \vdash p\, c\langle\sigma\rangle}\ (\text{TYPE})$$

This rule checks that the owner parameters of the type satisfy the ordering declared in the class header, as well as checking that all the permissions are valid in the present context. In addition—and this is a subtle point—if an owner parameter in the class header was declared with the mode immutable, then the owner that instantiates the parameter in the type must also be immutable. Without this requirement, one could pass non-immutable objects where immutable objects are expected.

On the other hand, we allow parameters with read to be instantiated with write and *vice versa*. In the latter case, only methods that do not have write effects on the owners in question may be invoked on a receiver with the type in point.

$$\frac{\Gamma \vdash p\, c\langle\sigma\rangle}{\Gamma \vdash \mathtt{unique}_p\, c\langle\sigma\rangle}\;(\textsc{unique-type})$$

By (UNIQUE-TYPE), a unique type is well-formed if a non-unique type with the movement bound as owner is well-formed.

To simplify the formal account, we chose to make loss of uniqueness explicit using a movement operation rather than making it implicit via subtyping and subsumption, as such a rule would require a destructive read to be inserted. Instead, we require conversion to be explicit, as in the following rule:

$$\frac{\Gamma \vdash e : \mathtt{unique}_q\, c\langle\sigma\rangle \quad \Gamma \vdash p \prec^* q}{\Gamma \vdash (p)\; e : p\, c\langle\sigma\rangle}\;(\textsc{expr-lose-uniqueness})$$

This "owner-cast" expression moves the contents of a unique into a subheap of some object or block (whatever the p owner corresponds to). This is well-formed if the expression has a unique type and if the movement bound of the type is outside the owner of the type cast to.

3.2 Brief Explanation of the System

In this section, we take a hands-on approach to showing how the system works by applying it to the example in Figure 1. For simplicity, we ignore everything that is not related to preserving read-only.

The key rules of the system are (METHOD), (EXPR-LVAL-DREAD), (STAT-FIELD-ASGN), (EXPR-INVOKE), and (STAT-BORROW).

In (METHOD), any write permissions revoked in the revocation clause E are removed from Γ. Thus, the method body must be well-typed under a restricted Γ.

Destructively reading, borrowing or assigning to a field in an object, (EXPR-LVAL-DREAD), (STAT-BORROW) and (STAT-FIELD-ASGN) requires a write permission to the object containing the field in the current context.

Method invocation is a little trickier. If a formal owner parameter requires write access, or that an object is immutable, the calling context must satisfy those requirements (by $\Gamma \vdash \sigma^\circ(\overline{\alpha})\ perm$). Furthermore, the current context must also have write permission to every owner in the set of owners to which the method is allowed to write ($E_c \setminus E$).

3.2.1 Type Checking Figure 1

By (CLASS), `addFirst()`, `filter()` and `getFirst()` must be well-formed under an environment $\Gamma = \mathtt{owner+} \prec^* \mathtt{world}, \mathtt{this+} \prec^+ \mathtt{owner}, \mathtt{data-} \succ^+ \mathtt{owner}, \mathtt{this} : \mathtt{owner}\, \mathtt{List}\langle\mathtt{data}\rangle$ for `List` to be a good class. By (METHOD), every statement in a method must be well-formed under Γ' equal to Γ extended by the formal parameters of the method and possible revocation of write rights. For `addFirst()`, $\Gamma' = \Gamma, \mathtt{obj} : \mathtt{data}\, \mathtt{Object}\langle\rangle$. We now look at the statements in `addFirst()`.

As we do not have constructors, (EXPR-NEW) does not care about permissions and is trivially well-formed with respect to write effects.

By (STAT-FIELD-ASGN), the second line of `addFirst()` requires that the owner of `tmp` is writeable under Γ', *i.e.,* it is in $\mathrm{dom}(\Gamma')$. By (EXPR-FIELD), the type of `tmp` is $\mathtt{this}\, \mathtt{Link}\langle\mathtt{owner}\rangle$. As `this+` $\in \mathrm{dom}(\Gamma')$, the field update is allowed. The next line follows the same pattern: reading a field is always allowed, and we have already established that we are allowed to assign to fields in `tmp`.

The last line of the method updates a field in `this`. By (STAT-FIELD-ASGN), `owner+` must be in $\mathrm{dom}(\Gamma')$, as the type of `this` is $\mathtt{owner}\, \mathtt{List}\langle\mathtt{data}\rangle$, which it is.

The method `filter()` type checks in a manner similar to that of `addFirst()`. However, `getFirst()` is different as it revokes the right to modify `owner` (and thus self). By (METHOD), the only line in `getFirst()` must type check under $\Gamma\, \mathsf{revoke}\, E$ where $E = \{\mathtt{owner}\}$. This is equivalent to $\mathtt{owner-} \prec^* \mathtt{world}, \mathtt{this-} \prec^+ \mathtt{owner}, \mathtt{data-} \succ^+ \mathtt{owner}, \mathtt{this} : \mathtt{owner}\, \mathtt{List}\langle\mathtt{data}\rangle$—the context has no write permissions. The field access is still allowed as reading fields does not require any write permissions.

3.2.2 Trapping Writes in a Read-Only Context

We now show how the system would trap an unpermitted write added to a method in the `List` class of Figure 1. Assume `Object` was defined thus:

```
class Object {
  this:Object state = null;
  void mutate() { this.state = null; }
}
```

and any of the methods in `List` included the line `this.first.data.mutate();`. Γ is the same as in the previous section.

The key to trapping this violation of read-only is the 8th clause in (EXPR-INVOKE). By (EXPR-FIELD) (applied two times), the type of `this.first.data`, the receiver of the mutating message, is $\mathtt{data}\, \mathtt{Object}\langle\rangle$.

$E_{\mathtt{Object}} = \{\mathtt{owner}\}$ (remember E_c returns the set of names of owners to which a class has write right) and $E = \epsilon$ as no rights are revoked. Consequently $E_{\mathtt{Object}} \backslash E = E_{\mathtt{Object}}$.

As `mutate` is not owner-polymorphic, σ_a is empty and thus $\sigma_2 = \{\mathtt{owner} \mapsto \mathtt{data}\}$ and $\sigma_2(E_{\mathtt{Object}} \backslash E) = \{\mathtt{data}\}$.

Thus, by the 8th clause of (EXPR-INVOKE), $\Gamma \vdash \mathtt{data+}$ must hold. By (GOOD-α), this amounts to $\mathtt{data+} \in \mathrm{dom}(\Gamma)$ which it clearly is not as we had $\mathtt{data-} \in \Gamma$ and `data-` and `data+` cannot occur simultaneously in Γ when Γ is well-formed.

Note that assignment to public fields is not allowed unless the receiver is `this`, which is why the modification had to be done through a method invocation.

3.3 Potentially Identical Owners with Different Modes

The list class in Figure 1 requires that the owner of its data objects is strictly outside the owner of the list itself. This allows for a clearer separation of the objects on the heap—for example, the list cannot contain itself.

The downside of defining the list class in this fashion is that it becomes impossible to store representation objects in a list that is also part of the representation. To allow that, the list class head must not use *strictly* outside:

```
class List< data- outside owner > { ... }
```

The less constraining nesting however leads to another problem: data and owner may be instantiated with the same owners. As data is read-only and owner is mutable, at face value, this might seem like a problem.

We choose to allow this situation as the context where the type appears might not care, or might have additional information to determine that the actual owners of data and the list do not overlap. If no such information is available, we could simply issue a warning. Of course, it is always possible to define different lists with different list heads for the two situations.

For immutables, this is actually a non-problem. The only way an immutable owner can be introduced into the system is through borrowing (or regions, see Section 4.2) where the immutable owner is ordered strictly inside any other known owner. As (TYPE) requires that write and immutable owner parameters are instantiated with owners that are write and immutable (respectively) in the context where the type appears, a situation where p+ and q* could refer to the same owner is impossible. As (TYPE) allows a read owner parameter to be instantiated with any mode, it is possible to have overlapping p- and q* in a context if a read owner was instantiated by an immutable at some point. Since objects owned by read owners will not be mutated, immutability holds.

3.4 Soundness of Joe$_3$

We have not formally proven soundness of Joe$_3$. Modulo omitting inheritance, the formal description of Joe$_3$ is a very simple and straightforward extension of that of Joline [32]. As modes have no run-time semantics, the crucial formal results of Joline should apply to Joe$_3$ as well — type soundness, owners as dominators and external uniqueness as dominating edges. Future work will extend the Joline formalism with modes and object-based regions and do the extra legwork to prove that the extended system is sound.

4. Extensions and Encodings

In this section we briefly discuss extensions to our system not included in the formalism, and the encoding of the modes from flexible alias protection.

4.1 Immutable Classes

In our system, an object always has permission to write to `owner` and `this` unless this permission is explicitly revoked in an effects clause for a specific method. Consequently, creating an immutable class requires every method to explicitly revoke its right to modify self. To relieve the programmer of this burden and to make a class' semantics clearer in the program text, we can introduce immutable classes through a class modifier:

```
immutable class String ...
```

The immutable class would be checked just as a regular class, but with the weaker permissions `owner*` and `this*` in Γ. Thus, methods that have write effects on `this` or `owner` would not type check. As fields may not be updated, except through `this`, this makes the object effectively immutable. To allow initialisation of immutable classes, the constructor would be allowed to initialise fields, similar to how final field initialisation is treated in Java.

4.2 Regions

In order to increase the precision of effects, we introduce explicitly declared regions, both at object-level and within method bodies. For simplicity, we have excluded regions from the formal account of the system. Object-based regions are similar to the regions of Greenhouse and Boyland [17] and the domains of Aldrich and Chambers [1], but we enable an ordering between them. Our method-scoped region construct is essentially the same as *scoped regions* in Joline [32], which is an object-oriented variant of classical regions [23, 30], adapted for use with ownership types.

Object-based regions As discussed in Section 3.3, defining the list class without the use of *strictly* outside places the burden of determining whether data objects are modified by changes to the list on the client of the list. This is because the list cannot distinguish itself from its data objects, as they (potentially) have the same owner.

By virtue of owners-as-dominators, an object that needs to keep rep objects in a list must include the list in the rep, or the list will not have the necessary permissions to reference the objects. As the list owner and data owner are the same, modifications to the list are indistinguishable from modification to its contents.

To tackle this problem and make our system more expressive, we extend Joe$_3$ with a regions system. A class declaration can contain any number of regions that each introduce a new owner nested strictly inside an owner in the scope. Thus, a class' rep is divided into multiple, disjoint parts (except for nested regions), and an object owned by one region cannot be referenced by another. The syntax for regions is `region` α `{` e `}`. Example:

```
class Example {
  this:Object datum;
  region inner+ strictly-inside this {
    inner:List<this> list;
  }

  void method() { list.add(datum); }
}
```

By virtue of the owner nesting, objects inside the region can be given permission to reference representation objects, but not vice versa as such types would not type check (*e.g.*, `this` is not inside `inner`). Thus, representation objects outside a region cannot reference objects in the region and consequently, effects on objects outside a region cannot propagate to objects in inside the region. In our example above, as there are no references from `datum` to the `list`, changes to `datum` cannot change the `list`.

Method-scoped regions The *scoped regions* construct in Joline [32] can be added to Joe$_3$ to enable the construction of method-scoped regions, which introduces a new owner for a temporary scope within some method body. Scoped regions allow the creation of stack-local objects which can be mutated regardless of what other rights exist in the context, even when `this` is read-only or immutable. Such objects act as local scratch space without requiring that the effects propagate outwards. The effects can be ignored.

The following line illustrates a pattern that occurs several times in the implementation of the Joline compiler:

```
<d- inside world> void method(d:Something arg)
  revoke this {
  region temp+ strictly-inside this {
    temp:Gamma<d> t = new temp:Gamma<d>();
    t.calculationsWithSideEffectsOnTemp(arg);
  }
}
```

Several times in the Joline compiler, we create a temporary object reminiscent of the type environment (Γ) to check whether certain addtions of owner nestings would be permitted. This object is completely temporary and its sole purpose is throwing an exception on an attempt at adding invalid owner nestings.

4.3 Encoding Modes from Flexible Alias Protection

In work [27] that led to the invention of Ownership Types, Noble, Vitek and Potter suggested a set of modes on references to manage the effects of aliasing in object-oriented systems. The modes were *rep*, *free*, *var*, *arg* and *val*. In this section, we indicate how these modes are (partially) encoded in our system.

The *rep* mode denotes a reference to a representation object that should not be leaked outside of the object. All ownership type systems encode *rep*; in ours, it is encoded as `this` $c\langle\sigma\rangle$.

The *free* expression holds a reference that is uncaptured by any variable in the system. This is encoded as `unique`$_p$ $c\langle\sigma\rangle$, a unique type. Any l-value of that type in our system is (externally) free.

The *var* mode denotes a mutable non-rep reference and is encoded as $p\, c\langle\sigma\rangle$, where `this` $\neq p$.

The *arg* mode is the most interesting of the modes. It denotes an argument[3] reference with a guarantee that the underlying object will not be (observably) changed under foot: "that is, *arg* expressions only provide access to the immutable interface of the objects to which they refer. There are no restrictions upon the transfer or use of *arg* expressions around a program" [27]. We support *arg* modes in that we can parameterise a type by an immutable owner in any parameter. It is also possible for a class to declare all its owner parameters as immutable to prevent its instances from ever relying on a mutable argument object that could change under foot. On the other hand, we do not support passing *arg* objects around freely—the program must still respect owners-as-dominators.

The final mode, *val*, is like *arg*, but it is attached to references with value semantics. These are similar to our immutable classes.

5. Related Work

Boyland et al.'s Capabilities for sharing [9] generalise the concepts of uniqueness and immutability. The system uses capabilities, which are pointers combined with a set of rights. What really distinguishes this proposal from other work is the exclusive rights which allow the revocation of rights of other references. Boyland et al.'s system can model uniqueness with the ownership capability. However, exclusive rights make the system difficult to check statically.

Table 2 summarises several proposals and their supported features. The systems included in the table represent the state of the art of read-only and immutable. In addition to Joe$_3$, our own proposal, the table includes (in order) SafeJava [4], Universes [24, 16, 15, 25], Jimuva [18], Javari [31], IGJ [35], JAC [21, 20] and ModeJava [28, 29]. SafeJava is probably the closest in spirit to our proposal, but the lack of crucial features, such as borrowing to immutables, makes it less powerful. We now discuss the different features covered in the table.

Expressiveness As discussed in Section 2.1.6, our system allows us to perform staged construction of immutable objects. This is also possible to do in SafeJava.

In our example in Figure 3, we show how we can encode fractional permissions [7]. Boyland suggests that copying rights may lead to observational exposure and proposes that the rights instead be split. Only the one with a complete set of rights may modify an object. SafeJava does not support borrowing to immutables and hence cannot model fractional permissions. It is unclear how allowing borrowing to immutables in SafeJava would affect the system, especially in the presence of inner classes which can break the owners-as-dominators property of deep ownership types.

In order to be able to retrieve writable objects from a read-only list, the elements in the list cannot be part of the list's representation. Joe$_3$, Universes, Jimuva and SafeJava can express this in a straightforward fashion, by virtue of ownership types. However, only our system, because of owner nesting information, can have two non-sibling lists sharing mutable data elements. Javari and IGJ have taken a more ad hoc course introducing mutable fields. It is possible in those systems to circumvent read-only if an object stores a reference to itself (or an object that does so) in a mutable field.

Flexible Alias Protection Modes The five alias modes proposed by Noble et al [27] were discussed in Section 4.3, where we also describe how these can be (partially) encoded in our system. Here we only describe how the modes have been interpreted for the purpose of the table (Table 2). The *rep* mode denotes a reference belonging to the representation of an object and should not be present in the interface. A defensive interpretation of *arg* is that all systems that have object or class immutability partially support *arg*,

[3] An object external to another object.

Feature	Joe$_3$	SafeJava [4]	Universes [24, 16, 15]	Jimuva [18]	Javari [31]	IGJ [35]	ModeJava [28, 29]	JAC [21, 20]
Expressiveness								
Staged constr. of immutables	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
Fractional permissions	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
Non rep fields	$\surd$	$\surd^1$	$\surd^1$	$\surd^1$	$\times^2$	$\times^2$	$\times$	$\times$
Flexible Alias Protection Modes								
arg	$\surd^3$	$\times^4$	$\times$	$\times^4$	$\times^4$	$\times^4$	$\times^4$	$\times$
rep	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$
free	$\surd$	$\surd$	$\surd^9$	$\times$	$\times$	$\times$	$\times$	$\times$
val [5]	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
var	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Immutability								
Class immutability	$\surd$	$\times$	$\times$	$\surd$	$\surd$	$\surd$	$\times^6$	$\times^6$
Object immutability	$\surd$	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\times$	$\times$
Read-only references	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$
Context-based immutability	$\surd$	$\times$	$\surd$	$\times$	$\times^7$	$\times^7$	$\times^7$	$\times$
Confinement and Alias Control								
Ownership types	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$
Owner-polymorphic methods	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$
Owners-as-modifiers	$\times$	$\times^8$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$
Unique references	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$

Table 2. Brief overview of related work. [1]) not as powerful as there is no owner nesting; two non sibling lists cannot share *mutable* data elements; [2]) mutable fields can be used to store a reference to `this` and break read-only; [3]) see Section 4.3; [4]) no modes on owners, and hence no immutable parts of objects; [5]) none of the systems deal with value semantics for complex objects; [6]) if all methods of a class are read-only the class is effectively immutable; [7]) limited notion of contexts via `this`-mutability; [8]) allows breaking of owners-as-dominators with inner classes and it is unclear how this interplays with immutables; [9]) support is forthcoming [25].

but only our system can have parts of an object being immutable. The *free* aliasing mode, interpreted as being equal to uniqueness, is supported by our system and SafeJava. None of the systems handle value semantics for complex objects and thus not the *val* mode (even though Javari include Java's primitive types in their system). The *var* aliasing mode expresses non-*rep* references which may be aliased and changed freely as long as they do not interfere with the other modes, for example, in assignments.

Immutability Immutability takes on three forms in *class immutability*, where no instance of a specific class can be mutable, *object immutability*, where no reference to a specific object can be mutable and *read-only* or *reference immutability*, where there may be both mutable and read-only aliases to a specific object.

Universes and our system provide what we call context-based immutability. In these two systems it is possible to create a writable list with writable elements and pass it to some other object to whom the elements are read-only. This other object may add new elements to the list which will be writable by the original creator of the list. The other systems in our table do not support this as they cannot allow *e.g.*, a list of writeables to be subsumed into a list of read-only references. In these systems, this practice could lead to standard covariance problems—adding a supertype to a list containing a subtype. Javari, IGJ and ModeJava all have a notion of `this`-mutable fields which inherit the mode of the accessing reference. This counts for some notion of context, albeit an ad hoc and inflexible one. In ModeJava a read-only list cannot return writable elements. In Javari and IGJ, this is only possible if the elements are stored in mutable fields, which causes other problems as discussed above.

Confinement and Alias Control Joe_3, SafeJava, Universes and Jimuva all support ownership types. This is what gives Joe_3 and Universes its context-based immutability. SafeJava and Jimuva, despite having ownership types, do not have context-based immutability due to their lack of read-only references. Universes is the only system supporting the owners-as-mutators property, meaning that representation exposure is allowed for read-only references.

Other approaches to confinement and alias control include Confined Types [3, 34], which constrain access to objects to from within their *package*. Bierhoff and Aldrich recently proposed a modular protocol checking approach [2] based on typestates. They partly implement Boyland's fractional permissions [7] in their access permissions.

Object-Oriented Regions and Effects systems Leino [22], Greenhouse and Boyland [17] and (to some degree) Aldrich and Chambers [1] take a similar approach to dividing objects into regions, and using method annotations to specify which parts of an object may be modified by a specific method. Adding effects to ownership, a la Clarke and Drossopoulou's Joe_1 [11], gives a stronger notion of encapsulation and enables more accurate description of effects. The addition of scoped regions to our system (*c.f.*, Section 4.2), combines both of these approaches.

Effect systems were first studied by Lucassen and Gifford [23] and Talpin and Jouvelot [30].

6. Future Work

6.1 Safe Representation Exposure

Müller and Poetzsch-Heffter's Universe system [24] allows representation exposure in a safe way (Boyland might disagree)—an object's representation can be exposed outside the object, but only via read-only references.

Extending our system to allow rep exposure for non-mutables will probably require an additional type that may only appear in contexts where its owner is read-only or immutable, but allows any valid owner in scope to be used as the owner of the type. This is a direction for future work.

6.2 Inheritance

Extending our system with inheritance is one of the next directions this research will take. We believe this to be a straightforward extension. Ownership, uniqueness and owner-polymorphic methods are already shown to work in the presence of inheritance [32], subtyping and downcasts [5, 33].

In the simplest way, for our Joe_3-specific extensions, an overriding method must revoke (at least) the same rights as the method it overrides, and argument and parameter types must be invariant and modes on owners must be preserved in subclasses.

7. Concluding Remarks

We have proposed Joe_3, an extension of Joe_1 and Joline with access modes on owners that can encode class, object and reference immutability, fractional permissions and context-based ownership with surprisingly little syntactical overhead. Future work will see a complete formalisation of the system, extended with inheritance and regions, including a dynamic semantics, and appropriate immutability invariants and soundness proofs.

References

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25, Oslo, Norway, Jan 2004. Springer Verlag.

[2] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. Submitted to OOPSLA 2007.

[3] Boris Bokowski and Jan Vitek. Confined Types. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1999.

[4] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Electrical Engineering and Computer Science, MIT, February 2004.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Safe runtime downcasts with ownership types. In Dave Clarke, editor, *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming*, UU-CS-2003-030. Utrecht University, July 2003.

[6] John Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 31(6):533–553, May 2001.

[7] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[8] John Boyland. Why we should not add readonly to Java (yet). *Journal of Object Technology*, 5(5):5–29, June 2006. Special issue: ECOOP 2005 Workshop FTfJP.

[9] John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2072, June 2001.

[10] David Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.

[11] David Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, November 2002.

[12] David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1998.

[13] David Clarke and Tobias Wrigstad. External uniqueness. In *10th Workshop on Foundations of Object-Oriented Languages (FOOL)*, New Orleans, LA, January 2003.

[14] David Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP)*, Darmstadt, Germany, 2003.

[15] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

[16] Werner Dietl and Peter Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[17] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, number 1628 in Lecture Notes in Computer Science, pages 205–229, Berlin, Heidelberg, New York, 1999. Springer.

[18] Christian Haack, Erik Poll, Jan Schäfer, and Aleksy Schubert. Immutable objects for a Java-like language. In Rocco De Nicola, editor, *16th European Symposium on Programming (ESOP'07)*,

volume 4421 of *LNCS*, pages 347–362. Springer, March 2007. Won the ETAPS award for the best theory paper at ETAPS.

[19] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, November 1991.

[20] Günter Kniesel and Dirk Theisen. JAC – Java with transitive readonly access control. In *Intercontinental Workshop on Aliasing in Object-Oriented Systems*, At ECOOP'99, Lisbon, Portugal, June 1999.

[21] Günter Kniesel and Dirk Theisen. JAC—access right based encapsulation for Java. *Software — Practice and Experience*, 31(6):555–576, May 2001.

[22] K. Rustan M. Leino. Data Groups: Specifying the Modification of Extended State. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1998.

[23] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1988.

[24] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.

[25] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007. To appear.

[26] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[27] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98—Object-Oriented Programming*, volume 1445 of *Lecture Notes In Computer Science*, pages 158–185, Berlin, Heidelberg, New York, July 1998. Springer-Verlag.

[28] Mats Skoglund and Tobias Wrigstad. A mode system for read-only references in Java. In *Formal Techniques for Java Programs, in Conjunction with ECOOP 2001*, Budapest, Hungary, 2001.

[29] Mats Skoglund and Tobias Wrigstad. Alias control with read-only references. In *Sixth Conference on Computer Science and Informatics*, March 2002.

[30] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[31] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[32] Tobias Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Department of Computer and Systems Science, Royal Institute of Technology, Kista, Stockholm, May 2006.

[33] Tobias Wrigstad and Dave Clarke. Existential owners for ownership types. *Journal of Object Technology*, 6(4):141–159, May–June 2007. Available from http://www.jot.fm/issues/issue_2007_03/article5.

[34] Tian Zhao, Jens Palsberg, and Jan Vitek. Type-based confinement. In *Journal of Functional Programming*, volume 15(6), pages 1–46, 2005.

[35] Yoav Zibin, Alex Potanin, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and reference immutability using Java generics. Technical Report MIT-CSAIL-TR-2007-018, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, March 16, 2007.

Ownership Meets Java

Christo Fogelberg
Victoria University of Wellington
cgf.vicmail@syntilect.com

Alex Potanin
Victoria University of Wellington
alex@mcs.vuw.ac.nz

James Noble
Victoria University of Wellington
kjx@mcs.vuw.ac.nz

Abstract

Ownership Generic Java (OGJ) is a language with ownership types as an extension to Java. In this position paper we outline the state of OGJ. We hope that the other aliasing and ownership researchers would benefit from the discussion around how to add ownership into a modern generic and annotation-capable typed object-oriented language like Java.

1. Introduction

With the lively state of ownership research [1, 3, 4, 8] a question comes up: "What stops us from adding ownership to Java today?" This position paper claims that there is nothing substantial which stops us from starting to use ownership today. The only problem is how it can be presented to the programming community to promote its usefulness.

OGJ [7] is a language with deep, reference-based ownership support. Over the recent months we have been working on resolving the remaining issues which arise when ownership and Java meet in an actual language implementation. Section 2 outlines these by dealing with statics, exceptions [5], arrays, equals and clone methods, and wildcards. The problem of arrays is a consequence of Java language design. The other problems are deeper issues which face many ownership systems, and we discuss them in the context of OGJ in this paper. Section 3 wonders what else we need to do before we could propose an ownership extension to Java (ultimately as a JSR).

The solutions described in this paper are in no way designed to be definitive, rather we pose a question to the community as to what could be the best alternatives to solving these issues.

2. Ownership Meets Java

2.1 Statics

Because static members cannot be owned by any instances or instance-associated ownership types it means that the possible owners are limited to `World`, `Package`, and `Class`. We propose that public static members be implicitly owned by `World`, that package-private static members be implicitly owned by `Package` and that protected and private static members be implicitly owned by `Class`.

Each of these approaches is, superficially, fairly straightforward, however, in the case of inheritance complexities can arise. This is because a public or protected static member of some class `C` which is owned by `<Class>` is not visible to any subclass of `C`, according to the original definition of OGJ.

We propose to modify the visibility rules in OGJ to take into account the inheritance hierarchy, in exactly the same way as the object typing rules do. For example, if `D extends C`, then `C.foo<Class>` would be visible and could be referred to by any instance of type `D`.

```
class Super<SuperOwner extends World> {
  public static Integer pub = 1; // assumes World
  static Integer pac = 2; // assumes Package
  protected static Integer pro = 3; // assumes Class
  private static Integer pri = 4; // assumes Class

  // Cannot use the instance type parameters or This
  // static String<SuperOwner> s = "illegal!";
}

class Sub<SubOwner extends World>
  extends Super<SubOwner> {

  public static String<Class>
    weird1 = "legal,␣but␣strange!";
  private static String<World>
    weird2 = "legal,␣but␣strange!";

  void DoFoo() {
    // Legal, because we made Class visible
    // to subclasses
    Sub.pro = 99;
  }
}
```

Figure 1. Static members and ownership.

Additionally, the presence of owner generic methods in OGJ allows for static methods to have additional owners supplied via generic parameters.

Figure 1 shows an example of statics in OGJ. The static fields gain an implicit owner based on their visibility. An explicit owner can be used as long as it doesn't contradict the name visibility of the static field.

2.2 Exceptions

Incorporating exceptions into an ownership type system raises two main problems: (1) the ownership of the exception itself, (2) and the possibility of leaking references. Ownership and exceptions were also addressed by Werner Dietl and Peter Müller [5].

An object throwing an exception could specify either that its owner was a global type like `World`, or that it was owned by one of the ownership types which were in scope. Specifying `World` would mean that non-`World` objects could not be assigned as fields to the reference. Similarly, using one of the other visible ownership types would mean that exceptions could not propagate very far. Thus exceptions would be either greatly weakened or method calls would be unduly restricted by static type checking. This would mean that many design idioms could not be expressed.

For OGJ we have chosen to introduce a special owner `Exception` (that is a subtype of `World`) to resolve this problem. All exceptions are owned by `Exception`. `Exception` is just like `World`, in that any object can refer to or create an object of type `Exception`. Using

```
class FooException <EOwner extends Exception>
  extends Throwable {

  public Foo<EOwner> causeOfException;

  public FooException(
    Foo<EOwner> causeOfException) {

    this.causeOfException = causeOfException;
  }
}

class A<Owner extends World>() {
  private Foo<This> f;

  public void SomeMethod() {
    throw new FooException<Exception>(f);
  }
}

class B<Owner extends World> {
  public static Foo<World> globalF;
  private Foo<This> myF;

  public void <AnotherOwner extends World>
    DoSomething(A<AnotherOwner> a) {

    Foo<This> localF;
    try {
      a.SomeMethod();
    } catch(FooException<Exception> e) {
      e.causeOfException.fix();

      // These lines cause compile-time errors:
      // B.globalF = e.causeOfException;
      // this.myF = e.causeOfException;
      // localF = e.causeOfException;
    }
  }
}
```

Figure 2. Exception handling and ownership.

a subtype of `World` marks off the relevant subset of global objects and allows them to be treated slightly differently.

In particular, we adopt the following conditions for `Exception`. Firstly, an object owned by `Exception` can only be created as part of a throw statement. Secondly we allow the ownership typing rules to be briefly broken during the exception throwing, so long as all other typing rules are not. This means that any object can be passed as a parameter to and used in the constructor of an `Exception`. In this situation the language implementation will only emit a warning saying that ownership will have been temporarily and locally broken.

The key advantages of this approach are as follows. Firstly, borrowing or uniqueness[6, 2] do not need to be introduced into OGJ just so that exceptions can be used. This minimises the learning curve for users of the language. Secondly, some exceptions in the API already expose references. An example of this is `omg.org.CORBA.portable.ApplicationException`. Rewriting exceptions like this to work in an ownership environment could also mean that the underlying architecture would need to be redesigned. Our approach maximises legacy interoperability and minimises code conversion costs.

Figure 2 shows an example of exception handling in OGJ. Inside method `SomeMethod` we can pass a field `f` owned by `This` to a publicly owned exception only because we are doing as part of the throw statement. Inside the catch block, the exception's fields can be referenced and modified via method calls, however copying the exception fields to variable not local to the catch block will generate a compile-time error.

```
class Foo<Owner> {}

class MyArrays<Owner extends World> {
  // What the specification claims is allowed:
  public Foo<Owner>[] myOtherArray = new Foo<?>[20];

  // What javac actually allows, they are equivalent:
  public Foo<Owner>[] myArray = new Foo[20];
}
```

Figure 3. Java arrays and ownership.

2.3 Arrays

In general, arrays are not a problem for ownership, however, some design decisions do need to be made. Although much of the discussion in this subsection is concerned with the details of implementing owned arrays with type erasure in Java, it generalises nicely to typed languages with generics.

We propose that arrays not have an owner [9]. Their one owner parameter refers to both the objects in the array and to the array itself. Although it is possible to give arrays their own owner, and while arrays have some minimal functionality added to them in Java there is very little that can be done with an array that does not directly involve its elements. This is not the case with collections. In our experience the added complexity of giving each array its own owner distinct from its elements' was not justified by the changes that would be required in Java. In the very rare cases of separate owner being useful, the use of a collection (e.g., `ArrayList` with appropriate owners) was more appropriate.

We consider compatibility with old Java virtual machines to be crucial. This means that, just like generics, ownership-checked programs must run on pre-ownership and pre-generic virtual machines. At the point of array creation, due to the restrictions imposed by Java's type erasure, there are really only two options: (1) allow ownership (but not generics) to be specified in the allocation statement and the type of the reference, or (2) only allow ownership in the type of the reference. Currently, if an approach like the second were used the language implementation would raise an "unchecked warning". We would recommend not raising a warning in the case of ownership types. This is because OGJ guarantees that the references will be properly type-checked.

These approaches work because of the way in which the Java type checks its programs. Aside from checking the type of the allocation statement at the time of construction against the type of the reference, Java otherwise type checks references against each other, not against the type of the underlying object. Because all references to array objects will include owner information as part of their type, ownership remains sound with arrays.

This means that if we can just get past the hurdle of object creation, owned arrays will be handled automatically by Java's type checking rules, just as existing generic arrays are handled. Even though these parameters will be erased by the language implementation, they are always type checked first.

Figure 3 shows an example of arrays in OGJ. These approaches work because although the object itself is unowned the only reference to the object is through its reference, which is properly type checked and which is forbidden by OGJ from having its owner cast away. In addition, the implementation still does the type substitution into the class.

```
class A<EO1 extends World, EO2 extends World,
  Owner extends World> {

  Foo<EO1, This> f1;

  boolean <O> equals(OObject<O> o) {
    if (o == this) return true;
    if (!(o instanceof List))
      return false; // compare to raw type

    A<OtherEO1, OtherEO2, O> oA =
     (A<OtherEO1, OtherEO2, O>) o;

    if(!this.f1.equals(oA.f1)) return false;
    else return true;
  }
}
```

Figure 4. Object comparison and ownership.

2.4 Equals

Equals and clone methods in OGJ suffer from the problem of not being able to have the same signature (and thus be overridden in the subclasses) due to a varying set of owners required by the method to perform equals or clone operation on various objects having access to many owners [8].

Since each object only has one "main" owner and the rest are simply those additional "outside" owners that it has access to, existential owners [8] shows why it is safe to sometimes lose track of the non-main owners and then gain them back by downcasting and introducing existential ownership types distinct from any other ownership types which are visible.

For example, this allows us to implement an equals method in OGJ as shown in Figure 4. Notice that downcasting introduces new owner type variables (e.g. `OtherEO1` and `OtherEO2`) following the existential ownership proposal [8]. The downcast uses owner `O` (coming via the method parameter `o`) which is not necessarily the same as the owner of the class `A` (`Owner`).

In Figure 4 class `A` has additional owners `EO1` and `EO2` in addition to the main owner `Owner`. Its equals method is a generic method that accepts the owner parameter `O` of the object being compared to (which maybe `This` for the other object). The equals method thus has access to two (potentially unrelated and private) objects in the object graph only for the duration of the method itself. Note that the equals method has the same signature for every class even though there may be multiple owners involved.

2.5 Clone

Figure 5 illustrates cloning in OGJ. For cloning, a method call to the assignee (which may have a new owner if one would like to change ownership of the clone) is required as shown in Figure 5. We call this *assisted cloning*. A C++-like copy constructor could be used in place of the special assistant method `assignClone`. For the sake of exposition we have omitted null and self-reference checks. The code as written would fail because of this but it illustrates the assistant method approach.

2.6 Wildcard bounds in OGJ

In this section we will discuss how wildcards and bounded wildcards are taken into account in OGJ. The Java type system makes this relatively easy.

An unbounded `?` can only be used as a wildcard in limited situations. This is because it is entirely anonymous. Although at compilation the static types of actual instances will be substituted in place of this wildcard and can be checked it cannot be referred to or used by the code. For example, a class specified with `?` as its owner could not use that owner in any of its method or field definitions.

```
class A<Owner extends World> {
  A<This> f; int i;

  <NewOwner> OObject<NewOwner> clone() {
    A<NewOwner> temp = new A<NewOwner>();
    temp.i = this.i;
    temp.f.<This>assignClone(this.f);
    return temp;
  }

  <NewOwner> void assignClone(A<NewOwner> a) {
    this = (A<Owner>) a.<Owner>clone();
  }
}
```

Figure 5. Object cloning and ownership, other version

Named bounded wildcards (e.g. `Owner super Package`) simply act as statically checkable restrictions on owner types which are valid for that method parameter, class or generic method. Note that an anonymous bounded wildcard (e.g. `? extends Package`) also faces the same restrictions as an unbounded anonymous wildcard and cannot be referred to in the code.

Because only `World`, `Package`, `Class` and `This` are ownership types known to exist, only a few bounds can be specified for a class which is not an inner class. For example, usefully. a class could have its owner specified as `Owner extends Package` or `Owner super Package`. In the first case it would mean that no instance of the class could be accessible outside the `Package`. In the latter it would mean that all instances of this class must be either `Package`-visible or `World`-visible. In both cases the valid owners are restricted to one of the globally defined types. This provides region-like capabilities.

In addition, because $\texttt{This}_l$ is a subtype of the owner of the object l, inner classes can be bound to have owners which are sub or super types of their enclosing classes owners. In relationships between unnested classes which are siblings in the class hierarchy it is impossible to express bounds like "the owner of `Foo` must be a subtype of the owner of `Bar`". This is because the owner variable of `Bar` is only visible to `Foo` if `Bar` is a supertype of `Foo` or if `Foo` is an inner class of `Bar`.

3. What Next?

In this position paper we presented our design choices for five corner cases in ownership language design: static ownership matching static visibility, global `Exception` owners for exceptions, ownership of array being the same as that of array's elements, existential owners for equals, and assisted clone implementation. We also discussed bounds on ownership types.

For ownership to be successful a large number of issues still remain to be resolved in a consistend and agreed upon manner in collaboration with other aliasing language researchers. These include formalising interaction between owned and *unowned* code, the choice between effective and reference-based ownership, using implicit or explicit owner parameters, adding additional features such as immutability or external uniqueness, developing a collection of language implementations, and writing fully ownership-aware collections and libraries.

An agreed upon compromise and implementation support would attract more users to the ownership-enabled languages and help resolve any issues stopping the ownership research proposal becoming a JSR reality.

Acknowledgments

This work is supported by the Victoria University of Wellington URF Grant Number 26214.

References

[1] Cameron, N., Drossopoulou, S., Noble, J., and Smith, M. Multiple ownership. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2007).

[2] Clarke, D., and Wrigstad, T. External Uniqueness is Unique Enough. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (Darmstadt, Germany, July 2003), vol. 2473 of *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag, Berlin, Heidelberg, Germany, pp. 176–200.

[3] Clifton, C., Leavens, G. T., and Noble, J. MAO: Ownership and effects for more effective reasoning about aspects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (2007).

[4] Dietl, W., Drossopoulou, S., and Müller, P. Generic universe types. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (2007).

[5] Dietl, W., and Müller, P. Exceptions in ownership type systems. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming (FTfJP)* (2004).

[6] Müller, P., and Poetzsch-Heffter, A. *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999, ch. Universes: a Type System for Controlling Representation Exposure. Poetzsch-Heffter, A. and Meyer, J. (editors).

[7] Potanin, A., Noble, J., Clarke, D., and Biddle, R. Generic ownership. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2006).

[8] Wrigstad, T., and Clarke, D. Existential owners for ownership types. *Journal of Object Technology* (May 2007). Accepted for publication.

[9] Zhao, T., Palsberg, J., and Vitek, J. Type-Based Confinement. *Journal of Functional Programming 16*, 1 (2006), 83–128.

2007 State of the Universe Address

Werner Dietl
ETH Zurich, Switzerland
Werner.Dietl@inf.ethz.ch
http://www.sct.inf.ethz.ch/

Peter Müller
Microsoft Research, USA
mueller@microsoft.com

Abstract

This position paper summarizes recent developments related to the Universe type system and suggests directions for future work.

1. Universe Type System

The Universe type system is an ownership type system that enforces the owner-as-modifier discipline. In this section, we summarize recent developments and suggest future work to improve the expressiveness and formal foundation.

1.1 Expressiveness

The Gang-of-Four design patterns are common design idioms for object-oriented programs. In [16], we compare how Ownership Types, Ownership Domains, and Universe Types handle these patterns. Based on this experience, we extended the Universe type system to support generics and ownership transfer.

Recent Developments. Generic Universe Types [4] extend Universe Types to generic types. Like Universe Types, Generic Universe Types enforce the owner-as-modifier discipline which does not restrict aliasing, but requires modifications of an object to be initiated by its owner.

Universe Types with Transfer [15] is an extension of Universe Types that supports ownership transfer. UTT combines ownership type checking with a modular static analysis to control references to transferable objects. UTT is very flexible because it permits temporary aliases, even across certain method calls. Nevertheless, it guarantees statically that a cluster of objects is externally-unique when it is transferred and, thus, that ownership transfer is type safe. UTT provides the same encapsulation as Universe Types and requires only negligible annotation overhead.

Future Work. Generic Universe Types reduce the number of necessary ownership casts in a program. Currently, we investigate Path-dependent Universe Types [18] to express additional relationships between objects and thereby further reduce the number of ownership casts.

Universe Types provide a very limited support for static fields and methods. The main problem is that global data enables a form of re-entrant method calls that is otherwise prevented by the type system. This form of re-entrancy causes problems for the verification of object invariants. We will formally integrate the Universe Type System with the Boogie methodology for the verification of object invariants [11], which can handle arbitrary forms of re-entrancy.

For the verification of object invariants, one has to control aliasing between fields of one object that are declared in different classes [13, 11]. We are currently extending Universe Types to enforce an ownership structure where each object has a context for each superclass of its dynamic type. This will allow us to enforce that the contexts for different superclasses are disjoint.

Another line of work to support program verification is to build an effects system on top of Universe Types. This effects system will be similar to Clarke and Drossopoulou's work [2], but has to handle `any` references, which makes read effects more complex. We plan to use the effects system to check side effects of methods and to support reasoning about pure methods.

1.2 Formal Foundation

Recent Developments. We proved in the theorem prover Isabelle that the Universe type system is sound and that the owner-as-modifier discipline is enforced [9]. We also wrote a detailed type safety proof on paper for Generic Universe Types [3]. A similar, but less comprehensive proof is available for Universe Types with Transfer [14].

Future Work. We aim at extending our Isabelle formalization of Universe Types to Generic Universe Types.

2. Type Inference

One strength of the Universe type system is the low annotation overhead; it is further reduced by appropriate defaults. However, the resulting ownership structure is flat and annotating existing software remains a considerable effort. We work on inferring deep ownership structures using static and dynamic techniques.

2.1 Static Universe Type Inference

Recent Developments. We generate constraints from the Java AST of a program and use a pseudo-boolean solver to find possible ownership modifiers [8, 17, 7]. The weighting function of the solver is used to find a deep ownership structure.

We allow partially annotated programs as input. The programmer can simply annotate some fields and method signatures and can then use the static inference to propagate the ownership modifiers and to achieve complete code coverage.

Future Work. Currently, the inference tools work with Universe Types. We will investigate how to incorporate the inference of Generic Universe Types and Universe Types with Transfer.

Another form of static inference is to infer the types of local variables from field and parameter types. We are pursuing this line of work in the context of Universe Type with Transfer. Here, inference for local variables is particularly interesting because locals can change their type from program point to program point as objects get transferred. We are currently implementing our ideas in the JML compiler.

2.2 Runtime Universe Type Inference

Recent Developments. We analyze the execution of standard Java programs and infer Universe modifiers from the execution traces [5, 12, 1, 7]. The advantage of runtime Universe type inference is that the deepest possible ownership structure is deduced. Good code coverage is needed for runtime inference. We allow the user to combine multiple program traces as input to the inference in order to achieve good coverage.

Static and runtime Universe type inference can be combined to get a deep ownership structure and ensure sound results. The result of the runtime inference is used as weight for the static inference. The static inference achieves perfect code coverage and can still change ownership modifiers if that improves the overall structure.

Future Work. A major topic for future work is to apply our inference to real applications. This will provide valuable insights in the expressiveness of Universe Types, the power of our inference tools, and especially to ownership structures that can be found in large systems. We expect especially the last result to be important for the ownership community as a whole.

3. Tool Support

3.1 Compiler and Runtime Support

Recent Developments. The type checker for Universe Types is implemented in the JML tool suite [10] since 2004. The JML compiler also produces the code needed for the runtime check of ownership downcasts. It also stores the ownership modifiers in the bytecode, which allows to typecheck programs without having the Java source code. We will commit the extensions for Generic Universe Types soon. The type checker for Universe Types is also implemented in recent version of ESC/Java2.

To make the interaction with the command-line tools easier for programmers we developed a set of Eclipse [6] plug-ins. The JML checker and runtime assertion checking (RAC) compiler can be invoked from within Eclipse and we created comfortable configuration dialogs. Error messages are parsed and displayed in a separate window and code with RAC can be executed from Eclipse. We also provide code templates that make entering ownership modifiers easy. See Fig. 1 for a screen shot.

Future Work. We are finishing the implementation of the Universe Types with Transfer type checker and runtime support. This extension of the JML compiler also supports inference for local variables. We work on integrating Universe Types with Transfer and Generic Universe Types.

3.2 Inference Tools

Recent Developments. Executing the command-line inference tools requires some knowledge to configure the programs correctly. We provide Eclipse plug-ins that allow the configuration through dialogs and that make management of temporary results easy.

The results of static inference are displayed in a comfortable tree view, see left pane in Fig. 2. The user can change ownership modifiers directly in this pane and see what effects a modification has—without parsing the source code again.

For the runtime inference, we provide a visualization of the ownership structure, see right panels in Fig. 2. The programmer can step through the program execution and observe how the ownership structure is built up.

Both inference tools create their results in a special annotation XML format that describes what ownership modifiers need to be added to a program source. We provide a customized editor for this XML format and the annotations can be automatically inserted into the Java source code.

Future Work. We are working on optimizing the inference tools to handle large programs and will then evaluate the tracing overhead and inference time. The visualizer for the inference tools is an interesting playground for visualizing ownership structures.

References

[1] M. Bär. *Practical Runtime Universe Type Inference*. Master's thesis, Department of Computer Science, ETH Zurich, 2006.

[2] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310. ACM Press, 2002.

[3] W. Dietl, S. Drossopoulou, and P. Müller. Formalization of Generic Universe Types. Technical Report 532, ETH Zurich, 2006.

[4] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

[5] W. Dietl and P. Müller. Runtime universe type inference. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2007. To appear.

[6] The Eclipse Foundation. Eclipse — an open development platform. `http://www.eclipse.org/`.

[7] A. Fürer. *Combining Runtime and Static Universe Type Inference*. Master's thesis, Department of Computer Science, ETH Zurich, 2007.

[8] N. Kellenberger. *Static Universe Type Inference*. Master's thesis, Department of Computer Science, ETH Zurich, 2005.

[9] M. Klebermaß. *An Isabelle Formalization of the Universe Type System*. Master's thesis, Department of Computer Science, ETH Zurich, 2007.

[10] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from `www.jmlspecs.org`, 2006.

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

[12] F. Lyner. *Runtime Universe Type Inference*. Master's thesis, Department of Computer Science, ETH Zurich, 2005.

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

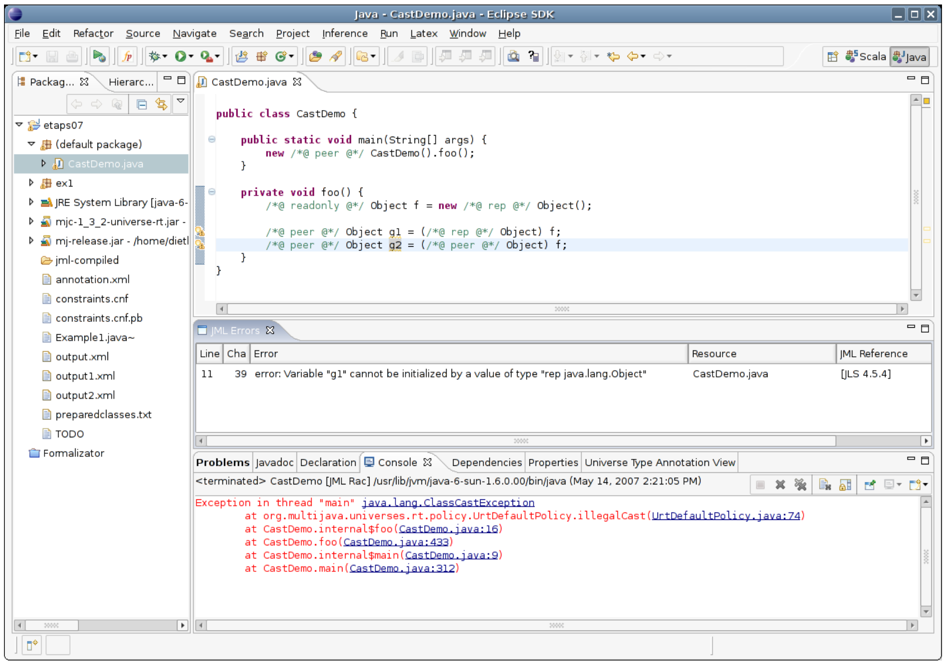
[14] P. Müller and A. Rudich. Formalization of ownership transfer in Universe Types. Technical Report 556, ETH Zurich, 2007.

[15] P. Müller and A. Rudich. Ownership Transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007. To appear.

[16] S. Nägeli. *Ownership in Design Patterns*. Master's thesis, Department of Computer Science, ETH Zurich, 2006.

[17] M. Niklaus. *Static Universe Type Inference using a SAT-Solver*. Master's thesis, Department of Computer Science, ETH Zurich, 2006.

[18] D. Schregenberger. *Universe Type System for Scala*. Master's thesis, Department of Computer Science, ETH Zurich, 2007.

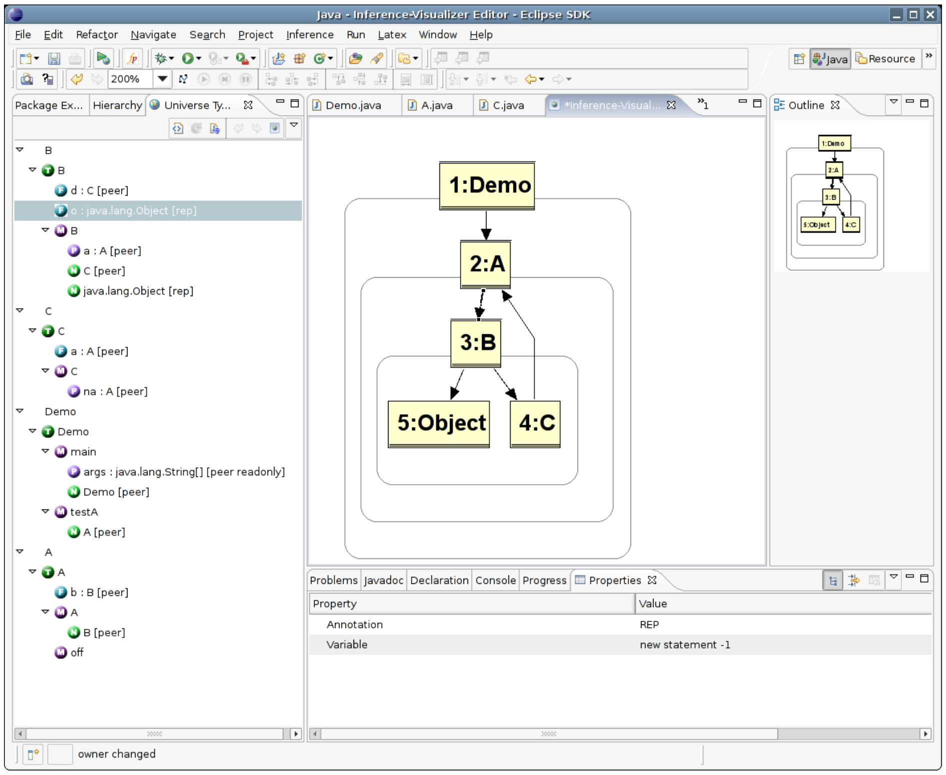

Java - CastDemo.java - Eclipse SDK
File Edit Refactor Source Navigate Search Project Inference Run Latex Window Help
Scala Java
Packag... Hierarc...
etaps07
(default package)
CastDemo.java
ex1
JRE System Library [java-6-
mjc-1_3_2-universe-rt.jar -
mj-release.jar - /home/dietl
jml-compiled
annotation.xml
constraints.cnf
constraints.cnf.pb
Example1.java~
output.xml
output1.xml
output2.xml
preparedclasses.txt
TODO
Formalizator
CastDemo.java
JML Errors
Problems Javadoc Declaration Console Dependencies Properties Universe Type Annotation View

```
public class CastDemo {

    public static void main(String[] args) {
        new /*@ peer @*/ CastDemo().foo();
    }

    private void foo() {
        /*@ readonly @*/ Object f = new /*@ rep @*/ Object();

        /*@ peer @*/ Object g1 = (/*@ rep @*/ Object) f;
        /*@ peer @*/ Object g2 = (/*@ peer @*/ Object) f;
    }
}
```

Line	Cha	Error	Resource	JML Reference
11	39	error: Variable "g1" cannot be initialized by a value of type "rep java.lang.Object"	CastDemo.java	[JLS 4.5.4]

```
<terminated> CastDemo [JML Rac] /usr/lib/jvm/java-6-sun-1.6.0.00/bin/java (May 14, 2007 2:21:05 PM)
Exception in thread "main" java.lang.ClassCastException
        at org.multijava.universes.rt.policy.UrtDefaultPolicy.illegalCast(UrtDefaultPolicy.java:74)
        at CastDemo.internal$foo(CastDemo.java:16)
        at CastDemo.foo(CastDemo.java:433)
        at CastDemo.internal$main(CastDemo.java:9)
        at CastDemo.main(CastDemo.java:312)
```

Figure 1. Eclipse Integration of JML tools: JML compiler error message in the middle and JML RAC runtime error at bottom.


Java - Inference-Visualizer Editor - Eclipse SDK
File Edit Refactor Navigate Search Project Inference Run Latex Window Help
200%
Java Resource
Package Ex... Hierarchy Universe Ty...
Demo.java A.java C.java *Inference-Visual...
Outline
1:Demo
2:A
3:B
5:Object
4:C
Problems Javadoc Declaration Console Progress Properties
owner changed

```
B
  B
    d : C [peer]
    o : java.lang.Object [rep]
    B
      a : A [peer]
      C [peer]
      java.lang.Object [rep]
C
  C
    a : A [peer]
    C
      na : A [peer]
Demo
  Demo
    main
      args : java.lang.String[] [peer readonly]
      Demo [peer]
    testA
      A [peer]
A
  A
    b : B [peer]
    A
      B [peer]
    off
```

Property	Value
Annotation	REP
Variable	new statement -1

Figure 2. Inference mode. Static inference results on the left. Visualization of runtime inference in the center.

Iterators can be Independent “from” Their Collections

John Boyland
Nanjing University, China
University of Wisconsin-Milkaukee, USA
boyland@cs.uwm.edu

William Retert Yang Zhao
University of Wisconsin-Milwaukee, USA
williamr@cs.uwm.edu yangzhao@cs.uwm.edu

Abstract

External iterators pose problems for alias control mechanisms: they have access to collection interals and yet are not accessible from the collection; they may be used in contexts that are unaware of the collection. And yet iterators can benefit from alias control because iterators may fail “unexpectedly” when their collections are modified. We explain a novel aliasing annotation “from” that indicates when a collection intends to delegate its access to internals to a new object and how it can be given semantics using a fractional permission system. We sketch how a static analysis using permissions can statically detect possible concurrent modification exceptions.

1. Introduction

Iterators in Java™and related languages are objects that give sequential access to collections, while being external to the collection. In particular, multiple iterators can operate on a collection at the same time, and the collection is not directly involved in the operation of iterators. Iterators are an improvement over previous related concepts, such as *cursors*, precisely because of this independence. Typically a collection only had one cursor, and moving the cursor had an effect on the collection. Multiple cursors are possible but hard to manage. Noble [18] has surveyed a wide variety of iterator architectures; we focus here on “external iterators.”

An iterator is originally created by the collection, but after creation, it may be used in contexts in which the collection is neither visible nor in scope. The very independence that makes iterators so powerful also makes programs that use them more complex because of the interactions, notably aliasing, between the collection and the iterators. In particular, an iterator typically has pointers into the internals of the collection representation, and may even perform changes on this representation.

Figure 1 gives two interface declarations for the kinds of iterators that will be discussed in this paper. In Java, these interfaces are conflated by making `remove` an optional operation. In this work, to make it easier to reason about iterators and to make the distinction visible in the type system, we use separate interfaces. C++ similarly makes a type-level distinction between iterators that can be used to modify a collection and those that cannot. Other kinds of iterators (such as `ListIterators` that can change an element in a collection) could be defined. We will use the term *mutating iterator* to refer to generally to any iterator that can modify the underlying collection.

Additionally, we have annotated the iterator methods with “method effects” indicating what state the methods are intended or permitted to access. We will assume that all methods are so annotated. In this case, the methods are declared as accessing only the state of the iterator and not reading or writing any other state. The fact that we can mask away the effects on the collection is non-trivial and is one of the main discussion topics of this paper.

```
interface Iterator {
  reads(this.All)
  boolean hasNext();

  writes(this.All)
  Object next();
}
```

```
interface RIterator
  extends Iterator {

  writes(this.All)
  void remove();
}
```

Figure 1. Two classes of iterators.

```
class App {
  this List list = new List();
  ⋮
  writes(All) void run() {
    Iterator it = list.iterator();
    ⋮ What if we mutate list?
    if (Util.member(null,it)) { ... }
  }
}
```

Figure 2. Using an iterator.

Normally, we will follow standard OO convention and abbreviate *this.All* as *All*, since `All` is a (model) instance field.

Figure 2 shows an example of using iterators. It uses classes `List` and `Util` defined in the following section. The example includes some omitted code that may or may not mutate the collection. A collection may be changed directly using a method such as `clear` or `add`, as well as indirectly through a mutating iterator. When this happens, all iterators currently active on the collection (excepting only the iterator through which the mutation took place, if any) are potentially invalid: they may be referring to internals that have been discarded or are otherwise reorganized. For instance, if a linked list is cleared, then existing iterators may refer to nodes that are (otherwise) garbage, and indeed in a language such as C++, the node may have already been returned to the memory allocation system. If a new entry is added to a hash table, an existing iterator may find its node pointer has been rehashed to a new location and continuing the iteration may result in repeating some elements previously encountered, and omitting others. In C++, the programmer is warned by the documentation that existing iterators may be “invalid” after a mutation.

It is possible to implement iterators so that they are robust in the face of change, albeit with some additional complexity. In Java, rather than following this route or letting a potentially confusing situation emerge, a *fail fast* semantics is used: an iterator will (almost) always detect when a mutation outside of its control has happened, and throw a `ConcurrentModificationException` (CME) if it is used afterwards. Typically this is implemented by version stamping the iterator and collection, but the implementation details are not the focus here. Instead we are interested in how static alias control mechanisms can (1) describe external iterators, (2) explain the effects of using a (mutating) iterator, (3) explain why and when concurrent modification exceptions are thrown, and (4) statically prevent these exceptions from occurring.

In the following section, we look at a linked list class with iterators annotated to express the design intent of the aliasing. Then in Sects. 3 and 4, we look at previously described alias control systems and evaluate them by these four criteria. In Section 5, we describe how the design intent of the example can be expressed using our "fractional permission" system.

2. Example

Figure 3 defines a simple linked list class with two kinds of iterators. The code is annotated (*italic* words) with "design intent" showing how aliases are intended to be controlled. Except for *from* (explained below), these annotations have appeared in one form or another in previous work. The example also uses class parameters (inside `< >`) to pass objects that may be used in annotations on fields of the class. For simplicity, we don't make use of generic classes (as in Java 5), although it has been shown that one can fruitfully use both class parameters for ownership and for generic classes [19].

Every field, parameter or return value is annotated by an aliasing annotation: *shared* (owned by the global context), *name* (owned by *name*), *readonly-name* (read-only state owned by *name*), *from(...)* (explained below). Another possibility is *unique*, not used here. The default is *borrowed*, which can be applied only to parameters (including method receivers), which means the method can only access state from the parameter (receiver) if the state is present in the *method effect*. A method effect is of the form *reads(...)* or *writes(...)* and permits the method access the mutable state named; write access includes read access. Here *All* means all state of this object, or any object owned by it (transitively). Constructors are implicitly permitted to write any part of the constructed object's state.

For example, the `add` method of class `List` is annotated *writes(All)* which permits it to read or write any field of the list object (or its nodes, which it owns). Here it simply updates the `head` field. The parameter is *shared* which means there is no alias control intended here.

Next consider the `iterator()` method. Its effect *reads(All)* permits it to read any field or node of the list. The return value is annotated *from(All)* which means that the iterator is "unique" (unaliased with anything else) but that it gets (some of) its state from the method effect on *All*. The idea is that the collection temporarily *yields* its own state to the iterator, an independent (even unique) object. Indeed, the iterator becomes the owner of the (read-only) state of this list, as can be seen by the annotation on the parameter `list` of class `ListIter`.

We will continue with Figure 3, but first glance at Figure 4 which shows how code can use the iterator without reference to the collection. Methods `member` and `removeAll` both take iterators and are annotated that they modify the iterator (*writes(it.All)*) and nothing else.

Back at Figure 3, looking at class `ListIter`, one sees that field `cur` points to a node owned by list. The method `next()` is allowed to access the `cur.next` field because through its read-only ownership of the list, the iterator has read-only access to the internals.

```
class ListNode<owner> {
 owner ListNode<owner> next;
 shared Object datum;

 ListNode(shared Object d, owner Node n)
 { datum = d; next = n; }
}

class List {
  this ListNode<this> head;

  List() { head = null; }

  writes(All) void add(shared Object datum)
  { head = new ListNode<this>(datum,head); }

  writes(All) void clear() { head = null; }

  reads(All) from(All) Iterator iterator()
  { return new ListIter<this>(head); }

  writes(All) from(All) RIterator riterator()
  { return new ListRemoveIter<this>(this); }
}

class ListIter<readonly-this list>
    implements Iterator {
  list ListNode cur;
  Iterator(list ListNode head) { cur = head; }

  reads(All) boolean hasNext()
  { return cur!=null; }

  writes(All) shared Object next()
  { if (cur == null) return null;
    Object temp = cur.datum;
    cur = cur.next;
    return temp; }
}

class ListRemoveIter<this list>
    extends ListIter<list> implements RIterator {
  this List list; // MUST == class parameter list
  list ListNode prev, last;

  RIterator(this List list) // MUST == class param. list
  { super(list.head); this.list = list; }

  writes(All) shared Object next()
  { prev = last; last = cur;
    return super.next(); }

  writes(All) void remove()
  { if (prev == null) list.head = cur;
    else prev.next = cur;
    last = prev; }
}
```

Figure 3. Listed list with iterators annotated with design intent.

```
class Util {
  writes(it.All)
  static boolean member(Object x, Iterator it)
  { while (it.hasNext())
    { if (it.next() == x) return true; }
    return false; }

  writes(it.All)
  static void removeAll(Object x, RIterator it)
  { while (it.hasNext())
    { if (it.next() == x) it.remove(); } }
}
```

Figure 4. Independence of iterators.

```
class Iterator2 implements Iterator {
  this Iterator it1, it2;
  Iterator2(this Iterator it1,
            this Iterator it2) { ... }

  reads(All) boolean hasNext()
  { return it1.hasNext()||it2.hasNext();}

  writes(All) shared Object next()
  { return it1.hasNext() ?
           it1.next() : it2.next(); }
}
```

Figure 5. Constructing iterators using iterators.

The remove iterator ListRemoveIter extends the ListIter class with three more fields. The first field shares the same name as the class parameter and must indeed be the same (identical) pointer. The new fields are implicitly included in All and thus the overriding of next() is permitted to access the prev and next fields. More importantly ListRemoveIter requires that it be made (temporary) owner of the collection (for both read and write access). The remove() method uses this ownership to perform modifications to the list under the effect *writes(All)*.

What about the problems with concurrent modification? In Figure 2, the run() method gets an iterator and after some time uses it to check for nulls in the list. If there is an intervening modification, the call to member would "fail fast" with Java iterators. In this case, the design intent indicates that the iterator it has (temporarily) taken over read-only ownership of the list; if we permit a write of the list to happen, we are indeed permitting a write to occur concurrently with a read, a classic error.

According to the annotation, therefore, the list may not be mutated until the iterator is no longer in use. Dually, write access is permitted only at the cost of disallowing any later use of the iterator. With a mutating iterator, even read access is prohibited until when the mutating iterator is done, or dually, the mutating iterator may not be used once any (even read) access is performed.

One reason to permit the iterator to be used separately from the collection is to support existing code patterns. Examining the open-source Eclipse project for uses of iterators, we have found some cases of interest. One example (greatly simplified here) concerns building an iterator that is constructed from other iterators (see Figure 5). The compound iterator indirectly takes over the (temporary, read-only) ownership of the collections' internals (there may be more than one collection involved). Again this means that effects on the element iterators are mapped into effects on the compound iterator.

```
class OtherCollection {
  this LinkNode<this> head;
  .
  .
  .
  reads(All) from(All) Iterator iterator()
  { return new List(head).iterator(); }
}
```

Figure 6. Delegating iterator creation.

Figure 6 shows another pattern, whereby a class does not implement its own iterators and instead creates an instance of a collection and gets an iterator for it. (Here we assume a new constructor for List that takes a read-only list of nodes.) In the code we saw with this pattern, the delegation involved creating a List around an existing array.

In this section, we have showed several ways in which iterators are defined and used. Undoubtedly, the annotations and explanations here reflect our own biases (and we indeed show how they are realized in our "fractional permission" system), but the code itself is conventional. In the following sections, we survey previously described alias control systems and the extent to which they can describe what is going on in the code.

3. Ownership-based Alias Control

Ownership is a recognized alias control technique. With ownership, each object has another object as its *owner*. The root of the ownership hierarchy is often called "world." Clarke and others propose a owners-as-dominator model: any reference to an object must pass that object's owner [10, 11]. This encapsulation property prevents any access to an object from objects outside its owner, but rules out external iterators: If the iterator is totally outside of the representation of the collection object (Figure 7(a)), then the iterator is not able to access the internals. On the other hand, putting the iterator as part of the representation enables the references to the internals, but disables the references from outside of the collection (Figure 7(b)).

Since the iterator is a common idiom in OO programs, alternative models have been proposed. Clarke and Drossopoulou [9] permit iterator-like objects that violate owners-as-dominators to exist in stack variables but not to be stored in fields. Because they only have dynamic extent (rather than indefinite extent), these *dynamic aliases* are deemed less dangerous than "static aliases." However, the typing of these dynamic aliases requires that the collections be in scope. Therefore, the dynamic aliases solution cannot handle the iterator usages in Figs. 4, 5 and 6

A related relaxation of owners-as-dominators was formalized by Boyapati and others [4]. Here objects of inner classes are permitted to access the internals of the outer object, even though the inner class object is not necessarily owned by its outer object. An iterator implementation is declared as an inner class implementing a global interface. Again, the aliasing between the object and its inner class objects is deemed less hazardous since it is restricted to a single compilation unit. Originally, Boyapati permitted iterators to be passed outside of the scope of collections (as in Fig. 4), but in this case, the effects were imprecise: they operated on the "world." This extension proved unamenable to ownership-based checking of synchronization, and was dropped in subsequent work [3, page 36].

Ownership domains permit an owner to make some owned objects public while protecting others from external access [1]. The objects owned by an owner are partitioned into several "domains."

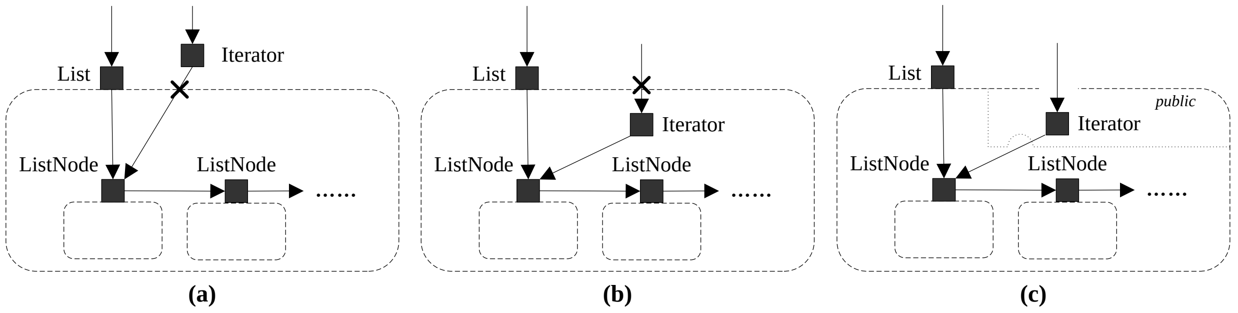


Figure 7. Owners-as-dominators (relaxed in (c)).

For instance, a collection object may own two domains: one for its internal representations and one for iterators. The latter domain can be public (see (c) in Figure 7). The iterator object can be referred to by outsiders (since it resides in a public domain), and it is able to access internal representations of the collection object (since its domain has the same owner as the representation domain). To precisely describe the states the iterator needs to access, Smith [20] proposes that the effects of an iterator be expressed as accessing state in its "sibling" domains, all other domains belonging to the same owner. This only works if the iterator is owned by the same object as the collection. Again there are problems with trying to use the iterator outside of the context of its collection's owner because then the sibling would be unknown.

Another alternate model is the "owners-as-modifier" model which distinguish between read-write and read-only references [13]. Read-write reference must pass through objects' owners, while read-only references may be created arbitrarly. The Universe type system distinguish three kinds of reference: *peer* references between objects in the same context; *rep* references from an object to any directly owned objects; *readonly* (or *any*) references between objects in arbitrary contexts. The last kind of references can not be used to modify objects. External iterators can be implemented in this model (see Figure 8), since any iterator object may use the readonly reference (represented as dashed line) to the internal representation of collections. Modifying iterators need to delegate mutations to the collection, which in turn must rely on runtime support to downcast the *any* reference to a *rep* reference.

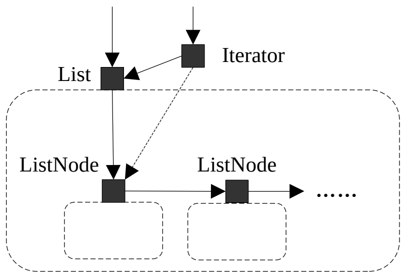


Figure 8. Owners-as-modifiers.

There are two difficulties, however. One is that object invariants cannot be guaranteed for objects referred to with "any" references. Thus a non-mutating iterator cannot rely on invariants of the collection (or its internals) to hold. Indeed, because of concurrent modifications, it is the case that a non-mutating iterator may fail unexpectedly. Furthermore, a mutating iterator must be a "peer" of the collection and thus cannot be used outside the context of the owner of the collection.

4. Handling Iterator Validity

Recently, a number of researchers tackled the problem of iterator validity, that is avoiding CME in Java-like languages. In particular, participants considered avoiding interference between calls that directly modify the collection and interleaved uses of one or more iterators to read that collection.

One solution, proposed by Weide [21] is to modify the semantics of the language such that the iterator copies out the contents of the collection upon its creation, and copies them back when it is finished. Changes made to the collection while the iterator is in use may be overwritten when the iterator is ended, using an explicit function in the collection. This behavior may be specified and checked using the standard tools of full program verification. These changes would affect neither asymptotic efficiency of iterator usage nor expressiveness when interference does occur; however, they are a significant departure from current usage.

The C# patterns for iterator usage and implementation are syntactically different from those of Java; in particular, enumerator functions can define iterators using `yield return` statements. Even so, the underlying problems of interference are generally the same: changes to the collection conflict with use of an iterator. Jacobs, Piessens, and Schulte [15] suggest defining *reads* clauses for enumerator methods that would declare some owned state as immutable while the enumerator-controlled loop is in effect, roughly correlating with the lifetime of the iterator in Java. This tracks well with C# enumerators' reading but not altering their collections; mutating iterators are not supported. Iterator objects can only be used directly to control loops, and may not be used independently of the collection. Every object is given special fields representing both overall writability and number of current readers. The reads clause is translated as modifications of and conditions on these fields, which may then be checked using the Boogie static verifier or using dynamic checks if necessary.

One may instead use fields to directly model the standard "timestamp" approach. Every collection is given an integer field which is incremented whenever the collection is modified; every iterator has an integer field with the value of the collection's timestamp at the time of the iterator's creation. David Cok [12] has instrumented this approach to iterators using model fields in JML. As these timestamps are only implemented in model fields, there is no concrete state underlying them. Rather than throw an exception when the collection's integer exceeds that its iterator, requirements on the relative values of the integers are included in the formal specification of the iterator. This specification can then be checked using ESC/Java2. In practice, the ESC/Java2 checker appears to detect both legal and illegal uses by static inference; however, the correctness of this specification apparently cannot be proven.

Instead of specifying the values of a special fields as a signal of whether the collection has been or can be modified, one may directly encode modifications of the collection in an abstract predicate representing the collection as a whole. Krishnaswami [16] has done this using separation logic, whose predicates are comparable to permissions. Both the collection and the iterator get a high-level predicate that includes an abstract representation of the state of the collection. Most methods of the collection both require and return the predicate of the collection; methods that write the collection return it with a different abstract state. Creating a new iterator consumes the predicate for (permission to) the collection and produces that of the iterator. At any time, however, the predicate for the collection may be carved out of the predicate for its iterator, providing both the collection predicate and a "magic wand" implication that can consume the collection's predicate to recover the iterator. This implication essentially represents the non-collection portions of the iterator's state. The former may be used to call any of the collection's methods, including creating another iterator. Thus one may simultaneously have the predicate for the collection and any number of implications allowing one to exchange the collection predicate for some iterator. This must be done to use the iterator, after which the collection can be carved out once more. However, calling a method that modifies the collection returns a predicate with a different abstract state, which cannot be used to recover any of the current iterators–they are useless. Because this formalization lacks fractions [5, 8], even non-mutating iterators interfere with each other.

Bierhoff [2] describes another linear-based system using fractions and (linear) typestates. Typestates can represent sole, write, or read access to a program variable. Both the collection and the iterator have permissions defined for them; these also detail state changes in the objects themselves. For example, the `hasNext` method is needed to establish that the iterator's next is available as a prerequisite to calling `next`. The absence of this typestate precludes calling the method.

The iterator method returns a linear implication which consumes the permission for the collection, and produces only permission for the iterator. While the collection's permission is unavailable, methods that change it cannot be called. The `finalize` method of the iterator returns the reverse linear implication, consuming the iterator permission and producing that of the collection. The iterator class is parametrized by the collection to permit `finalize` to return permission to the collection. The iterator permission may be fractional, for a read-only iterator, or unique for a modifying iterator. The linear & is cleverly used to delay deciding whether an iterator is read-only or fractional. If the iterator permission is fractional, collection methods that only require fractional state may be called.

5. Explaining Iterators Using Permissions

The key issue with the design intent of the iterators is that access to the collection must be reduced to read-only (for non-mutating iterators) or completely prevented (for mutating iterators) while the iterator is active. In other words, the alias control system must be flow-sensitive. While some ownership type systems are moving to include flow-based analysis (see ownership transfer in Universes [17]), this is a significant increase in complexity.

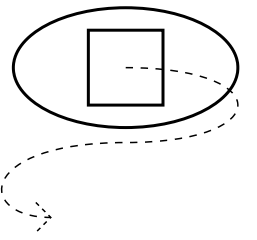
On the other hand, systems based on linear logic (such as Bierhoff's permission system or separation logic), lead to complex management of state. Linearity is powerful but is difficult to manage. Fänhdrich and Deline [14] recognized this problem and designed a relation called "adoption" which permitted non-linearity to co-exist with linearity. We have since shown that adoption can model object ownership [6].

Our permission system combines the simplicity of ownership—allowing the iterator interface to hide the fact that it has access to the collection internals—with the power of linearity—expressing the constraint that the collection is encumbered by the iterator. In this section, we describe our permissions system and how it can account for the annotated design intent in the examples. Permissions are used to give a *semantics* to annotations, and then a flow-sensitive type system can be used to *check* that the code does indeed follow the design intent prescribed by the annotations.

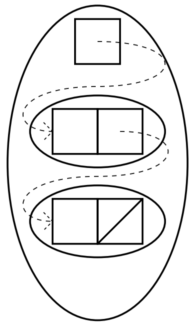
5.1 Permissions overview

A *permission* is a token that permits access to mutable state. Each field in a Java program is associated with exactly one field permission. This permission can be *split* into *fractions*: in order to write a field one needs the whole field permission, but read access is permitted with only a non-zero fraction. Permissions cannot be copied—only transferred. As a result, although two read accesses can be carried out "at the same time" in different parts of the program, if one or both of the accesses is a write, the permission system will flag an error. This is the basic intuition of fractional permissions.

In order to support information hiding and "non-linear" access to state, we add the concept of *permission nesting* (a generalization of adoption); in which an arbitrary permission (often composite) is "nested" in a field permission. As a result, one who has access to the field (and knowledge of the nesting relationship) can get at the nested permissions, by "carving" them out of the field permission.

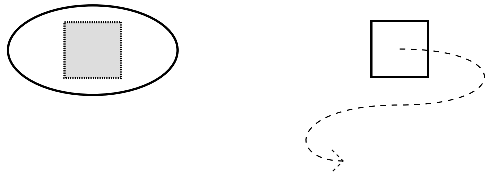


Nesting is used for two main purposes. On the left, we are using nesting to model "data groups." The square represents permission to access a field with a pointer value. The oval shows the (model) field "All." The digram demonstrates that the permission to access the field is nested in "All."



On the right, we are using nesting to express ownership as well as data groups. We now see that the field (small square near the top) points to a node whose entire state is nested in the first object's "All" mode field. This object has two fields (think of them as a data field and a next field), the second of which points to another owned node. The second node has a "null" next pointer.

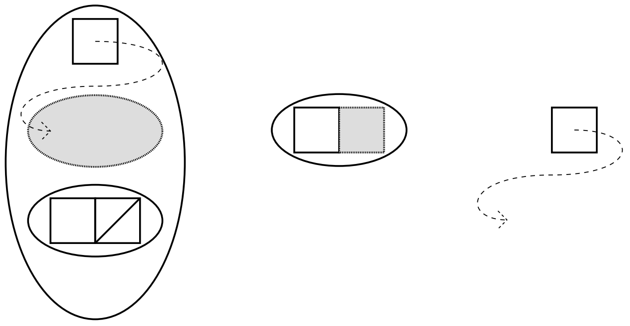
Carving temporarily removes the permissions from the field permission; the field permission now has a "hole" in it. This situation is represented by a kind of "linear implication" $\Psi \multimap \Psi'$, where Ψ is the nested permission and Ψ' is the nester field permission.



$(o.f \to r) \multimap o.\mathrm{All}$ $\qquad\qquad$ $o.f \to r$

The permission $\Psi \multimap \Psi'$ can be read as referring to all the state implied by Ψ' except that part of which is implied by Ψ. In our permission system, "$\multimap$" (read "scepter") functions very similarly to "$-\!\!\star$" in separation logic (or "$\multimap$" in linear logic). The peculiar distinction of "$\multimap$" is that it requires that the consequent (right-hand side) include the antecedent (left-hand side).

Carving is used to get any nested state. Thus to read the next field of the first node, we first carve out the node and then carve the field out:



The inversion of "carving" is "replacing." *Replacing* is simply linear *modus ponens*: it takes $\Psi \multimap \Psi'$ and Ψ and puts them together to retrieve the consequent Ψ'. In standard linear fashion, the process consumes both of the inputs (the antecedent and the implication).

Our permission logic also includes composition (represented by a comma), conditionals (to handle possibly null pointers) and existentials.

5.2 Annotation semantics

In current work [7], we describe the semantics of annotations precisely in terms of permissions. Here, we content ourselves with informal explanation:

data groups State encapsulation (such as *All* referring to all the state of an object) is handled by unit-typed fields that nest the state that belongs to it.

ownership Ownership is represented by nesting x*.All* of the owned object x in a field (ownership domain) of the owner. Multiple ownership domains [1] can be modelled. Every ownership domain is nested in *this.All*.

readonly ownership Read-only ownership is represented by nesting an unspecified *fraction* of the state of the owned object in the owner's domain.

borrowed Borrowed references are references that are transmitted without permissions; in order to access state through a "borrowed" parameter (or receiver), a method uses effects:

effects Method effects are represented by permissions that are passed to the method and which are returned afterwards.

unique A unique reference is always associated with permission to access the object it refers to (if any). For instance, a *unique* return value will be returned along with the necessary permissions.

from As with *unique*, a *from* return value has permissions to access it returned by the method, but unlike *unique*, these permissions encumber the effect named, the permissions for the effect are returned in linear implication:

$$(r.\text{All}) \multimap (\mathit{effect}, v)$$

where v is an unspecified permission. In other words, the permissions represented by the effect are unusable until the state pointed to by the return value is no longer needed. At this time, the linear implication can be applied, releasing the effect and some unknown "residual" permission (v) that can be discarded. In our examples, the v is the iterator permission itself bereft of permission to access the collection.

$$\texttt{iterator}: \begin{array}{l} \forall zt \cdot (zt.\text{All} \rightarrow \\ \exists rv \cdot r.\text{All}, (r.\text{All} \multimap zt.\text{All}, v)) \end{array}$$

Figure 9. The permission type of the `iterator()` method.

Annotations are translated into permissions using a simple substitution (not described here). An example is given in Fig. 9 and is explained below.

5.3 Controlling access

If one has permission to access an object's state, the carve operation gives access to the state nested in that object (other objects owned by it). Allowing this situation in general would break encapsulation; indeed it would permit clients to mutate list internals, resulting in complete chaos. The difficulty this situation presents cannot be solved simply by permissions because one still wishes to permit the client to call methods that use the permission to access internals. Instead, protection of internals is solved in the traditional way with visibility: carving is only permitted to access *visible state*.

The iterator examples show the list iterators accessing internals of the list, including its list nodes. In order to allow iterators to do these operations, we use Java's nested classes. The `ListIter` and `ListRemoveIter` classes are made (private) nested classes of `List` and thus given access to the list internals. The `ListNode` class serves as a structure and can make its fields public.

5.4 Explaining Iterators

In the `iterator()` method of class `List`, the read-only permission to access the list is passed to the new `ListIter` object so that it can nest this permission in its ownership domain. The permission for the effect is not returned to the caller as normal; a linear implication *Iter* $\multimap$ (*reads(All)*, v) is returned instead. The read permission in the consequent cannot be used, as a read permission or to reform a write permission for the list, until the iterator is no longer in use, However, as fractions permit splitting a permission into an arbitrary number of read permissions, other read-only operations may be performed on the list. Figure 9 gives the full permission type of the method (after annotations are translated). The variables t and r refer to the receiver (`this`) and return value respectively. The variable z refers to the non-zero fraction of access required and v refers to the unspecified permission to be discarded when the linear implication is applied. The `riterator()` method is similar except that it requires, and so makes inaccessible, full (write) permission to the list.

The type in Fig. 9 does not disclose that the iterator uses ownership, or indeed anything in how the iterator is implemented. Indeed the permission implication $r.\text{All} \multimap (zt.\text{All}, v)$ is implemented internally by an empty permission since the return value's state $r.\text{All}$ includes the entire list read permission on the right-hand side of the implication. But the client does not need to know this; should *not* know this fact. The information hiding is essential to ensure that the list permission is inaccessible until the implication is applied to the iterator permission, consuming it and releasing the list permission.

In the iterator implementation, as the iterator is temporarily the owner of the list, it may be used without explicit mention of the list. In the implementation of the `next` method, we can access `cur.next` by first carving the (read-only) permission list from the iterator's "All" permission (provided by the caller per the method effect), and then getting access to the node by carving its permission from the list, and finally to get access to the field by carving its permission from the node's "All" permission. This permission is existential (the next pointer is not determined by the node) and must be unpacked. At this point, we have a series of permission implications plus one field permission:

$$zl.\text{All} \multimap t.\text{All}, \qquad zc.\text{All} \multimap zl.\text{All},$$
$$(\exists p \cdot z(c.\texttt{next} \to p, \ldots)) \multimap zc.\text{All} \qquad zc.\texttt{next} \to n$$

Here t is the value of this, l for `list`, c for `cur` and `n` for its next field. When we are done, the entire set of permissions can be packed back up into t.All and returned to the caller. The caller need not be aware of the existence of the list at all.

For example, the `Util` methods work by borrowing iterators: its caller provides the permissions that are then returned. It is impossible to also provide permission to access the collection in a conflicting way—writing the collection is precluded by the presence of a read permission in the consequent of the linear implication–and thus call-back problems are prevented.

Remove iterators avoid throwing a CME because they nest the full write permission for the list preventing alteration of the list state until the remove iterator is no longer used. And, because creating iterators requires at least some part of the list's state, no other iterator can be created while the remove iterator is in use. Neither can a remove iterator be created if another iterator is in use. This is more strict than Java requires, but does ensure the absence of CMEs.

Regular iterators are also prevented from throwing a CME. In Fig 2, after the application requests for the iterator `it` to be created, the collection is encumbered. A linear implication for recovering the access to the list is made available. This linear implication can be applied at any time (in the static flow-analysis of the method). However, once it is applied, permission to access the iterator is irrevocably consumed. Thus, once the permission type checker is "forced" to apply the implication to permit a collection mutation, the iterator is no longer usable and later uses of the iterator will not type check.

The class to concatenate two iterators (see Fig. 5) requires ownership of the iterators, so that effects on them can be attributed to the compound iterator. This can be granted (the iterator was essentially "unique" before) at the price that the respective collections are still encumbered. In order to unencumber the collections, it necessary to retrieve the permissions for the iterators now nested in compound iterator; this can be done when the compound iterator is being discarded.

Delegation of iterator creation (see Fig. 6) uses an iterator on a newly constructed collection. The "*from*" annotation is not actually needed since the linked nodes are copied; it merely expresses the design intent that the collection should not be changed while the iterator is active.

5.5 Analysis

Our current work [7] gives a type system based on permissions. It is flow-sensitive, keeping track of the current permissions at each point in the program. When an field access is processed or a method is called, it checks whether the operation can be permitted. Nesting, carving and replacing operations are carried out implicitly as needed, perhaps several levels deep as the example for `cur.next` showed. The type system described is non-algorithmic, but we have a (more complex) algorithmic type system that is implemented for Java. Currently the implementation does not support "from" annotations; however, adding support for them appears straightforward.

5.6 Comparisons

Compared to approached based solely on ownership, our system is able to detect when an iterator is invalidated.

Compared to approaches based on program verification, our permissions system is not as powerful. It is based on logic that is similar to power to decidable logics. It remains to be shown that our permission type system (not given here) is decidable. By abstracting out just the access to mutable state, permissions represent a more high-level view of the program than any system that uses model fields (say) to represent version stamps.

Compared to approaches based on linear logic, our system is simpler because it uses information hiding (ownership). Both Bierhoff's and Krishnaswami's approaches have explicit mention of the collection state in the iterator state. Thus it appears that these systems could not support iterator patterns that use the iterator in places where the collection is unknown. Indeed while we require both aliasing annotations and effects annotations, the effect annotations on iterators are very simple. On the other hand, Bierhoff's system tracks type state too: a positive return from `hasNext()` ensures that `next()` can be called safely.

5.7 Other Applications

The "from" annotation is a general solution to the problem of how to grant temporary access to internals. Consider a buffered stream. Internally it has an unbuffered stream. Sometimes a client may wish to perform actions on the unbuffered stream and then resume using the buffered stream. One technique is for the client to be given access to the underlying stream and "hope" that the client will remember to flush the buffered stream before any unbuffered access. A less error-prone approach is to use an enforced "from" annotation:

```
class BufferedOutputStream
    implements OutputStream {
  ⋮
  writes(All)
  from(All) OutputStream getUnderlying() {
    flush();
    return underlying;
  }
}
```

Here the underlying stream encumbers the buffered output stream. If the client wishes to use the buffered stream again, it must give up access (permission) to the underlying stream. Thus we see that "from" is a general solution for a class of problems.

6. Conclusion

We have informally described the concept of permissions which combines an ownership-like system (nesting) with linear types, and is flow sensitive. Permissions can be used to express the design intent on our examples; it can be enforced that a collection cannot be modified while non-mutating iterators are active. The system is flexible enough to permit several interesting iterator usage patterns with minor annotation overhead. The "from" annotation can also be used more generally whenever a class wishes to grant temporary access to internal data structures.

References

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *ECOOP'04 — Object-Oriented Programming, 18th European Conference*, Oslo, Norway, June 14–18, volume 3086 of *Lecture*

Notes in Computer Science, pages 1–25. Springer, Berlin, Heidelberg, New York, 2004.

[2] Kevin Bierhoff. Iterator specification with typestates. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, Portland, Oregon, USA, pages 79–82. ACM Press, New York, 2006.

[3] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. Ph.D., Massachusetts Institute of Technology, February 2004.

[4] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, California, June 8–11, *ACM SIGPLAN Notices*, 38:324–337, May 2003.

[5] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, San Diego, California, USA, June 11-13, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, Berlin, Heidelberg, New York, 2003.

[6] John Boyland and William Retert. Connecting effects and uniqueness with adoption. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Long Beach, California, USA, January 12-14, pages 283–295. ACM Press, New York, 2005.

[7] John Boyland, William Retert, and Yang Zhao. Comprehending annotations on object-oriented programs using fractional permissions. Submitted to OOPSLA '07. Manuscript available at http://www.cs.uwm.edu/faculty/boyland/papers/oo-permissions.pdf.

[8] Stephen Brookes. Variables as resource for shared-memory programs: Semantics and soundness. In *Twenty-second Conference on the Mathematical Foundations of Programming Semantics*, Genova, Italty, May 24–27, *Electronic Notes in Theoretical Computer Science*, 158:123–150, 2006.

[9] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Seattle, Washington, USA, November 4–8, *ACM SIGPLAN Notices*, 37(11):292–310, November 2002.

[10] David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Sydney, Australia, 2001.

[11] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, *ACM SIGPLAN Notices*, 33(10):48–64, October 1998.

[12] David R. Cok. Specifying Java iterators with JML and Esc/Java2. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, Portland, Oregon, USA, pages 71–74. ACM Press, New York, 2006.

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

[14] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 17–19, *ACM SIGPLAN Notices*, 37:13–24, May 2002.

[15] Bart Jacobs, Frank Piessens, and Wolfram Schulte. VC generation for functional behavior and non-interference of iterators. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, Portland, Oregon, USA, pages 67–70. ACM Press, New York, 2006.

[16] Neelakantan R. Krishnaswami. Reasoning about iterators with separation logic. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, Portland, Oregon, USA, pages 83–86. ACM Press, New York, 2006.

[17] Peter Müller and Arsenii Rudich. Formalization of ownership transfer in universe types. Technical Report 556, ETH Zurich, 2007.

[18] James Noble. Iterators and encapsulation. In *TOOLS Europe 2000*, pages 431–442. IEEE Computer Society, Los Alamitos, California, 2000.

[19] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic Java. In *OOPSLA'06 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, USA, October 22–26, *ACM SIGPLAN Notices*, 41(10), October 2006.

[20] Matthew Smith. Toward an effects system for ownership domains. In *7th ECOOP Workshop on Formal Techniques for Java-like Programs*, Glasgow, UK, July 26. 2005.

[21] Bruce W. Weide. SAVCBS 2006 challenge: Specification of iterators. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, Portland, Oregon, USA, pages 75–77. ACM Press, New York, 2006.

Simple and Flexible Stack Types

Frances Perry [*]
Princeton University
frances@cs.princeton.edu

Chris Hawblitzel Juan Chen
Microsoft Research
{chrishaw, juanchen}@microsoft.com

Abstract

Typed intermediate languages and typed assembly languages for optimizing compilers require types to describe stack-allocated data. Previous type systems for stack data were either undecidable or did not treat arguments passed by reference. This paper presents a simple, sound, decidable type system expressive enough to support the Micro-CLI source language, including by-reference arguments. This type system safely expresses operations on aliased stack locations by using singleton pointers and a small subset of linear logic.

1. Introduction

Java and C# are safe, high-level languages. The safety of Java and C# protects one program from another: safe applets cannot crash a browser, safe servlets cannot crash a server, and so on. The high level of abstraction makes programming easier, but makes compilation more challenging. Java and C# require sophisticated optimizing compilation to achieve performance competitive with programs written directly in C or assembly language.

Unfortunately, a large, complex compiler is likely to have bugs, and these bugs may cause the compiler to produce unsafe assembly language code. Proof-carrying code (PCC) [14] and typed assembly language (TAL) [13] solve this problem by verifying the safety of the assembly language code generated by the compiler, thus removing the compiler from the trusted computing base. Because the behavior of an assembly language program is undecidable in general, PCC and TAL require machine-checkable evidence to verify a program's safety. A type-preserving compiler generates this evidence by transforming a well-typed source program into a well-typed assembly language program, preserving the well-typedness of the program during each compilation phase in between the source and assembly language levels [13]. To do this, the compiler must define type systems for each intermediate language in the compilation. Java bytecode [11] and CIL [4] are well-known typed intermediate languages, but these still contain many high-level abstractions, such as single instructions for invoking virtual methods and platform-independent storage slots for local data. Below the Java bytecode and CIL levels, these abstractions break down into smaller pieces. A virtual method invocation turns into a method table lookup, instructions for pushing arguments onto a stack, a call instruction, plus prologue and epilogue code in the called method. Local data storage slots turn into machine-specific registers and stack slots. These lower-level concepts need lower-level types.

This paper describes SST (**S**imple **S**tack **T**ypes), a type system that is appropriate for type-checking stack operations in the lowest levels of a type-preserving compiler, including the final typed assembly language generated by the compiler. Previous type systems for stacks were either undecidable without explicit proof annotations [2, 9] or could not represent arguments passed and returned by reference [12]. By contrast, SST has a simple decision procedure, making it easy to use in an intermediate language. It expresses by-reference arguments, even when multiple references point to the same aliased location. It is provably type-safe, via standard preservation and progress lemmas. Finally, SST is simple and elegant enough to be a trustworthy component of a typed assembly language.

To represent stacks in the presence of aliasing, SST builds on ideas from stack-based TAL [12], alias types [18], and linear logic [6, 19]. Section 2 discusses these systems and related systems in more detail. Sections 3 and 4 introduce SST's types and instructions formally. Section 5 describes a translation from the Micro-CLI [9] source language to SST, demonstrating SST's expressiveness. Section 6 concludes.

2. Background and Related Work

Stack-based TAL (STAL) was the first TAL to support stacks. Its central idea, shared by SST, was a *stack type*, which specifies the known types of values on the stack at any point in a TAL program. For example, the STAL stack type "int :: int :: ρ" specifies that two integers live at the top of the stack, but all types deeper in the stack are unknown, specified only by the stack type variable ρ. Code blocks in STAL may be polymorphic over stack type variables.

In addition to the concatenation operator " :: ", STAL contains a compound stack type that can express some pointers into the middle of the stack. Unfortunately, STAL cannot express the possibly aliased pointers that C# compilers use to implement by-reference arguments. Consider the three C# methods below. The `swap` method takes two integer references and swaps the integers. The `f` method instantiates arguments `x` and `y` with pointers to local variables `a` and `b`, while `g` instantiates `x` and `y` with pointers to `c`:

```
void f() {
    int a = 10, b = 20;
    swap(ref a, ref b); }
void g() {
    int c = 30;
    swap(ref c, ref c); }
void swap(ref int x, ref int y) {
    int t = x;
    x = y;
    y = t; }
```

STAL cannot give a useful type to the `swap` method: even with compound types, STAL stack types must list the types of stack slots in precisely the order that they appear in memory. The STAL type for `swap` must reserve one particular stack slot for `x` and another for `y`, making it impossible for a caller to instantiate `x` and `y` with aliased pointers (as `g` does), with heap pointers (as is allowed by C#), or with two stack pointers in the opposite order. Regarding these limitations, Morrisett *et al.* say that, "it appears that this limitation could be removed by introducing a

[*] The work by Frances Perry was done during an internship at Microsoft Research

$$\frac{}{\varsigma \Rightarrow \varsigma}\ \text{s-imp-eq} \qquad \frac{\varsigma \Rightarrow \varsigma'}{\ell : \tau :: \varsigma \Rightarrow \ell : \tau :: \varsigma'}\ \text{s-imp-concat} \qquad \frac{\ell : \sigma \Rightarrow \ell : \sigma'}{\ell : (\sigma \wedge \{\ell_t : \tau\}) \Rightarrow \ell : (\sigma' \wedge \{\ell_t : \tau\})}\ \text{s-imp-alias}$$

$$\frac{\varsigma_1 \Rightarrow \varsigma_2 \quad \varsigma_2 \Rightarrow \varsigma_3}{\varsigma_1 \Rightarrow \varsigma_3}\ \text{s-imp-trans} \qquad \frac{}{\ell : (\tau :: \varsigma) \Rightarrow \ell : (\tau :: \varsigma \wedge \{\ell : \tau\})}\ \text{s-imp-add-alias} \qquad \frac{}{\ell : (\sigma \wedge \{\ell_t : \tau\}) \Rightarrow \ell : \sigma}\ \text{s-imp-drop-alias}$$

$$\frac{}{\ell : (\tau_1 :: \ell_q : (\sigma \wedge \{\ell_2 : \tau_2\})) \Rightarrow \ell : ((\tau_1 :: \ell_q : \sigma) \wedge \{\ell_2 : \tau_2\})}\ \text{s-imp-expand-alias}$$

$$\frac{\varsigma \Rightarrow \ell : (\sigma \wedge \{\ell_1 : \tau_1\}) \quad \varsigma \Rightarrow \ell : (\sigma \wedge \{\ell_2 : \tau_2\})}{\varsigma \Rightarrow \ell : (\sigma \wedge \{\ell_1 : \tau_1\} \wedge \{\ell_2 : \tau_2\})}\ \text{s-imp-merge-alias}$$

Figure 1. Logical Stack Implication Rules

limited form of intersection type, but we have not yet explored the ramifications of this enhancement." (In fact, one subsequent TAL [2] did add intersection types, but did not explore its use for stacks. Furthermore, this type system was undecidable [2].) SST uses a form of intersection type, rather than using STAL's compound types.

A key advantage of stack allocation is the ease of stack deallocation: a program simply pops data from the top of the stack to deallocate the data. In general, popping may leave dangling pointers to popped data. STAL deals with this safely but awkwardly, applying a special validation rule before each use of any potentially dangling pointer. SST follows a more direct and flexible approach introduced by alias types [18] (although alias types handled heaps objects, not stack data). Alias types split a pointer type into two parts: the location ℓ of the data, and the type of the data at location ℓ. The pointer to the data has a singleton type Ptr(ℓ), which indicates that the pointer points exactly to the location ℓ, but deliberately does not specify the type of the data at location ℓ. Instead, a separate *capability* specifies the current type at ℓ. For example, the capability $\{\ell \mapsto \text{int}\}$ specifies that ℓ currently holds an integer. Because of the separation between singleton pointer types and capabilities, the capabilities can evolve, independently of the pointer types, to track updates and deallocation.

To ensure that no two capabilities specify contradictory information about a single location, alias types impose a linearity discipline on the program's treatment of capabilities, prohibiting arbitrary duplication of the information contained in a capability. In particular, the capability $\{\ell \mapsto \text{int}\}$ is not equivalent to the capability $\{\ell \mapsto \text{int}, \ell \mapsto \text{int}\}$. However, alias types (and the similar capability calculus [3]) use non-standard operators and rules for controlling linearity. Following recent advice [20, 7, 5], SST uses operators and rules directly inspired by standard linear logic [6, 19] and separation logic [17, 8]. Linear logic and separation logic share a core of basic operators. Two are of particular interest for stacks: multiplicative conjunction "$\otimes$" (written as "$*$" in separation logic) and additive conjunction "&" (written as "$\wedge$" in separation logic). For example, to have "coffee $\otimes$ tea" is to have both coffee and tea. To have "coffee&tea" is to have a choice between coffee and tea, but not both. Ahmed and Walker observe that additive conjunction "allows us to specify different 'views' of the stack" [1] (though [1] did not explore applications of this observation); we take this observation as a starting point for representing by-reference arguments.

Jia, Spalding, Walker and Glew [9] used linear logic as the basis for a typed low-level language of stacks and heaps (we refer to this low-level language as "JSWG"). In contrast to STAL, JSWG expressed by-reference arguments. To demonstrate this, the authors also introduced the high-level "Micro-CLI" source language (modeled on the CLI intermediate format targeted by C# compilers [4]) and provided a translation from Micro-CLI programs to JSWG programs. In contrast to SST's decidable logic, JSWG's linear logic (which includes the standard linear operators $\otimes$, &, $\oplus$, $\multimap$, and !) is undecidable [10], making SST more practical than JSWG's system for a compiler intermediate language. Furthermore, JSWG expresses pointers using a heavyweight notion of "frozen" capabilities (with version numbers and "tag trees" for pointers into the stack) while SST relies solely on singleton pointer types and a minimal linear logic. Despite its smaller set of features, SST is still powerful enough to express Micro-CLI; Section 5 describes a translation of Micro-CLI programs to SST programs.

3. Simple Stack Types

Consider the STAL stack type int :: int :: ρ from the Section 2. In alias type notation, each integer on the stack would have a capability $\{\ell \mapsto \text{int}\}$. In linear logic notation, the $\otimes$ operator would glue capabilities together to form a complete stack capability: $\{\ell_2 \mapsto \text{int}\} \otimes \{\ell_1 \mapsto \text{int}\} \otimes \rho$, where ℓ_2 and ℓ_1 are the locations of each of the two integers on the stack. SST takes this notation as a starting point, but makes two modifications. First, to simplify the type checking algorithm, SST replaces the commutative, associative $\otimes$ operator with the non-commutative, non-associative :: operator, resulting in a stack capability $\{\ell_2 \mapsto \text{int}\} :: \{\ell_1 \mapsto \text{int}\} :: \rho$. Second, rather than showing one location per stack slot, SST's notation puts stack slots in between locations, writing $\ell_2 : \text{int} :: \ell_1 : \text{int} :: \ell_0 : \rho$ to indicate that one integer falls between locations ℓ_2 and ℓ_1, and the other falls between locations ℓ_1 and ℓ_0. Note that this adds the extra location ℓ_0 to the example — for instance, the stack pointer might have type Ptr(ℓ_2), pointing to the top of the stack, while the frame pointer might have type Ptr(ℓ_0), pointing to the bottom of the frame.

The following grammar generates labeled stack types ς and unlabeled stack types σ (where τ indicates a single-word type, such as int):

labeled stack type	ς	$::=$	$\ell : \sigma$
unlabeled stack type	σ	$::=$	$\rho \mid \text{Empty} \mid \tau :: \varsigma \mid \sigma \wedge \{\ell : \tau\}$

The unlabeled stack type variables ρ, empty stack Empty, and stack concatenation operator :: give SST the same expressiveness as the core of STAL, but little else. The real power of SST comes from the $\wedge$ operator, indicating aliasing. The stack type $\sigma \wedge \{\ell : \tau\}$ implies three things. First, σ holds. Second, the location ℓ resides either in the heap or in the part of the stack described by σ. Third, ℓ currently contains a word of type τ. Figure 1 shows the rules governing stack types; "$\varsigma \Rightarrow \varsigma'$" means that if ς holds, then ς' also holds. Some rules (s-imp-concat, s-imp-alias, s-imp-eq, s-imp-trans) are basic structural rules. The s-imp-add-alias and s-imp-merge-alias rules allow a program to add one or more aliases to a stack type. The s-imp-drop-alias rule lets a program drop unneeded aliases. The s-imp-expand-alias rule expands the scope of an alias, as described in more detail below.

As an example, consider the `swap` function from Section 2. Suppose that the compiler pushes arguments to `swap` onto the stack from right-to-left, and stores the return address in a register. Upon entry to `swap`, the stack will hold the arguments `x` and `y`, each of which is a pointer to some location inside ρ:

$$\ell_2 : \text{Ptr}(\ell_x) :: \ell_1 : \text{Ptr}(\ell_y) :: \ell_0 : (\rho \wedge \{\ell_x : \text{int}\} \wedge \{\ell_y : \text{int}\})$$

Note that locations ℓ_x and ℓ_y may appear anywhere in ρ, in any order. In fact, ℓ_x and ℓ_y may be the same location. For example, suppose that just before calling `swap`, the stack has type $\ell_0 : \text{int} :: \varsigma$. Figure 1's s-imp-add-alias and s-imp-merge-alias rules prove:

$$\begin{array}{ll} & \ell_0 : \text{int} :: \varsigma \\ \Rightarrow & \ell_0 : ((\text{int} :: \varsigma) \wedge \{\ell_0 : \text{int}\} \wedge \{\ell_0 : \text{int}\}) \end{array}$$

Using this, the program can choose $\rho = (\text{int} :: \varsigma)$, choose $\ell_x = \ell_y = \ell_0$, push two pointers to ℓ_0 onto the stack, and call `swap`.

Figure 1's rules also allow reordering of aliases. For example, the s-imp-drop-alias, s-imp-alias, and s-imp-merge-alias rules prove:

$$\begin{array}{ll} & \ell_0 : (\rho \wedge \{\ell_y : \text{int}\} \wedge \{\ell_x : \text{int}\}) \\ \Rightarrow & \ell_0 : (\rho \wedge \{\ell_x : \text{int}\} \wedge \{\ell_y : \text{int}\}) \end{array}$$

Section 2 mentioned the danger of pointers left dangling after the program pops a word from the stack. The syntax $\sigma \wedge \{\ell : \tau\}$ expresses a clear scope in which ℓ remains safe to use: ℓ definitely contains type τ as long as σ remains unmodified. If the program pops a word from σ, for example, then the alias $\{\ell : \tau\}$ must be discarded (see section 4.1 for details). The rules governing this scope are simple: s-imp-expand-alias expands the scope of an alias, but there is no rule to contract the scope. Expansion is safe, and allows a caller to pass a reference on to another method. The `h` method shown below expands the scope of `c` before calling `swap`. Contraction, on the other hand, could leave unsafe dangling pointers, as shown by the illegal and unsafe C# method `illegalMethod`:

```
void h(ref int c)       { swap(ref c, ref c); }
ref int illegalMethod() { int c; return ref c; }
```

Relation to linear logic. Just as $::$ is a limited version of the linear logic $\otimes$ operator, the $\wedge$ operator is a limited version of the linear logic $\&$ operator. More specifically, the notation $\sigma \wedge \{\ell : \tau\}$ corresponds to the linear logic formula $\sigma\&(\{\ell \mapsto \tau\} \otimes \top)$, where $\top$ is the linear logic notation to indicate any resource. Intuitively, knowing $\sigma\&(\{\ell \mapsto \tau\} \otimes \top)$ means that you can choose to look at the stack in one of two ways: either consider the stack to have type σ, or consider the stack to have type $\{\ell \mapsto \tau\} \otimes \top$. The latter case tells you that the stack holds type τ at location ℓ, plus some other data represented by $\top$.

The s-imp-expand-alias rule and lack of a contraction rule also correspond to linear logic, where $A \otimes (B\&(C \otimes \top))$ implies $(A \otimes B)\&(C \otimes \top)$, but $(A \otimes B)\&(C \otimes \top)$ does not imply $A \otimes (B\&(C \otimes \top))$; linear logic can expand, but not contract, the scope of "$\&(C \otimes \top)$". Unlike JSWG [9]'s scoping via version numbers and tag trees, SST's scoping follows naturally from linear logic rules.

Decidability. Deciding whether one linear logic formula implies another is undecidable in general [10], but is decidable for formulas consisting only of atoms, the $\otimes$ operator, and the $\&$ operator [10]. Since SST's $::$ and $\wedge$ operators are limited versions of linear logic's $\otimes$ and $\&$ operators, it is not surprising that SST's logic is also decidable. The companion technical report [15] presents a simple and efficient (near linear-time) algorithm to decide $\varsigma \Rightarrow \varsigma'$, based on a syntax-directed reformulation of Figure 1's rules. The existence of such a decision algorithm is the key to the decidability of type checking in SST (stated formally in Section 4).

Locations. A location ℓ may be a location variable "η", the location of the bottom of the stack "base", the next location towards the top of the stack "next(ℓ)", or a heap location "p" (assuming an infinite supply of locations p for heap allocation):

$$\textbf{location} \quad \ell \quad ::= \quad \eta \mid \text{base} \mid \text{next}(\ell) \mid p$$

For example, the STAL type int :: int :: ρ may be written in SST as "$\text{next}^2(\eta) : \text{int} :: \text{next}(\eta) : \text{int} :: \eta : \rho$", where $\text{next}^2(\eta)$ is an abbreviation for $\text{next}(\text{next}(\eta))$. For convenience, we frequently use the following abbreviation:

$$(\tau_n \ldots \tau_1)@(\ell : \sigma) = \text{next}^n(\ell) : \tau_n :: \ldots :: \text{next}^1(\ell) : \tau_1 :: \ell : \sigma$$

With this, the STAL type int :: int :: ρ may be written in as $(\text{int}; \text{int})@(\eta : \rho)$.

4. Formalization

Types. SST supports integer type "int", nonsense type "Nonsense" for uninitialized stack slots, heap pointer type "HeapPtr(τ)" for pointers to heap values of type τ, singleton type "Ptr(ℓ)", and code type "$\forall[\Delta](\Gamma, \varsigma)$" for code blocks.

$$\begin{array}{llll} \textbf{type} & \tau & ::= & \text{int} \mid \text{Nonsense} \mid \text{HeapPtr}(\tau) \\ & & & \mid \text{Ptr}(\ell) \mid \forall[\Delta](\Gamma, \varsigma) \end{array}$$

Type $\forall[\Delta](\Gamma, \varsigma)$ describes preconditions for code blocks. The location environment Δ is a sequence of location variables and stack type variables. The register file Γ is a partial function from registers to types. Γ and ς describe the initial register and stack state for the blocks. They may refer to the variables in Δ.

Values and Operands. A stack location d is either "base" or the next stack location "next(d)".

A word-sized value w may be an integer "i", the "nonsense" value for uninitialized stack slots, a heap location "p", a stack location "d", or instantiated values "$w[\ell]$" and "$w[\sigma]$" where w points to code blocks polymorphic over location variables and stack type variables. Contents of registers and stack slots are word-sized. As in STAL [12], word-sized values are separated from operands to prevent registers from containing registers.

$$\begin{array}{rlll} \textbf{stack loc} & d & ::= & \text{base} \mid \text{next}(d) \\ \textbf{word value} & w & ::= & i \mid \text{nonsense} \mid p \mid d \mid w[\ell] \mid w[\sigma] \\ \textbf{operand} & o & ::= & r \mid w \mid o[\ell] \mid o[\sigma] \end{array}$$

An operand o may be a register "r", a word-sized value "w", or instantiated operands "$o[\ell]$" and "$o[\sigma]$". A special register sp is used for the stack pointer.

Instructions. Most instructions are standard. Values on the heap or stack are accessed through explicit load and store instructions.

$$\begin{array}{llll} \textbf{instr} & \text{ins} & ::= & \text{mov } r, o \mid \text{add } r, o \mid \text{sub } r, o \mid \text{ladd } r, i \\ & & & \mid \text{load } r_1, [r_2 + i] \mid \text{store } [r_1 + i], r_2 \\ & & & \mid \text{jumpif0 } r, o \mid \text{heapalloc } r = \langle o \rangle \\ & & & \mid (\eta, r) = \text{unpack}(o) \end{array}$$

SST uses "ladd" instructions for stack location arithmetic. The first operand points to a stack location. The second operand is a constant integer (positive or negative). A "ladd" instruction moves the stack pointer along the stack according to the integer value. The standard add and subtract instructions deal with only integer arithmetic.

The heap allocation instruction "heapalloc $r = \langle o \rangle$" allocates a word on the heap with initial value o and assigns the new heap location to r.

The unpack instruction "$(\eta, r) = \text{unpack}(o)$" coerces a heap pointer o to a heap location. It introduces a fresh location variable η for o and assigns η to r.

4.1 Type Checking Instructions

The type checker maintains a few environments. The location environment Δ and the register file Γ were explained previously. The heap environment Ψ is a partial function from heap locations to heap pointer types. Stack-related rules are shown here. Appendix B contains all rules.

Operand Typing Rules. The judgment $\Delta; \Psi; \Gamma \vdash o : \tau$ means that operand o has type τ under the environments. Note that a heap location can be typed in two ways: the type in the heap environment (o-p-H) or a singleton type (o-p). A stack location has a singleton type (o-d).

If an operand o has a polymorphic type $\forall[\Delta](\Gamma, \varsigma)$, $o[\ell]$ and $o[\sigma]$ instantiate the first variable in Δ with ℓ and σ respectively. The judgments $\Delta \vdash \ell$ and $\Delta \vdash \sigma$ mean that ℓ and σ are well-formed under Δ respectively.

$$\frac{}{\Delta; \Psi; \Gamma \vdash r : \Gamma(r)}\ \text{o-reg} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash i : \text{int}}\ \text{o-int}$$

$$\frac{}{\Delta; \Psi; \Gamma \vdash \text{nonsense} : \text{Nonsense}}\ \text{o-ns} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash d : \text{Ptr}(d)}\ \text{o-d}$$

$$\frac{}{\Delta; \Psi; \Gamma \vdash p : \Psi(p)}\ \text{o-p-H} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash p : \text{Ptr}(p)}\ \text{o-p}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \forall[\eta, \Delta'](\Gamma', \varsigma) \quad \Delta \vdash \ell}{\Delta; \Psi; \Gamma \vdash o[\ell] : \forall[\Delta'](\Gamma'[\ell/\eta], \varsigma[\ell/\eta])}\ \text{o-inst-l}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \forall[\rho, \Delta'](\Gamma', \varsigma) \quad \Delta \vdash \sigma}{\Delta; \Psi; \Gamma \vdash o[\sigma] : \forall[\Delta'](\Gamma'[\sigma/\rho], \varsigma[\sigma/\rho])}\ \text{o-inst-Q}$$

The judgment $\vdash (\Gamma, \varsigma)\{r \leftarrow \tau\}(\Gamma', \varsigma')$ means that assigning a value of type τ to register r results in new environments Γ' and ς'. Only Γ is changed if r is not sp. Otherwise the stack grows or shrinks according to the new value of sp.

$$\frac{r \neq \text{sp} \quad \Gamma' = \Gamma[r \mapsto \tau]}{\vdash (\Gamma, \varsigma)\{r \leftarrow \tau\}(\Gamma', \varsigma)}\ \text{a-not-esp}$$

$$\frac{\vdash \text{Resize}(\ell, \varsigma) = \varsigma' \quad \Gamma' = \Gamma[\text{sp} \mapsto \text{Ptr}(\ell)]}{\vdash (\Gamma, \varsigma)\{\text{sp} \leftarrow \text{Ptr}(\ell)\}(\Gamma', \varsigma')}\ \text{a-esp}$$

Stack Rules. *Resize.* When the stack grows or shrinks, SST uses the judgment $\vdash \text{Resize}(\ell, \varsigma) = \varsigma'$ to get the new stack type. The judgment means that resizing stack ς to location ℓ results in stack ς'. The location ℓ will be the top of ς'. The stack shrinks if ℓ is inside ς (s-shrink) and grows if ℓ is beyond the top of ς (s-grow). The stack drops all aliases beyond ℓ when shrinking to avoid dangling pointers.

$$\frac{\varsigma \Rightarrow \vec{\tau}\ @(\ell : \sigma)}{\vdash \text{Resize}(\ell, \varsigma) = \ell : \sigma}\ \text{s-shrink}$$

$$\frac{\varsigma' = (\text{Nonsense}_n; \ldots; \text{Nonsense}_1)@(\ell : \sigma)}{\vdash \text{Resize}(\text{next}^n(\ell), \ell : \sigma) = \varsigma'}\ \text{s-grow}$$

Location Lookup. The judgment $\varsigma \vdash \ell + i = \ell'$ means that in stack ς going i slots from location ℓ leads to location ℓ'. A positive i means going toward the stack top and negative means toward the stack bottom. The notion n represents natural numbers. (The requirement $\varsigma \Rightarrow \vec{\tau}\ @(\ell : \sigma)$ ensures that ℓ is a stack location, not a heap location.)

$$\frac{\varsigma \Rightarrow \vec{\tau}\ @(\ell : \sigma)}{\varsigma \vdash \ell + n = \text{next}^n(\ell)}\ \text{s-offset-next}$$

$$\frac{\varsigma \Rightarrow \vec{\tau}\ @(\ell : \sigma)}{\varsigma \vdash \text{next}^n(\ell) + (-n) = \ell}\ \text{s-offset-prev}$$

Type Lookup. The judgment $\varsigma \vdash \ell : \tau$ means that the location ℓ in stack ς has type τ. The location ℓ can be either an alias in ς, or be on the spine of ς (the stack type obtained by dropping all aliases from ς).

$$\frac{\varsigma \Rightarrow \ell' : (\sigma \wedge \{\ell : \tau\})}{\varsigma \vdash \ell : \tau}\ \text{s-lookup}$$

Stack Update. The judgment $\varsigma \vdash \ell \leftarrow \tau \rightsquigarrow \varsigma'$ means that updating the location ℓ in stack ς with type τ results in stack ς'. Weak updates do not change the stack type (s-update-weak). Strong updates change the type of ℓ and drop all aliases beyond ℓ because they may refer to the old type of ℓ (s-update-strong).

$$\frac{\varsigma \vdash \ell : \tau}{\varsigma \vdash \ell \leftarrow \tau \rightsquigarrow \varsigma}\ \text{s-update-weak}$$

$$\frac{\varsigma \Rightarrow \vec{\tau}\ @(\ell : \tau :: \varsigma')}{\varsigma \vdash \ell \leftarrow \tau' \rightsquigarrow \vec{\tau}\ @(\ell : \tau' :: \varsigma')}\ \text{s-update-strong}$$

Instruction Typing Rules. Figure 2 lists instruction typing rules. $\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{ins}\}(\Gamma'; \varsigma')$ means that checking instruction "ins" changes the environments Γ and ς to new environments Γ' and ς'.

The location arithmetic instruction "ladd r, i" requires that r point to a location ℓ and i be a multiple of 4. The stack grows toward lower addresses. If i is negative, the result location is further outward from ℓ.

Loads and stores can operate on heap locations (i-load-p and i-store-p), stack locations on the spine (i-load-concat and i-store-concat), and aliases (i-load-aliased and i-store-aliased). SST supports weak updates on heap locations and aliases, and both strong and weak updates on stack locations on the spine.

The rule for heap allocation assigns a heap pointer type to the register that holds the pointer, instead of a singleton type, because the new heap location is statically unknown. The heap environment does not change after heap allocation because the rest of the program does not refer to the new heap location by name.

When control transfers, the type checker matches the current environments with those of the target. The location environment of the target should have been fully instantiated. $\Gamma \Rightarrow \Gamma'$ requires that Γ' be a subset of Γ.

4.2 Blocks and Programs

A heap value v is either a code block "block" or a heap word "$\langle w \rangle$". A code block "$\forall[\Delta](\Gamma, \varsigma)\ b$" describes the precondition $\forall[\Delta](\Gamma, \varsigma)$ and its body b. The block body is a sequence of instructions that ends with a jump instruction. Only variables in Δ can appear free in Γ, ς, and the block body.

A program consists of a heap H, a register bank R, a stack s, and a block body as the entry point. H is a partial function from heap locations to heap values. R is a partial function from registers to word-sized values. The stack s records values on the spine. It is either the empty stack "empty" or a concatenation of a word-sized value with a stack "$w :: s$".

heap value	v	$::=$	block $\mid \langle w \rangle$
block	block	$::=$	$\forall[\Delta](\Gamma, \varsigma)\ b$
block body	b	$::=$	ins; b $\mid$ jump o
heap	H	$::=$	$p_1 \mapsto v_1, \ldots, p_n \mapsto v_n$
reg bank	R	$::=$	$r_1 \mapsto w_1, \ldots, r_n \mapsto w_n$
stack value	s	$::=$	empty $\mid w :: s$
program	P	$::=$	(H, R, s, b)

A program $P = (H, R, s, b)$ is well-formed (illustrated by the judgment $\vdash P$) if H matches a heap environment Ψ, R matches a register file Γ, s matches a stack type ς, and b is well-formed under Ψ, Γ, and ς. The notion "$\bullet$" means empty environments.

$$\frac{\Delta;\Psi;\Gamma \vdash o : \tau \quad \vdash (\Gamma,\varsigma)\{r \leftarrow \tau\}(\Gamma',\varsigma')}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{mov } r, o\}(\Gamma';\varsigma')}\ \text{i-mov} \qquad \frac{\begin{array}{c}\Gamma(r) = \text{Ptr}(\ell) \quad \varsigma \vdash \ell + i = \ell' \\ \vdash (\Gamma,\varsigma)\{r \leftarrow \text{Ptr}(\ell')\}(\Gamma',\varsigma')\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{ladd } r, -4 * i\}(\Gamma';\varsigma')}\ \text{i-ladd}$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : \text{int} \quad r \neq \text{sp} \quad \Gamma(r) = \text{int}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{add } r, o\}(\Gamma;\varsigma)}\ \text{i-add} \qquad \frac{\Delta;\Psi;\Gamma \vdash o : \text{int} \quad r \neq \text{sp} \quad \Gamma(r) = \text{int}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{sub } r, o\}(\Gamma;\varsigma)}\ \text{i-sub}$$

$$\frac{\begin{array}{c}\Gamma(r_2) = \text{HeapPtr}(\tau) \\ \vdash (\Gamma,\varsigma)\{r_1 \leftarrow \tau\}(\Gamma',\varsigma')\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{load } r_1, [r_2 + 0]\}(\Gamma';\varsigma')}\ \text{i-load-p} \qquad \frac{\begin{array}{c}\Gamma(r_2) = \tau \\ \Gamma(r_1) = \text{HeapPtr}(\tau)\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{store } [r_1 + 0], r_2\}(\Gamma;\varsigma)}\ \text{i-store-p}$$

$$\frac{\begin{array}{c}\Gamma(r_2) = \text{Ptr}(\ell) \quad \varsigma \vdash \ell + i = \ell' \\ \varsigma \vdash \ell' : \tau \quad \vdash (\Gamma,\varsigma)\{r_1 \leftarrow \tau\}(\Gamma',\varsigma')\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{load } r_1, [r_2 + (-4 * i)]\}(\Gamma';\varsigma')}\ \text{i-load-concat} \qquad \frac{\begin{array}{c}\Gamma(r_1) = \text{Ptr}(\ell) \quad \Gamma(r_2) = \tau \\ \varsigma \vdash \ell + i = \ell' \quad \varsigma \vdash \ell' \leftarrow \tau \rightsquigarrow \varsigma'\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{store } [r_1 + (-4 * i)], r_2\}(\Gamma;\varsigma')}\ \text{i-store-concat}$$

$$\frac{\begin{array}{c}\Gamma(r_2) = \text{Ptr}(\ell) \quad \varsigma \vdash \ell : \tau \\ \vdash (\Gamma,\varsigma)\{r_1 \leftarrow \tau\}(\Gamma',\varsigma')\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{load } r_1, [r_2 + 0]\}(\Gamma';\varsigma')}\ \text{i-load-aliased} \qquad \frac{\begin{array}{c}\Gamma(r_1) = \text{Ptr}(\ell) \\ \varsigma \vdash \ell : \tau \quad \Gamma(r_2) = \tau\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{store } [r_1 + 0], r_2\}(\Gamma;\varsigma)}\ \text{i-store-alised}$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash o : \tau \\ \vdash (\Gamma,\varsigma)\{r \leftarrow \text{HeapPtr}(\tau)\}(\Gamma',\varsigma')\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{heapalloc } r = \langle o \rangle\}(\Gamma';\varsigma')}\ \text{i-heapalloc} \qquad \frac{\begin{array}{c}\Gamma(r) = \text{int} \quad \Delta;\Psi;\Gamma \vdash o : \forall[\,](\Gamma',\varsigma') \\ \Gamma \Rightarrow \Gamma' \quad \varsigma \Rightarrow \varsigma'\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{jumpif0 } r, o\}(\Gamma;\varsigma)}\ \text{i-jump0}$$

Figure 2. Instruction Typing Rules

$$\frac{\vdash H : \Psi \quad \bullet;\Psi \vdash s : \varsigma \quad \bullet;\Psi \vdash R : \Gamma \quad \bullet;\Psi;\Gamma;\varsigma \vdash b}{\vdash (H, R, s, b)}\ \text{m-tp}$$

A heap H matches a heap environment Ψ if they have the same domain and each heap value in H has the corresponding type in Ψ (h-tp). Matching a register bank with a register file is defined similarly (g-tp).

$$\frac{\begin{array}{c}\Psi = \{\ldots, p \mapsto \tau, \ldots\} \quad H = \{\ldots, p \mapsto v, \ldots\} \\ \ldots \quad \bullet;\Psi \vdash v : \tau \quad \ldots\end{array}}{\vdash H : \Psi}\ \text{h-tp}$$

$$\frac{\begin{array}{c}\Gamma = \{\ldots, r \mapsto \tau, \ldots\} \quad R = \{\ldots, r \mapsto w, \ldots\} \\ \ldots \quad \Delta;\Psi;\bullet \vdash w : \tau \quad \ldots\end{array}}{\Delta;\Psi \vdash R : \Gamma}\ \text{g-tp}$$

A stack value s matches a stack type ς if all the locations on the spine have the corresponding type in ς (s-base and s-concat) and ς contains only aliased locations to heap pointers (s-alias) and to stack locations on the spine (s-imp).

$$\frac{}{\Delta;\Psi \vdash \text{empty} : (\text{base} : \text{Empty})}\ \text{s-base}$$

$$\frac{\Delta;\Psi \vdash s : (\ell : \varsigma) \quad \Delta;\Psi;\bullet \vdash w : \tau}{\Delta;\Psi \vdash w :: s : (\text{next}(\ell) : \tau :: \ell : \sigma)}\ \text{s-concat}$$

$$\frac{\Delta;\Psi,\{p \mapsto \text{HeapPtr}(\tau)\} \vdash s : (\ell : \sigma)}{\Delta;\Psi,\{p \mapsto \text{HeapPtr}(\tau)\} \vdash s : (\ell : (\sigma \wedge \{p : \tau\}))}\ \text{s-alias}$$

$$\frac{\Delta;\Psi \vdash s : \varsigma \quad \varsigma \Rightarrow \varsigma'}{\Delta;\Psi \vdash s : \varsigma'}\ \text{s-imp}$$

To type check a block body, the checker checks the instructions in order (b-ins) until it reaches the jump instruction (b-jump).

The unpack instruction "$(\eta, r) = \text{unpack}(o)$" requires o have a heap pointer type (b-unpack). The rule introduces a fresh location variable η to Δ, assigns r a singleton type $\text{Ptr}(\eta)$, and updates the stack type to contain η.

$$\frac{\begin{array}{c}\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{ins}\}(\Gamma';\varsigma') \\ \Delta;\Psi;\Gamma';\varsigma' \vdash b\end{array}}{\Delta;\Psi;\Gamma;\varsigma \vdash \text{ins}; b}\ \text{b-ins}$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash o : \forall[\,](\Gamma',\varsigma') \\ \Gamma \Rightarrow \Gamma' \quad \varsigma \Rightarrow \varsigma'\end{array}}{\Delta;\Psi;\Gamma;\varsigma \vdash \text{jump } o}\ \text{b-jump}$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash o : \text{HeapPtr}(\tau) \qquad r \neq \text{sp} \qquad \eta \notin \Delta \\ (\Delta;\eta);\Psi;\Gamma[r \mapsto \text{Ptr}(\eta)];\ell : (\sigma \wedge \{\eta : \tau\}) \vdash b\end{array}}{\Delta;\Psi;\Gamma;\ell : \sigma \vdash (\eta, r) = \text{unpack}(o)}\ \text{b-unpack}$$

A block is well-formed if under the heap environment and the specified precondition, the block body type-checks.

$$\frac{\Delta;\Psi;\Gamma;\varsigma \vdash b}{\Psi \vdash \forall[\Delta](\Gamma,\varsigma)\ b}\ \text{block-tp}$$

The judgment $P \to P'$ means that program P evaluates to program P'. Evaluation rules are listed in Appendix B.3.

We proved soundness and decidability of SST. The proofs can be found online [16].

THEOREM 1 (Preservation). *If $\vdash P$ and $P \to P'$, then $\vdash P'$.*

THEOREM 2 (Progress). *If $\vdash P$, then $\exists P'$ such that $P \to P'$.*

THEOREM 3 (Decidability). *Given Ψ and block, there is an algorithm to decide whether "$\Psi \vdash$ block" holds.*

5. Source Language and Translation

As mentioned in Section 2, we translate JSWG's Micro-CLI [9] to SST. Micro-CLI supports both heap and stack allocation. A managed pointer can point to either a heap-allocated or a stack-allocated value. Managed pointers have the same constraints as those in CLI, such as they cannot be stored in objects nor returned from functions.

The syntax of Micro-CLI is restated here.

qualifiers	q	$::=$	$S \mid H$
types	τ	$::=$	int $\mid \tau *_q$
values	v	$::=$	$n \mid x$
program	p	$::=$	$fds\ \ rb$
function decls	fds	$::=$	$\cdot \mid fd\ \ fds$
function decl	fd	$::=$	$\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ \ rb$
return block	rb	$::=$	$\{lds; ss;$ return $v\}$
local decls	lds	$::=$	$\cdot \mid ld; lds$
local decl	ld	$::=$	$\tau\ x = v \mid \tau\ x =$ new$_q\ v$
statement list	ss	$::=$	$\cdot \mid s; ss$
statement	s	$::=$	if v then ss else $ss \mid x = v$
			$\mid x = v_1 + v_2 \mid x = v_1 - v_2$
			$\mid x = f(v_1, \ldots, v_n)$
			$\mid x =\ !v \mid v_1 := v_2$

Micro-CLI supports only the integer type and pointer types. Each pointer type is qualified by "S" (stack pointer) or "H" (heap pointer). Heap pointer types are subtypes of stack pointer types with the same referent types, that is, $\tau *_H$ is a subtype of $\tau *_S$.

A Micro-CLI program consists of a sequence of function declarations and a return block. A function declaration specifies the return type, the function name, the parameters, and the body (a return block). A return block contains a sequence of local variable declarations and a sequence of statements. A local variable declaration declares the type and the initial value of a local variable that can be used in subsequent declarations and statements.

The detailed translation from Micro-CLI to SST is described in the companion technical report. Because SST deals with aliasing differently from JSWG, the two translations differ in rules around managed pointers which introduce aliasing. For example, if a source function has a parameter with type "pointer-to-pointer-to-int", the translation to SST creates two aliases for the pointers while the translation to JSWG uses existential types to abstract the locations and version numbers to relate the scopes. The precondition of the function in SST would have a stack type "next(η) : Ptr(η_1) :: $\eta : (\rho \wedge \{\eta_1 :$ Ptr$(\eta_2)\} \wedge \{\eta_2 :$ int$\})$" where the function is polymorphic over η_1 and η_2.

We use the following example to show the result of translation. The "swap" function in Section 2 is rewritten into Micro-CLI syntax as follows:

$$\begin{array}{l}
\text{int swap(int } *_S\ x, \text{int } *_S\ y)\{ \\
\quad \text{int } t = 0; \\
\quad \text{int } t' = 0; \\
\quad t =\ !x; \\
\quad t' =\ !y; \\
\quad x :=\ t'; \\
\quad y := t; \\
\quad \text{return } 0; \\
\}
\end{array}$$

Micro-CLI does not allow such syntax as "$x :=\ \ !y$". A new variable "t'" holds the value of "$!y$" and is then assigned to x. Local variables can be initialized only by values. The local variables t and t' are initialized to 0 first and then assigned "$!x$" and "$!y$" respectively. Micro-CLI does not allow functions with no return values. The "swap" function simply returns an integer value.

The function is translated to the following SST function:

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall[\eta_x, \eta_y, \eta_0, \rho](\Gamma, \varsigma)} \\
\quad \text{mov } r_{fp}, \text{sp} & \\
\quad \text{mov } r_1, 0 & ; r_1 = 0; \\
\quad \text{ladd sp}, -4 & \\
\quad \text{store } [\text{sp} + 0], r_1 & ;\text{push } r_1 \text{ (for } t') \\
\quad \text{mov } r_1, 0 & ; r_1 = 0; \\
\quad \text{ladd sp}, -4 & \\
\quad \text{store } [\text{sp} + 0], r_1 & ;\text{push } r_1 \text{ (for } t) \\
\quad \text{load } r_1, [r_{fp} + 0] & ; r_1 = x \\
\quad \text{load } r_1, [r_1 + 0] & ; r_1 = [r_1] \\
\quad \text{store } [r_{fp} + (-8)], r_1 & ; t = r_1\ (t =!x) \\
\quad \text{load } r_1, [r_{fp} + 4] & ; r_1 = y \\
\quad \text{load } r_1, [r_1 + 0] & ; r_1 = [r_1] \\
\quad \text{store } [r_{fp} + (-4)], r_1 & ; t' = r_1\ (t' =!y) \\
\quad \text{load } r_1, [r_{fp} + 0] & ; r_1 = x \\
\quad \text{load } r_2, [r_{fp} + (-4)] & ; r_2 = t' \\
\quad \text{store } [r_1 + 0], r_2 & ; [r_1] = r_2\ (x := t') \\
\quad \text{load } r_1, [r_{fp} + 4] & ; r_1 = y \\
\quad \text{load } r_2, [r_{fp} + (-8)] & ; r_2 = t \\
\quad \text{store } [r_1 + 0], r_2 & ; [r_1] = r_2\ (y := t) \\
\quad \text{ladd sp}, 16 & ;\text{pop } t, t', x, y \\
\quad \text{mov } r_1, 0 & ; r_1 = 0 \\
\quad \text{ladd sp}, -4 & \\
\quad \text{store } [\text{sp} + 0], r_1 & ;\text{push } r_1 \\
\quad \text{jump } r_{ra} & ;\text{jump } r_{ra} \\
\multicolumn{2}{l}{\text{where } \Gamma = \text{sp} \mapsto \text{Ptr}(\text{next}^2(\eta_0)),} \\
\multicolumn{2}{l}{\qquad\qquad r_{ra} \mapsto \forall[\,](\text{sp} \mapsto \text{Ptr}(\text{next}(\eta_0)), \text{next}(\eta_0) : \text{int} :: \eta_0 : \rho)} \\
\multicolumn{2}{l}{\text{and } \varsigma = \text{next}^2(\eta_0) : \text{Ptr}(\eta_x) :: \text{next}(\eta_0) : \text{Ptr}(\eta_y) ::} \\
\multicolumn{2}{l}{\qquad\quad \eta_0 : (\rho \wedge \{\eta_x : \text{int}\} \wedge \{\eta_y : \text{int}\})}
\end{array}$$

The translation is straightforward. Many optimizations can be applied to improve the SST code, which is beyond the scope of this paper. The translation reserves register sp for the stack pointer, r_{fp} for the frame pointer, and r_{ra} for the return address. Two temporary registers r_1 and r_2 are used to hold intermediate values during the translation of a Micro-CLI instruction. Parameters and return values are passed through the stack. Local variables are allocated on the stack.

The SST function is polymorphic over four variables: η_x, η_y, η_0, and ρ. The first two represent the values of x and y. The third represents the location of the rest of the stack (abstracted by the stack type variable ρ). The parameters x and y are on the stack upon entry to the function. Section 3 explained the initial stack state. The parameters and the local variables are accessed through the frame pointer: t, t', x, and y have addresses $r_{fp} - 8$, $r_{fp} - 4$, r_{fp}, and $r_{fp} + 4$ respectively.

At the beginning of the function, the frame pointer r_{fp} is assigned sp and the initial values for t and t' are pushed onto the stack. At the end, the local variables and the parameters are popped from the stack, the return value is pushed onto the stack, and the control transfers to the return address, which is kept in register r_{ra}.

We proved the type-preservation theorem of the translation:

THEOREM 4 (Type-preserving Translation). *Well-typed Micro-CLI programs translate to well-typed SST programs.*

6. Conclusions

With a simple stack type ς, SST safely supports many low-level idioms: stack pointers, frame pointers, by-value arguments, and by-reference arguments, where by-reference arguments may point to both stack data and heap data.

This paper presented one particular type system built around the stack type ς, but many variations are possible. For example, we treated the stack pointer register as a special register to safely accomodate kernel-mode code in the presence of interrupts, but some other settings could treat the stack pointer as an ordinary register. For GC safety, we allowed pointer arithmetic on stack pointers but disallowed pointer arithmetic on heap pointers. For simplicity, we assumed infinite stack space to grow in, but a type checker based on SST could also verify stack overflow checks (perhaps in cooperation with virtual-memory-based overflow checks). Also for simplicity, our heap consisted of one-word objects, but this extends naturally to objects with multiple fields. Finally, to ensure simple, efficient type checking, we used a small, restricted linear logic, but we could trade efficiency for expressiveness by varying the linear logic, without abandoning the basic SST approach.

References

[1] Amal Ahmed and David Walker. The logical approach to stack typing. In *2003 ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2003.

[2] Karl Crary. Toward a foundational typed assembly language. In *Symposium on Principles of Programming Languages*, 2003.

[3] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275. ACM Press, 1999.

[4] ECMA. *Standard ECMA-335 Common Language Infrastructure (CLI)*. 2006.

[5] Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In *15th European Symposium on Programming (ESOP'06)*, 2006.

[6] Jean-Yves Girard. Linear logic. In *Theoretical Computer Science*, 1987.

[7] Chris Hawblitzel. Linear types for aliased resources (extended version). Technical Report MSR-TR-2005-141, Microsoft Research, 2005.

[8] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[9] Limin Jia, Frances Spalding [Perry], David Walker, and Neal Glew. Certifying compilation for a language with stack allocation. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 407–416, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Patrick Lincoln, John C. Mitchell, Andre Scedrov, and Natarajan Shankar. Decision problems for propositional linear logic. *Ann. Pure Appl. Logic*, 56(1-3):239–311, 1992.

[11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 1999.

[12] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 13(5):957–959, 2003.

[13] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 21, pages 527–568. ACM Press, 1999.

[14] George Necula. Proof-Carrying Code. In *ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[15] Frances Perry, Chris Hawblitzel, and Juan Chen. Simple and flexible stack types. Technical Report MSR-TR-2007-51, Microsoft Corporation. ftp://ftp.research.microsoft.com/pub/tr/TR-2007-51.pdf.

[16] Frances Perry, Chris Hawblitzel, and Juan Chen. Proofs for SST, 2007. http://research.microsoft.com/users/juanchen/stack.

[17] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000)*, 2002.

[18] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *In European Symposium on Programming*, 2000.

[19] P. L. Wadler. A taste of linear logic. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science, Gdánsk*, New York, NY, 1993. Springer-Verlag.

[20] David Walker. Mechanical reasoning about low-level programs. lecture notes, http://www.cs.cmu.edu/˜dpw/papers.html, 2001.

A. SST Syntax

location	ℓ	$::=$	$\eta \mid \text{base} \mid \text{next}(\ell) \mid p$
labeled stack type	ς	$::=$	$\ell : \sigma$
unlabeled stack type	σ	$::=$	$\rho \mid \text{Empty} \mid \tau :: \varsigma$
			$\mid \sigma \wedge \{\ell : \tau\}$
type	τ	$::=$	$\text{int} \mid \text{Nonsense} \mid \text{Ptr}(\ell)$
			$\mid \text{HeapPtr}(\tau) \mid \forall[\Delta](\Gamma, \varsigma)$
stack loc	d	$::=$	$\text{base} \mid \text{next}(d)$
word value	w	$::=$	$i \mid \text{nonsense} \mid p \mid d$
			$\mid w[\ell] \mid w[\sigma]$
operand	o	$::=$	$r \mid w \mid o[\ell] \mid o[\sigma]$
instr	ins	$::=$	$\text{mov } r, o \mid \text{add } r, o$
			$\mid \text{sub } r, o \mid \text{ladd } r, i$
			$\mid \text{load } r_1, [r_2 + i]$
			$\mid \text{store } [r_1 + i], r_2$
			$\mid \text{jumpif0 } r, o$
			$\mid \text{heapalloc } r = \langle o \rangle$
			$\mid (\eta, r) = \text{unpack}(o)$
heap value	v	$::=$	$\text{block} \mid \langle w \rangle$
block	block	$::=$	$\forall[\Delta](\Gamma, \varsigma)\ b$
block body	b	$::=$	$\text{ins}; b \mid \text{jump } o$
loc env	Δ	$::=$	$\bullet \mid \eta; \Delta \mid \rho; \Delta$
heap	H	$::=$	$p_1 \mapsto v_1, \ldots, p_n \mapsto v_n$
heap env	Ψ	$::=$	$p_1 \mapsto \tau_1, \ldots, p_n \mapsto \tau_n$
reg bank	R	$::=$	$r_1 \mapsto w_1, \ldots, r_n \mapsto w_n$
reg file	Γ	$::=$	$r_1 \mapsto \tau_1, \ldots, r_n \mapsto \tau_n$
stack value	s	$::=$	$\text{empty} \mid w :: s$
program	P	$::=$	(H, R, s, b)

We use the following abbreviation:

$$(\tau_n \ldots \tau_1)@(\ell : \sigma) = \text{next}^n(\ell) : \tau_n :: \ldots :: \text{next}^1(\ell) : \tau_1 :: \ell : \sigma$$

B. SST Semantics

B.1 Well-formedness

$\boxed{\Delta \vdash \ell}$

$$\frac{}{\{\ldots, \eta, \ldots\} \vdash \eta}\ \text{wf-l-var} \qquad \frac{}{\Delta \vdash \text{base}}\ \text{wf-l-base}$$

$$\frac{\Delta \vdash \ell}{\Delta \vdash \text{next}(\ell)}\ \text{wf-l-next} \qquad \frac{}{\Delta \vdash p}\ \text{wf-l-p}$$

$\boxed{\Delta \vdash \varsigma}$

$$\frac{\Delta \vdash \ell}{\Delta \vdash \ell : \text{Empty}}\ \text{wf-S-empty} \qquad \frac{\Delta \vdash \ell \quad \rho \in \Delta}{\Delta \vdash \ell : \rho}\ \text{wf-S-P}$$

$$\frac{\begin{array}{c}\Delta \vdash \ell \qquad \Delta \vdash \tau \qquad \Delta \vdash \ell_q : \sigma \\ \forall\, \ell'_q, \tau', \sigma' : \tau = \tau' \text{ if } \ell_q : \sigma \Rightarrow \ell'_q : (\sigma' \wedge \{\ell : \tau'\})\end{array}}{\Delta \vdash \ell_q : (\sigma \wedge \{\ell : \tau\})}\ \text{wf-S-alias}$$

$$\frac{\begin{array}{c}\Delta \vdash \ell \qquad \Delta \vdash \tau \qquad \Delta \vdash \varsigma \\ \forall\, \ell'_q, \ell', \tau', \sigma' : \ell \neq \ell' \text{ if } \varsigma \Rightarrow \ell'_q : (\sigma' \wedge \{\ell' : \tau'\})\end{array}}{\Delta \vdash \ell : (\tau :: \varsigma)}\ \text{wf-S-concat}$$

$\boxed{\Delta \vdash \tau}$

$$\frac{}{\Delta \vdash \text{int}}\ \text{wf-t-int} \qquad \frac{}{\Delta \vdash \text{Nonsense}}\ \text{wf-t-ns}$$

$$\frac{\Delta \vdash \tau}{\Delta \vdash \text{HeapPtr}(\tau)}\ \text{wf-t-hp} \qquad \frac{\Delta \vdash \ell}{\Delta \vdash \text{Ptr}(\ell)}\ \text{wf-t-single}$$

$$\frac{\Delta, \Delta' \vdash \Gamma' \quad \Delta, \Delta' \vdash \varsigma' \quad \Delta \cap \Delta' = \{\}}{\Delta \vdash \forall[\Delta'](\Gamma', \varsigma')}\ \text{wf-t-code}$$

$\boxed{\Delta \vdash \Gamma}$

$$\frac{\ldots\ \Delta \vdash \tau\ \ldots}{\Delta \vdash \{\ldots, r \mapsto \tau, \ldots\}}\ \text{wf-G}$$

B.2 Static Semantics

$\boxed{\Delta; \Psi; \Gamma \vdash o : \tau}$

$$\frac{}{\Delta; \Psi; \Gamma \vdash r : \Gamma(r)}\ \text{o-reg} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash i : \text{int}}\ \text{o-int}$$

$$\frac{}{\Delta; \Psi; \Gamma \vdash \text{nonsense} : \text{Nonsense}}\ \text{o-ns}$$

$$\frac{}{\Delta; \Psi; \Gamma \vdash p : \Psi(p)}\ \text{o-p-H} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash p : \text{Ptr}(p)}\ \text{o-p}$$

$$\frac{}{\Delta; \Psi; \Gamma \vdash d : \text{Ptr}(d)}\ \text{o-d}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \forall[\eta, \Delta'](\Gamma', \varsigma) \quad \Delta \vdash \ell}{\Delta; \Psi; \Gamma \vdash o[\ell] : \forall[\Delta'](\Gamma'[\ell/\eta], \varsigma[\ell/\eta])}\ \text{o-inst-l}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \forall[\rho, \Delta'](\Gamma', \varsigma) \quad \Delta \vdash \sigma}{\Delta; \Psi; \Gamma \vdash o[\sigma] : \forall[\Delta'](\Gamma'[\sigma/\rho], \varsigma[\sigma/\rho])}\ \text{o-inst-Q}$$

$\boxed{\vdash (\Gamma, \varsigma)\{r \leftarrow \tau\}(\Gamma', \varsigma')}$

$$\frac{r \neq \text{sp} \quad \Gamma' = \Gamma[r \mapsto \tau]}{\vdash (\Gamma, \varsigma)\{r \leftarrow \tau\}(\Gamma', \varsigma)}\ \text{a-not-esp}$$

$$\frac{\vdash \text{Resize}(\ell, \varsigma) = \varsigma' \quad \Gamma' = \Gamma[\text{sp} \mapsto \text{Ptr}(\ell)]}{\vdash (\Gamma, \varsigma)\{\text{sp} \leftarrow \text{Ptr}(\ell)\}(\Gamma', \varsigma')}\ \text{a-esp}$$

$\boxed{\vdash \text{Resize}(\ell, \varsigma) = \varsigma'}$

$$\frac{\varsigma \Rightarrow \vec{\tau}\ @(\ell : \sigma)}{\vdash \text{Resize}(\ell, \varsigma) = \ell : \sigma}\ \text{s-shrink}$$

$$\frac{\varsigma' = (\text{Nonsense}_n; \ldots; \text{Nonsense}_1)@(\ell : \sigma)}{\vdash \text{Resize}(\text{next}^n(\ell), \ell : \sigma) = \varsigma'}\ \text{s-grow}$$

$\boxed{\varsigma \vdash \ell + i = \ell'}$

$$\frac{\varsigma \Rightarrow \vec{\tau}\ @(\ell : \sigma)}{\varsigma \vdash \ell + n = \text{next}^n(\ell)}\ \text{s-offset-next}$$

$$\frac{\varsigma \Rightarrow \vec{\tau}\ @(\ell : \sigma)}{\varsigma \vdash \text{next}^n(\ell) + (-n) = \ell}\ \text{s-offset-prev}$$

$\boxed{\varsigma \vdash \ell : \tau}$

$$\frac{\varsigma \Rightarrow \ell' : (\sigma \wedge \{\ell : \tau\})}{\varsigma \vdash \ell : \tau}\ \text{s-lookup}$$

$\boxed{\varsigma \vdash \ell \leftarrow \tau \rightsquigarrow \varsigma'}$

$$\frac{\varsigma \vdash \ell : \tau}{\varsigma \vdash \ell \leftarrow \tau \rightsquigarrow \varsigma}\ \text{s-update-weak}$$

$$\frac{\varsigma \Rightarrow \vec{\tau}\ @(\ell : \tau :: \varsigma')}{\varsigma \vdash \ell \leftarrow \tau' \rightsquigarrow \vec{\tau}\ @(\ell : \tau' :: \varsigma')}\ \text{s-update-strong}$$

$\boxed{\Gamma \Rightarrow \Gamma'}$

$$\frac{\Gamma' \subseteq \Gamma}{\Gamma \Rightarrow \Gamma'}\ \text{G-imp}$$

$\boxed{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{ins}\}(\Gamma'; \varsigma')}$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \tau \quad \vdash (\Gamma, \varsigma)\{r \leftarrow \tau\}(\Gamma', \varsigma')}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{mov}\ r, o\}(\Gamma'; \varsigma')}\ \text{i-mov}$$

$$\frac{\begin{array}{c}\Gamma(r) = \text{Ptr}(\ell)\ \varsigma \vdash \ell + i = \ell' \\ \vdash (\Gamma, \varsigma)\{r \leftarrow \text{Ptr}(\ell')\}(\Gamma', \varsigma')\end{array}}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{ladd}\ r, -4 * i\}(\Gamma'; \varsigma')}\ \text{i-ladd}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \text{int} \quad r \neq \text{sp} \quad \Gamma(r) = \text{int}}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{add}\ r, o\}(\Gamma; \varsigma)}\ \text{i-add}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \text{int} \quad r \neq \text{sp} \quad \Gamma(r) = \text{int}}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{sub}\ r, o\}(\Gamma; \varsigma)}\ \text{i-sub}$$

$$\frac{\Gamma(r_2) = \text{HeapPtr}(\tau) \quad \vdash (\Gamma, \varsigma)\{r_1 \leftarrow \tau\}(\Gamma', \varsigma')}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{load}\ r_1, [r_2 + 0]\}(\Gamma'; \varsigma')}\ \text{i-load-p}$$

$$\frac{\Gamma(r_2) = \tau \quad \Gamma(r_1) = \text{HeapPtr}(\tau)}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{store}\ [r_1 + 0], r_2\}(\Gamma; \varsigma)}\ \text{i-store-p}$$

$$\frac{\begin{array}{c}\Gamma(r_2) = \text{Ptr}(\ell) \quad \varsigma \vdash \ell + i = \ell' \\ \varsigma \vdash \ell' : \tau \quad \vdash (\Gamma, \varsigma)\{r_1 \leftarrow \tau\}(\Gamma', \varsigma')\end{array}}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{load}\ r_1, [r_2 + (-4 * i)]\}(\Gamma'; \varsigma')}\ \text{i-load-concat}$$

$$\frac{\begin{array}{c}\Gamma(r_1) = \text{Ptr}(\ell) \quad \Gamma(r_2) = \tau \\ \varsigma \vdash \ell + i = \ell' \quad \varsigma \vdash \ell' \leftarrow \tau \rightsquigarrow \varsigma'\end{array}}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{store}\ [r_1 + (-4 * i)], r_2\}(\Gamma; \varsigma')}\ \text{i-store-concat}$$

$$\frac{\begin{array}{c}\Gamma(r_2) = \text{Ptr}(\ell)\ \varsigma \vdash \ell : \tau \\ \vdash (\Gamma, \varsigma)\{r_1 \leftarrow \tau\}(\Gamma', \varsigma')\end{array}}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{load}\ r_1, [r_2 + 0]\}(\Gamma'; \varsigma')}\ \text{i-load-aliased}$$

$$\frac{\Gamma(r_1) = \text{Ptr}(\ell) \quad \varsigma \vdash \ell : \tau \quad \Gamma(r_2) = \tau}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{store}\ [r_1 + 0], r_2\}(\Gamma; \varsigma)}\ \text{i-store-aliased}$$

$$\frac{\begin{array}{c}\Delta; \Psi; \Gamma \vdash o : \tau \\ \vdash (\Gamma, \varsigma)\{r \leftarrow \text{HeapPtr}(\tau)\}(\Gamma', \varsigma')\end{array}}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{heapalloc}\ r = \langle o \rangle\}(\Gamma'; \varsigma')}\ \text{i-heapalloc}$$

$$\frac{\begin{array}{c}\Gamma(r) = \text{int}\ \Delta; \Psi; \Gamma \vdash o : \forall[\,](\Gamma', \varsigma') \\ \Gamma \Rightarrow \Gamma'\ \varsigma \Rightarrow \varsigma'\end{array}}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{jumpif0}\ r, o\}(\Gamma; \varsigma)}\ \text{i-jump0}$$

$\boxed{\vdash P}$

$$\frac{\vdash H : \Psi \quad \bullet; \Psi \vdash s : \varsigma \quad \bullet; \Psi \vdash R : \Gamma \quad \bullet; \Psi; \Gamma; \varsigma \vdash b}{\vdash (H, R, s, b)}\ \text{m-tp}$$

$\boxed{\varsigma \Rightarrow \varsigma'}$

$$\frac{\varsigma \Rightarrow \varsigma'}{\ell : \tau :: \varsigma \Rightarrow \ell : \tau :: \varsigma'}\ \text{s-imp-concat} \qquad \frac{\ell : \sigma \Rightarrow \ell : \sigma'}{\ell : (\sigma \wedge \{\ell_t : \tau\}) \Rightarrow \ell : (\sigma' \wedge \{\ell_t : \tau\})}\ \text{s-imp-alias}$$

$$\frac{}{\varsigma \Rightarrow \varsigma}\ \text{s-imp-eq} \qquad \frac{}{\ell : (\tau :: \varsigma) \Rightarrow \ell : (\tau :: \varsigma \wedge \{\ell : \tau\})}\ \text{s-imp-add-alias}$$

$$\frac{\varsigma_1 \Rightarrow \varsigma_2 \quad \varsigma_2 \Rightarrow \varsigma_3}{\varsigma_1 \Rightarrow \varsigma_3}\ \text{s-imp-trans} \qquad \frac{}{\ell : (\sigma \wedge \{\ell_t : \tau\}) \Rightarrow \ell : \sigma}\ \text{s-imp-drop-alias}$$

$$\frac{}{\ell : (\tau_1 :: \ell_q : (\sigma \wedge \{\ell_2 : \tau_2\})) \Rightarrow \ell : ((\tau_1 :: \ell_q : \sigma) \wedge \{\ell_2 : \tau_2\})}\ \text{s-imp-expand-alias}$$

$$\frac{\varsigma \Rightarrow \ell : (\sigma \wedge \{\ell_1 : \tau_1\}) \quad \varsigma \Rightarrow \ell : (\sigma \wedge \{\ell_2 : \tau_2\})}{\varsigma \Rightarrow \ell : (\sigma \wedge \{\ell_1 : \tau_1\} \wedge \{\ell_2 : \tau_2\})}\ \text{s-imp-merge-alias}$$

Figure 3. Stack Implication Rules

$\boxed{\vdash H : \Psi}$

$$\frac{\Psi = \{\ldots, p \mapsto \tau, \ldots\}\ H = \{\ldots, p \mapsto v, \ldots\} \quad \ldots\ \bullet; \Psi \vdash v : \tau\ \ \ldots}{\vdash H : \Psi}\ \text{h-tp}$$

$\boxed{\Delta; \Psi \vdash R : \Gamma}$

$$\frac{\Gamma = \{\ldots, r \mapsto \tau, \ldots\}\ R = \{\ldots, r \mapsto w, \ldots\} \quad \ldots\ \Delta; \Psi; \bullet \vdash w : \tau\ \ \ldots}{\Delta; \Psi \vdash R : \Gamma}\ \text{g-tp}$$

$\boxed{\Delta; \Psi \vdash s : \varsigma}$

$$\frac{}{\Delta; \Psi \vdash \text{empty} : (\text{base} : \text{Empty})}\ \text{s-base}$$

$$\frac{\Delta; \Psi \vdash s : (\ell : \sigma) \quad \Delta; \Psi; \bullet \vdash w : \tau}{\Delta; \Psi \vdash w :: s : (\text{next}(\ell) : \tau :: \ell : \sigma)}\ \text{s-concat}$$

$$\frac{\Delta; \Psi, \{p \mapsto \text{HeapPtr}(\tau)\} \vdash s : (\ell : \sigma)}{\Delta; \Psi, \{p \mapsto \text{HeapPtr}(\tau)\} \vdash s : (\ell : (\sigma \wedge \{p : \tau\}))}\ \text{s-alias}$$

$$\frac{\Delta; \Psi \vdash s : \varsigma \quad \varsigma \Rightarrow \varsigma'}{\Delta; \Psi \vdash s : \varsigma'}\ \text{s-imp}$$

$\boxed{\Delta; \Psi; \Gamma; \varsigma \vdash b}$

$$\frac{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{ins}\}(\Gamma'; \varsigma')\ \Delta; \Psi; \Gamma'; \varsigma' \vdash b}{\Delta; \Psi; \Gamma; \varsigma \vdash \text{ins}; b}\ \text{b-ins}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \forall[\,](\Gamma', \varsigma')\ \Gamma \Rightarrow \Gamma' \quad \varsigma \Rightarrow \varsigma'}{\Delta; \Psi; \Gamma; \varsigma \vdash \text{jump } o}\ \text{b-jump}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \text{HeapPtr}(\tau) \qquad r \neq \text{sp} \qquad \eta \notin \Delta \quad (\Delta; \eta); \Psi; \Gamma[r \mapsto \text{Ptr}(\eta)]; \ell : (\sigma \wedge \{\eta : \tau\}) \vdash b}{\Delta; \Psi; \Gamma; \ell : \sigma \vdash (\eta, r) = \text{unpack}(o)}\ \text{b-unpack}$$

$\boxed{\Psi \vdash \text{block}}$

$$\frac{\Delta; \Psi; \Gamma; \varsigma \vdash b}{\Psi \vdash \forall[\Delta](\Gamma, \varsigma)\ b}\ \text{block-tp}$$

$\boxed{\Delta; \Psi \vdash v : \tau}$

$$\frac{\Psi \vdash \forall[\Delta'](\Gamma', \varsigma')\ b \quad \Delta \vdash \forall[\Delta'](\Gamma', \varsigma')}{\Delta; \Psi \vdash \forall[\Delta'](\Gamma', \varsigma')\ b : \forall[\Delta'](\Gamma', \varsigma')}\ \text{v-code}$$

$$\frac{\Delta; \Psi; \bullet \vdash w : \tau}{\Delta; \Psi \vdash \langle w \rangle : \text{HeapPtr}(\tau)}\ \text{v-hp}$$

B.3 Dynamic Semantics

$\boxed{d + i = d'}$

$$\begin{array}{lcl} d + 0 & = & d \\ d + (n+1) & = & \text{next}(d) + n \\ \text{base} + (-(n+1)) & = & \text{base} \\ \text{next}(d) + (-(n+1)) & = & d + (-n) \end{array}$$

$\boxed{\text{size}(s) = d}$

$$\begin{array}{lcl} \text{size}(\text{empty}) & = & \text{base} \\ \text{size}(w :: s) & = & \text{next}(\text{size}(s)) \end{array}$$

$\boxed{\text{resize}(d, s) = s'}$

$$\begin{array}{l} \text{resize}(\text{size}(s), s) \ = \ s \\ \text{resize}(\text{size}(s) + (n+1), s) \ = \ \text{nonsense} :: \text{resize}(\text{size}(s) + n, s) \\ \text{resize}(\text{size}(s) + (-(n+1)), w :: s) \ = \ \text{resize}(\text{size}(s) + (-n), s) \end{array}$$

$\boxed{s(d) = w}$

$$\frac{}{(w :: s)(\text{size}(w :: s)) = w}\ \text{s-lookup-top}$$

$$\frac{s(d) = w}{(w' :: s)(d) = w}\ \text{s-lookup}$$

$\boxed{s' = s[d \leftarrow w]}$

$$\frac{d = \text{size}(w :: s)}{w' :: s = (w :: s)[d \leftarrow w']}\ \text{s-assign-top}$$

$$\frac{s' = s[d \leftarrow w]}{w' :: s' = (w' :: s)[d \leftarrow w]}\ \text{s-assign}$$

$\boxed{R \vdash o \mapsto w}$

$$\frac{}{R \vdash r \mapsto R(r)}\ \text{eo-r} \qquad \frac{}{R \vdash w \mapsto w}\ \text{eo-w} \qquad \frac{R \vdash o \mapsto w}{R \vdash o[\ell] \mapsto w[\ell]}\ \text{eo-inst-l} \qquad \frac{R \vdash o \mapsto w}{R \vdash o[\sigma] \mapsto w[\sigma]}\ \text{eo-inst-Q}$$

$\boxed{(R, s)\{r \leftarrow w\}(R', s')}$

$$\frac{r \neq \text{sp} \quad R' = R[r \mapsto w]}{(R, s)\{r \leftarrow w\}(R', s)}\ \text{u-not-esp} \qquad \frac{R' = R[\text{sp} \mapsto d]}{(R, s)\{\text{sp} \leftarrow d\}(R', \text{resize}(d, s))}\ \text{u-esp}$$

$\boxed{P \rightarrow P'}$

$$\frac{R \vdash o \mapsto w \quad (R, s)\{r \leftarrow w\}(R', s')}{(H, R, s, (\text{mov}\ r, o;\ b)) \rightarrow (H, R', s', b)}\ \text{e-mov}$$

$$\frac{R \vdash r \mapsto d \quad (R, s)\{r \leftarrow d + i\}(R', s')}{(H, R, s, (\text{ladd}\ r, -4 * i;\ b)) \rightarrow (H, R', s', b)}\ \text{e-ladd}$$

$$\frac{R \vdash r \mapsto i_1 \quad R \vdash o \mapsto i_2 \quad (R, s)\{r \leftarrow i_1 + i_2\}(R', s')}{(H, R, s, (\text{add}\ r, o;\ b)) \rightarrow (H, R', s', b)}\ \text{e-add}$$

$$\frac{R \vdash r \mapsto i_1 \quad R \vdash o \mapsto i_2 \quad (R, s)\{r \leftarrow i_1 - i_2\}(R', s')}{(H, R, s, (\text{sub}\ r, o;\ b)) \rightarrow (H, R', s', b)}\ \text{e-sub}$$

$$\frac{R \vdash r_2 \mapsto p \quad H(p) = \langle w \rangle \quad (R, s)\{r_1 \leftarrow w\}(R', s')}{(H, R, s, (\text{load}\ r_1, [r_2 + 0]; b)) \rightarrow (H, R', s', b)}\ \text{e-load-p}$$

$$\frac{R \vdash r_2 \mapsto d \quad s(d + i) = w \quad (R, s)\{r_1 \leftarrow w\}(R', s')}{(H, R, s, (\text{load}\ r_1, [r_2 + (-4 * i)]; b)) \rightarrow (H, R', s', b)}\ \text{e-load-d}$$

$$\frac{R \vdash r_1 \mapsto p \quad H(p) = \langle w \rangle \quad R \vdash r_2 \mapsto w'}{(H, R, s, (\text{store}\ [r_1 + 0], r_2; b)) \rightarrow (H[p \leftarrow \langle w' \rangle], R, s, b)}\ \text{e-store-p}$$

$$\frac{R \vdash r_1 \mapsto d \quad R \vdash r_2 \mapsto w \quad s' = s[d + i \leftarrow w]}{(H, R, s, (\text{store}\ [r_1 + (-4 * i)], r_2; b)) \rightarrow (H, R, s', b)}\ \text{e-store-d}$$

$$\frac{R \vdash o \mapsto w \quad p \notin \text{domain}(H) \quad H' = H, p \mapsto \langle w \rangle \quad (R, s)\{r \leftarrow p\}(R', s')}{(H, R, s, (\text{heapalloc}\ r = \langle o \rangle; b)) \rightarrow (H', R', s', b)}\ \text{e-heapalloc}$$

$$\frac{R \vdash r \mapsto i \quad i \neq 0}{(H, R, s, (\text{jumpif0}\ r, o; b)) \rightarrow (H, R, s, b)}\ \text{e-jump0-false}$$

$$\frac{R \vdash r \mapsto 0 \quad R \vdash o \mapsto p[\text{subst}] \quad H(p) = \forall[\Delta](\Gamma, \varsigma)\ b_2}{(H, R, s, (\text{jumpif0}\ r, o; b_1)) \rightarrow (H, R, s, b_2[\text{subst}/\Delta])}\ \text{e-jump0-true}$$

$$\frac{R \vdash o \mapsto p \quad (R, s)\{r \leftarrow p\}(R', s')}{(H, R, s, ((\eta, r) = \text{unpack}(o); b)) \rightarrow (H, R', s', b[p/\eta])}\ \text{e-unpack}$$

$$\frac{R \vdash o \mapsto p[\text{subst}] \quad H(p) = \forall[\Delta](\Gamma, \varsigma)\ b}{(H, R, s, \text{jump}\ o) \rightarrow (H, R, s, b[\text{subst}/\Delta])}\ \text{e-jump}$$

Figure 4. Instruction Evaluation Rules

See the Pet in the Beast: How to Limit Effects of Aliasing

Franz Puntigam
Technische Universität Wien, Institut für Computersprachen
franz@complang.tuwien.ac.at

Abstract

Aliasing is quite powerful, but difficult to control. Often clients need exclusive access to objects for some concerns, and sometimes we see no other way than to ensure this by controlling aliasing. Instead, we propose to restrict what clients can do when accessing objects. To invoke methods in an object clients need tokens issued by this object. Static type checking enforces the tokens to be available and ensures exclusive access for specific concerns without avoiding aliasing. We show by examples how this concept works and discuss several possibilities to improve its flexibility.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Object-Oriented Programming, Aliasing

Keywords Types, Tokens

1. Motivation and Overview

Aliasing is like a beast causing troubles. It shows up where we do not expect it and perverts our statements. It is slippery and escapes when we think we have caught it.

Aliasing is also like a pet. Object-oriented programmers love it. It opens doors to objects that seem far far away. Science fiction authors would be surprised if they knew how easy we walk from one object to another at a completely different part of the world.

The beast and the pet are actually the same animal. Aliasing gives programming languages much expressive power, so much that we easily lose control. Programming systems become weaker whenever we cage or tame the beast. We must be careful not to destroy flexibility: Although it is possible to develop nearly every kind of system without the undesirable properties of aliasing (for example, in a referentially transparent functional language) we have to do so with considerably less flexibility in structuring the code. Such flexibility is essential in object-oriented programming to achieve good factorization.

In this paper we discuss an approach to annotate object references with constraints on how to access referenced objects. This approach fully supports aliasing; there is no cage for it as in many other approaches [2, 3, 12, 15, 39]. However, we limit what the beast can do by ensuring that constraints on references are preserved when introducing new aliases. For example, if we want a specific method in an object to be invokable at most once, then we annotate the only reference existing on object creation with a corresponding constraint. After introducing further references to the object we still have this property: There is only one reference through which the message can be sent although we usually do not know where to find this reference. We need not restrict aliasing by itself; we just limit effects of aliasing.

We express what can be done through an object reference by a set of tokens (or just names) associated with the reference. Method specifications give semantics to tokens: A method can require specific tokens to be associated with the reference through which it is invoked; these tokens are removed from the reference on invocation, and further tokens (specified by the method) can be added on return. We express constraints by tokens because they are easily understood by both programmers and tools like compilers.

This approach was introduced as part of a type system expressing synchronization to ensure linearity at the presence of aliasing [29]. Applications of this technique are usually related to synchronization and coordination. In the above example of a single invocation, all clients of an object must be coordinated such that at most one of them invokes the specific method. Such kinds of coordination are inevitably connected with constraints on single references (as opposed to the whole referenced object) at the presence of aliasing.

The goal of this paper is to survey how this approach works and show by examples what can be done with it and where its limits are. Thereby, the focus is on limiting effects of aliasing, not synchronization and coordination. In Section 2 we give the basics of our token-based approach, and in Section 3 we show how to distribute tokens within a system. In Section 4 we briefly describe static type checking. In the remaining sections we use various concepts to add flexibility – dependences between tokens in Section 5, relationships between values and tokens in Section 6, type parameters in Section 7, and a dynamic concept in Section 8 – before we discuss related work in Section 9 and give concluding remarks in Section 10.

2. Tokens to Ensure Limited Access

We show how to specify constraints by examples in a Java-like pseudo-language. The first example gives a simplified interface of a window, where method invocations depend on tokens (in square brackets – tokens removed on invocation to the left and those added on return to the right of arrows):

```
interface Window {
   [init -> shown,ready] void initialize (...);
   [ready -> ready] void update (...);
   [shown -> icon] void iconify ();
   [icon -> shown] void uniconify ();
   [shown,ready ->] void close ();
   int getCreationTime ();
}
```

In our pseudo-language, brackets denote token sets associated with types and methods, they do not denote arrays. Let us assume that new windows are of type `Window[init]`, this is, we have a reference to an instance of `Window` associated with a token `init`. Through this reference we can invoke only `initialize` and `getCreationTime`. All other methods require tokens (as specified to the left of `->`) not available in the reference; they are not invokable. When we invoke `initialize` the type of the reference changes first to `Window[]` (or equivalently just `Window`) and on return to `Window[shown,ready]`. Further methods become invokable, but `initialize` cannot be invoked again. We can distribute references to the window all over the system (see Section 3 how to do that in a type-save way). The type checker prevents tokens from being duplicated thereby. Maybe, we get a reference of type `Window[ready]` in a client feeding the window with new data, a reference of type `Window[shown]` in a window control panel, and any number of references of type `Window` through which only `getCreationTime` can be invoked. However, we cannot get two references to the same window, both of type `Window[ready]`, because there exists always at most one token `ready`. In the control panel we can invoke `iconify` and `uniconify` only in alternation: Invoking `iconify` changes the type associated with the window reference to `Window[icon]`, and invoking `uniconify` again to `Window[shown]`. The method `close` becomes invokable only if we rearrange the system such that all available tokens (`ready` and `shown`) occur again in the type of a single reference. After invoking `close` all tokens are gone, and only `getCreationTime` remains invokable.

The example shows how we can easily specify nontrivial constraints on method invocations and how clients are forced to satisfy them. This technique is expressive: We can specify all prefix-closed trace sets [30]. Clients that hold tokens have got partial control over the corresponding object: The client holding `ready` is the only one being able to invoke `update`, and the client holding `icon` or `shown` completely controls whether the window is iconified. Clients cannot influence each other in this respect. Moreover, a client who holds any token of a window can prevent `close` ever to be invoked. This kind of "separation of concerns" works without any knowledge about aliases in the system. We need not know which client has control over a concern. This client can even change dynamically.

To ensure limited access we usually want to have at most one token of each name per object. Separation would be weaker if we had several tokens `ready` and possibly several clients invoking `update` in the same window object. As proposed in [29] this approach supports several tokens of the same name in order to express limited resources (that are not necessarily limited to one; for example, buffer sizes) for the purpose of synchronization. For the purpose of aliasing control we need no limited resources of this kind. In this paper we implicitly assume tokens to occur at most once per object (this is, in the types of all references to the same object).

Explicit result types of constructors play an important role in specifying initial object states:

```
class MyWindow implements Window {
   MyWindow[init] () {};
   ...
}
```

An invocation of `new MyWindow()` returns a new instance with a single token `init`. Based on this information we can compute the maximum of tokens for this object available in the whole system (see Section 4). Since `MyWindow` does not add tokens to those inherited from `Window`, there can always be at most an `init`, or a `ready` and either a `shown` or an `icon` in the types of all references to an instance of `MyWindow`.

3. How to Distribute Tokens

In the next example we show how to handle tokens in types of parameters and variables:

```
class Test {
   void play (Window[ready -> ready] w) {
      w.update(...);
      w.update(...);
   }
   void blink (Window[icon -> shown] w) {
      w.uniconify();
      w.iconify();
      w.uniconify();
   }
   Window[ready] win;
   [unique -> unique]
    Window[ready] swap (Window[ready ->] w) {
      Window[ready] old = win;
      win = w;
      return old;
   }
   [unique -> unique] void condUpdate() {
      if (win != null) { win.update(...); }
   }
   Test[unique] () { win = null; }
}
```

Let `y` be a variable of type `Window[ready,icon]` and `x` one of type `Test[unique]`. We can invoke `x.play(y)` since `y` has the token `ready` as required in the formal parameter type to the left of `->`. On invocation the type of `y` changes to `Window[icon]` (this is, `ready` is removed) and on return from `play` again to `Window[ready,icon]` (this is, the token to the right of `->` in the formal parameter type is added). In the body of `play` the parameter `w` has a token `ready` on invocation as well as on return; we can invoke `update` as often as we want to do so.

Invocations of `blink` change argument types: On return from `x.blink(y)` variable `y` will be of type `Window[ready,shown]`. In the body of `blink` we must invoke `uniconify` at least once to ensure `w` to have the appropriate token on return. We can invoke `x.play(y)` and `x.blink(y)` in any ordering and even concurrently because the token sets required from `y` (as well as the empty token sets required from `x`) do not overlap.

Parameter passing does not produce or consume tokens. Tokens just move from the argument type to the parameter type on invocation and vice versa on return.

Whenever we introduce an alias (in this case by binding a formal parameter to an argument) we perform *type splitting:* The tokens specified in the argument type are split into two groups. Tokens specified in the formal parameter type (to the left of the arrow) move to the formal parameter while all other tokens remain in the argument's type. After return the formal parameter is no longer in use. We combine the previously split types again; thereby tokens (specified to the right of the arrow) move from the formal parameter to the argument.

Assignment resembles parameter passing on method invocation: When assigning a reference to a variable where the variable type specifies tokens, these tokens are removed from the reference; this is, the tokens move from the assigned value to the variable. In the body of `swap` the token `ready` moves from the parameter `w` to the instance variable `win`. Since the token finally belongs to `win`, it cannot move back to the argument `y` on return from `x.swap(y)`.

Types specifying tokens in square brackets frequently change. For example, `w` in the body of `blink` is of type `Window[icon]` before invoking `uniconify` and of type `Window[shown]` afterwards. There is no difficulty for a type checker and usually also for a human reader to determine what is the current type of a local variable at some position in the program. However, such type changes cause troubles on instance variables: There can be independent accesses of the same variable through concurrent threads as well as through aliases. If one of the clients accessing the variable causes tokens to be removed from the variable, others do not know about this change and can assume the tokens still to be available; there can be an unexpected and undesired duplication of tokens. To avoid such problems we require tokens of instance variables to be visible only if we can exclude simultaneous accesses through concurrent threads and aliases. In our example we get uniqueness by requiring the token `unique` on `x` when executing `x.swap(y)` and `x.condUpdate()`.[1] Static type checking ensures such variable accesses to be actually unique (see Section 4). Furthermore, such variables must not be accessed from outside (except though getter and setter methods). Of course, on return from methods all such variables must hold their declared tokens.

As in all Java-like languages `null` is an appropriate instance of each reference type. Since no method is invokable through `null`, we can assume this special value to be associated with any token. Tokens do not compromise the use of `null`.

If we have two references of the types `Window[ready]` and `Window[shown]`, then it is in general not possible to invoke `close` through any of them. However, the following method allows us to combine the token sets:

```
Window[ready,shown] comb (Window[ready ->] x,
                          Window[shown ->] y) {
   if (x == y) { return x; }
   else { return null; }
}
```

The method is correct because in this case `x` and `y` are known to be aliases with the common type `Window[ready,shown]`. Whenever we know two variables (or parameters) to refer to the same object (after comparing identity) we assume all tokens belonging to any of the two variables to belong to both of them. The essential part is just the conditional statement with an identity comparison as condition; the rest of this example just gives us a setting where this statement may be useful.

Our approach supports subtyping considering tokens. We give just a raw idea of it (see [29, 30, 33] for more complete descriptions): Subtypes specify all (relevant) tokens specified by supertypes. Hence, `MyWindow[ready,shown]` is a subtype of `Window[ready]`, but is not related to `Window[icon]` by subtyping. Methods declared in subtypes have to the left of `->` at most and to the right at least those tokens that occur to the left or right of the arrow in the corresponding method declaration in the supertype. Irrelevant tokens (these are tokens no method depends upon) need not be considered. As a consequence we can invoke at least each sequence of methods through a reference to an instance of a subtype that we can invoke through a reference to an instance of a corresponding supertype. Supertypes are more restrictive than (or equal to) subtypes.

4. Static Type Checking

Static type checking in our approach is rather simple and can be performed at a class by class basis (separate compilation). Programmers give all information the checker needs by specifying tokens in types and together with methods. The type checker must ensure all specified types and tokens to be consistent (which is much simpler than inferring information about aliasing or synchronization from a program). It can do so by a single walk through the code of a class. In detail, the checker has to ensure the following properties:

1. *At any time there cannot be several tokens of the same name for the same object.* To ensure this property we apply a simple fixed-point algorithm to compute for each class an upper bound of token sets that can become available: Initially we have the sets of tokens specified in constructors (one set per constructor). We construct further token sets by updating each token set according to each method where the token set contains all tokens occurring to the left of `->` in the method; tokens to the left of the arrow are removed and tokens to the right are added. The algorithm terminates if no new token sets can be constructed this way. Type checking fails if a token set contains the same token twice. Usually the fixed point is reached quickly because there are only few different tokens in a class. Since new tokens can be introduced only by method invocations (as ensured by the properties mentioned below) this fixed-point construction is sufficient to ensure that two tokens of the same name can never exist for any object.
2. *Methods are invoked only through references associated with all needed tokens.* Initially we assume types of variables to carry tokens as in the variable declarations, and types of parameters as to the left of arrows in parameter declarations. While walking through the code according to the control flow we ensure for each method invocation that the type of the reference the method is invoked through which contains all tokens occurring to the left of `->` in the declaration of this method. Furthermore, we update the type of the reference by removing all tokens occurring to the left and adding all tokens occurring to the right of the arrow in the method declaration. Whenever the control flow is split (for example, in a conditional statement) we perform these checks for each path separately. At joins of several paths we remove all tokens that do not occur in all corresponding types constructed independently in the paths to be joined.
3. *Tokens are not duplicated when introducing aliases.* While walking through the code according to the control flow we ensure for each method invocation that types of arguments have all tokens occurring to the left of the arrow in the corresponding formal parameter type. These types are updated by removing all tokens occurring to the left and adding all tokens occurring to the right of the arrow in the formal parameter type. For each assignment of a value to a variable we ensure that the value has all tokens specified in the type of the variable and remove these tokens from the value's type. At the end of the control flow of each method and constructor we ensure that
   - each parameter has all tokens that occur to the right of the arrow in the parameter declaration,
   - and each instance variable has at least all tokens that occur in the variable declaration.
4. *Always at most one method can make use of tokens associated with an instance variable.* Such variables are not directly accessible from outside the object they belong to which. To ensure the absence of simultaneous accesses to each such variable within an object we use the set of methods accessing the variable and the upper bound of token sets constructed while checking property 1: If there is no token set in the upper bound that contains all tokens occurring to the left of the arrows of any pair of methods in the method set, then these methods cannot be invoked simultaneously and the variable access is unique.

   For example, in class `Test` (in the previous section) only `swap` and `condUpdate` accesses the instance variable `win`. Both methods have `unique` to the left of the arrow. Each of the four possible method pairs has two tokens `unique` to the left of the arrows. The upper bound constructed from `Test` contains only a single token set with a single token `unique`. Since no token set in the upper bound contains two tokens `unique`, several concurrent or overlapping invocations are impossible. In this case (and in many similar cases) we do not need the upper bound to show this property because we know that no token set

[1] Declaring `swap` and `condUpdate` as synchronized is not sufficient because there is still the possibility of a simultaneous access through aliasing. Requiring a unique token is a stronger condition. It ensures the absence of any other client also invoking one of these methods.

in the upper bound contains the same token twice. Sometimes the use of upper bounds increases accuracy. For example, in MyWindow the methods iconify and uniconify cannot be invoked simultaneously because no token set in the upper bound contains both shown and icon.

The type system is strong and sound in the sense that methods can be invoked only when objects are in appropriate states as specified by tokens. Essential parts of a corresponding proof can be found in [29, 30]. To get this result we need not restrict aliasing, and we need no knowledge of aliases (except of local information about statements possibly introducing new aliases to ensure property 3). This is an important difference to many seemingly similar approaches like the Fugue protocol checker [10].

There is a (still incomplete) implementation of the type checker for a simple language similar to the language we use in this paper. From early experiences with this checker we see that the type system is quite good in detecting errors where programmers get tokens wrong. Wrong tokens in method declarations usually show up as diverging upper bounds (as constructed to ensure property 1) or cause methods not to be invocable. Wrong tokens in types cause methods not to be invocable or references not to be usable as method arguments. The type checker complains about such errors.

Concerning type safety it does not matter if tokens are lost or hidden in the type of unused references. In such cases, clients just do not make use of services offered by objects. To enforce clients to make use of services we can extend the type checker as proposed in [34] at the cost of flexibility.

5. Dependent Tokens

In this and the following sections we discuss a number of approaches to improve the expressiveness and flexibility of our technique. An important step in this direction is to make use of known relationships between tokens that belong to different objects.

In the following example we show a possibility to specify tokens belonging to an instance variable in dependence of tokens of the object that contains the variable [33]:

```
class IconButtons {
   Window[icon for down][shown for up] window;
   [down -> up] void pressUp() {
      window.uniconify();
   }
   [up -> down] void pressDown() {
      window.iconify();
   }
   IconButtons[up] (Window[shown] w) {
      window = w;
   }
}
```

We think of IconButtons as a wrapper for the part of Window dealing with icons. The variable window has one token icon for each token down known to occur in the corresponding instance of IconButtons and one token shown for each up in the instance. In general, we regard a set of tokens to the left of for as available it there exists the set of tokens to the right of for. In the body of pressUp we know down to be available at method invocation and up on return because of [down -> up]. Hence, we assume window to have a token icon on invocation, and we must ensure that window has a token shown on return. An invocation of uniconify changes the token appropriately. Because of [up -> down] specified for pressDown we assume window to have a shown on invocation of this method, and we have to ensure the variable to have an icon on return. On object creation we must initialize window with a reference having a shown because the new instance of IconButtons is associated with an up.

Checking for-clauses in instance variable specifications is straightforward because type safety follows from the construction of this language concept. There is only a small difference to type checking as proposed in Section 4: To ensure property 3 we have to compute the tokens carried by variables from token specifications in methods instead of having them declared directly.

Using class IconButtons we control both buttons in a single class. Distributing a concern (like controlling the state of iconification) over several classes is a much more difficult topic that occurs in practice. In the next example we show an alternative solution to IconButtons based on separate classes for each button:

```
class ButtonA {
  Window[shown for activeA] window;
  ButtonB[passiveB for activeA] button;
  [passiveA -> activeA] void activate() {...}
  [activeA -> passiveA] void press() {
     window.iconify();
     button.activate();
  }
  [initA -> activeA]
   void init (Window[shown for activeA ->] w,
              ButtonB[passiveB for activeA->] b) {
     window = w;
     button = b;
  }
  ButtonA[initA] () {}
}

class ButtonB {
  Window[icon for activeB] window;
  ButtonA[passiveA for activeB] button;
  [passiveB -> activeB] void activate() {...}
  [activeB -> passiveB] void press() {
     window.uniconify();
     button.activate();
  }
  [initB -> passiveB]
   void init (Window[icon for activeB ->] w,
              ButtonA[passiveA for activeB->] b) {
     window = w;
     button = b;
  }
  ButtonB[initB] () {}
}
```

The variable window carries a token shown in ButtonA (and icon in ButtonB) when the button in active. Otherwise we do not know any token of window. After pressing the active button we activate the other button, and the pressed button becomes passive. These classes work essentially in the same way as IconButtons once the objects have been initialized. On invocation of activate the variables window in the two objects implicitly exchange the only available token. The initialization is the tricky part: We have to tell the two objects that they can safely assume to have the only token shown or icon of window when they are active. In ButtonA the parameter type Window[shown for activeA ->] specifies that w refers to a window carrying shown only while the button is active (and has no tokens on return from init); this parameter is assigned to window of essentially the same type. To initialize the objects we may use the following piece of code:

```
w = new MyWindow();
a = new ButtonA();
b = newButtonB();
```

```
    a.initialize(w,b);
    b.initialize(w,a);
```

The variable w occurs in both invocations of initialize and gives away its only token to both objects depending on the states of the objects as specified by the for clause in the formal parameter type. To ensure this initialization to be correct we have to show the following properties:

- The two objects never have the tokens activeA and activeB at the same time. Because of two different objects this property is not obvious. Using least bounds of token sets constructed for both classes as in Section 4 we can show this property: No token set constructed from ButtonA contains both activeA and passiveA, and no token set constructed from ButtonB contains activeB and passiveB. Since button in ButtonA carries passiveB if there is a token activeA, there cannot exist a token activeB at the same time, and analogously for button in ButtonB. Hence, activeA and activeB cannot exist at the same time.
- When the other object becomes active, there exists the token on window needed by the other object. This means, window must carry a token icon (or shown) at the end of each method where activeA (or activeB) occurs to the left of the arrow and does not also occur to the right.

Because these checks are ad hoc and compromise separate compilation, it is an open question whether for clauses in formal parameter types shall be supported or not.

6. Values as Tokens and Tokens as Values

Dependent tokens are safe and (without for clauses in parameter types) easy to handle where they are appropriate. However, in many situations we need more freedom. Especially, we want to relate the availability of tokens to values in variables. In the next example we show how to establish such relationships:

```
class SwapButton {
   int state;
   Window[icon if state < 0]
         [shown if state > 0] window;
   [unique -> unique] void press() {
      if (state < 0) {
         window.uniconify();
         state = 1;
      }
      else if (state > 0) {
         window.iconify();
         state = -1;
      }
   }
   SwapButton[unique] (Window[shown] w) {
      window = w;
      state = w == null ? 0 : 1;
   }
}
```

The variable window is associated with a token icon if state holds an integer value below zero, and with shown if the value is larger than zero. There is no token for zero. Before we can make use of these tokens we have to ensure corresponding conditions (considering the value of state) to be satisfied. After changing tokens associated with window we must update state.

This approach to relate tokens with values raises a large number of problems:

- Tokens are allowed to depend only on side-effect-free conditions that read only instance variables of the object. Such variables like state must not be written from outside, and there must not exist aliases of them. Otherwise it would be impossible to keep results of evaluating the conditions synchronized with the available tokens. In the programming language Ada we have similar requirements on conditions in when clauses belonging to protected types (Ada's notion for monitors) [16].
- The compiler must be able to determine whether conditions specified in square brackets correspond to other occurrences of the same conditions in conditional statements. Usually the compiler can determine only structural equivalence. The use of named conditions (based on name equivalence) can be helpful in this respect. For example, we define a parameterless boolean function that implements the condition and invoke this function instead of using the condition directly. This way it is easy to determine equivalence of conditions.
- On return from a method that changes tokens of variables or assigns new values to variables like state we have to ensure tokens and variable values to correspond to each other. We can do so by checking the conditions. In general, we can perform these checks only at run time and thereby lose static type safety. To avoid this problem we restrict values assigned to variables like state (where conditions depend upon) to be constant. In this case we can perform the checks at compilation time and keep static type safety. This restriction reduces the expressiveness, but tokens depending on values are still quite expressive.

Each of these problems can be solved (although the first and the last one are serious) and dependence of tokens on values does not compromise static type checking. However, since we need rather heavy machinery, we may prefer to use another approach that allows us to express more directly what we want to have:

```
class SwapButton2 {
   Window[?] window;
   [unique -> unique] void press() {
      if ([icon]window) {
         window.uniconify(); }
      else if ([shown]window) {
         window.iconify(); }
   }
   SwapButton2[unique] (Window[shown] w) {
      window = w;
   }
}
```

The question mark in the declaration of window states that we do not know statically which tokens will be associated with the variable. The tokens associated with window are stored in an implicit variable. An expression of the form [...]window returns true if this implicit variable contains all tokens specified in the square brackets. In the body of press we dynamically check if window is associated with icon or shown and make use of the found token. On return from the method (as well as from the constructor) the tokens of window are automatically stored in the implicit variable.

Up to now we regarded tokens to be a purely static language concept. The approach taken in SwapButton2 handles tokens dynamically. Nonetheless we can ensure static type safety without any difficulty because types are split and updated in the same way as in the purely static concept. By storing tokens in implicit variables (not directly modifiable by the programmer) we avoid the difficulties we have to address in the approach taken in SwapButton.

The implicit variable in SwapButton2 corresponds essentially to state in SwapButton. These two classes differ mainly in the syntax. In the approach of SwapButton we can use state information also for purposes not related to tokens, while the approach of SwapButton2 requires less program code and is simpler to check.

Using tokens as values as well as letting tokens depend on values adds much flexibility to the whole language concept.

7. Type Parameters

Tokens encoded into types and changes of types cause a difficulty together with homogeneous genericity as in Java: Each use of a type parameter refers to the same type while we often want to refer to types with different token sets, and we have to consider tokens to avoid unexpected token duplication. We need some notation to express tokens for type parameters. In the next example we show our first approach where we use essentially the same notation as for types:

```
class IconList<W extends Window> {
   [i -> i] void add (String s, W[icon] w) {...}
   [i -> s] void uniconifyAll () {...}
   [s -> s] W[shown] delete (String s) {...}
   [s -> s] W get (String s) {...}
   IconList[i] () {}
}
```

An instance of `IconList<MyWindow>` can be used as expected: We can add objects of type `MyWindow[icon]`, cause all added windows to become uniconified, and delete uniconified windows and thereby get back instances of `MyWindow[shown]`. An invocation of `get` cannot return any token because the returned instance of `MyWindow` remains in the list where the token still is needed. The compiler would complain if we tried to return the token and at the same time keep it in the list.

However, for types like `IconList<MyWindow[ready]>` this approach is inappropriate. The type parameter `W` must not carry tokens because `get` cannot return any reference associated with tokens as explained above. Otherwise we would implicitly duplicate tokens and destroy type safety.

If we need type parameters carrying tokens, we must declare the parameters with a question mark to make our intention clear:

```
class IconList<W[?] extends Window> {...}
```

In this variant the compiler complains about possible token duplication in `get`.

In Java we have no access to types substituting type parameters at run time. Therefore, it is most natural to keep also tokens in these types invisible. In languages with run-time support of genericity (like C#) we regard tokens associated with type parameters as being stored in an implicit variable. Then, we can use the boolean expression `[ready]W` to dynamically determine if each instance of `W` is associated with a token `ready` in a similar way as we did in `SwapButton2`. As a special case we can use `[]W` to ensure in methods like `get` no token to be associated with `W`.

8. Dynamic Tokens

A simple and seemingly still powerful approach to further increase flexibility introduces a dynamic pool of tokens into each object. We differentiate between static tokens (used so far) and dynamic tokens stored in dynamic pools. Dynamic tokens required on invocation (this is, dynamic tokens to the left of `->` in brackets associated with methods) are taken from the dynamic pools of the objects the methods belongs to (not from references to them). On return dynamic tokens are added to the pools, not to references. If a required dynamic token is not available on invocation, then the invocation is delayed until the token becomes available. The main purpose of dynamic tokens is synchronization [31, 33].

In this paper we prefix dynamic tokens with `$` to distinguish them syntactically from static tokens. By replacing `unique` in our `SwapButton` example with a dynamic token we get:

```
[$unique -> $unique] void press () {...}
SwapButton[$unique] (Window[shown] w) {...}
```

Each client can invoke `press` without needing a token. Several simultaneous invocations will be synchronized and executed in any sequential ordering. In this respect the use of dynamic tokens resembles that of “synchronized” in Java. However, we consider recursive invocations[2] of `press` as erroneous while recursive synchronized methods are supported. Unfortunately, there is no easy way to statically determine indirect recursive invocations especially together with separate compilation. We can detect erroneous recursive invocations practically only at run time as deadlocks.

Dynamic tokens are not as useful in controlling aliasing as they seem to be at a first glance. A client does not get unique access for some concern for a sequence of invocations – just for a single invocation. In simple cases (like ensuring unique access to a variable carrying tokens) dynamic tokens give us more flexibility at the cost of lost static safety and lost control over effects of aliasing. As we can see from dynamic tokens there is a fundamental difference between conventional synchronization and limiting the effects of aliasing although these concepts are related. Synchronization is a much weaker concept.

9. Related Work

The work presented in this paper is closely related to process types [27, 29, 30], a type concept where we express synchronization in types of active objects and in types of references to active objects. Process types were developed as abstractions over expressions in object-oriented process calculi like Actors [1] and build the formal basis of the present work. Static type checking ensures that only acceptable messages can be sent and thereby enforces required synchronization. Process types allow us to specify nearly arbitrary constraints on the acceptability of messages: We can specify all prefix-closed trace sets, type equivalence is based on trace-set equivalence, and subtyping on trace-set inclusion [28]. A notation based on tokens helps us to keep static type checking as well as deciding type equivalence and subtyping simple [29, 30]. The process type concept considers types to be partial behavior specifications [19, 20] especially useful in specifying the behavior of software components [4, 18, 25].

Recent work regards process types as a synchronization concept in Java-like object-oriented programming languages [31, 32, 33]. This work adds a further dynamic level of synchronization while keeping the completely static level of (required) synchronization. To control aliasing we need mainly the static level.

There are several approaches similar to process types. Some approaches ensure subtypes to show the same deadlock behavior as supertypes, but do not enforce message acceptability [24, 25]. Other approaches consider dynamic changes of message acceptability, but do not guarantee message acceptability in all cases [8, 9, 35]. Few approaches ensure all sent messages to be acceptable [17, 23]. There is essentially the same idea behind the well-known work on linear types [17] based on the π-calculus [21] and process types based on an Actor-like model. However, since there is no natural notion of message acceptability in the π-calculus as in the Actor model, static checking of linear types has to prevent deadlocks and (therefore) is much more restrictive than checking of process types that can ensure message acceptability without preventing deadlocks.

The Fugue protocol checker [10, 11] uses a different approach to specify client-server protocols: Rules for using interfaces are recorded as declarative specifications. These rules can limit the or-

[2] In general, this restriction applies to invocations of all methods that require the same dynamic tokens, not just recursive invocations of the same method.

der in which methods are called as well as specify pre- and post-conditions. Tokens in this protocol checker represent typestates. Other than in our approach, they can be used only for unique references. Since there is no concept resembling type splitting (as in our approach), the Fugue protocol checker cannot statically ensure all methods to be invoked in specified orders at the presence of aliasing. In these cases the checker introduces pre- and post-conditions to be executed at run time. Hence, our approach can statically ensure type safety in many cases where the Fugue protocol checker can perform only dynamic checks. There is a number of further similar approaches to express (abstract) object states in types and check protocol compatibility [6, 7, 36, 37, 38].

Several programming languages [5, 13, 26] were developed based on the Join calculus [14]. For example, in Polyphonic C# [5] we combine methods like put and get in a buffer to a chord to be executed as a single unit. Clients can see how methods in a chord are synchronized. Since only one method in a chord is executed synchronously and all other methods are asynchronous, only specific forms of synchronization are supported. Communication in Polyphonic C# and similar languages resembles that of the rendezvous concept while (dynamic versions of) process types extend monitors. There is no way to constrain method invocation sequences as with process types, and there is no obvious way to use chords in controlling aliasing.

Synchronization with tokens has a long tradition: Petri Nets have been explored for nearly half of a century as a basis of synchronization [22]. In general, expressing object states by abstract tokens often has clear (both practical and theoretical) advantages over expressing them more concretely by values in instance variables: Tokens are much easier tractable than concrete states especially when used in types. Many proposals use tokens to express abstract object states [6, 10, 37].

The major contribution of this paper is to explore process types from the perspective of aliasing control. Different from earlier work on process types we assume each token to occur at most once in a system. As a consequence we get clear separation of concerns, better error detection from static type checking, and more flexibility in specifying tokens associated with instance variables. Dependent tokens distributed over several classes as well as values used as tokens and tokens depending on values have not been considered so far in the context of process types.

10. Conclusions

The basic approach to limit effects of aliasing is simple: Objects issue tokens, and clients need tokens to interact with objects. A client holding a token gets exclusive access to the object that issued the token for the concern associated with the token because there exist only one token for this concern in the whole system. Static type checking ensures that methods can be invoked only in specified sequences by clients holding the required tokens. We apply a number of techniques to manage tokens in more or less complicated situations to increase the flexibility of this approach. For example, with dependent tokens we safely specify tokens to be available if other tokens are available, and with specific boolean expressions we get dynamic access to (otherwise static) tokens. We also detected some cases where this concept causes difficulties or reaches its limits: Access to instance variables carrying tokens must be exclusive, dependent tokens distributed over several objects are difficult to handle, and dynamic tokens (which are quite useful for synchronization) do not help much for our purpose. Nonetheless, we already have a number of techniques to avoid most cases of undesirable effects of aliasing while we need not restrict aliasing by itself.

References

[1] Agha, G., Mason, I. A., Smith, S., and Talcott, C. Towards a theory of actor computation. In *Proceedings CONCUR'92* (1992), no. 630 in Lecture Notes in Computer Science, Springer-Verlag, pp. 565–579.

[2] Aldrich, J., Kostadinov, V., and Chambers, C. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Nov. 2002).

[3] Almeida, P. S. Balloon types: Controlling sharing of state in data types. In *Proceedings ECOOP'97* (1997), no. 1241 in Lecture Notes in Computer Science, Springer-Verlag.

[4] Arbab, F. Abstract behavior types: A foundation model for components and their composition. *Science of Computer Programming 55*, 1–3 (2005), 3–52.

[5] Benton, N., Cardelli, L., and Fournet, C. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems 26*, 5 (Sept. 2004), 269–804.

[6] Bierhoff, K., and Aldrich, J. Lightweight object specification with typestates. In *ESEC/FSE-13* (Lisbon, Portugal, Sept. 2005), ACM Press, pp. 217–226.

[7] Boyland, J. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium* (Berlin, Heidelberg, New York, 2003), R. Cousot, Ed., vol. 2694 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 55–72.

[8] Caromel, D. Toward a method of object-oriented concurrent programming. *Communications of the ACM 36*, 9 (Sept. 1993), 90–101.

[9] Colaco, J.-L., Pantel, M., and Salle, P. A set-constraint-based analysis of actors. In *Proceedings FMOODS'97* (Canterbury, United Kingdom, July 1997), Chapman & Hall.

[10] DeLine, R., and Fähndrich, M. The fugue protocol checker: Is your software baroque? Tech. rep., Microsoft Research, 2004. http://www.research.microsoft.com.

[11] DeLine, R., and Fähndrich, M. Typestates for objects. In *ECOOP 2004 – Object-Oriented Programming* (Oslo, Norway, June 2004), no. 3086 in Lecture Notes in Computer Science, Springer-Verlag.

[12] Drossopoulou, S., Clarke, D., and Noble, J. Types for hierarchic shapes. In *ESOP* (2006), pp. 1–6.

[13] Drossopoulou, S., Petrounias, A., Buckley, A., and Eisenbach, S. School: A small chorded object-oriented language. In *Proceedings of ICALP Workshop on Developments in Computational Models* (2005).

[14] Fournet, C., and Gonthier, G. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages* (1996), pp. 372–385.

[15] Hogg, J. Islands: Aliasing protection in object-oriented languages. *ACM SIGPLAN Notices 26*, 10 (Oct. 1991), 271–285. Proceedings OOPSLA'91.

[16] ISO/IEC 8652:1995. Annotated ada reference manual. Intermetrics, Inc., 1995.

[17] Kobayashi, N., Pierce, B., and Turner, D. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems 21*, 5 (1999), 914–947.

[18] Lee, E. A., and Xiong, Y. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing 16*, 3 (Aug. 2004), 210–237.

[19] Liskov, B., and Wing, J. M. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices 28*, 10 (Oct. 1993), 16–28. Proceedings OOPSLA'93.

[20] Meyer, B. *Object-Oriented Software Construction*, second edition ed. Prentice Hall, 1997.

[21] Milner, R. The polyadic π-calculus: A tutorial. In *Logic and*

Algebra of Specification (1992), Springer-Verlag, pp. 203–246.

[22] MURATA, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE 77*, 4 (Apr. 1989), 541–580.

[23] NAJM, E., AND NIMOUR, A. A calculus of object bindings. In *Proceedings FMOODS'97* (Canterbury, United Kingdom, July 1997), Chapman & Hall.

[24] NIELSON, F., AND NIELSON, H. R. From CML to process algebras. In *Proceedings CONCUR'93* (1993), no. 715 in Lecture Notes in Computer Science, Springer-Verlag, pp. 493–508.

[25] NIERSTRASZ, O. Regular types for active objects. *ACM SIGPLAN Notices 28*, 10 (Oct. 1993), 1–15. Proceedings OOPSLA'93.

[26] ODERSKY, M. Functional nets. In *Proceedings of the European Symposium on Programming* (2000), Springer-Verlag.

[27] PUNTIGAM, F. Type specifications with processes. In *Proceedings FORTE'95* (Montreal, Canada, Oct. 1995), IFIP WG 6.1, Chapman & Hall.

[28] PUNTIGAM, F. Types for active objects based on trace semantics. In *Proceedings FMOODS'96* (Paris, France, Mar. 1996), E. Najm and J.-B. Stefani, Eds., Chapman & Hall, pp. 4–19.

[29] PUNTIGAM, F. Coordination requirements expressed in types for active objects. In *Proceedings ECOOP'97* (Jyväskylä, Finland, June 1997), M. Aksit and S. Matsuoka, Eds., no. 1241 in Lecture Notes in Computer Science, Springer-Verlag, pp. 367–388.

[30] PUNTIGAM, F. *Concurrent Object-Oriented Programming with Process Types*. Der Andere Verlag, Osnabrück, Germany, 2000.

[31] PUNTIGAM, F. Client and server synchronization expressed in types. In *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)* (San Diego, USA, Oct. 2005).

[32] PUNTIGAM, F. From static to dynamic process types. In *First International Conference on Software and Data Technologies* (Setubal, Portugal, Sept. 2006), INSTICC Press 2006, pp. 21–28.

[33] PUNTIGAM, F. Internal and external token-based synchronization in object-oriented languages. In *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006* (Oxford, UK, Sept. 2006), no. 4228 in Lecture Notes in Computer Science, Springer-Verlag, pp. 251–270.

[34] PUNTIGAM, F., AND PETER, C. Types for active objects with static deadlock prevention. *Fundamenta Informaticae 48*, 4 (2001), 315–341.

[35] RAVARA, A., AND VASCONCELOS, V. T. Behavioural types for a calculus of concurrent objects. In *Proceedings Euro-Par'97* (1997), no. 1300 in Lecture Notes in Computer Science, Springer-Verlag, pp. 554–561.

[36] STROM, R. E., AND YELLIN, D. M. Extending typestate checking using conditional liveness analysis. *IEEE Transactions on Software Engineering 19*, 5 (May 1993), 478–485.

[37] STROM, R. E., AND YEMINI, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering 12* (1986), 157–171.

[38] YELLIN, D. M., AND STROM, R. E. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems 19*, 2 (Mar. 1997), 292–333.

[39] ZHAO, T., PALSBERG, J., AND VITEK, J. Lightweight confinement for featherweight Java. In *OOPSLA 2003 Conference Proceedings* (Anaheim, California, Oct. 2003), ACM, pp. 135–148.

Using ownership types to support library aliasing boundaries

Luke Wagner Jaakko Järvi Bjarne Stroustrup
Texas A&M University
{lw,jarvi,bs}@cs.tamu.edu

Abstract

This paper describes a library for concurrency used in a 10-developer videogame project. The developers were inexperienced, yet there were no problems with data races in the multi-threaded application. We credit this to the explicit representation of ownership in the design of the library. Correct library usage implies aliasing boundaries which bear a strong resemblance to the owners-as-dominators property enforced by ownership types. We explore other situations where analogous aliasing boundaries exist and discuss a family of related libraries that could benefit from a design explicitly representing ownership. The ownership relations in the library currently have no support from the type system. We examine approaches to embed static checking of the aliasing boundaries in our implementation language, C++.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming; D.3.3 [*Software*]: Language Constructs and Features—Control structures

General Terms Design

Keywords Ownership types, C++, Data Races, Concurrency

1. Introduction

In research on type systems for object-oriented languages, an important property of interest is local reasoning. The challenge lies in the fact that an object's state is comprised not only of its immediate data members, but also the transitive closure of all the states of the objects on which it depends [1]. To provide a "deeper" form of encapsulation than directly supported by current languages, ownership types [2, 3] allow a class to identify its dependencies on other objects and then prevent outsiders from acquiring references to those dependencies. With these limitations on aliasing, it is possible to reason about the correctness of a class by looking only at the code for that class and its dependencies.

However, local reasoning for the programmer is not the only benefit from using aliasing boundaries. Researchers have demonstrated that higher-level program guarantees can be made by building on ownership type systems [4–8]. This paper presents an additional example where aliasing boundaries in a program can be beneficial: a library for concurrency developed and successfully used in a large student videogame project. We show that the aliasing boundaries required for correct library usage strongly resemble the owners-as-dominators property enforced on an object graph by ownership types [2]. Based on this, the paper presents a method by which code using the library could be checked using ownership types.

Based on the positive experience with the concurrency library, this paper considers a family of related libraries that could benefit from a similar approach. Together, these libraries can be seen as the decomposition of the separation facilities built into a traditional process, so that each individual separation facility may be applied at the sub-process level.

In sum, this paper is an experience report and a position paper that (1) describes a set of library abstractions and programming conventions that restrict aliasing in order to guarantee the absence of data-races; (2) identifies the correspondence of the aliasing boundaries required by the library with those expressed with ownership types; (3) describes the type system extensions necessary to move from a documented usage rule to statically ensuring that aliasing boundaries are respected; (4) outlines an economical implementation approach to embed those extensions into standard C++, by inspecting the program's AST with an extended type checker; and (5) identifies the aliasing boundaries discussed as a general pattern for a related class of libraries, justifying the effort to develop the checking mechanisms.

This paper is organized as follows. Section 2 describes the concurrency library and Section 3 how ownership types can be used to check its correct use. Section 5 discusses the C++ embedding, and the other libraries that could benefit from the same technique. Section 6 mentions related work. Section 7 concludes and discusses future plans.

2. The Library

This section describes a simple concurrency library that was developed for a videogame project written in C++. The game is called "...and then the World was Consumed by Monsters" and can be downloaded from the development team's website [9]. The project, organized by the Texas Aggie Game Developers, included 10 undergraduate student developers over a period of 6 months with no other experienced oversight. Thus, simplicity and understandability were key to the success of the project.

In the videogame development community, amateurs are often strongly discouraged from using concurrency by the more experienced because of the difficult class of bugs it can introduce. However, several game constraints made it necessary to offload computation and blocking API calls to other threads. First, as with most interactive videogames, there is an underlying rendering loop which repaints the screen. To maintain a visually smooth animation, each frame should take less than 30 ms. Second, the game allows the user to control a character that roams around a virtual world. The representation of the virtual world can be much larger than what fits in memory. This requires the world to be cut into smaller chunks which contain all the geometry, collision data, and creatures for a small area of the world. As the user moves into new areas, chunks get loaded and dropped, which requires I/O operations to load the memory, OS, and graphics resources for those chunks. Since some of these operations do not provide an asynchronous option and can have a high latency, most videogames either try to perform them all at once before the game starts or batch the operations and stall the user at chosen points when executing the batch. Such stalls did not fit nicely into the gameplay, so a separate thread was needed to handle concurrent world loading.

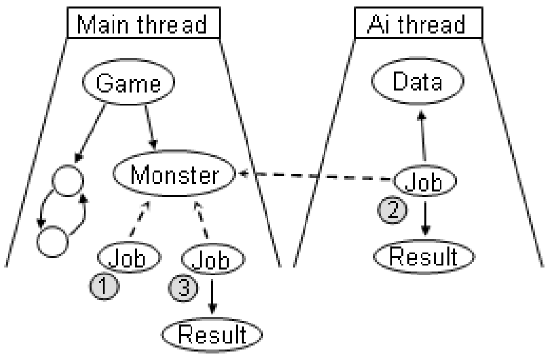


Figure 1. Travelling object with a tether, demonstrating the three states 1–3 of the Job object.

The same mechanism used for loading was later expanded to simplify AI programming. Here, the problem is a possibly expensive algorithm to find a path between two points. While the algorithm could be extended to moderate its execution time, a simpler approach is sending requests for computation to another thread with a lower priority. Thus, for slower machines where rendering takes a greater percentage of the time, enemies will simply take longer to decide where to go instead of hurting the framerate. An obvious additional benefit is the ability to utilize multiple hardware threads.

To avoid requiring the rest of the program to deal with shared mutable data and locking, the model for concurrency was based on the communicating sequential processes (CSP) metaphor (where CSP processes are just threads and the shared address space is not explicitly used). When extended to object-oriented programming, all objects are understood to belong to a single thread and messages between threads take the form of objects that can dynamically change membership. Objects that are allowed to change membership are called *travelling objects* and all others are called *local objects*.

A common need in the project was for a travelling object to create a reference to a local object, travel to a different thread to do some work, and then return to the original thread to use the reference. The synchronization implied by travelling prevents such behavior from being a data race. However, the situation is complicated by the fact that the local object may be destroyed while the travelling object is away. In a single-threaded scenario, a *weak pointer* library primitive comparable, e.g., to Boost weak pointer [10], is used when the pointee is allowed to be destroyed while another objects points to it. Although, the weak pointer implementation could have been extended to be made thread-safe, at the expense of synchronization overhead for all operations, this is more powerful than is necessary: a weak pointer maintains the liveness of the local object while the travelling object is in other threads. All a travelling object needs is to discover, when it returns, if the pointee has been destroyed in the interim.

To address the need for a simplified cross-thread weak pointer, the thread library provides a new, thread-aware smart pointer called a *tether*. Taking advantage of a tether's restricted semantics, the library is able to use the synchronization points already in place for transferring travelling objects between threads to keep the tethers coherent when they change threads.

Figure 1 demonstrates a typical usage scenario for a tether. In the figure: labels 1-3 show three steps in execution, the solid arrows represent normal references, and the dashed arrows represent tethers. In this scenario, a **Monster** local object needs a path in order to attack the player. Since the path finding algorithm should not be run in the rendering thread, the **Monster** creates a **Job** object (1) and ships it off to do the work in another thread. Before leaving, the **Job** creates a tether to the **Monster**. Next, the **Job** arrives (2) and is given a temporary local reference to the **Data** object which it can use to do the path finding computation. While in a different thread than the **Monster**, the tether held by the **Job** cannot be dereferenced. After **Job** finishes and returns to the original thread (3), it uses the tether to check whether the tethered **Monster** is still alive, and if so, the **Job** hands over the computation results.

Although more general usage of travelling objects could be supported using these library metaphors, the functionality required by the project only needed threads to act like assembly lines which processed jobs FIFO in the manner just described. Accordingly, **AssemblyLine** is the library primitive for creating such threads:

```
template <class GenericHost>
class AssemblyLine {
  GenericHost *host;
public:
  ...
  void send(typename GenericHost::Guest *);
  void receive_returning();
};
```

Since **AssemblyLine** does not know what to do with travelling objects, it is parametrized by a **GenericHost**. The host's responsibility is to receive incoming travelling objects and to provide them access to the necessary local data structures. Additionally, the type of travelling objects is determined by the **Guest** associated type of **GenericHost**. After starting a new OS thread in its constructor, an **AssemblyLine** will create an instance of **GenericHost**, which will be the first client object local to the new thread. To allow returning travelling objects to reenter the thread, the main thread synchronizes with **AssemblyLine** by calling **receive_returning()**.

To give a better idea of the library's use, we now walk through some skeleton code using the library in the path finding scenario. At the top-level of the application, a **Game** is created which, in turn, creates an **AssemblyLine**:

```
class Game {
  AssemblyLine<Host> ai_thread;
  ...
public:
  void run() {
    while (!quit) {
      ...
      ai_thread.receive_returning();
    }
  }
};
int main() {
  Game g;
  g.run();
}
```

When **ai_thread** is destroyed by **Game**, job processing will be stopped, all pending jobs will be deleted, and the OS thread will be released. **Host** parametrizes **AssemblyLine** and holds the path finding data that is needed by **Job**s:

```
class Host {
  Data data;
public:
  typedef Job Guest;
  void arrived(Job &guest) {
    guest.do_work(data);
  }
};
```

When a travelling object is sent from the main thread and gets pulled off the queue by **AssemblyLine**'s thread, it is handed over to **Host** by calling **arrived()**. When **arrived()** returns, **AssemblyLine** will send the travelling object back to the main thread. In addition to being compatible with **Host::Guest**, a travelling object's class must inherit from the **TravellerBase** library base class:

```
class Job : public TravellerBase {
  Tether<Monster> tether;
  ...
public:
  Job(Monster &m, ...) : tether(create_tether(m)) { ... }
  void do_work(Data &);
  void welcome_back() {
    if (tether)
      tether−>found_path(...);
  }
};
```

Travelling objects can choose to have any number of tethers to local objects using the **Tether** template class, parametrized by the type of the pointee. **Tether** follows the C++ smart pointer idiom and guards access to the pointee through **operator−>()**. For **AssemblyLine** and **Tether** to cooperate in keeping the tether coherent when it changes threads, construction of **Tether** is abstracted by the **create_tether()** protected member function inherited from **TravellerBase**. When a travelling object is accepted back into the main thread, **welcome_back()** is called. **Job** can then safely use its **Tether** after testing that the **Monster** object it was pointing to has not been destroyed.

Finally, using **Job** in **Monster** is fairly simple:

```
class Monster {
  AssemblyLine<Host> &ai_thread;
  ...
public:
  ...
  void think() {
    if (... I want to attack ...)
      ai_thread.send(new Job(*this, ...));
  }
  void found_path(...);
};
```

The project did not need **Job** objects after they returned to the main thread, so the **AssemblyLine** takes the liberty of deleting them. Altogether, the end-to-end order of function calls corresponding to Figure 1 is: **Monster::think()**, **Job::Job()**, **AssemblyLine::send()**, **Host::arrived()**, **Job::do_work()**, **AssemblyLine::receive_returning()**, **Job::welcome_back()**, **Monster::found_path()**, **Job::~Job()**.

Although message-based schemes are often viewed as more complex than shared-memory schemes when used for low level parallel programming, as used in the videogame project for simple task-level parallelism, we found the message-passing approach to be a clear mental model of concurrent execution for the programmer compared to shared memory with locking. Programming with this model, we did not experience data races. This could be attributed to the smaller scale of the student project, or the fear of concurrency imbued in the team by horror stories, but we believe the library design was an important part.

3. Checking Usage

The library described in Section 2 helps programmers by providing a simple mental model and set of tools for programming concurrency. This section describes how the type system could be enlisted to help as well. What is described is a correspondence between ownership typing judgements and aliasing restrictions in the concurrency library. The code shown is what the ownership type system needs to see, not what needs to be written in the actual C++ code. A lightweight embedding in C++ is discussed in Section 4. The syntax used to express the ownership typing concepts is based on Joline [11] and Ownership Generic Java (OGJ) [3]. In some places, features of C++ will be mixed in where they are needed by the library.

Another point to clarify is the meaning of *ownership*. Ownership types are traditionally presented in the context of a language with garbage collection and so the main issue is accessibility. However, in the context of C++, ownership can also refer to the responsibility of an object to manage the lifetime of the resources it owns. This paper limits its discussion of ownership to issues of accessibility; static guarantees involving object lifetimes are not addressed.

This section first discusses the basics of ownership types and then describes how they can be used by each piece of the library.

3.1 Background

Ownership types can be used to statically limit what references are allowed between objects. Considering objects and their references as a graph, ownership types allow the user to draw boundaries around parts of the graph, limiting incoming references. What follows is a brief explanation of how this is accomplished. Although it sounds like extra runtime state and checking is being added, none of it is needed after type checking; the runtime behavior of the program is not modified.

First, every object is given a unique *ownership context*. An ownership context can be thought of as a value of an opaque type. The only purpose of an ownership context is to be part of the type of an object. An object's class is augmented to take, as a generic parameter, the ownership context of some other object, which becomes its owner. Because ownership contexts are values, this creates a relation between objects, not types. Additionally, there is an omnipresent, disembodied **world** ownership context which is not associated with any object. Because an owner has to be constructed before the objects it owns, ownership is acyclic. Furthermore, all objects have exactly one owner, so the ownership relation forms a tree rooted at **world**.

For an object to hold or use a reference to another object, static type checking demands that the reference have a type. Ownership types limit aliasing by controlling what types can be constructed: if a type cannot be named, the reference cannot be held. Because ownership contexts have been embedded in types, controlling aliasing reduces to controlling what objects have access to what ownership contexts.

Ownership contexts are accessible in a few ways. As the base case: every object can access its own ownership context using the overloaded **this** keyword; the **world** ownership context can be accessed using **world** keyword; and an object can access its owner's ownership context using the **owner** keyword. Next, ownership types allow an arbitrary number of extra ownership contexts to be passed to an object, as type parameters, with the restriction that all parameters are ancestors of the **owner** in the ownership tree.

The following code snippet shows an example of these concepts in the syntax of the Joline language [11]:

```
class Bar {}

class Foo<P1 outside owner> {
  this:Bar owned_by_me;
  owner:Bar owned_by_my_owner;
  owner:Foo<P1> same_type_as_me;
  this:Foo<owner> can_access_my_siblings;
}
```

Because every class must take an owner parameter, Joline makes the owner parameter implicit. Other ownership parameters are declared between angle brackets, like type parameters. Ownership parameters are bounded to be *outside* other parameters (meaning an ancestor in the ownership tree), with **owner** as the most general bound. When supplying the actual parameters to a class, the owner

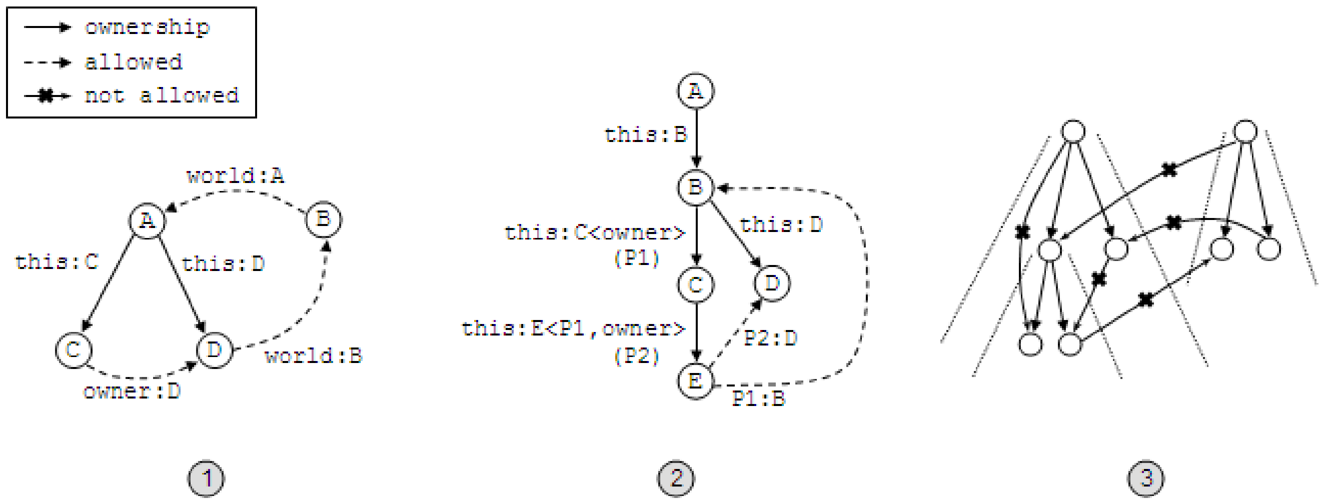


Figure 2. Examples of aliases allowed and not allowed by ownership types

is also distinguished from the other parameters by placing it before the type, separated by a colon.

Figure 2 illustrates the effect of ownership typing on the object graph. Diagram 1 shows the use of only **this**, **owner**, and **world**. Nodes represent objects and are labelled with the object's class. Arrows represent references between objects and are labeled with the type of the reference. Thus, the arrow labeled **this:C** indicates that the object of class **C** is owned by the object of class **A**.

The second diagram shows how additional ownership parameters can allow objects to access the ownership contexts of owners higher up in the ownership tree. For example, **E** can reference **D** because **E** has access to the ownership context of **D**'s owner, **B**. The identifier in parenthesis is the name of the formal parameter.

The third diagram shows references that are not allowed based on the ownership tree. Looking at the pattern of what references are and are not allowed, we can see one-way boundaries emerge on the object graph (drawn by the dotted lines). Visualizing these boundaries can help in understanding ownership types. A more formal statement is that ownership types guarantee the *owners-as-dominators* property on the object graph: an owner is a dominator on the path from **world** to all objects it transitively owns [2].

This forms the core of ownership types. On top of this, there are three additional extensions that need to be discussed. The first is the ability to parametrize a class by another class. OGJ allows type parameters and ownership parameters to be mixed compactly as follows:

```
class Box<Node extends Object<NodeO>> {
  Node held_in_box;
}
```

In this code, **NodeO** is the owner of **Node** and can be used to instantiate new classes. However, a subtle result of OGJ's treatment of ownership parameters and Java's type erasure semantics for generics is that **Node** represents a class that has already been instantiated with an owner. This means it is an error to try to give it a new type because:

```
class Outside<Inside extends Object<O>> {
  this:Inside mine; // wrong
}
```

really means (swapping the formal parameter **Inside** with the actual parameter **SomeType**):

```
class Outside<Inside extends Object<O>> {
  this:O:SomeType mine; // wrong: two owners
}
```

and supplying two owners is obviously wrong. What is needed is to pass an uninstantiated class that can be instantiated with arbitrary ownership parameters. This would be analogous to the "template template parameter" mechanisms in C++ and will be denoted in the parameter list by using the **class** keyword:

```
class Outside<class Inside> {
  this:Inside mine; // OK: Inside not already instantiated
}
```

Uninstantiated class parameters will be used extensively by the library types in the next section.

Another extension, which is also part of OGJ is *manifest ownership*. This allows a class to hard-code its owner by inheriting from a class instantiated with an owner:

```
class Foo extends world:Object { ... }
```

Written this way, **Foo** cannot be given an owner and will be the sibling of all **Foo**s in the ownership tree.

The last extension is owner polymorphic methods, which are part of Joline. This feature is one of several extensions which offer "principled violations of the ownership type system" (e.g., as described in [12]). Generally, such extensions are included to support common constructs such as iterators [13]. An owner polymorphic method lets the caller give the callee access to an ownership context for the duration of the call:

```
class Person {
  <You inside world> void lend(You:Gold yours) {
    You:Gold local_ref = yours;
    // mine = yours; (error)
  }
  this:Gold mine;
  // You:Gold stolen; (error)
}
```

This example shows how **You** is only available for the duration of the call, so references to the **You:Gold** cannot live past the call. This gives the concurrency library a tool to allow *temporary* aliasing between two objects dynamically determined to be in the same thread without the possibility that a reference will escape.

The height of the ownership tree described in this section is at most three. This might suggest a lighter-weight type system, like Universes [14], to achieve the same static guarantees. However, (1) the extensions used are based on an ownership type system, and (2) using this library in combination with similar libraries, as described in Section 5, involves nesting, which creates more complicated ownership trees requiring the full owners-as-dominators guarantee.

3.2 Typing

This section describes how the concurrency library can use ownership types as a tool to prevent data races, analogous to how a library can use **const** or accessibility modifiers to prevent clients from modifying returned references or accessing implementation details. The facility that ownership types add is statically enforced aliasing boundaries. By creating aliasing boundaries around threads and travelling objects, the concurrency library can guarantee to the library user: *if you can hold a reference to an object, it is safe to access it*.

The first step is to disallow client usage of **world**, which would allow allows client code to make and reference objects that are not local to any thread. The library types that are roots of the various ownership subtrees can then use the manifest ownership feature described in Section 3.1 to allow creation by library users without mentioning **world**. We can now revisit the parts of the ownership library that were introduced in Section 2. First, we consider the modified **AssemblyLine**:

```
class AssemblyLine<class GenericHost> extends world:Object {
  this:GenericHost host;
public:
  void send(Traveller<GenericHost::Guest>);
  void receive_returning();
}
```

AssemblyLine takes an uninstantiated **GenericHost** parameter and instantiates it with **this**. Without the **world** ownership context available, all objects created by **host** will necessarily be owned by the **AssemblyLine**. To guarantee that only travelling objects get moved between threads, **send()** only accepts the **Traveller** wrapper type, which is shown next:

```
class Traveller<class TravObjT> extends world:Object {
  this:TravObjT obj;
public:
  <O inside world> Traveller(O:TravObjT::InitArgs a) {
    obj = new<O> this:TravObjT(this, a);
  }
  <O inside world, class LocObjT>
  this:Tether<LocObjT> create_tether(O:LocObjT);
}
```

Traveller uses the same technique as **AssemblyLine** for owning a generic object. However, **TravObjT** cannot be default constructed like **GenericHost**, so **Traveller** takes in generic initialization data to pass to **TravObjT**'s constructor. Since **create_tether()** returns a **Tether** owned by **this**, only travelling objects can create tethers. To prevent direct construction, **Tether** has a private constructor:

```
class Tether<class PtrT> {
  // private constructor, only available to friend Traveller
  PtrT ptr;
public:
  bool alive();
  void request_access(owner:TetherUser<PtrT> p) {
    if (... same thread ...)
      p.access_granted(ptr);
    else
      ... error
  }
}
```

```
interface TetherUser<class PtrT> {
  <O inside world> void access_granted(O:PtrT);
}
```

Tether's main job is to guard access to the pointee. To do this, **Tether** requires that its users implement the **TetherUser** interface. Similar to the double virtual dispatch in the Visitor pattern, **access_granted()** gets called by **Tether** in response to calling **request_access()**. This approach does three things for **Tether**: first, it lets **Tether** dynamically guard access to the reference; second, the owner polymorphic method **access_granted()** allows **Tether** to prevent the given reference or any copies from outliving **access_granted()**; and third, **Tether** knows the duration of the reference's visibility and can thus prevent the travelling object from getting sent to another thread somewhere in the call stack.

As shown in the **Tether** pseudo-code, the library implementation ignores ownership types internally: **Tether** stores a plain reference to the object and calls **access_granted()** without any ownership context. This is similar in spirit to how, for example, a C++ **std::vector** presents a typed container interface to its users, but internally works with **malloc()**s, **void∗**s, and **memcpy()**s. As with **world**, this exemption should only exist for classes that are part of the library.

Lastly, client objects in the main thread need an owner. Without **world** and starting in the non-member **main()** function, however, there is no way to create objects. Following the same pattern as **AssemblyLine** and **Traveller**, **MainThread** allows the client to generically embed an object which will be owned by the **MainThread** object:

```
class MainThread<class ClientMain> extends world:Object {
  this:ClientMain host = new this:ClientMain;
public:
  int main() { return host.main(); }
}
```

With the library types covered, we can now consider what ownership types are needed for the user's code. The **Host** needs to modify its **arrived()** member function which gets called by **AssemblyLine** to reflect that it can only reference the arrived travelling object temporarily:

```
class Host {
  owner:Data data;
public:
  typedef Job Guest;
  <O inside world> void arrived(O:Job guest) {
    guest.do_work<owner>(data);
  }
}
```

The **data** member is a local object, so it is owned by the **AssemblyLine**. To pass a reference to the **Job**, **Host** needs to pass **owner** as well. As the travelling object, **Job** requires the most modifications:

```
class JobArgs {
  ...
  owner:Monster m;
}

class Job implements TetherUser<Monster> {
  owner:Tether<Monster> tether;
public:
  typedef JobArgs InitArgs;
  <O inside world> Job(Traveller t, O:JobArgs j) {
    tether = t.create_tether<O,Monster>(j.m);
  }
  <O inside world> void do_work(O:Data d);
  void welcome_back() {
    tether.request_access(this);
  }
  <O inside world> void access_granted(O:Monster m) {
    m.found_path(...);
  }
}
```

First, to support initialization in the generic **Traveller**, **Job** has to specify what data it needs with the **JobArgs** class and **InitArgs** associated type. **Job** also receives its owning **Traveller** as a constructor parameter, which it uses to create a tether. In **welcome_back()**, **Job** calls **request_access()**, passing itself to be the receiver of the **access_granted()** call. Instead of creating a **Job** directly, **Monster** now makes a **Traveller**:

```
class Monster {
  AssemblyLine<Host> ai_thread;
public:
  void think() {
    if (... I want to attack ...)
      ai_thread.send
        (new Traveller<Job>(new owner:JobArgs(this)));
  }
  void found_path(...);
}
```

Having the **Job** embedded in the **Traveller** prevents **Monster** from holding any references to the travelling object when it leaves.

In summary, the library requires all user objects to be owned by a library object. User objects that share the same owner are statically guaranteed to be in the same thread. Additionally, objects that are temporarily in the same thread can be allowed to reference each other in a controlled manner using owner polymorphic methods. A key part of this approach is that ownership types are not modified to include concepts of thread, local, travelling, and tethers. Rather, these concepts are in the library, which then uses ownership types as a tool for library design.

4. Embedding Ownership

Section 3 demonstrates how the primitives provided by the concurrency library of Section 2 could be checked if everything is written in an idealized language with ownership types. What is needed is a translation to this checkable form from Standard C++. We do not have such a translation implemented, however we outline what we believe is a promising approach to a minimal embedding in the language.

The first problem to address is how to attach ownership to references. In the simplest case, no annotation is needed at all. First, references to the library types that use manifest ownership (**AssemblyLine**, **Traveller**, and **MainThread**) do not need any ownership parameters. Next, when an owner is needed, **owner** may be used as a default. Defaulting has already been applied to Ownership Generic Java [15] to allow Generic Java programs to compile unmodified. For users of the concurrency library, code that does not deal with travelling objects will only refer to objects owned by the same thread. Thus, depending on how much code deals with concurrency, having **owner** be the default can eliminate much of the need for annotations.

When the default does not work, the programmer needs to make an annotation. There are many ways a programmer could make explicit the intent that a pointer or reference should represent ownership. The goal is to allow programmers and tools to verify that the ownership rules are obeyed. The most primitive approach is to use a special class of names for variables such as:

```
Foo *this_owned_a;
Foo *owner_owned_b;
```

where portions of identifiers are used as cues. Another approach is annotations in smart comments, which is the approach used by Universes [14]:

```
/** this: */Foo *a;
/** owner: */Foo *b;
```

However, the least intrusive interface to an analysis tool is a trivial template wrapper, such as:

```
this_owned_ptr<Foo> a;
owner_owned_ptr<Foo> b;
```

The templates are defined as any other template, using the standard syntax of C++. The type checker, however, can recognize the templates as an explicit ownership annotation. In addition to providing a solid handle for an analysis tool to work on, the wrappers can naturally introduce or remove operations on the wrapped type. The reason for using a technique that does not require language changes is that we eventually want to handle a large class of annotations and do not want to define our own set of dialects with their own compiler infrastructure. This is the SELL (Semantically Enhance Library Language) approach which we support with a simple tools infrastructure called "The Pivot" [16].

Considering in particular annotations needed for the concurrency library, the primary case is when using an owner polymorphic method:

```
class Job {
  <O inside world> void do_work(O:Data data) {
    // use data reference
  }
}
```

To annotate **data** we can write the following:

```
class Job {
  void do_work(caller_owned_ptr<Data> data) {
    // use data smart pointer
  }
};
```

Here, the presence of the **caller_owned_ptr** template wrapper indicates to the translation to both declare an owner polymorphic parameter and bind it to **data**.

Aside from annotating references with ownership, some of the constructs of the library had to be changed to accommodate ownership types. In particular: global variables and non-member functions need to be wrapped into a global object owned by a **MainThread**; **Tether**s are "dereferenced" indirectly through a double dispatch instead of using the more natural arrow operator; and inheriting from **TravellerBase** is changed to embedding in **Traveller**. For these special cases, a translation from C++ should be able to make simple patterned substitutions. For example, consider the following:

```
A *global = new A;
void foo(A *a) {}
int main() { foo(global); }
```

For type checking purposes, these globals can be collected into a single **Process** class that gets embedded in **MainThread**:

```
class Process {
  A *global = new A;
  void foo(A *a) {}
public:
  int main() { foo(global); }
};
```

```
int main() {
  MainThread<Process> mt;
  return mt.main();
}
```

With the default **owner** applied, the code can type check. A more involved example is converting uses of the arrow operator in **Tether** to double dispatch. Here, the translation involves hoisting the member function call, the arguments, and the return value into an automatically generated **TetherUser**. For example:

```
class Job {
  Tether<Monster> tether;
public:
  void welcome_back() {
    if (tether)
      tether->found_path(...);
  }
};
```

can be automatically translated into:

```
class AutoUser : public TetherUser<Monster> {
  ...
public:
  AutoUser(...);
  void access_granted(caller_owned<Monster> m) {
    m.found_path(...);
  }
};
class Job {
  Tether<Monster> tether;
public:
  void welcome_back() {
    if (tether)
      tether.request_access(AutoVisitor(...));
  }
};
```

With these and related transformations, the syntactic burden over normal use of the library can be reduced while internally generating the fully ownership-annotated source for checking.

5. Discussion

The presentation so far has been concerned with describing our experience with a single library on a single project. This section branches out to consider a wider range of features and applications of this kernel experience.

5.1 Variations on Tethers

The **Tether** construct presented in this paper was motivated by the specific needs of a project, but other variations on the same approach make sense for different situations. The essential ideas are: (1) regardless of aliasing boundaries, objects need to be able to point to objects in other threads, and (2) these pointers can have different operations in place of the standard "dereference". We present two further examples here.

An opposite approach to calling **AssemblyLine::send()** is for a local object to use a **Tether** to "pull" a travelling object into the same thread. The pull operation waits until the target object is not in use in its current thread and transfers it to the caller's thread.

```
class Worker {
  PullTether<RenderPipeline> rpipe;
  void render_data(...) {
    // prepare data for rendering
    // expensive computation...
    PulledObject<RenderPipeline> po = rpipe.pull();
    po->render(...);
  }
};
```

We can see that these semantics are analogous to that of a traditional lock which protects the object getting pulled. However, without any additional work on the part of the user, the runtime system can make optimizations over plain locks. First, by keeping track of the tethers to an object, the runtime can tell which threads can possibly request a lock at the same time. With this knowledge, the runtime can use cheaper locks when, for example, it knows that all contending threads are assigned to the same physical processor. Conversely, in a non-uniform memory architecture, the runtime system could look at the tethers that exist between objects and place threads which have many tethers between them "closer" together, with respect to the machine topology.

Another variation is to treat a tether as a homing device for the object to which it points. Instead of pulling a distant object close, tether could be augmented to provide a "take me to this object" operation which allows a travelling object to go to the thread that owns the pointee:

```
class UpdateCourier {
  Update update;
public:
  void update_data(HomingTether<Data> d) {
    d.go_to_thread(*this);
  }
  void arrived_at_thread(Data &d) {
    d.apply(update);
  }
};
```

In this example, a control thread updates data structure that are local to different processing threads by sending courier objects to the threads with the update. Courier objects are given **HomingTether**s to indicate which data set needs to receive the update. Finally, **arrived_at_thread()** is called by **HomingTether** when the transfer is complete.

5.2 Variations on Libraries

In this section we identify the design and typing of the concurrency library as an instance of a more general pattern of library design. The pattern is defined by: (1) providing library primitives whose semantics imply aliasing boundaries, and (2) providing the user of the library semantically-modified pointers to refer across these boundaries. We now consider two other examples, how their primitives imply aliasing boundaries, and how users can refer to objects across these boundaries.

5.2.1 Memory Protection

Fine grained memory protection has been used for security, fault isolation, and efficient IPC since early capability-based architectures [17] and continues to be researched. Recent work includes Mondriaan Memory Protection (MMP) which has been applied to the Linux kernel [18]. The idea is to associate *protection domains* with allocated memory regions and threads. Threads are then prevented from accessing memory outside their current protection domain. This approach helps find errors that might have gone undetected and catches errant program behavior closer to the source.

A straightforward API for a memory protection library would provide functions for: allocating and deallocating opaque protection domain handles; adding and removing memory regions to and from domains; and changing the domain of the currently executing thread. These API calls could be abstracted by an object-oriented library in the same manner that the concurrency library in Section 2 abstracted low level locking and thread operations. The aliasing boundaries in this case would align with protection domains and a library pointer type would be provided to point to objects in other protection domains. The library could then either offer travelling mechanisms similar to the concurrency library or simply provide a dereference operation. In addition to allowing a flat partitioning of memory, systems like MMP allow a region of memory to be in more than one protection domain. This lets the user create a nesting structure of permissions which directly corresponds to the owners-as-dominators property enforced by ownership types.

5.2.2 Resource Accounting

A good operating system will release all resources requested by a process when the process exits. This requires the system to record which resources have been allocated by the process. Thus, a simple way to do "garbage collection", not only for memory but all OS resources, is to fork child processes to handle work items and then exit, automatically freeing the resources used to process the work item. This approach has several performance disadvantages and consequently developers usually need to use multiple threads and careful resource management instead.

The utility of a process, with respect to resource management, is that it provides a single collection point for resources. To achieve the same effect at a finer granularity, we can introduce "resource domains". Each resource domain owns a set of objects and keeps track of all allocation requests made by objects it owns. One challenge for the library is to keep track of the current resource domain as execution passes between objects owned by different resource domains. By aligning aliasing boundaries with resource domains, the library user would be required to use a library mechanism when pointing to objects in other resource domains. By controlling access to objects in other resource domains, the library can keep track of changes:

```
class EnemyAI : ResourceDomainVisitor<EnemyGraphics> {
  CrossDomainPtr<EnemyGraphics> ptr;
public:
  void think() {
    if (... decide to hold a fireball ...)
      ptr.access_resource_domain(*this);
  }
  void in_resource_domain(EnemyGraphics &g) {
    // allocate Fireball in graphics resource domain
    g.shoot(new Fireball);
  }
};
```

In this example, the AI component of an enemy creates a **Fireball** for the graphics component to show. The two objects are in different resource domains, so the **EnemyAI** needs to use the library-supplied pointer type **CrossDomainPtr**. To access the object, the same double virtual dispatch technique used by **Tether** in Section 3 is used. This allows the library to change domains for the duration of **in_resource_domain()** so that **Fireball** is allocated in the graphics resource domain.

Hierarchical resource management is normally done in C++ using constructors and destructors following the Resource Acquisition Is Initialization idiom [19]. On the opposite end of the resource management spectrum, garbage collection tries to hide when resources are released and does not associate an owner. The approach presented in this section is therefore somewhere in between: resources have owners and deterministic bounds on their allocation, but these bounds are more like catch-alls than proper manual resource management. Thus, allocation domains can be seen as a fine-grained way to handle leaks or a way to recover resources when an error has left a portion of the system in an undefined state.

5.2.3 Summary

In the examples above, the library provides primitives that organize objects in the program hierarchically. To fully utilize this library design, however, several libraries need to be able to coexist in the same ownership tree in the same program. For example, consider a modern web browser. Concurrency boundaries can be associated with different browsing windows, security boundaries with the scripting interpreters, memory protection boundaries with less-than-stable modules, and resource accounting boundaries where leaks are difficult to avoid. This implies a heterogeneous nesting of boundaries which we have not considered thus far. For the same reason it is necessary to cross homogeneous boundaries, it will be necessary to compose each library's semantically-modified pointers to cross multiple heterogeneous boundaries. This ventures far from the experience and example focused on by this paper but we feel it points to an exciting use of ownership types as a tool for future library design.

6. Related Work

Since the widespread recognition of the problems of aliasing in object-oriented programming, and the need for local reasoning, more than a decade ago [1], many type systems have emerged to address the problems. The approaches vary from completely outlawing aliasing using variants of linear types [20,21], to cutting the object graph into fully encapsulated partitions [22,23], to enforcing an owners-as-dominators property on the object graph using ownership types [2], to even more flexible and/or less intrusive type systems with less guarantees [24–26]. Of these approaches, ownership types have emerged as a promising compromise and many different aspects of the type system have been researched [3, 27, 28]. Boyapati *et al.* have used and extended ownership types to guarantee the absence of data races and deadlocks [4], statically safe region-based memory management [5], and safe lazy upgrades to persistent object stores [6].

The work most similar to ours is SafeJava [4], which also uses ownership types. More recent work to statically ensure the absence of data-races has been done by Jacobs *et al.* using automatically verified annotations in the Spec# compiler [29, 30]. The main difference between our approach and these two is the basis for concurrency: in our model, nothing is shared and objects travel between threads; in the other two, there are shared objects which are owned by **world** and synchronized with locks. SafeJava does allow unique types to be passed between threads via a synchronized global shared variable, but this places aliasing constraints on the unique object which would not allow constructs like tethers. Another difference is how the data-race freedom guarantees are made. These approaches use concurrency constructs built into the language and build concurrency guarantees into the type system. In the approach we outline, the library both provides the concurrency primitives and uses a generic ownership type system to make guarantees about use of the library.

7. Conclusion and Future Work

In this paper we presented a simple library for concurrency, successfully used in a large student project, and demonstrated how ownership types could be used to statically check that client code respect the aliasing boundaries imposed by the library. To provide flexible support for objects travelling between threads while carrying aliases to thread local objects, we combine owner polymorphic methods with dynamic checks performed by the library to guarantee the absence of data races. Finally, we present an approach to embed the necessary ownership annotations in C++ and to use an extended type checker to enforce the rules on top of the language.

We also found the strategy used to support the concurrency library was also found to apply to a family of related libraries including memory protection and resource accounting. One direction for future work is to examine existing programs that exhibit task-level parallelism, like the videogame example in this paper. By looking at more and larger programs, we can further develop both the concurrency model and typing approach introduced here to address more usage scenarios.

Acknowledgements: We are grateful to the reviewers for their feedback, members of the program committee for their patience, and Alex Potanin for his help.

References

[1] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.

[2] David Gerard Clarke. *Object ownership and containment*. PhD thesis, University of New South Wales, 2002.

[3] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic Java. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 311–324, New York, NY, USA, 2006. ACM Press.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM Press.

[5] Chandrasekhar Boyapati, Alexandru Salcianu, Jr. William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, New York, NY, USA, 2003. ACM Press.

[6] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 403–417, New York, NY, USA, 2003. ACM Press.

[7] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[8] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[9] Texas Aggie Game Developers: `http://tagd.cs.tamu.edu`.

[10] C++ Boost Library Collection, 2007. Boost Smart Pointers: `http://www.boost.org/libs/smart_ptr/smart_ptr.htm`.

[11] Tobias Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Stockholm University, 2006.

[12] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, Louisiana, January 2003.

[13] James Noble. Iterators and encapsulation. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 431, Washington, DC, USA, 2000. IEEE Computer Society.

[14] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[15] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Defaulting generic Java to ownership. In *In Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming*, Oslo, Norway, 2004. Springer-Verlag.

[16] Bjarne Stroustrup. A rationale for semantically enhanced library languages. In *Proceedings of the First International Workshop on Library-Centric Software Design (LCSD '05)*, 2006. As technical report 06-12 of Rensselaer Polytechnic Institute, Computer Science Department.

[17] R. M. Needham and R. D.H. Walker. The Cambridge CAP computer and its protection system. In *SOSP '77: Proceedings of the sixth ACM symposium on Operating systems principles*, pages 1–10, New York, NY, USA, 1977. ACM Press.

[18] Emmett Witchel, Junghwan Rhee, and Krste Asanovic. Mondrix: memory isolation for Linux using Mondriaan memory protection. *SIGOPS Oper. Syst. Rev.*, 39(5):31–44, 2005.

[19] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[20] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[21] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 176–200, 2003.

[22] John Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285, New York, NY, USA, 1991. ACM Press.

[23] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32–59, 1997.

[24] Jan Vitek and Boris Bokowski. Confined types. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–96, New York, NY, USA, 1999. ACM Press.

[25] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight generic confinement. *J. Funct. Program.*, 16(6):793–811, 2006.

[26] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. *SIGPLAN Not.*, 37(11):311–330, 2002.

[27] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *SIGPLAN Not.*, 37(11):292–310, 2002.

[28] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Safe runtime downcasts with ownership types. Technical Report TR-853, MIT Laboratory for Computer Science, June 2002.

[29] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *8th International Conference on Formal Engineering Methods*, pages 420–439, 2006.

[30] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 137–147, Washington, DC, USA, 2005. IEEE Computer Society.

Runtime Universe Type Inference

Werner Dietl
ETH Zurich, Switzerland
Werner.Dietl@inf.ethz.ch
http://www.sct.inf.ethz.ch/

Peter Müller
Microsoft Research, USA
mueller@microsoft.com

Abstract

The Universe type system is an ownership type system for object-oriented languages that enforces the owner-as-modifier discipline. One strength of the Universe type system is its low annotation overhead. Still, annotating existing software is a considerable effort.

In this paper, we describe how we can analyze the execution of programs and infer ownership modifiers from the execution. These modifiers help to understand the organization of a system and can also be re-inserted into the original source code. This allows a programmer to enforce the maintenance of a specific ownership structure. We implemented runtime Universe type inference as a C program that traces the JVM execution, a Java application that infers the Universe annotations, and a set of Eclipse plug-ins that integrates the interaction with the other tools.

1. Introduction

The Universe type system [13] is an ownership type system for object-oriented languages that enforces the owner-as-modifier discipline. The type checker and runtime support for Universe Types are implemented in the JML tool suite [21].

At runtime, the *owner* of an object is either another object in the store or the special *root object*. Objects that share the same owner are grouped into a *context*; objects that have the root object as owner are in the *root context*. Ownership builds a tree rooted at the root object.

The owner-as-modifier discipline ensures that the owner of an object controls all modifications of an owned object, that is, only references to objects in the same context and to owned objects can be used for modifications. This discipline enables the modular verification of invariants [27].

Statically, the Universe type system uses three different *ownership modifiers* to build this ownership structure. The modifier `peer` expresses that the current object `this` is in the same context as the referenced object, the modifier `rep` expresses that the current object is the owner of the referenced object, and the modifier `any` does not give any static information about the relationship of the two objects. References with an `any` modifier convey less information as references with a `peer` or `rep` modifier with the same class and are therefore supertypes of the two more specific types.

The owner-as-modifier discipline is enforced by forbidding field updates and non-pure method calls through `any` references. An `any` reference can still be used for field accesses and to call pure methods. The method modifier `pure` is used to mark methods that leave objects in the pre-state of a method call unchanged.

A distinguishing characteristic of the Universe type system is its low annotation overhead compared to other ownership type systems. The annotation effort is further reduced by default modifiers. Reference types by default have the `peer` ownership modifier; only exceptions and immutable types default to `any`. These defaults make the conversion from Java to Universe Types simple, as all programs that do not directly modify caught exceptions continue to compile. However, these defaults only provide a flat ownership structure.

Standard techniques for static type inference [10] are not applicable. First, we do not have to check the existence of a correct typing. Such a typing trivially exists by making all ownership modifiers `peer`, that is, by having a flat ownership structure. Second, there is no notion of a best or most precise Universe typing. Usually, there are many possible typings, and it depends on the intent of the programmer which one to prefer.

In this paper, we describe how ownership modifiers for deep ownership structures can be found by *runtime inference*, that is, by observing the execution of a program. This approach does not require that the source code of the program is available. By using the dominator algorithm we ensure that the result is the deepest possible ownership structure that conforms to the Universe type rules. A deep ownership structure maximizes encapsulation and facilitates program verification. Nevertheless, it might not be what the programmer intended. The solution of our program therefore still needs to be reviewed by the programmer to ensure that it corresponds to the intended design.

Runtime inference depends on good code coverage to produce meaningful results. To achieve better coverage we use multiple program traces to infer the ownership modifiers. We also combine the results of runtime inference with our static inference tools [29, 16] to ensure that the final solution gives valid Universe Types for the complete program.

1.1 Related Work

Wren's work on inferring ownership [32] provided a theoretical basis for our work. It developed the idea of the Extended Object Graph and how to use the dominator as a first approximation of ownership. It builds on ownership types [8, 3, 7, 9] which uses parametric ownership and enforces the owner-as-dominator property. The number of ownership parameters for parametric type systems is not fixed and is usually determined by the programmer, as is the number of type parameters for a class. Trying to automatically infer a good number of ownership parameters makes their system complex. No implementation is provided.

Daikon [14] is a tool to detect likely program invariants from program traces. Invariants are only enforced at the beginning and end of methods and therefore also snapshots are only taken at these spots. From these snapshots we cannot infer which references were used for reading and which were used for writing. Therefore we could not directly use Daikon, but our tool has a similar architecture. In the future we hope to apply optimizations from Daikon to our tool.

SafeJava [7] provides intra-procedural type inference and default types to reduce the annotation overhead. Agarwal and Stoller [1] describe a run-time technique that infers even more annotations. AliasJava [4] uses a constraint system to infer alias annotations.

Another static analysis for ownership types resulted in a large number of ownership parameters [19]. In contrast, by using runtime information we achieve a deep ownership structure and the simplicity of Universe Types makes the mapping to static annotations possible.

Rayside et al. [30] present a dynamic analysis that infers ownership and sharing, but does not map the result back to an ownership type system. Mitchell [26] analyzes the runtime structure of Java programs and characterizes them by their ownership patterns. The tool can work with heaps with 29 million objects and creates succinct graphs. The tool does not distinguish between read and write references and the results are not mapped to an ownership type system.

Work on the dynamic inference of abstract types [18] uses the flow of values in a program execution to infer abstract types. Yan et al. [33] use state machines to map implementation events to architecture events and thereby deduce architectures. Both approaches do not seem to be applicable to infer ownership information.

1.2 Running Example

We use the classes in Fig. 1 to illustrate how the algorithm works. This is a very simple and artificial example to illustrate all aspects of the algorithm. The main class is `Demo`; the Java entry-point `main` creates an instance of class `Demo` and calls method `testA` on that instance. The argument is a boolean that depends on the number of command line arguments. Method `testA` creates an `A` instance. Class `A` stores the boolean flag and creates an instance of class `B`. Class `B` creates a `C` instance and a `java.lang.Object` instance. Finally, class `C` stores a reference to the `A` object it receives and depending on the value of the `mod` field calls the `off` method on the `A` instance. The execution of the `main` method in class `Demo` results in the objects depicted in Fig. 2.

Outline. Sec. 2 describes the algorithm to infer ownership modifiers from runtime information, Sec. 3 gives implementation details, and Sec. 4 describes the Eclipse plug-ins. Finally, Sec. 5 discusses future work and concludes.

2. Runtime Universe Type Inference

The inference of Universe Types from program executions is performed in the following five steps:

1. Build the representation of the object store
2. Build the dominator tree
3. Resolve conflicts with the Universe type system
4. Harmonize different instantiations of a class
5. Output Universe Types

We describe these steps in the following subsections. We discuss static methods at the end of this section.

2.1 Build the Representation of the Object Store

From a program execution we get a sequence of modifications of the object store. Instead of looking at only single snapshots of the store (as in [26]), we build a cumulative representation of the object store. This so-called *Extended Object Graph* (EOG) [32] represents all objects that ever existed in the store, all references between these objects that were ever observed, and, in particular, which objects modified which other objects. The information about modifications is particularly important since Universe Types do not restrict references in general (unlike other ownership type systems), but the modification of objects.

For each object in the EOG, we record information about its fields as well as the parameters and results of its methods. We use this information to infer ownership modifiers for these variables. Local variables are treated in a subsequent step as we describe in Sec. 2.5.

```
public class Demo {
    public static void main( String[] args) {
        new Demo().testA(args.length > 0);
    }

    public void testA(boolean b) {
        new A(b);
    }
}

class A {
    boolean mod;
    B b;

    A(boolean m) {
        mod = m;
        b = new B(this);
    }

    void off() {
        mod = false;
    }
}

class B {
    C c;
    Object o;

    B(A a) {
        c = new C(a);
        o = new Object();
    }
}

class C {
    A a;

    C(A na) {
        a = na;
        if( a.mod ) {
            a.off();
        }
    }
}
```

Figure 1: Running example to illustrate our inference algorithm.

We distinguish between two types of references in the EOG: write references and naming references. *Write references* are used to update a field or call a non-pure method on an object; these references mainly determine the ownership structure of an application. In addition we store references that were only used for reading fields and calling pure methods. These *naming references* are needed to map the resulting EOG back to the source code.

For example, a call x.`foo`(y) introduces two edges in the EOG. A write references from the current receiver object `this` to x represents that `this` modifies x by calling the non-pure method `foo`. This reference will later influence the ownership relation between `this` and x. A naming reference from x to y represents that a

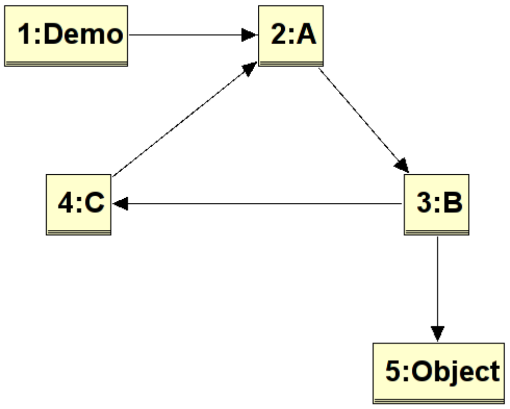


Figure 2: The store at the end of method `main` in class `Demo`. Objects are depicted by rectangles and are labeled with an identifier and the class name. References are depicted by arrows.

method of x takes y as parameter. This naming reference is labeled with the name of the formal parameter and will later be used to infer the ownership modifier of the parameter.

To determine whether a method call constitutes a modification, we need purity information. We require that the purity of methods is provided as input to our tool. There are algorithms [31] to infer method purity and we also implemented a tool [17] to help with this task.

In our running example (Fig. 1), class `A` contains the statement `b = new B(this)`. On the bytecode level, this corresponds to two steps, first the creation of a new object and then the update of the field `b` of the current object. For an object creation, we insert a write edge from the current receiver object to the newly created object. In Fig. 2, this corresponds to the edge from object 2 to object 3. This write edge ensures that the ownership modifier for the object creation is either `peer` or `rep`, which is a requirement of the Universe type system. For a field update, we store a write reference from the current object to the receiver of the field update and a naming reference from the receiver of the field update to the object on the right-hand side. The naming reference is labeled with the field name. All naming references for a field can later be used to infer the correct ownership modifier for that field.

Arrays in the Universe type system use two ownership modifiers, one for the relation between `this` and the array object, and one for the relation between the array object and the objects stored in the array. For arrays, we added a special kind of naming reference that stores the relationship between the array object and the objects that are stored in the array. These references can then be used to determine the second ownership modifier.

2.2 Build the Dominator Tree

Universe Types require that all modifications of an object are initiated by its owner. For the EOG, this means that all chains of write references from the root object to an object x must go through x's owner. Therefore, we can identify suitable candidates for the owner of x by computing the dominators of x. The concept of dominators is well-known in the compiler field [2], and efficient algorithms have been developed [22].

Universe Types do not restrict references that are merely used for reading. Therefore, the naming references in the EOG do not carry information that helps us to determine ownership relations between objects. Consequently, we ignore them when we build the dominator graph. They are later used to find the correct static ownership modifiers.

The result of finding the dominators for the graph from Fig. 2 is shown in Fig. 3a. Domination is depicted by rounded rectangles. A direct dominator sits atop the rounded rectangle that groups the objects it dominates. It is a candidate for becoming the owner of this group of objects.

2.3 Resolve Conflicts with the Universe Type System

Domination is a good approximation of ownership, but it cannot be directly used to infer Universe Types. The Universe type system only allows write references within a context and from an owner to an owned object. On the other hand, a dominator graph can have references from an object to an object in an enclosing context. Such write references are not permitted in the Universe type system. If such references are found in the EOG, the involved objects are raised to a common level until no more conflicts are present.

This problem is illustrated by the code in Fig. 1. If we observe an execution of the constructor of class `C` when `a.mod` is `false` then the `off` method is not called on the `a` reference. In this case, the reference from object 4 to object 2 is used in a read-only manner, that is, the EOG contains a naming reference between object 4 and object 2. Under this assumption, the dominator graph in Fig. 3a is a valid ownership structure in Universe Types. The reference between object 4 and object 2 is stored in field `a` of class `C`. This field will be annotated with an `any` ownership modifier.

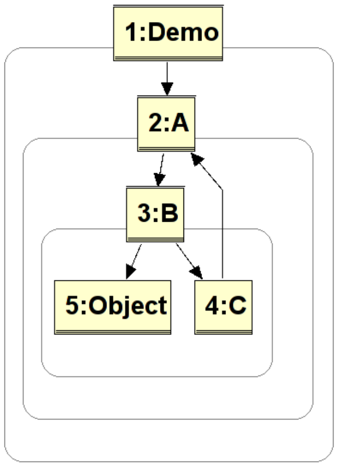
However, if `a.mod` is `true`, the non-pure method `off` is called on `a`. This results in a write reference from object 4 to object 2. In this case, the dominator graph does not represent a valid ownership structure because there is a write reference to an object in an enclosing context. This write reference can neither be typed with a `rep` nor with a `peer` modifier and is, therefore, not admissible in Universe Types. To solve this problem, we flatten the ownership structure to make the write reference from object 4 to object 2 admissible. This is done by raising the origin of the write reference (object 4) to the context that contains the destination of the write reference (object 2). This makes the two objects peers, and the write reference between them is admissible as it can be typed with modifier `peer`.

However, raising object 4, creates a conflict for the write reference from object 3 to object 4 since now object 4 is neither owned by nor a peer of object 3. Therefore, we apply the same solution again; this time, object 3 is raised to be in the same context as object 4. The resulting dominator graph is depicted in Fig. 3b. In this graph, all write references are from a direct dominator to an object it dominates or between objects with the same direct dominator. Therefore, this graph represents a valid ownership structure that can be expressed in Universe Types.

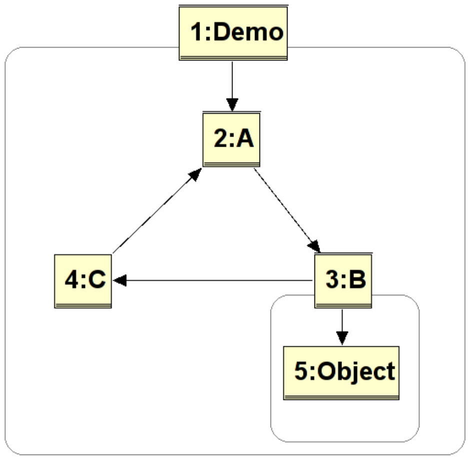
Our example shows that conflict resolution has to be applied repeatedly because resolving one conflict can cause others. Nevertheless, conflict resolution can be implemented efficiently without visiting the same write reference twice. To achieve that, we use a list of conflicting write references and process the list in a top-down way, that is, objects higher-up in the dominator graph are processed first. Moreover, we resolve conflicts that cross a large number of context boundaries before conflicts that cross fewer contexts. For details see [24].

2.4 Harmonize Different Instantiations of a Class

After conflict resolution, the EOG is consistent with the owner-as-modifier discipline. However, it might not be possible to statically type the EOG because different instances of a class might be in different ownership relations. To enforce uniformity of all instances of a class, we traverse all instances of each class and compare the ownership properties of each variable (field or parameter). This step



(a) Dominator Tree



(b) After conflict resolution

Figure 3: Contexts are depicted by rounded rectangles. Owner objects sit atop the context of objects they own.

has to take into account both write and naming references in the EOG.

If for any given variable the ownership relations are the same (for instance, they all point to peer objects), the variable can be typed statically. If they differ, we apply a resolution that is similar to the conflict resolution described in the previous subsection. If at least one instance of a variable is the origin of a peer reference and the other instances of this variable are rep references, we raise the targets of the rep references to make them peers and type the variable with modifier `peer`. If at least one instance of a variable is the origin of a reference that is neither a peer nor a rep reference, the variable is typed with modifier `any`. In this case, downcasts are needed at the point where this variable is used for field updates and calls to non-pure methods.

For example, imagine that method `testA` in class Demo is once called with `false` and once with `true` as the argument. Then we have two instances of class `A`, once with a deep ownership structure as in Fig. 3a and once with a flat structure as in Fig. 3b. The annotation for field `b` in class `A` is once `rep` and once `peer`. The algorithm then decides to use `peer` as annotation for field `b` and raises the non-conforming instance to a higher level. Because we raise an object together with all peers that reference it or are referenced by it, this step cannot create new conflicts in the ownership graph.

2.5 Output Universe Types

After the first four steps of the algorithm, we have determined ownership modifiers for field declarations, method parameters and results, and allocation expressions. The last step is to output these ownership modifiers and insert them into the source code, if it is available.

Local variables are not inferred from the EOG because that would require monitoring every assignment of a local variable, which would slow down the inference. As an implementation problem, Java JVMTI does not support monitoring of local variable assignments, and we deemed a solution using bytecode instrumentation too heavy-weight.

Inferring ownership modifiers for locals is very similar to Java's bytecode verification [23]. Both infer the types of local variables based on the types of fields and method signatures. Like bytecode verification, we symbolically execute the bytecode of a method body to obtain the ownership modifiers of local variables. This step might introduce downcasts when `any` references are used to modify objects. These casts are not guaranteed to succeed at runtime. Therefore, they should be reviewed by the programmer.

Fig. 4 shows the result of our inference for the example source code from Fig. 1. The ownership modifiers are inferred after processing program executions with and without command-line arguments. This source code complies to the Universe type system. By inserting the ownership modifiers into the source code, we ensure that future revisions of this code will maintain the ownership structure.

2.6 Static Methods

In Universe Types, static methods are either executed in the context in which the caller is executed or in the context owned by `this`. In the former case, the target type of the call to the static method has a `peer` modifier; in the latter case, it has a `rep` modifier. `any` is not permitted.

When we monitor the execution of a program, no object exists that corresponds to the target of the static call. In the EOG, we create an artificial target object as the receiver of a static method call. The relationship between the current object and the artificial object determines the ownership modifier for the static call. To enforce that the target of a static call does not have the `any` modifier, we always treat static method calls as non-pure. This creates a write reference in the graph and ensures that a `peer` or `rep` modifier is inferred.

Our treatment of static methods is illustrated by the example in Fig. 5. Consider the call x`.foo(`y`)`. The execution of `foo` affects three objects in the EOG: the receiver x, the parameter y, and an artificial target object for the call to `process`, say z. We add a write edge from x to z because x calls the static method. We also add a write reference from z to y because `process` calls a non-pure method on y. Since Universe Types do not allow `rep` modifiers in static methods, the latter write reference forces the parameter `p` of `process` to have a `peer` modifier. The modifier of the target type of the call to `process` is determined by the relation between the current receiver x and parameter y. Since `process` expects a `peer` parameter, y and the artificial target object z must have the same owner. Therefore, if x owns y, then x also owns z, and the annotated call will be `rep S.process(q)`. If x and y are peers, the call will be `peer S.process(q)`. In all other cases, step 2 of

```
public class Demo {
    public static void main(any any String[] args) {
        new peer Demo().testA(args.length > 0);
    }

    public void testA(boolean b) {
        new rep A(b);
    }
}

class A {
    boolean mod;
    peer B b;

    A(boolean m) {
        mod = m;
        b = new peer B(this);
    }

    void off() {
        mod = false;
    }
}

class B {
    peer C c;
    rep Object o;

    B(A a) {
        c = new peer C(a);
        o = new rep Object();
    }
}

class C {
    peer A a;

    C(peer A na) {
        a = na;
        if( a.mod ) {
            a.off();
        }
    }
}
```

Figure 4: Running example with inferred ownership modifiers.

the inference will automatically adapt the relation between x and y and, thereby, the relation between x and z.

3. Implementation

Fig. 6 shows the architecture of the implementation. The tool is split into two parts: Sec. 3.1 describes the tracing agent, which monitors the execution of Java programs. Sec. 3.2 describes the inference tool, which determines the ownership modifiers.

3.1 Tracing Agent

We monitor a Java Virtual Machine (JVM) execution with a Java Virtual Machine Tooling Interface (JVMTI) agent written in C. JVMTI is the low-level interface provided by the Java Platform Debugger Architecture (JPDA) [20].

```
class S {
    static T process(T p) {
        p.nonpureOperation();
        return p;
    }

    T foo(T q) {
        return S.process(q);
    }
}
```

Figure 5: Example for static methods.

The agent receives events from the virtual machine and produces a trace file that documents the execution of the program. The trace file is in a simple XML format. Storing the execution of a program in a trace file gives the following advantages: (1) Multiple trace files can be generated to achieve good code coverage. In our example, one should trace the execution of class Demo once without any command-line arguments and once with an argument. (2) Interactive or long-running programs need to be traced only once for each desired code path. This trace file can then be reused later without requiring human interaction or recomputing results.

On the other hand, storing the trace files on disc and then parsing them again in the next phase sometimes leads to a performance overhead. In the beginning of this project, we investigated the Java Debug Interface (JDI) as high-level alternative to the low-level JVMTI. The JDI versions up to Java 5 did not provide enough information to allow our Universe inference, especially the value returned by a method was not accessible. In Java 6 the JDI API was enhanced and we investigate adding JDI support as an alternative source of program traces.

JVMTI does not provide the necessary events for array component updates. Therefore we used instrumentation of the Java bytecode to create artificial events for array updates.

3.2 Inference Tool

The main inference tool is an independent Java 5 application that performs the steps outlined in Sec. 2. It reads (multiple) trace files generated by the tracing agent and builds one Extended Object Graph from the available information. Then the dominators are determined, conflicts are resolved, multiple instances are harmonized, and the output is written to an XML file. The different steps of the algorithm are implemented as visitors that manipulate the EOG.

The application is configured by a simple XML file that determines what input and output files to use and which visitors and observers to use. This extensible architecture allows us to support a command line interface and the Eclipse plug-ins described in Sec. 4, and will also allow us to add JDI as an alternative input.

The output of our inference tool is an annotation XML file that contains the ownership modifiers for the encountered types. For this annotation XML file, we defined an XML schema that can provide ownership modifiers for the different types. If the source code of the traced program is available then the annotations can be inserted into the source code using a separate annotation tool we developed. Producing the output in XML will allow us to support several annotation tools, for instance, for the existing Universe syntax and for JSR 308-style annotations.

To build the correct EOG, we need to know which methods are pure. We use a separate annotation XML file as additional input to the inference tool to provide this purity information. This XML file has the same schema as the output file, which allows us to use the annotation editor, visualizer, and insertion tool to create

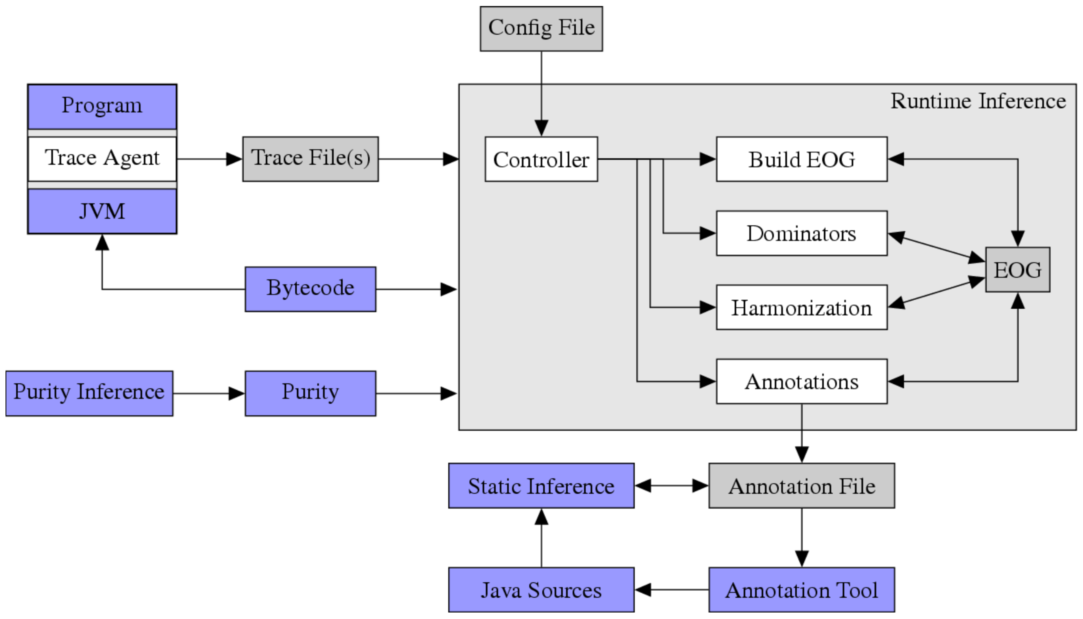


Figure 6: Architecture of the runtime inference tools. White boxes depict components of the inference tool. Boxes in light gray depict files and data structures that are part of the inference tool, and boxes in dark gray depict external components and files.

the input. To ease the creation of this purity information, we also implemented a purity inference tool [31, 17].

The XML file in Fig. 7 shows the result of applying the inference algorithm (without inference of local variables) to our running example (see Fig. 4 for the annotated source code). The Java structure is modeled in the XML structure, and the modifier attribute is used to provide the ownership modifier for the corresponding type or the purity for a method.

4. Eclipse Integration

To ease the usage of the command-line tools, we created a set of Eclipse 3.2 [15] plug-ins that integrate the runtime inference into the standard Java development environment.

4.1 Tracing

Eclipse allows one to execute Java programs directly from the IDE using “Run As” configurations. The programmer can use these configurations to set, for example, command-line arguments and the JVM to use. We added a new “Run As” configuration that allows one to trace program executions. The only additional information the user has to provide is the name of the trace file. The plug-in takes care of configuring the Java tracing agent correctly.

We provide the complete configuration information on a separate tab. This information can be copied into a script and allows the user to configure the tracing agent within Eclipse, but then use the command-line tool directly.

4.2 Inference

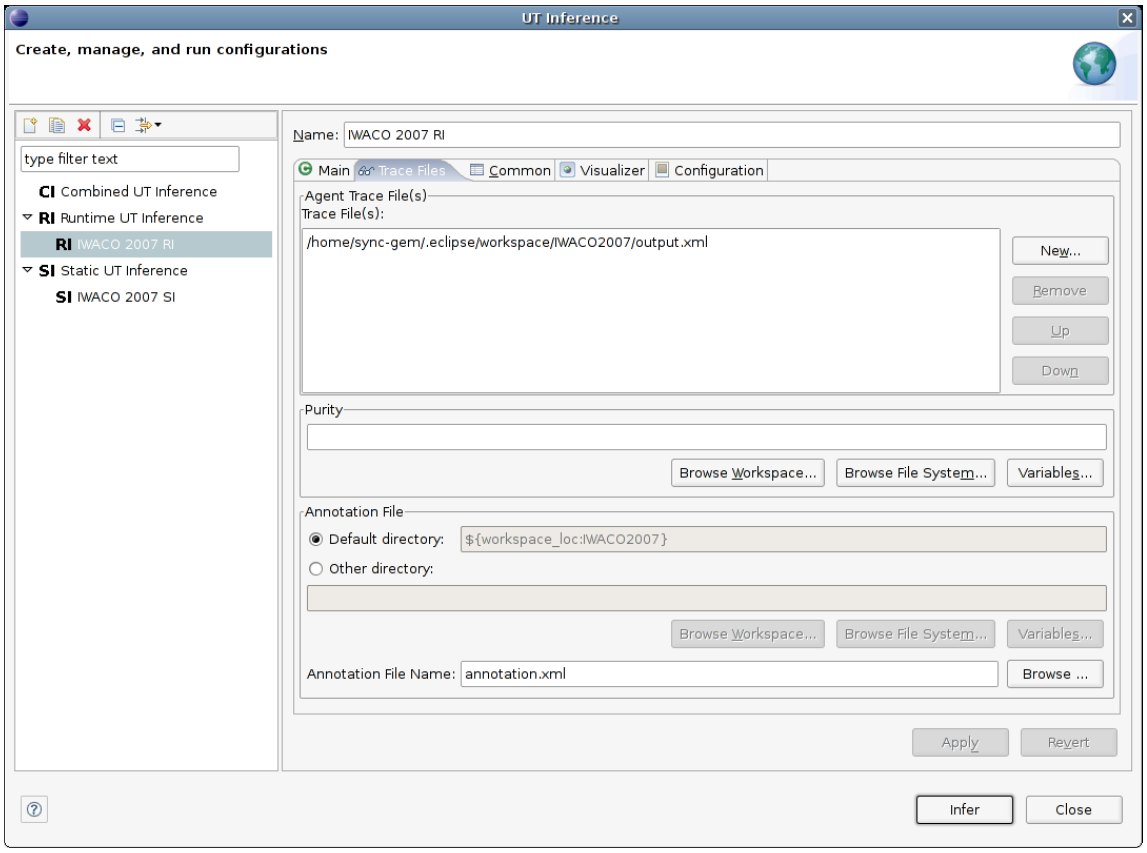
Once the program was traced, the Universe Types can be inferred with a separate plug-in. Similarly to the “Run As” dialog, we provide the possibility to manage different configurations. The main configuration tab (shown in Fig. 8) allows one to easily configure the trace files, purity information, and output file that should be used. Again, we provide a tab that allows one to use the configuration from the command line.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ann:annotations
  xmlns:ann="http://sct.inf.ethz.ch/annotations">
  <ann:head>
    <target>java</target>
    <style>types</style>
  </ann:head>
  <ann:class name="A">
    <ann:field modifier="rep" type="B" name="b"/>
  </ann:class>
  <ann:class name="B">
    <ann:field modifier="rep" type="C" name="c"/>
    <ann:field modifier="rep" type="java.lang.Object"
            name="o"/>
    <ann:method name="B" signature="A" modifier="">
      <ann:parameter index="0" modifier="any" type="A"
                 name="param0"/>
    </ann:method>
  </ann:class>
  <ann:class name="C">
    <ann:field modifier="any" type="A" name="a"/>
    <ann:method name="C" signature="A" modifier="">
      <ann:parameter index="0" modifier="any" type="A"
                 name="param0"/>
    </ann:method>
  </ann:class>
  <ann:class name="Demo"/>
</ann:annotations>
```

Figure 7: XML output of the inference tool.

4.3 Annotation Management

The result of the runtime inference is an XML file that contains the inferred ownership modifiers. This XML file can be either edited with the standard XML editor or with a special annotation editor. The annotation editor (shown in Fig. 9) allows one to edit the ownership information, for instance, by providing drop-down lists


UT Inference

Create, manage, and run configurations

CI Combined UT Inference
RI Runtime UT Inference
RI IWACO 2007 RI
SI Static UT Inference
SI IWACO 2007 SI

Name: IWACO 2007 RI

Agent Trace File(s)
Trace File(s):
/home/sync-gem/.eclipse/workspace/IWACO2007/output.xml

Purity

Annotation File
Default directory: ${workspace_loc:IWACO2007}
Other directory:

Annotation File Name: annotation.xml

Figure 8: Screen shot of the configuration dialog for the type inference. The user can set the tool options, for instance, which trace files to use and what output file to generate.

of possible ownership modifiers. If the source code of the program is available, we can automatically insert the ownership modifiers from the XML file into the source.

4.4 Visualizer

The flexible observer architecture that we chose for the inference tool allowed us to add a graphical visualizer to the inferer. This visualizer (shown in Fig. 10) uses the Eclipse Graphical Editing Framework (GEF) to display the extended object graph while it is built up and modified during the execution. This gives a clear understanding of how the program executes and how the inference algorithm works.

The visualizer adds a new toolbar to Eclipse. Here the user can set the zoom level, use automatic or manual layout of the graph, "play" the evolution of the inference algorithm, take a single step of the algorithm, or pause the animation. It further provides buttons that help in the manual layout of the graph. The automatic layout of the graph nodes is used by default. It automatically positions the nodes and routes the edges to have a nice diagram. It uses a simplex algorithm that tries to minimize the crossings of edges [16]. The manual layout can be used to manipulate the graph by hand.

The objects in the graph can be shown with and without the fields and methods that the corresponding class has. The display of this additional information follows the UML standard for object diagrams.

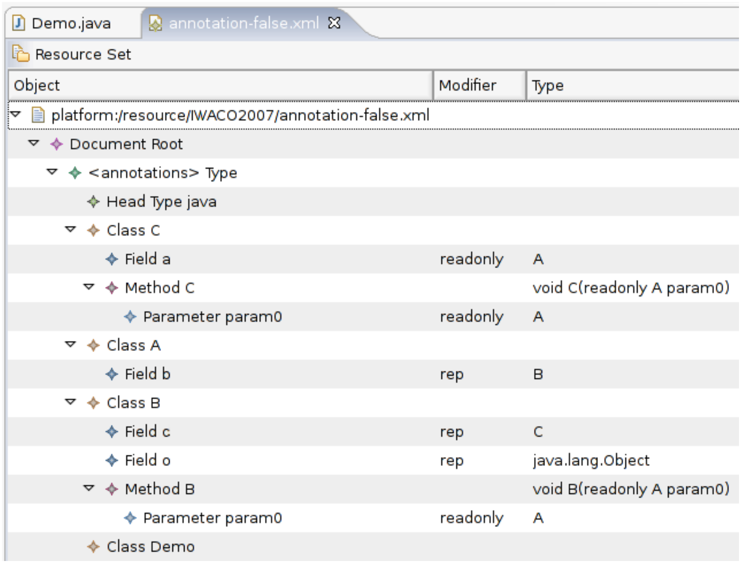
5. Conclusion and Future Work

This paper presented the current status of our work on runtime Universe type inference. We successfully used the tools in small case studies such as linked list and tree implementations. In these examples, the overhead of tracing the execution and the calculation of the ownership modifiers was reasonable. Even for small examples, the support for multiple trace files was very useful to increase the code coverage and, thus, the quality of the inferred ownership.

As future work, we plan to carry out non-trivial case studies. Inferring ownership for major applications will not only allow us to further evaluate and optimize our tools, but also provide insights into the structure of real applications. We expect this information to be valuable for further research on ownership in general.

Currently, our inference tool only works for non-generic Java. We recently developed Generic Universe Types [12, 11] and we will investigate whether runtime inference can be extended to generics. The problem is that genericity in Java 5 is implemented by erasure, that is, the type arguments are not visible to the virtual machine. It will also be interesting to study runtime inference in the presence of ownership transfer [28].

We plan to add JDI support to directly trace program executions without creating trace files. The inference visualizer is under active development and we have many ideas to make the interaction more convenient and to improve scalability to large object graphs. Examples include hierarchical folding of sub-trees, searching for instances of a particular class, and visual enhancements.



Object	Modifier	Type
platform:/resource/IWACO2007/annotation-false.xml		
Document Root		
<annotations> Type		
Head Type java		
Class C		
Field a	readonly	A
Method C		void C(readonly A param0)
Parameter param0	readonly	A
Class A		
Field b	rep	B
Class B		
Field c	rep	C
Field o	rep	java.lang.Object
Method B		void B(readonly A param0)
Parameter param0	readonly	A
Class Demo		

Figure 9: Screen shot of the annotation editor. The editor gives a tree view of the ownership information contained in an annotation XML file. Editing is simplified by drop-down lists of possible values.

Finally, we are integrating the runtime inference with our static inference tools [16]. This allows us to propagate and check the partial information that is inferred from program traces and ensures that the resulting annotations comply with the Universe type system.

Acknowledgments

This work builds on the Master's and semester theses of students at ETH Zurich: Frank Lyner [24] developed the first version of the command-line tool, Marco Bär [5] improved and extended the command-line tools, Marco Meyer [25] created the first annotation tool and visualizer, Paolo Bazzi [6] created the Eclipse plug-ins for runtime inference, David Graf [17] implemented the purity-inference tool, and Andreas Fürer [16] improved and created the Eclipse plug-ins and created the current version of the annotation tool and visualizer. Peter Müller's work was done at ETH Zurich.

References

[1] R. Agarwal and S. D. Stoller. Type Inference for Parameterized Race-Free Java. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2937 of *Lecture Notes in Computer Science*, pages 149–160. Springer-Verlag, January 2004.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition*. Addison-Wesley, 2007.

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

[4] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 311–330. ACM Press, 2002.

[5] M. Bär. *Practical Runtime Universe Type Inference*. Master's thesis, Department of Computer Science, ETH Zurich, 2006.

[6] P. Bazzi. *Integration of Universe Type System Tools into Eclipse*. Semester Project, Department of Computer Science, ETH Zurich, Summer 2006.

[7] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.

[8] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310. ACM Press, 2002.

[9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, 1998.

[10] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Principles of programming languages (POPL)*, pages 207–212. ACM Press, 1982.

[11] W. Dietl, S. Drossopoulou, and P. Müller. Formalization of Generic Universe Types. Technical Report 532, ETH Zurich, 2006. `sct.inf.ethz.ch/publications`.

[12] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

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

[14] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[15] The Eclipse Foundation. Eclipse — an open development platform. `www.eclipse.org/`.

[16] A. Fürer. *Combining Runtime and Static Universe Type Inference*. Master's thesis, Department of Computer Science, ETH Zurich, 2007.

[17] D. Graf. *Implementing Purity and Side Effect Analysis for Java Programs*. Semester Project, Department of Computer Science, ETH Zurich, Winter 2005/06.

[18] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In Lori L. Pollock and Mauro Pezzè, editors, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 255–265. ACM, 2006.

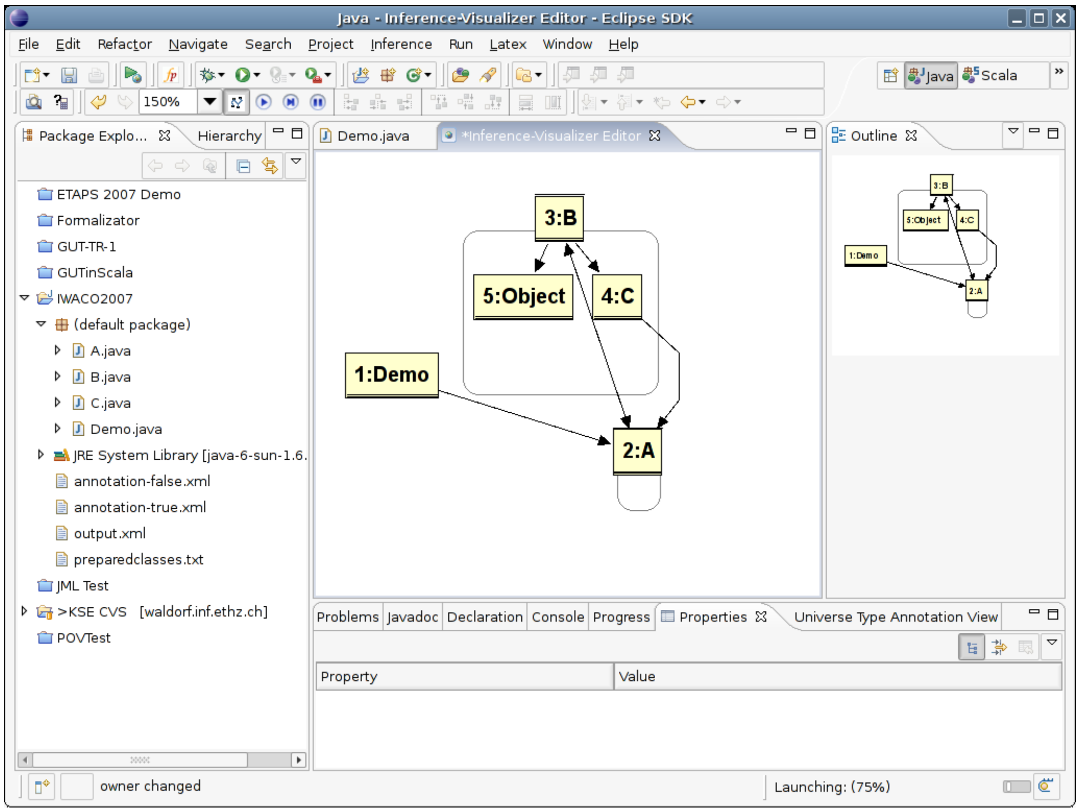

Java - Inference-Visualizer Editor - Eclipse SDK
File Edit Refactor Navigate Search Project Inference Run Latex Window Help
150%
Java Scala
Package Explo... Hierarchy
Demo.java *Inference-Visualizer Editor
Outline
ETAPS 2007 Demo
Formalizator
GUT-TR-1
GUTinScala
IWACO2007
(default package)
A.java
B.java
C.java
Demo.java
JRE System Library [java-6-sun-1.6.
annotation-false.xml
annotation-true.xml
output.xml
preparedclasses.txt
JML Test
>KSE CVS [waldorf.inf.ethz.ch]
POVTest
3:B
5:Object
4:C
1:Demo
2:A
Problems Javadoc Declaration Console Progress Properties Universe Type Annotation View
Property
Value
owner changed
Launching: (75%)

Figure 10: Screen shot of the inference visualizer. The main editor in the center shows a (zoomable) image of the EOG. The outline view on the right helps in keeping the overview. The properties tab at the bottom gives additional information about selected elements. The status bar outputs information about the steps of the algorithm.

[19] S. E. Moelius III and A. L. Souter. An object ownership inference algorithm and its application. In M. T. Morazan, editor, *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.

[20] Sun Microsystems Inc. Java Platform Debugger Architecture (JPDA). `java.sun.com/javase/technologies/core/toolsapis/jpda/`.

[21] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from `www.jmlspecs.org`, 2006.

[22] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979. Available from `doi.acm.org/10.1145/357062.357071`.

[23] X. Leroy. Java bytecode verification: An overview. In *Computer Aided Verification (CAV)*, volume 2102, pages 265–285, 2001.

[24] F. Lyner. *Runtime Universe Type Inference*. Master's thesis, Department of Computer Science, ETH Zurich, 2005.

[25] M. Meyer. *Interaction with Ownership Graphs*. Semester Project, Department of Computer Science, ETH Zurich, Summer 2005.

[26] N. Mitchell. The runtime structure of object ownership. In Dave Thomas, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 74–98. Springer-Verlag, 2006.

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

[28] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007. To appear.

[29] Matthias Niklaus. *Static Universe Type Inference using a SAT-Solver*. Master's thesis, Department of Computer Science, ETH Zurich, 2006.

[30] D. Rayside, L. Mendel, and D. Jackson. A dynamic analysis for revealing object ownership and sharing. In *Workshop on Dynamic systems analysis (WODA)*, pages 57–64. ACM Press, 2006.

[31] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag, 2005.

[32] A. Wren. Inferring ownership. Master's thesis, Department of Computing, Imperial College, June 2003. `www.cl.cam.ac.uk/~aw345/`.

[33] H. Yan, D. Garlan, B. R. Schmerl, J. Aldrich, and R. Kazman. DiscoTect: A system for discovering architectures from running systems. In *International Conference on Software Engineering (ICSE)*, pages 470–479, 2004.

Compile-Time Views of Execution Structure Based on Ownership

Marwan Abi-Antoun
School of Computer Science
Carnegie Mellon University
marwan.abi-antoun@cs.cmu.edu

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
jonathan.aldrich@cs.cmu.edu

Abstract

A developer often needs to understand both the code structure and the execution structure of an object-oriented program. Class diagrams extracted from source are often sufficient to understand the code structure. However, existing static or dynamic analyses that produce raw graphs of objects and relations between them, do not convey design intent or readily scale to large programs.

Imposing an ownership hierarchy on a program's execution structure through ownership domain annotations provides an intuitive and appealing mechanism to obtain, at compile-time, a visualization of a system's execution structure. The visualization conveys design intent, is hierarchical, and thus is more scalable than existing approaches that produce raw object graphs.

We first describe the construction of the visualization and then evaluate it on two real Java programs of 15,000 lines of code each that have been previously annotated. In both cases, the automatically generated visualization fit on one page, and gave us insights into the execution structure that would be otherwise hard to obtain by looking at the code, at existing class diagrams, or at unreadable visualizations produced by existing compile-time approaches.

1. Introduction

When modifying an object-oriented program, both the code structure (static hierarchies of classes) and the execution structure (dynamic networks of communicating objects) must be understood. "For a developer unfamiliar with the system to obtain this understanding is a non-trivial task. Little work has been done on minimizing this learning curve" [38].

In many cases, developers cannot rely that external design documentation is up-to-date. Many tools can automatically generate class diagrams from program source [21]. However, a class diagram shows the code structure and does not explain the execution structure of the system. In object-oriented design patterns, much of the functionality is determined by what instances point to what other instances. For instance, in the Observer design pattern [15, p. 293], understanding "what" gets notified during a change notification is crucial for the function of the system, but "what" does not usually mean a class, "what" means a particular instance. Furthermore, a class diagram often shows several classes depending on a single container class such as `java.util.ArrayList`. However, different instantiations of such a class often correspond to different elements in the design, hence the need for an instance-based view to *complement* a class diagram.

A running object-oriented program can be represented as an *object graph*: nodes correspond to objects and edges correspond to relations between objects. Existing dynamic analyses can describe the runtime object graph of a system for a particular set of inputs and exercised use cases [12, 33]. Obtaining at compile time a finite and conservative abstraction of all possible runtime object graphs is more challenging because of aliasing, precision and scalability issues. Static analyses [29, 40] that approximate the runtime object graph often produce large non-hierarchical graphs that do not convey design intent and do not scale to large programs (See visualizations [2] for examples).

Many type systems enforce *ownership* at compile time, i.e., make one object part of another object's representation [8, 7, 3, 11]. In the ownership domains type system [3], each object contains one or more public or private *ownership domains* — conceptual groups of objects — and each object is in exactly one domain. As with most other ownership type systems, adding ownership domain annotations to a program's source code can control aliasing and enforce *instance encapsulation* which is stronger than module visibility mechanisms. Moreover, ownership domains can express and enforce a tiered runtime architecture by representing a tier as an ownership domain. A *domain link* can abstract permissions of when objects can communicate [1].

Our contribution in this paper is to leverage ownership domain annotations to obtain at compile-time a sound visualization of the execution structure of a program with ownership domain annotations, the Ownership Object Graph. The visualization is hierarchical, conveys design intent and compares favorably with existing compile-time visualizations of two previously annotated Java programs, each consisting of 15,000 lines of code.

Currently, annotations are added mostly manually, however, active work in the area of semi-automated annotation inference [4, 9, 24, 25] promises to lower the annotation overhead. The visualization reflects the annotations, and the quality of the visualization reflects the quality of the annotations. The design intent is expressed by choosing the ownership domains and their structure, then adding annotations to the program — currently manually.

The ideas and techniques of ownership are fundamental for obtaining such a compile time visualization. First, ownership domains provide a coarse-grained ownership structure of an application with a granularity larger than an object or a class [37]. Second, ownership organizes a flat object graph into an ownership tree, and hierarchy is needed to achieve scalability and attain both high-level understanding and detail. Third, different ownership domains and different places in the hierarchy provide precision about inter-domain aliasing and conservatively describe all aliasing that could take place at runtime. Since two objects in two different domains cannot be aliased, the analysis can distinguish between instances that would be merged in a class diagram, allowing better understanding of the runtime structure of the system. Fourth, ownership domain names are specified by a developer and therefore can convey more design intent than the aliasing information obtained using a static analysis that does not rely on annotations [34].

We first define the Ownership Object Graph (Section 2) and describe the algorithm to construct it at compile time (Section 3). We then present concrete and in-depth examples of the visualization of two real annotated 15,000-line object-oriented programs (Section 4). Finally, we survey related work in Section 5 and conclude.

2. The Ownership Object Graph

This section discusses the challenges in visualizing an annotated program and describes the different intermediate representations we used to obtain the visualization.

A running object-oriented program can be represented as a *runtime object graph*: nodes correspond to *runtime objects* and edges correspond to relations between runtime objects such as creation, usage and reference [32]. The aim is to statically approximate all of the runtime object graphs that may be generated in any run of the program. The goals of the visualization are as follows:

- **Scalability:** to support high-level understanding, the visualization groups runtime objects into relatively few top-level "abstract" elements, each represented by a canonical object;
- **Hierarchy:** to provide detailed understanding, the visualization supports the ability to show the substructure of an abstract element. Thus the visualization can be viewed as a hierarchical tree of objects;
- **Design Intent:** the visualization groups runtime objects into clusters that are meaningful abstractions — e.g., that an object is in a tier — and documents design-level constraints using domain links — e.g., that two tiers may communicate. The user provides the design intent regarding object encapsulation and communication using ownership domain annotations [1];
- **Soundness:** to ensure that the visualization is a faithful representation of the runtime object graph, it must be *sound*. In particular, all objects and relations present at runtime should be represented. Furthermore, if two variables may alias at runtime, they should appear in the graph as a single "abstract" element.

The analysis builds two intermediate representations, an *abstract graph*, which is converted into a *visual graph*, which is then displayed as the Ownership Object Graph.

2.1 Abstract Graph

The *abstract graph* is built from ownership domain annotations in the source code (Figure 1). The syntax for declaring and using ownership domains follows that used for Java generics [3].

For each type in the program, the abstract graph shows the ownership domains declared in it, and shows field and variable declarations as *abstract objects* declared inside *abstract domains*. The abstract graph provides scalability through ownership hierarchy and captures design intent as described above, but is not adequate for visualization for several reasons (See Figure 2).

First, the abstract graph is not really hierarchical in the sense of an object having children; rather, an object has a type and the type has domains and the domains have object children. Second, it does not include all objects: a domain contains abstract objects only for the locally declared fields, but if that domain is passed as a domain parameter to another object, and that object declares its fields in that domain, those non-local fields will not be represented. Third, it does not show all aliasing: different field declarations — and therefore different abstract objects, could be aliased and thus must be shown as one. To realize the properties above, the abstract graph is converted into a *visual graph*.

2.2 Visual Graph

The visual graph is an intermediate representation which instantiates the types in the abstract graph and shows only objects and domains: each *visual object* contains *visual domains* and each *visual domain* contains *visual objects*. Thus, in the visual graph, one can view the children of an object without going through its declared type. Furthermore, to support the visualization goals listed earlier, the construction of the visual graph takes into account *object merging*, *object pulling* and *type abstraction*.

We visualize ownership domains as follows: a dashed border white-filled rectangle represents an actual ownership domain. A solid border grey-filled rectangle with a bold label represents an object. A dashed edge represents a link permission between two ownership domains. A solid edge represents a creation, usage, or reference relation between two objects. An object labeled "obj : T" indicates an object of type T as in UML object diagrams.

Object Merging. In the visual graph, a canonical visual object is created to represent all the abstract objects of a given type in a given source-level domain declaration. Two abstract objects in the same domain in the abstract graph, if related by inheritance, could indeed refer to the same runtime object, and thus are merged for soundness. In general, this object may summarize multiple runtime objects. For the annotated code in Figure 1, the visual graph in Figure 3 merges into one visual object (labelled with `t1: Teller`)

```
class Branch< CUSTOMERS > /* Formal domain parameter */ {
  public domain TELLERS, VAULTS;
  link TELLERS -> VAULTS;

  CUSTOMERS Customer c1;
  TELLERS Teller t1;
  TELLERS Teller t2;
  VAULTS Vault v1;
  VAULTS Vault v2;
}
class Bank {
  domain owned; /* Private default domain */

  /* Bind Branch<CUSTOMERS> formal to 'owned' actual */
  owned Branch<owned> b1;
}
```

Summary of syntax for ownership domains annotations [3]:
d T o: declare object o of type T in domain d;
[public] domain a: declare private [or public] domain;
class C<d>: declare formal domain parameter d on class C;
C<actual> cObj: provide actual for formal domain parameter;
link b -> d: give domain b permission to access domain d;

Figure 1. Ownership domains illustrated with a simplified Bank system [3]. `Branch` declares two domains, `TELLERS` for `Teller` objects and `VAULTS` for `Vault` objects. `Branch` also declares a domain link from the `TELLERS` domain to the `VAULTS` domain to allow `Teller` objects to access `Vault` objects. `Branch` also takes a `CUSTOMERS` formal domain parameter to hold `Customer` objects. `Bank` references a `Branch` object in field `b1`, binding the `CUSTOMERS` formal domain of `Branch` to the `Bank`'s own private domain `owned`.

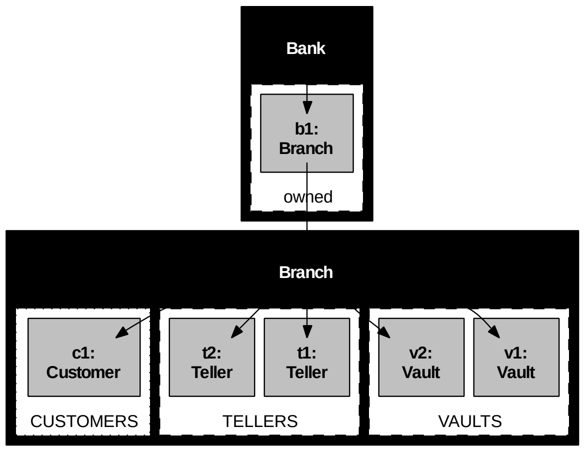


Figure 2. The abstract graph for the Bank system. A black-filled box represents a type, with white-filled domains declared inside it and grey objects declared inside each domain.

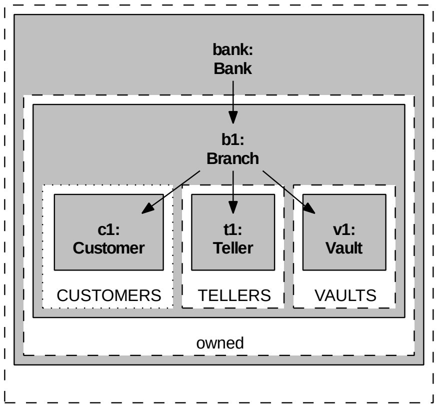


Figure 3. The visual graph for a `Branch` object *without* pulling: objects `t1` and `t2` are merged in domain `TELLERS`, and similarly, objects `v1` and `v2` in domain `VAULTS`. Object `c1` is shown in the formal domain parameter `CUSTOMERS` (dotted border).

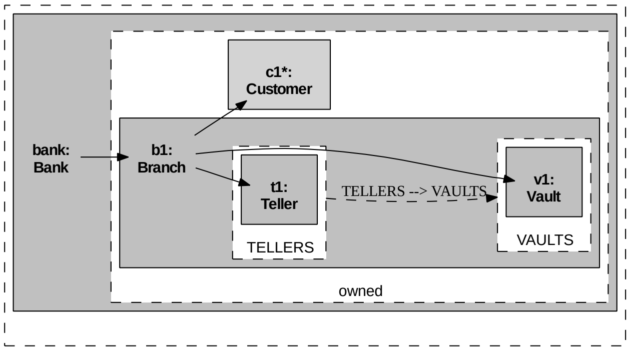


Figure 4. Object `c1*` was *pulled* from the formal domain parameter `CUSTOMERS` in Figure 3 into the actual domain `Bank.owned` to which it is bound. The dashed edge represents a domain link between `TELLERS` and `VAULTS`.

the abstract objects `t1` and `t2` declared in domain `TELLERS` since they have the same declared type.

Merging objects of the same declared type that are in the same domain may be imprecise. For instance, two `Vector` objects in the same domain would get merged even if they are never aliased. Our analysis remains more precise than a class diagram which also summarizes objects by type, because the type system guarantees that two objects that are in two different domains can never be aliased. In some cases, adding generic types where applicable, e.g., for generic containers, can minimize excessive merging.

A developer can also prevent merging by placing two objects that should never get merged in separate domains, e.g., by defining two domains `CASHVAULT` and `GOLDVAULT` to store `v1` and `v2` in Figure 1 instead of using a single domain `VAULTS`.

Object Pulling. The abstract graph may display an object only in the domain where the domain is declared as a formal parameter. But in the visual graph, each runtime object that is actually in a domain must appear where that domain is declared. To ensure this property of visual graphs, an abstract object declared inside a formal domain is *pulled* into each domain that the formal domain is transitively bound to. Figure 3 shows object `c1` in the formal domain parameter `CUSTOMERS` (dotted border). In Figure 4, object `c1` — marked with $*$ — was pulled from the formal domain `CUSTOMERS` in `Branch` to the actual domain `owned` in `Bank` (the former is bound to the latter using the annotation `Branch<owned>` on field `b1` in Figure 1).

Type Abstraction. For soundness, it may be necessary to merge abstract objects of different but compatible declared types. For example, consider the classes from the Java Abstract Window Toolkit (AWT) library in Figure 5. A variable of type `Window` and a different variable of type `Frame` in the same domain may alias each other, the corresponding abstract objects must therefore be merged for soundness.

In addition, it may be useful to do further heuristic merging to improve abstraction and reduce clutter in the graph. For example, if abstract objects of type `Button`, `Panel` and `Frame` were declared in the same domain, it may make sense to merge them into a single visual object of type `Component` or `Accessible`. On the other hand, merging can be taken too far: merging all the abstract objects in a domain into a single visual object of type `java.lang.Object` would result in a trivial and uninteresting visual graph. Thus, we heuristically merge abstract objects whenever they share one or more non-trivial *least upper bound types*. The resulting visual object is marked as having an intersection type that includes all the least upper bounds. In the example above, the least upper bound would be the intersection of the set {`Component`, `Accessible`}.

The definition of "trivial" is user-configurable; typically types such as `Object` and `Serializable` are trivial, and so abstract objects which share these as a supertype are not merged according to this heuristic. Again, a developer controls this heuristic by adding or removing types from the list of trivial types.

Instantiation-Based View. Merging abstract objects based on non-trivial least-upper-bound types can sometimes lead to unwanted merging. For instance, in the JHotDraw case study discussed in Section 4.2, both interfaces `Command` and `Tool` are in the same `Controller` domain and both extend the same interface `ViewChangeListener`. As a result, the abstract objects for `Command` and `Tool` get merged into the same visual object unless interface `ViewChangeListener` is added to the list of trivial types. However, this would not work since several variables have `ViewChangeListener` as their declared type.

The key insight however is that there are no object allocations of the interface `ViewChangeListener` since an interface cannot be instantiated directly. As an alternative to merging abstract objects, it is possible to achieve soundness by scanning object allocations instead of field and variable declarations, and then only adding visual objects for types that are actually instantiated and not the ones that are just declared. This technique is similar to how Rapid Type Analysis (RTA) [5] determines the receiver of a method call during the construction of a call graph.

In the example above, if the analysis encounters an object allocation of a `Tool` object but never that of a `ViewChangeListener` object, the analysis would only create a visual object for `Tool`, and similarly for `Command`, thus achieving the desired effect of keeping `Command` and `Tool` distinct. This solution can also prevent merging all the abstract objects in a domain into a single visual object of

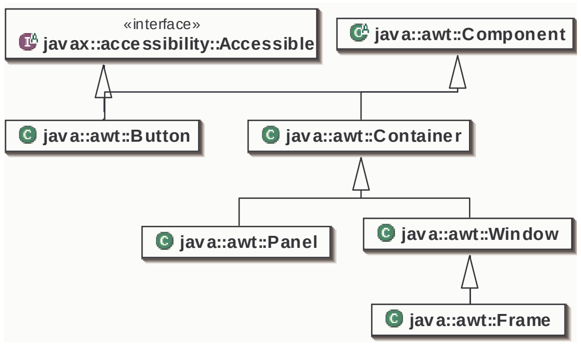


Figure 5. Type hierarchy excerpts from AWT.

type `java.lang.Object`. If the analysis does not encounter an allocation expression of the form `new Object()` in the code, it never creates a visual object for the `java.lang.Object` abstract type.

A class hierarchy analysis could determine that a variable of type `ViewChangeListener` could alias a variable of type — of course, an alias analysis could do better. A newly allocated object can be considered un-aliased or `unique` [3]. A standard flow analysis can track the flow of an object from its point of creation to the point at which it is first assigned to an ownership domain.

Design Intent Types. Since the visualization is instance-based, labelling instances is important for conveying design intent. A visual object can merge one or more abstract objects, and each abstract object has an abstract type corresponding to a declared type in the program. A visual object is labelled "`obj: T`" as in UML object diagrams – where `obj` is an optional instance name and `T` is an optional type name. An abstract object maintains the field name or variable name in the program. `obj` is selected from one of the abstract objects merged into a visual object. `T` is a list of least upper bound types as discussed above. The user can optionally specify a list of informative *design intent* types. A *design intent type* is the preferred abstract type used to label a visual object. A trivial type is not used in the label unless it occurs as a declared type in the program. Design intent types do not affect the soundness of the Ownership Object Graph and are just for labelling.

2.3 Ownership Object Graph

A visual object can contain itself so the visual graph must represent a potentially unbounded runtime object graph with a finite graph. For example, consider a class `C` which declares a domain `d` and a field of type `C` in domain `d`:

```
class C {
    domain d; /* Declare domain d */
    d C f;
}
```

Since there is a unique canonical object for each type in each domain, the object representing `C` in domain `d` must also represent the child object of type `C` in domain `d` of the parent; it is therefore its own parent in this representation. A finite representation is essential to ensure that the analysis terminates, but we want to show the user a hierarchical view where no object is its own parent. We therefore compute the Ownership Object Graph as a finite, depth-limited, unrolling of the visual graph. In the example above, we would show one `C` object within another down to a finite depth.

To summarize, an Ownership Object Graph is a graph with two types of nodes, objects and domains. The nodes form a hierarchy where each object node has a unique parent domain and each domain node has a unique parent object. The root of the graph is a top-level domain. In addition, the Ownership Object Graph has the object merging, object pulling and type abstraction properties. Finally, there are two kinds of edges: edges between objects correspond to object creation, usage and reference relations, and edges between domains correspond to domain links. Compared to earlier definitions of object graphs [32], the Ownership Object Graph explicitly represents clusters of nodes, i.e., domains, and edges between these clusters, i.e., domain links.

2.4 Soundness

For the Ownership Object Graph to be most useful, it should be a *sound* approximation of the true runtime object graph for any possible run of the program. In this section, we only present an operational definition of the soundness of the Ownership Object Graph and leave a proof of soundness for future work.

Intuitively, soundness means that every object, domain, and edge in the runtime object graph is represented in the Ownership Object Graph. However, the Ownership Object Graph may be an approximation of the true runtime object graph, as it may represent multiple runtime objects with a single visual object, and similarly for domains and edges. The following invariants relate the Ownership Object Graph to the runtime object graph:

- **Unique Representatives:** Each object in the runtime object graph is represented by exactly one object in the visual graph. Similarly, each domain in the runtime object graph — as defined in the dynamic semantics of ownership domains [3, p. 15], is represented by exactly one domain in the visual graph;
- **Edge Soundness:** If there is a field reference from object o_1 to object o_2 in the runtime object graph, then there is a field reference edge between visual objects θ_1 and θ_2 in the visual graph, corresponding to o_1 and o_2 — similarly for domain links and edges;
- **Ownership Soundness:** If object o is in domain d in the runtime object graph, then object θ (corresponding to o) is in domain δ (corresponding to domain d) in the visual graph. Similarly, if o declares domain d in the abstract graph, then θ declares domain δ in the visual graph.

The Ownership Object Graph inherits other properties that are guaranteed by the soundness of the underlying ownership system — for example, that every object is assigned an owning domain which is consistent with all program annotations and does not change over time. These invariants are correct up to the following assumptions:

- **All Sources Available:** The program's whole source code is available, and the program operates by creating some main object and calling a method on it (this justifies the Ownership Object Graph's focus on a single root object, although multiple root objects could in principle be shown). The class of that main object is the type of the root of the Ownership Object Graph;
- **No Reflective Code:** Reflection and dynamic code loading may violate the above invariants by introducing unknown objects and edges, and possibly violating the guarantees of the underlying ownership system;
- **Flow Analysis:** Objects marked as `shared` and `unique` are not currently shown in the Ownership Object Graph. Objects that are `shared` would be trivial to add but would add many uninteresting edges to the Ownership Object Graph. Objects that are `unique` would require a flow analysis to be handled properly (See Section 3.5). Usage edges (e.g., method invocations, field accesses) could be generated for a system with only ownership, but a flow analysis is required for usage edges to be sound in the presence of `lent` objects.

Despite the assumptions about the whole program source being available and restrictions on reflection and dynamic loading, our system is still *relatively sound* in the presence of these features. In particular, as long as the reflective operations are annotated correctly and consistently with ownership information, then any object referred to by some field in the source code that is available will show up in the Ownership Object Graph, as specified above.

For edge soundness, all field references in external library code must be annotated. Since it is often not possible to annotate all such code, "virtual" [26] or "ghost" [13] fields may be declared as annotations in external files. A *virtual field* holds information that is closely related to the meaning of an object, but need not be kept directly in the object in a particular implementation [26]. These annotations do not affect the execution of the system at runtime but are treated as an object's actual fields by the analysis.

3. Analysis

At a high-level, the analysis works as follows: (1) Obtain an abstract graph from ownership domain annotations; (2) Collapse the inheritance hierarchy by copying fields into subclasses; (3) Instantiate abstractly the types in the abstract graph into objects in the visual graph, merging objects in the same domain by compatible types (two types are compatible if they have a non-trivial least upper bound); (4) Pull objects in the visual graph from formal domains to actual domains, again merging as necessary; (5) Add details to the visual graph, such as field references, domain links, etc.; and (6) Extract the display Ownership Object Graph as a depth-limited projection of the visual graph.

3.1 Data Representations

The analysis first creates from the program text an AbstractGraph and then converts it into a VisualGraph. The data type declarations of the AbstractGraph and VisualGraph are in Figure 6, and will be referred to by the metavariables shown in parentheses. To help keep the representations distinct, we use English letters ($o, d, \ldots$) for elements of the AbstractGraph, and Greek letters ($\theta, \delta, \ldots$) for elements of the VisualGraph.

The AbstractGraph consists of the AbstractTypes in the program, the AbstractDomains declared in each type, and the AbstractObjects declared in each domain. An AbstractType also lists AbstractEdges and AbstractLinks. The VisualGraph instantiates the types in the AbstractGraph and shows VisualObjects and VisualDomains: each VisualObject contains VisualDomains and each VisualDomain contains VisualObjects. The VisualGraph also has VisualEdges and VisualLinks.

The identifiers used for the elements in the AbstractGraph and VisualGraph do not correspond to the declared names of domains or objects (e.g., field or variable names) since these cannot be assumed to be globally unique, and do not take into account binding and scope. An implementation would typically have additional fields to hold the user-friendly display name. In addition, an AbstractType maintains its underlying TypeBinding to determine its sub-typing relationship with respect to other AbstractTypes.

The analysis maintains a one-to-one mapping between a VisualDomain δ and its corresponding AbstractDomain d to avoid extra copying. However, a VisualObject typically merges several AbstractObjects as discussed earlier.

3.2 Extract an AbstractGraph from Annotated Code

An AbstractGraph is obtained from the annotated program text using a visitor on the Abstract Syntax Tree of the annotated program. Most steps in Figure 7 are straightforward and are not shown in great detail. During the construction of the AbstractGraph, private ownership domains are given a *protected* semantics[1]. The default domain `owned` is considered to be declared at the first point of use and inherited thereafter. If `owned` were to be declared in `java.lang.Object`, all the objects declared in the `owned` domain would be in the same inherited domain and would get unnecessarily merged if they have the same declared type. Singleton `shared`, `lent` and `unique` AbstractDomains are created.

To simplify the treatment of inheritance when creating the VisualGraph, the AbstractGraph is post-processed by collapsing the type hierarchy, i.e., pushing field references declared in the AbstractType corresponding to a given type t into each AbstractType of the sub-types of t.

While the algorithm described in Figure 7 is presented in terms of the ownership domains type system, it can be easily applied to other ownership type systems that do not have the concept of multiple ownership domains per object and assume a single domain or "context" per object [8]. In those cases, we consider that each class implicitly declares a single ownership domain `owned` and proceed according to the algorithm. The other details of the transformation and visualization are unchanged.

[1] Domains declared in a class are inherited by its subclasses [3, *Aux-Domains rule* (Fig.14)], but are called somewhat confusingly `private`.

- AbstractGraph (g)
  - Root : AbstractObject `/* the root */`
  - Types: List<AbstractType>
- AbstractType (t)
  - TypeBinding: TypeBinding `/* Java type */`
  - Domains: List<AbstractDomain>
  - Links: List<AbstractLink>
  - Edges: List<AbstractEdge>
- AbstractDomain (d)
  - DomainType: public | private | parameter
  - Objects: List<AbstractObject>
  - DeclaringType: AbstractType
- AbstractObject (o)
  - Type: AbstractType `/* declared type */`
  - Domain: AbstractDomain `/* my owner */`
  - Bindings: List<Binding>
  - Visualized: boolean `/* bookkeeping */`
- Binding (b)
  - Formal: AbstractDomain
  - Actual: AbstractDomain
- AbstractEdge (e)
  - From: AbstractType `/* edge source */`
  - To: AbstractObject `/* edge target */`
  - EdgeType: creation | usage | reference
- AbstractLink (s)
  - From: AbstractDomain `/* link source */`
  - To: AbstractDomain `/* link target */`
- VisualGraph (γ)
  - Root: VisualObject
  - Objects: List<VisualObject>
  - Edges: List<VisualEdge>
  - Links: List<VisualLink>
- VisualObject (θ)
  - Domains: List<VisualDomain>
  - Merged: List<AbstractObject> `/* abstract objects merged into 'this' */`
  - Pulled: List<VisualObject> `/* visual objects 'this' was pulled into */`
  - IsPulled: boolean `/* bookkeeping */`
  - Parent: VisualDomain `/* my owner */`
- VisualDomain (δ)
  - Objects: List<VisualObject> `/* objects in this domain */`
  - Parents: List<VisualObject> `/* objects this domain is part of */`
  - AbstractDomain: AbstractDomain `/* map */`
- VisualEdge (η)
  - From: VisualObject `/* edge source */`
  - To: VisualObject `/* edge destination */`
  - EdgeType: creation | usage | reference
- VisualLink (σ)
  - From: VisualDomain `/* link source */`
  - To: VisualDomain `/* link destination */`

Figure 6. Data types used by AbstractGraph and VisualGraph. Some fields are for bookkeeping only.

3.3 Convert an AbstractGraph to a VisualGraph

Constructing the VisualGraph from an AbstractGraph takes into account the properties described earlier. The pseudo-code for the algorithm is presented in Figures 8, 9 and 10. The notation

```
for (T anObject : setOfObjects) ...
```

is similar to the Java 1.5 "enhanced `for`-loop" for iterating over collections and arrays. An overbar represents a sequence.

The transformation takes as input the AbstractGraph g whose root is the top-level AbstractObject o_{root}, and AbstractDomain d_{root} is the domain for o_{root}. The top-level procedure VISUALIZEGRAPH (Figure 8) first creates a top-level VisualDomain δ_{root} and then visualizes the AbstractObject o_{root}.

```
1. For each type declaration $C$ in the program
   (a) Create AbstractType $t$ and add it to $g$.*Types*
   (b) For each formal domain parameter in $C$
       i. Create corresponding AbstractDomain $d$
       ii. Add $d$ to $t$.*Domains*
   (c) For each declared ownership domain
       i. Create corresponding AbstractDomain $d$
       ii. Add $d$ to $t$.*Domains*
   (d) For each domain link between $d_1$ and $d_2$ in $C$
       i. Create AbstractLink between the AbstractDomain of $d_1$
          and the AbstractDomain of $d_2$
       ii. Add AbstractLink to $t$.*Links*
   (e) For each declaration $d\ C'<\overline{a}>\ o$ in $C$
       i. If $C'$ has no AbstractType, create $t'$ for $C'$
       ii. If AbstractType $t$ of Type $C$ has no AbstractDomain $d$,
           create $d$ and add $d$ to $t$.*Domains*
       iii. Create AbstractObject $o$ and add to $d$.*Objects*
       iv. Create bindings $\overline{b}$ from formals $\overline{f}$ of AbstractType $t'$ to
           actuals $\overline{a}$ of $t$ and add to $o$.*Bindings*
       v. If declaration is a field declaration
          A. Create AbstractEdge $e$ of type reference from Ab-
             stractType $t$ to AbstractObject $o$
          B. Add $e$ to $t$.*Edges*
2. Collapse inheritance hierarchy
   (a) Copy any public domains defined on an interface to the classes
       implementing the interface
   (b) Push field references from each super-class into its sub-classes
```

Figure 7. Obtaining the AbstractGraph.

The conversion involves two mutually recursive functions, VISUALIZEOBJECT to convert an AbstractObject into a VisualObject and VISUALIZEDOMAIN to convert an AbstractDomain into a VisualDomain. Each AbstractDomain declared in the AbstractType of an AbstractObject is visualized in turn.

Before a VisualObject θ is created for an AbstractObject o of type t inside a VisualDomain δ, the analysis calls FINDOBJECT to look for an existing VisualObject in δ with which o can be merged, i.e., if δ has a θ of type t' where t and t' have *non-trivial least upper bounds* using procedure GETLEASTUPPERBOUNDS. If such an object does not exist, a new VisualObject is created. If θ exists, then it is used and o is added to the list of AbstractObjects that are merged by θ. Each call to FINDOBJECT takes into account the AbstractTypes of all the AbstractObjects that are merged into a VisualObject.

Procedure ARENONTRIVIALTYPES excludes from the computed types any type mentioned in the list of trivial types. By default, the list includes `java.lang.Object`, `java.io.Serializable` and other user-selected types. However, a trivial type is allowed to be part of the least upper bounds, if the AbstractObject is declared of that type.

Once VisualObjects and VisualDomains have been created, procedure PULLOBJECTS uses a worklist to pull existing VisualObjects: each VisualObject is pulled from a formal to an actual domain, potentially creating a new VisualObject if it cannot be merged with an existing one. If a new AbstractObject is merged into an existing VisualObject, the VisualObject is added back to the worklist. New VisualObjects are also added to the worklist so they get pulled in turn. The analysis tracks the VisualObjects that a given VisualObject is pulled into.

Finally, the top-level procedure VISUALIZEGRAPH calls VISUALIZEFIELDREFS to add field references to the VisualGraph and VISUALIZEDOMAINLINKS to add the domain links.

When adding the field references associated with a VisualObject θ, ADDFIELDREFS (Figure 10) takes into account all the field references declared in the AbstractType of each AbstractObject merged into a VisualObject. ADDFIELDREFS also adds field references to all the pulled VisualObjects that are tracked by the bookkeeping fields.

The algorithm given in Figure 8 is sound for systems that use single inheritance and have no declared variables of a trivial type. In systems that do not meet these restrictions, the algorithm may produce multiple visual objects to represent the same runtime object. In this case, two possible approaches can be used to restore soundness. The first approach is the instantiation-based view described in Section 2 above, whereby visual objects are created for each object that is instantiated rather than for each field or variable declaration in the program.

In the second approach, the procedure FINDOBJECT in Figure 8 is modified to identify all VisualObjects that could be merged with the target TypeBindings. If there is more than one such VisualObject, the analysis unifies the VisualObjects and the resulting VisualObject has the union of the VisualDomains, merged AbstractObjects, etc. The analysis then unifies recursively all the VisualObjects that a unified VisualObject was pulled into. The FINDOBJECT procedure then returns the unified VisualObject.

3.4 Convert the VisualGraph into the Ownership Object Graph

The ownership object graph that is displayed is a depth-restricted projection of the visual graph, starting from a root object. The visualization currently uses the nested boxes discussed earlier but the algorithm is not tied to a specific graphical notation.

This step is depends on the visualization package used. In our prototype implementation, we use GraphViz [16]. Each dark grey box for each object and white-filled node for each domain must have a unique identifier — otherwise, nodes with the same identifer get unified. Since there is one VisualDomain corresponding to an AbstractDomain, and an AbstractDomain is shared across all the AbstractObject instances of a given AbstractType, each occurrence of a VisualDomain that appears in a VisualObject must be assigned a new identifier.

Because the Ownership Object Graph is a depth-limited projection, it may omit objects deeply nested in the ownership hierarchy. These objects are conceptually summarized by their containing object, and the visualization remains sound with this summarization. However, those objects may have field references to objects that are present in the projection; for soundness, the corresponding edges should be shown. In our approach, these field reference edges can be represented by summary fields in the leaf objects of the graph.

These summary fields are identified as follows. For each leaf object θ_{leaf} in the Ownership Object Graph, for each transitive child object θ_{child} of θ_{leaf}, in an *extended depth-limited projection* of the VisualGraph, we consider all actual field references from VisualObject θ_{child} to VisualObject θ_{target}, where θ_{target} is not a child of θ_{leaf}. Each such edge is represented by a summary edge from θ_{leaf} to θ_{parent}, where θ_{parent} is the nearest parent of θ_{target} that is visible in the Ownership Object Graph. This algorithm will find summary fields for all fields present at runtime as long as the *extended depth-limited projection* projects below the leaves of the graph until a cycle in the VisualGraph is reached — i.e., for each path downward from a leaf, the same VisualObject is reached a second time. This projection must still be depth-limited, as in general the VisualGraph may have an infinite depth due to reference cycles.

3.5 Limitations and Future Work

In future work, we plan on improving the precision of the analysis, proving the soundness of the Ownership Object Graph, and evaluating the scalability of the approach on large systems.

Precision. Merging objects of the same type that are in the same domain can lead to unwanted merging in some cases. Adding generic types improves the precision of the analysis, but for additional precision, an alias analysis may be needed [29].

```
Global: Map<AbstractDomain ,VisualDomain > $map$
Global: AbstractGraph $g$ (input)
Global: VisualGraph $\gamma$ (output)

VISUALIZEGRAPH()
    $\delta_{root}$ = new VisualDomain ()
    $\delta_{root}$.AbstractDomain = $d_{root}$
    $\gamma$ = new VisualGraph ()
    $\gamma$.Root = VISUALIZEOBJECT($\delta_{root}, o_{root}$)
    PULLOBJECTS()
    VISUALIZEFIELDREFS()
    VISUALIZEDOMAINLINKS()

VISUALIZEOBJECT(VisualDomain $\delta$, AbstractObject $o$)
    $\bar{t}$ = GETTYPEBINDINGS($o$.Type)
    $\theta$ = FINDOBJECT($\delta$, $\bar{t}$)
    if ( $\theta$ == NULL )
      then $\theta$ = new VisualObject ()
           $\delta$.Objects.add( $\theta$ )
           $\theta$.Parent = $\delta$
           $\gamma$.Objects.add( $\theta$ )
    $\theta$.Merged.add( $o$ )
    o.Visualized = TRUE
    for ( $d_i$ : $t$.Domains )
         do $\delta_i$ = VISUALIZEDOMAIN($\theta$, $d_i$)
            $\delta_i$.Parents.add( $\theta$ )
            $\theta$.Domains.add( $\delta_i$ )
    return $\theta$

VISUALIZEDOMAIN(AbstractDomain $d$)
    $\delta$ = $map$.get($d$)
    if ( $\delta$ == NULL )
      then $\delta$ = new VisualDomain ()
           $map$.put($d$, $\delta$)
           $\delta$.AbstractDomain = $d$
           for ( $o_i$ : $d$.Objects )
                do if ( $o_i$.Visualized )
                     then continue
                   VISUALIZEOBJECT($\delta$, $o_i$)
    return $\delta$

FINDOBJECT(VisualDomain $\delta$, List<TypeBinding> $\bar{t}$)
    for ( $\theta_i$ : $\delta$.Objects )
         do $\overline{t_m}$ = GETMERGEDTYPES($\theta_i$)
            $\bar{\ell}$ = GETLEASTUPPERBOUNDS($\overline{t_m}$, $\bar{t}$)
            if ( ARENONTRIVIALTYPES($\bar{\ell}$, $\bar{t}$) )
              then return $\theta_i$
    return NULL

GETTYPEBINDINGS(AbstractType $t$)
    ▷ Obtain list of transitive supertypes

GETLEASTUPPERBOUNDS(List $\bar{\ell}$, List $\bar{t}$)
    ▷ Compute least-upper-bounds if they exist

ARENONTRIVIALTYPES(List $\bar{\ell}$, List $\bar{t}$)
    ▷ Exclude from $\bar{\ell}$ trivial types such as java.lang.Object
    ▷ or in the user-specified list of trivial types
    ▷ EXCEPT if it is one of the declared types in $\bar{t}$
    return TRUE if remaining list of types non-empty

GETMERGEDTYPES(VisualObject $\theta$)
    List $l$ = new List()
    for ( $o_i$ : $\theta$.Merged )
         do $l$.add( $o_i$.Type )
    return $l$
```

Figure 8. Pseudo-code for creating VisualGraph.

```
PULLOBJECTS()
    Stack $worklist$ = new Stack()
    for ( $\theta$ : $\gamma$.Objects )
         do $worklist$.push($\theta$)
    while ( !$worklist$.isEmpty() )
         do VisualObject $\theta$ = $worklist$.pop()
            PULLOBJECT($\theta$, $worklist$)

PULLOBJECT(VisualObject $\theta$, Stack $worklist$)
    ▷ List.add first checks if element exists to avoid duplicates
    ▷ and returns TRUE if element is added, FALSE otherwise.
    ▷ $b_1 | = b_2$ is shorthand for $b_1 = b_1$ OR $b_2$
    $\delta_f$ = $\theta$.Parent
    $d_f$ = $\delta_f$.AbtractDomain
    for ( $d_a$ : GETACTUALS($d_f$ ) )
         do if ( $d_a$ == $d_f$ )
              then continue
            $\delta_a$ = $map$.get($d_a$)
            $\overline{t_m}$ = GETMERGEDTYPES($\theta$)
            $\theta_p$ = FINDOBJECT($\delta_a, \overline{t_m}$)
            $changed$ = FALSE
            if ( $\theta_p$ == NULL )
              then $\theta_p$ = new VisualObject ()
                   $\gamma$.Objects.add( $\theta_p$ )
                   $\theta_p$.Parent = $\delta_a$
                   $\theta_p$.IsPulled = TRUE
                   $\delta_a$.Objects.add( $\theta_p$ )
                   $changed$ = TRUE
            $\theta$.Pulled.add( $\theta_p$ )
            for ( $o$ : $\theta$.Merged )
                 do $changed \mid = \theta_p$.Merged.add( $o$ )
            ▷ Add domains from merged object
            for ( $\delta_i$ : $\theta$.Domains )
                 do $changed \mid = \theta_p$.Domains.add( $\delta_i$ )
                    $\delta_i$.Parents.add( $\theta_p$ )
            ▷ If anything changed, add back to $worklist$
            ▷ so that merged objects get pulled too...
            if ( $changed$ )
              then $worklist$.push( $\theta_p$ )

GETACTUALS(AbstractDomain $d_f$)
    List $l$ = new List()
    $\delta_f$ = $map$.get($d_f$)
    for ( $\theta_i$ : $\delta_f$.Parents ) ▷ Pull “up” only
         do for ( $o_i$ : $\theta_i$.Merged )
                 do for ( $b_i$ : $o$.Bindings )
                         do if ( $b_i$.Formal == $d_f$ )
                              then $l$.add( $b_i$.Actual )
    return $l$
```

Figure 9. Pseudo-code for creating VisualGraph (continued).

An object marked `unique` is not shown until it is assigned to a specific domain. Thus, an inter-procedural flow analysis is needed to track an object from its creation (at which point it is `unique`) until its assignment to a specific domain. In the current tool, this flow analysis is not implemented, so a `unique` object returned from a factory method must be annotated with the domain in which it should be displayed. In addition, the flow analysis can determine what domain a `lent` object is really in. A precise handling of the `lent` annotation is needed to add to the Ownership Object Graph usage edges corresponding to method invocations and field accesses since many method parameters are annotated with `lent`. Those edges are currently missing.

Scalability. Finally, we lack empirical evidence of the scalability of the approach to large systems. In the absence of semi- or fully-automated annotation inference (a separate research problem), the main difficulty would be adding the ownership domain annotations to legacy code.

```
VISUALIZEFIELDREFS()
    for ( $\theta$ : $\gamma$.Objects )
        do ADDFIELDREFS( $\theta$ )

ADDFIELDREFS(VisualObject $\theta_{src}$)
    for ( $o$ : $\theta_{src}$.Merged )
        do for ( $e$ : $o$.Type.Edges )
            do for ( $d_a$ : GETBINDINGS($o$, $e$.To.Domain ) )
                do $\delta_a$ = $map$.get($d_a$)
                   $\theta_{dst}$ = GETMERGED($\delta_a$, $e$.To )
                   if ( $\theta_{dst}$ != NULL )
                     then ADDFIELDREFS($\theta_{src}$, $\theta_{dst}$)

ADDFIELDREFS(VisualObject $\theta_{src}$, VisualObject $\theta_{dst}$)
    $\eta$ = new VisualEdge ()
    $\eta$.From = $\theta_{src}$
    $\eta$.To = $\theta_{dst}$
    if ( $\gamma$.Edges.add( $\eta$ ) )
      then for ( $\theta_{src_p}$ : $\theta_{src}$.Pulled )
            do for ( $\theta_{dst_p}$ : $\theta_{dst}$.Pulled )
                do ADDFIELDREFS($\theta_{src_p}$, $\theta_{dst_p}$)

GETBINDINGS(AbstractObject $o$, AbstractDomain $d$)
    List $l$ = new List()
    for ( $b$ : $o$.Bindings )
        do if ( $b$.Formal == $d$ )
            then $l$.add(b.Actual)
    return $l$

GETMERGED(VisualDomain $\delta$, AbstractObject $o$)
    for ( $\theta_i$ : $\delta$.Objects )
        do for ( $o_m$ : $\theta_i$.Merged )
            do if ( $o_m$ == $o$ )
                then return $\theta_i$
    return NULL

VISUALIZEDOMAINLINKS()
    for ( $t$ : $g$.Types )
        do for ( $s$ : $t$.Links )
            do VisualLink $\sigma$ = new VisualLink ()
               $\sigma$.From = $map$.get( $s$.From )
               $\sigma$.To = $map$.get( $s$.To )
               $\gamma$.Links.add( $\sigma$ )
```

Figure 10. Pseudo-code for creating VisualGraph (continued).

4. Evaluation

To evaluate our approach, we built tools and conducted two case studies on real object-oriented implementations.

4.1 Ownership Object Graph Tool

The tool obtains the Ownership Object Graph of an annotated program, represents it as a GraphViz clustered graph [16] and offers the following features:

- **Top-Level Objects:** the displayed Ownership Object Graph is a depth-limited projection of the visual graph — the depth is user-selectable but cannot be too large. The user can interactively select an object as the root of the graph to view its substructure;
- **Trivial Types:** the tool allows the user to specify an optional list of trivial types;
- **Design Intent Types:** the tool allows the user to specify an optional list of design intent types for labelling objects;
- **Object Labels:** objects can be labelled with an optional field name or variable name and an optional type name. The type used in the label consists of a least-upper-bound type or a design intent type as discussed earlier;
- **Elide Private Domains:** the tool allows the user to elide all the private domains at once and show only the public domains in the visible Ownership Object Graph;
- **User Elision:** the tool can elide temporarily uninteresting elements. When the sub-structure of an object is elided, the symbol $(+)$ is appended to its label;
- **Traceability:** the tool can show for a given visual object, the list of abstract objects and their abstract types merged into it, to help the user fine-tune the list of trivial types;
- **Navigation:** the tool supports zooming, searching by AbstractObject or AbsractType name, etc.

4.2 Case Study: JHotDraw

The subject system for the first case study is JHotDraw [20]. Version 5.3 has around 200 classes and around 15,000 lines of Java. The core types in JHotDraw were organized according to the Model-View-Controller pattern as follows:

- `Model`: consists of `Drawing`, `Figure`, etc. A `Drawing` is composed of `Figure`s which know their containing `Drawing`. A `Figure` has a list of `Handle`s to allow user interactions;
- `View`: consists of `DrawingEditor`, `DrawingView`, etc.;
- `Controller`: includes `Handle`, `Tool` and `Command`. A `Tool` is used by a `DrawingView` to manipulate a `Drawing`. A `Command` encapsulates an action to be executed.

Annotation Process. JHotDraw was annotated without making any structural refactoring such as extracting interfaces, etc. Since JHotDraw Version 5.3 did not use generic types and to improve the precision of the analysis, we used Eclipse refactorings [14] to infer the most specific generic types of containers such as `Vector` — and prevent objects of type `Vector<Handle>` and those of type `Vector<Figure>` from getting merged. The annotation process is described in detail elsewhere [1].

Ownership Object Graph. We made use of the visualization during the annotation process: for instance, visualizing the annotations encouraged us to make more use of the `owned` annotation since `owned` pushes objects down in the ownership hierarchy and avoids cluttering the top-level domains.

The list of trivial types includes interfaces implemented by many classes, e.g., `Storable`, `Animatable`, constant interfaces, e.g., `SwingConstants`[2], as well as interfaces implementing the Observer design pattern, e.g., `ViewChangeListener`. Both `Tool` and `Command` implement `ViewChangeListener` and are in the `Controller` domain, so they may get merged otherwise[3].

Evaluation. Existing compile-time analyses [40, 19] cannot produce, for a program the size of JHotDraw, a readable flat object graph that fits on one page (See other visualizations [2]). The top-level Ownership Object Graph obtained from the annotated program using our approach is shown in Figure 11 and clearly illustrates the Model-View-Controller design.

Each gray box corresponds to a "canonical object" that represents many instances at runtime and is labeled with one or more "design intent" type from the core `framework` package (variable names were not particularly informative and are not shown).

In the visualization, the `Controller` domain clearly shows `Command`, `Handle` and `Tool` instances. The self-edge on `Tool` is explained by the fact that an `UndoableTool` wraps a `Tool` and similarly, an `UndoableCommand` wraps a `Command`. The `View` domain shows instances of `DrawingEditor` (the application itself) and `DrawingView`. The `Model` domain shows instances of `Figure`:

[2] Inheriting from a constant interface to access the constants without qualifying them is a bad coding practice, the Constant Interface *antipattern* [6, Item #17] and Java 1.5 supports *static imports* to avoid it.

[3] The tool currently scans field and variable declarations and not object allocations as discussed in Section 2.

a Figure has one or more Connectors that define how to locate a "connection point".

Understanding why Drawing did not appear in the Model tier led us to discover that StandardDrawing, the base class implementing the Drawing interface, extends CompositeFigure, thus a Drawing *is-a* Figure[4]. Although this is not a design problem *per se*, it is inconsistent with the design intent in the core framework package: there, interface Drawing does not extend interface Figure. This finding was unexpected in a framework as carefully designed and as widely studied as JHotDraw. Although a class diagram could reveal that a StandardDrawing is a Figure, the Ownership Object Graph quickly pinpoints that.

The top-level domains have only 28 objects even though JHotDraw has 200 around types and presumably each type is instantiated at least once. This illustrates how the properties of the Ownership Object Graph provide more abstraction and more design intent than a visualization of the raw object graph [19, 40].

In fact, designers often employ similar techniques in a design-oriented class diagram, i.e., one not retrieved from an implementation using a tool: a) *merge interface and abstract implementation class* — although important for code reuse, such a code factoring is often unimportant from a design standpoint; and b) *subsume a set of similar classes under a smaller set of representative classes* — showing many similar subclasses that vary only in minor aspects on a class diagram often leads to needless clutter [36, pp. 139–140]. It seems the JHotDraw designers used similar techniques to present the JHotDraw design in their tutorials [36].

In the Ownership Object Graph, all runtime figure objects referenced in the program by the Figure interface, its abstract implementation class AbstractFigure, or any of its concrete subclasses DecoratorFigure, ConnectionFigure, etc., appear as a single Figure object in the Model domain.

The distinction between public and private domains within each object enables eliding all the private domains at once to show only the top-level Model, View and Controller domains in object Main. To illustrate the hierarchy however, objects were selected individually and their internals were elided — those have the symbol $(+)$ appended to their labels. DrawingEditor shows its internals: its private owned domain has an Iconkit object among others, and IconKit has its own substructure, but the latter is elided.

Currently, the visualization does not show multiplicities: at runtime, there is one DrawingEditor (the application itself), one IconKit, but one or more DrawingView objects.

4.3 Case Study: HillClimber

By many accounts, JHotDraw is considered the brainchild of experts in object-oriented design and programming. In comparison, the subject system for this case study, HillClimber, is another 15,000 line application that was mainly developed and maintained by undergraduates.

In HillClimber, the application *window* uses a *canvas* to display *nodes* and *edges* of a *graph* in order to demonstrate algorithms for constraint satisfaction problems provided by the *engine*.

Annotation Process. HillClimber was organized into a data ownership domain to store the graph, a ui domain to hold the user interface elements, and a logic domain to hold the engine, search objects, and associated objects. Unlike JHotDraw, adding annotations to HillClimber involved refactoring to decouple the code. Again, to increase the precision of the analysis, we refactored the code to use generics, mostly automatically using Eclipse. However, Eclipse cannot infer the generic type of a variable of type Vector storing arrays of Node objects: such code was manually refactored to use Vector<Vector<Node>>. The annotation process is described in detail elsewhere [1].

Evaluation. The Ownership Object Graph in Figure 12 shows clearly the core HillClimber top-level objects, window, canvas, engine and graph. Similarly, the Search object in the logicTier domain merges many instances of sub-classes of class Search such as MCHSearch, RandSearch, etc.

The Graph base class declares a nodes:Vector<Node> field and its subclass HillGraph refers to that same object. Generic types improved the precision of the analysis and prevented the merging of edges:Vector<Edge> and nodes:Vector<Node>. The graph:Graph object merges both Graph and HillGraph and shows objects nodes and edges in its owned domain.

Since a domain is introduced where it is declared and then is inherited according to the protected semantics, HillGraph and Graph share the same owned domain. However, when two "unrelated" objects, e.g., a Button object and a Panel object get merged (since they have a non-trivial least upper bound) and each has its declared owned domain, it is possible to have multiple domains of the same name in a given visual object — in that case, a domain name is fully qualified with the type name where it was declared in the abstract graph.

The visualization highlights the need to potentially make object edgesIn, the incident edges on a node, encapsulated inside object node:Entity. This would require changing the annotations and the code as necessary to abide by the rules of the type system. This in turn would push the object down the ownership tree and remove it from the top-level domain.

The mediator:ICanvasMediator object was introduced during a refactoring to decouple the code [1] and mediate between the graph and the canvas. Finally, the object labeled window:Frame merges several user interface objects representing dialogs, etc., thus illustrating the type abstraction property.

5. Related Work

Program Visualization. There is a large body of software visualization research where the emphasis is on novel kinds of visualization using colors, shapes, 3D, etc. Our contribution in this paper is not the visualization *per se* — we're using the simple but effective GraphViz package — it is in having developer-specified ownership annotations drive a sound compile-time visualization of the program's execution structure.

Many dynamic analyses visualize the execution structure but ignore ownership: they instrument the running program, filter the program traces based on various query criteria and then visualize the summarized information in novel ways, often with a granularity not larger than an object or a class [23, 37, 35, 17, 39, 30, 10]. On the other hand, such analyses handle programs for which source code is not available, do not require source code annotations or changes to the source code to add the annotations and allow more fine-grained user interaction in producing the visualization.

Ownership Annotation Inference. Annotation inference is an active area of research using both static [4, 9, 24, 25] and dynamic [41] analyses. However, a fully automated inference cannot create multiple public domains in one object and meaningful domain parameters to represent the design intent, such as the separate Model, View, and Controller in the JHotDraw case study. Existing inference algorithms produce for each class a long list of domain parameters, often place each field in a separate domain, or annotate many objects with shared or lent [4].

Dynamic Object Graph Analyses. Dynamic analyses can infer the ownership structure of a running program based on its heap structure. Although these techniques have the advantage of not requiring abundant source code annotations, they can only infer the equivalent of owned, shared, lent and unique annotations. This

[4] According to the Release Notes for JHotDraw Version 5.1, this change was made to support inserting a Drawing as a Figure inside another Drawing.

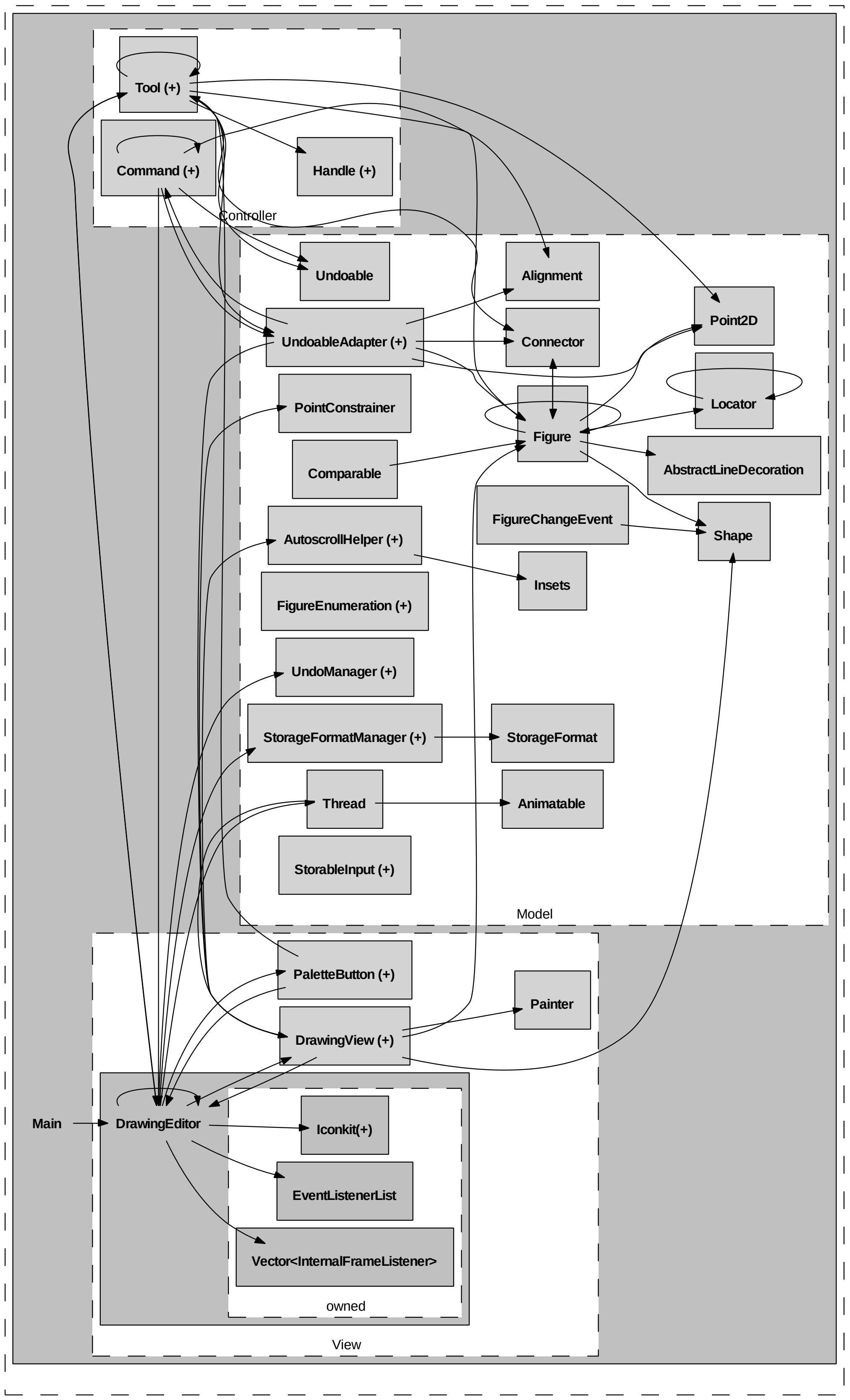


Figure 11. Top-level Ownership Object Graph for JHotDraw. This graph was laid out automatically by GraphViz without user intervention. The edges correspond to field references.

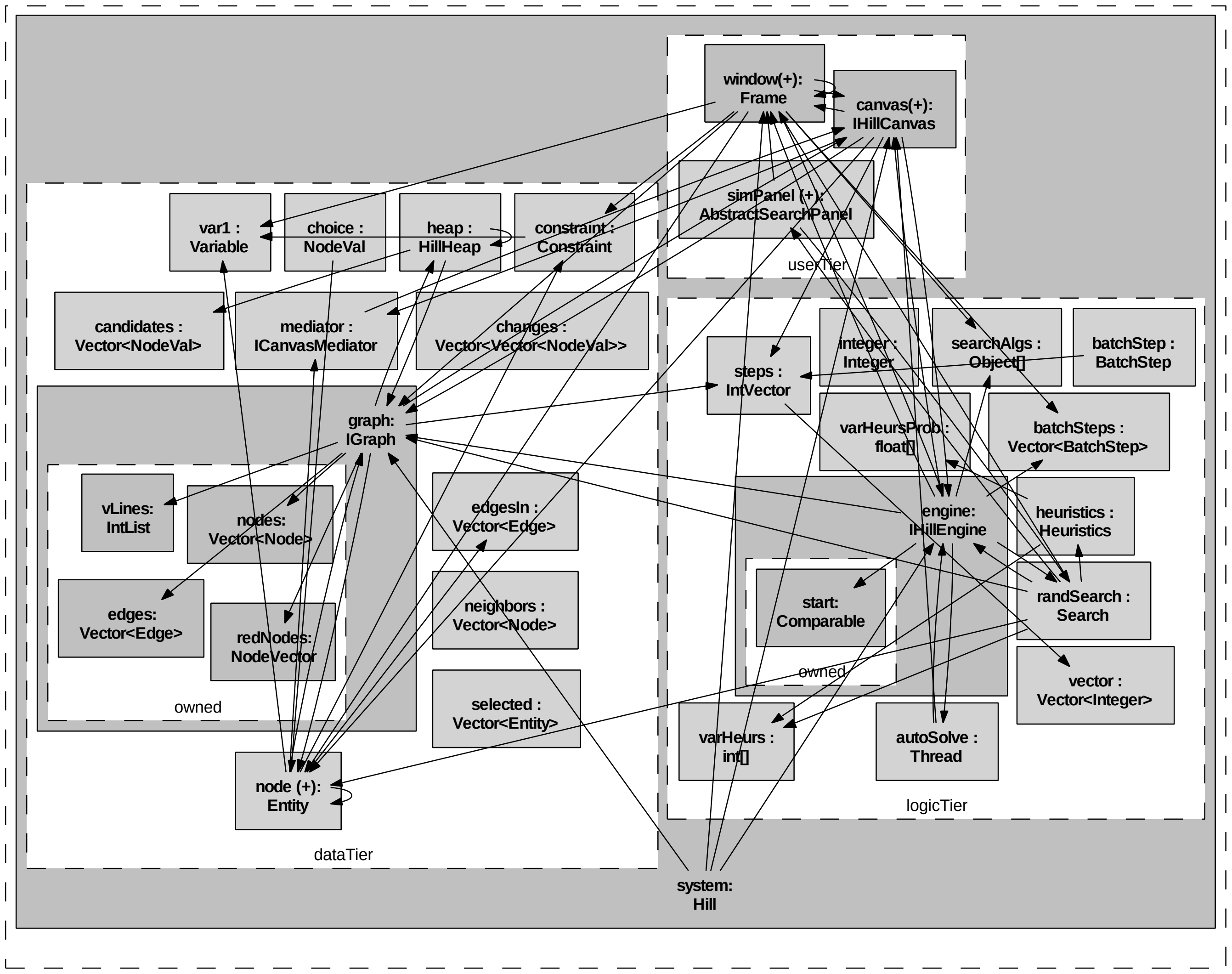


Figure 12. Ownership Object Graph for HillClimber, laid out automatically by GraphViz `dot` without user intervention.

assumes a strict owner-as-dominator hierarchy which is not flexible enough to represent design patterns such as the Composite pattern.

Rayside et al. [33] characterize sharing and ownership and produce a matrix display of the ownership structure. Similarly, Mitchell [27] uses lightweight ownership inference to examine a single heap snapshot rather the entire program execution, and scales the approach to large programs through extensive graph transformation and summarization. Flanagan and Freund [12] proposed a dynamic analysis to reconstruct each intermediate heap from a log of object allocations and field writes, then apply a sequence of abstraction-based operations to each heap, and combine the results into a single object model that conservatively approximates all observed heaps from the programs execution. Their tool, AARDVARK, has the notion of ownership and containment and uses simple heuristics to choose the most appropriate generalization. Noble et al. [18, 28] and Potanin et al. [31] also process heap snapshots and show both matrix and graph visualizations of ownership trees, indicating an object's "aliasing shadow" and "interior".

There are several problems with dynamic analyses: first, runtime heap information does not convey design intent. Second, a dynamic analysis may not be repeatable, i.e., changing the inputs or executing different use cases might produce different results. Compared to dynamic ownership analyses — which are descriptive and show the ownership structure in a single run of a program, the Ownership Object Graph obtained at compile time is prescriptive and shows ownership relations that will be invariant over all program runs. Third, a dynamic analysis cannot be used on an incomplete program still under development or to analyze a framework separately from a specific instantiation. Finally, some dynamic analyses carry a significant runtime overhead — a 10X-50X slowdown in one case [12], which must be incurred each time the analysis is run, whereas the main cost of adding annotations is incurred once.

Static Object Graph Analyses. Several static analyses produce various object graphs, but they do not use ownership and do not convey design intent. PANGEA [40] produces a flat object graph. WOMBLE [19] uses syntactic heuristics and hard-coded heuristics for container classes to obtain an object model including multiplicities, but its analysis does not attempt to be sound and the flat object graph it produces does not scale to large programs: in particular, the WOMBLE visualization of the 15,000-line JHotDraw does not fit on one readable page [2] nor does it convey the Model-View-Controller design.

AJAX [29] uses an alias analysis to build a refined object model as a conservative compile-time approximation of the heap graph reachable from a given set of root objects, and simplifies it through a series of transformations. However, AJAX does not use ownership and produces flat object graphs. Although AJAX has been evaluated on a system with as many as 36,000 lines of code, the object graphs it produces are manually post-processed to become readable, and its heavyweight analysis does not scale to much larger programs.

Lam and Rinard [22] proposed a type system for describing and enforcing design: developer-specified annotations guide the abstraction by merging objects with *tokens* and merging methods with *subsystems*, and are used to produce a flat object graph, that was evaluated on a 1,700-line program. However, the tokens and subsystems are statically fixed (unlike domains, all instances of a class use the same tokens declared in the class), so they do not model runtime hierarchy, do not describe data sharing as precisely as ownership domains, and do not handle inheritance. In contrast, our approach does not require additional annotations just to obtain a visualization: ownership annotations are useful in their own right, as demonstrated by the extensive research into ownership types [8, 7, 4, 3, 11]. Finally, our approach handles inheritance.

Rayside et al. had proposed earlier a static object graph analysis based on Bacon and Sweeney's Rapid Type Analysis (RTA) [5] but indicated that it produced unacceptable over-approximations for most non-trivial programs [34].

6. Conclusion

Ownership domain annotations with meaningful domain names add hierarchy to a flat object graph, precision about inter-domain aliasing, convey design intent, and enable an instance-based hierarchical visualization of the execution structure of a system, to complement views of the code structure provided by existing approaches.

Evaluating the approach on two previously annotated Java programs consisting of 15,000 lines of code each produced in both cases a visualization that fits on one page and conveys the complex design intent better than existing compile-time approaches that do not rely on ownership annotations.

Acknowledgments

This work was supported in part by NSF grant CCF-0546550, DARPA contract HR00110710019, the Department of Defense, and the Software Industry Center at Carnegie Mellon University and its sponsors, especially the Alfred P. Sloan Foundation.

References

[1] M. Abi-Antoun and J. Aldrich. Ownership Domains in the Real World. In *ECOOP IWACO*, 2007.

[2] M. Abi-Antoun and J. Aldrich. Ownership Object Graph Case Studies. `http://www.cs.cmu.edu/~mabianto/oog/`, 2007.

[3] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.

[4] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, 2002.

[5] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA*, 1996.

[6] J. Bloch. *Effective Java*. Addison-Wesley, 2001.

[7] C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In *POPL*, 2003.

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.

[9] W. Cooper. Interactive Ownership Type Inference. Senior Thesis, Carnegie Mellon University, 2005.

[10] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the Execution of Java Programs. In *Lectures on Software Visualization*, 2002.

[11] W. Dietl and P. Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8), 2005.

[12] C. Flanagan and S. N. Freund. Dynamic Architecture Extraction. In *Workshop on Formal Approaches to Testing and Runtime Verification*, 2006.

[13] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, 2002.

[14] R. M. Fuhrer, F. Tip, A. Kiezun, J. Dolby, and M. Keller. Efficiently Refactoring Java Applications to Use Generic Libraries. In *ECOOP*, 2005.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[16] E. R. Gansner and S. C. North. An Open Graph Visualization System and its Applications to Software Engineering. *Software Practice & Experience*, 30(11), 2000.

[17] J. Gargiulo and S. Mancoridis. Gadget: A Tool for Extracting the Dynamic Structure of Java Programs. 2001.

[18] T. Hill, J. Noble, and J. Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *Journal of Visual Languages and Computing*, 13(3), 2002.

[19] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *IEEE Trans. on Softw. Eng.*, 27(2), 2001.

[20] JHotDraw. `http://www.jhotdraw.org/`, 1996.

[21] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zundorf. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *WCRE*, 2002.

[22] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, 2003.

[23] D. B. Lange and Y. Nakamura. Interactive Visualization of Design Patterns Can Help in Framework Understanding. In *OOPSLA*, 1995.

[24] Y. Liu and A. Milanova. Ownership and Immutability Inference for UML-based Object Access Control. In *ICSE*, 2007.

[25] K.-K. Ma and J. S. Foster. Inferring Aliasing and Encapsulation Properties for Java. In *OOPSLA*, 2007. To appear.

[26] S. McCamant and M. D. Ernst. Early Identification of Incompatibilities in Multi-Component Upgrades. In *ECOOP*, 2004.

[27] N. Mitchell. The Runtime Structure of Object Ownership. In *ECOOP*, 2006.

[28] J. Noble. Visualising Objects: Abstraction, Encapsulation, Aliasing, and Ownership. In *Lectures on Software Visualization*, 2002.

[29] R. W. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, 2001.

[30] R. Oechsle and T. Schmitt. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams using the Java Debug Interface (JDI). In *Lectures on Software Visualization*, 2002.

[31] A. Potanin, J. Noble, and R. Biddle. Checking Ownership and Confinement. *Concurrency & Comput.: Pract. & Exp.*, 16(7), 2004.

[32] J. Potter, J. Noble, and D. Clarke. The Ins and Outs of Objects. In *Australian Software Engineering Conference*, 1998.

[33] D. Rayside, L. Mendel, and D. Jackson. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *WODA*, 2006.

[34] D. Rayside, L. Mendel, R. Seater, and D. Jackson. An Analysis and Visualization for Revealing Object Sharing. In *Eclipse Technology eXchange (ETX)*, 2005.

[35] T. Richner and S. Ducasse. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In *ICSM*, 1999.

[36] D. Riehle. *Framework Design: a Role Modeling Approach*. PhD thesis, Federal Institute of Technology Zurich, 2000.

[37] M. Sefika, A. Sane, and R. H. Campbell. Architecture-Oriented Visualization. In *OOPSLA*, 1996.

[38] F. Shull, F. Lanubile, and V. R. Basili. Investigating Reading Techniques for Object-Oriented Framework Learning. *IEEE Trans. on Softw. Eng.*, 26(11), 2000.

[39] M. P. Smith and M. Munro. Runtime Visualisation of Object-Oriented Software. In *VISSOFT*, 2002.

[40] A. Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FU Berlin, 2002.

[41] A. Wren. Ownership Type Inference. Master's thesis, Department of Computing, Imperial College, 2003.

Ownership Domains in the Real World

Marwan Abi-Antoun
School of Computer Science
Carnegie Mellon University
marwan.abi-antoun@cs.cmu.edu

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
jonathan.aldrich@cs.cmu.edu

Abstract

The Ownership Domains type system has had publicly available tool support for a few years. However, the previous implementation used non-backwards compatible language extensions to Java and ran on a research infrastructure, which made it difficult to conduct substantial case studies on interesting systems.

We first present a re-implementation of ownership domains using Java 1.5 annotations and the Eclipse infrastructure. We then use the improved tool to annotate two real 15,000-line Java programs while using refactoring tool support, generics and external libraries.

Ownership domains, as most other ownership type systems, provide useful encapsulation properties. We illustrate using actual examples from the subject systems how ownership domains also express and enforce design intent regarding object encapsulation and communication and help identify tight coupling. Finally, we mention some expressiveness gaps that we encountered.

1. Introduction

Researchers have proposed many ownership type systems, e.g., [15, 10, 3, 17, 41], but have not reported significant experience with most of them on real code. Only a few systems, notably Ownership Domains [6, 3], Universes [17] and Generic Ownership [41], have released tool support [46, 20, 40], and even fewer systems have been evaluated in substantial case studies [6, 25, 2, 38].

The previous implementation of ownership domains [3] used non-backwards compatible extensions of Java [46]. As a result, none of the rich tool support for Java programs was available to programs with ownership domain annotations[1].

In a previous case study [2], we identified that adding ownership domain annotations to existing code often highlights refactoring opportunities. For instance, a lengthy domain parameter list is often an indication of tightly coupled code that could benefit from refactoring — such as extracting an interface and programming to that interface. It is unrealistic to assume that it is possible to refactor all such code prior to annotating it. In our experience, having access to refactoring tool support during the annotation process was invaluable. Using language extensions also makes it harder to partially and incrementally annotate existing code and thus conduct case studies on interesting systems. Finally, the previous tool used a modified research infrastructure [8] that is no longer actively maintained and does not support Java generics — as of this writing.

To address these adoptability challenges, we re-implemented the Ownership Domains type system using the annotation facility in Java 1.5 [27], so that Java programs with ownership annotations remain legal Java 1.5 programs. We also implemented the tool as a plugin to the Eclipse open source development environment that has become popular with researchers and practitioners [24, 37].

[1] The Universes tools built on the Java Modeling Language (JML) infrastructure support both language extensions and stylized comments [20].

We believe this improved tool support promotes the adoptability of the ownership domains technique by Java developers as follows. First, all the Eclipse tool support such as syntax highlighting, refactoring, etc., remains available to annotated programs. Second, using annotations makes it easier to support in a non-breaking way additional annotations such as external uniqueness [14] or `readonly` [17]. Third, using annotations gives the ability to incrementally and partially specify annotations on large code bases. Fourth, using annotations will make it possible to study the evolution of programs with ownership annotations, an area that has not received much attention — since no one will maintain a program with limited tool support. Finally, annotating existing code is difficult and time-consuming and tools are being developed to add annotations semi-automatically [6, 16]. One of the benefits of using annotations over language extensions is that an inference algorithm cannot break an existing program by inserting potentially incorrect annotations.

We made the following design choices for the annotation system. First, we worked within the limits of Java 1.5 annotations [27], even though annotations may be more verbose than an elegantly designed language. Moreover, Java 1.5 annotations impose several restrictions, e.g., no annotations on generic type arguments. Other researchers have tried to eliminate some of these restrictions by proposing revisions of the language [19], but until such proposals are officially adopted, their prototype implementations are not Eclipse compatible, an important factor for adoptability. Second, to work around the Java 1.5 limitation of allowing annotations only on declarations, we consistently declare additional temporary variables and add annotations to them. This has worked well for new expressions, cast expressions (both implicit and explicit) and arguments for method and constructors. Third, checking ownership domain annotations only generates informational messages, i.e., no errors or warnings, and does not stop a developer from running the program. Fourth, we hard-code a minimal number of implicit defaults and provide a separate tool to supply explicit reasonable defaults to reduce the annotation burden. In the future, this tool can be replaced with a smarter annotation inference tool. Finally, the annotations are non-executable and do not affect the program's behavior [2]; unlike the earlier implementation, the current system does not include runtime checks. As a result, the annotation-based system is unsound at casts — but could be made sound using bytecode rewriting to add necessary dynamic checks.

The rest of the paper is organized as follows: we review ownership domains in Section 2, describe the annotation language in Section 3 and the salient tool features in Section 4. We discuss two case studies in Section 5 and show how ownership domains express and enforce design intent related to object communication and encapsulation. We discuss some expressiveness gaps that we encountered in Section 6 and conclude with related work in Section 7.

[2] Annotations may increase the memory footprint and slow down class loading as a result, but no empirical data has been reported to date.

2. Review of Ownership Domains

Ownership domains are conceptual groups of objects with explicit domain names and explicit policies that govern references between them. Each object belongs to a single ownership domain, and a top-level domain is assumed.

Public and Private Domains. Each object can declare one or more *public* or *private* domains to hold its internal objects, thus supporting hierarchy. A public domain is accessed using a syntax similar to field access. Domain declarations are added to a class, but for each instance of that class, fresh instances of these domains are created for that object, i.e., `obj1.DomainA` and `obj2.DomainA` are distinct if `obj1` and `obj2` are instances of the same class `T` and do not alias each other.

Explicit Domain Links. Each object can declare a policy describing the permitted aliasing among objects in its public domains, and between its private domains and public domains. Ownership domains support two kinds of policy specifications: a) a link from one domain to another allows objects in the first domain to access objects in the second domain; and b) permission to access an object implies permission to access its public domains. In addition to explicit domain links, the following implicit policy specifications are included: a) an object has permission to access other objects in the same domain; and b) an object has permission to access objects in any domain that it declares. Any reference not explicitly permitted by these rules is prohibited, and link permissions are not transitive.

Ownership Domain Parameters. Two objects can access objects in the same domain, as long as implicit or explicit permissions allow that access, by declaring a formal domain parameter on one object, and binding that formal domain parameter to another domain. Method domain parameters are also supported and are often needed for static methods.

Alias Types. In addition, the following special annotations are defined for increased expressiveness [3]:

- `unique`: indicates an object to which there is only one reference such as a newly created object. Unique objects can be passed linearly from one object to another, by destroying the old reference to the object when the new reference is created;
- `lent`: one ownership domain can temporarily lend an object to another ownership domain, with the constraint that the second ownership domain will not create any persistent references to that object: e.g., a method formal parameter is often annotated with `lent` to indicate that it is a temporary alias;
- `shared`: indicates that an object may be aliased globally. `shared` references may not alias non-`shared` references.

Unlike *owner-as-dominator* type systems [15], public domains in the ownership domains type system can express constructs such as iterators [3] (See Figure 1) or an instance of the Composite design pattern [22, p. 163] that does not encapsulate its subcomponents and gives clients the ability to add components to any composite of the hierarchy and not only to the root composite [30]. Developers can still express owner-as-dominator in ownership domains by: a) never declaring a public domain; and b) never linking a domain parameter to an internal domain [3].

3. Annotation Design

In this section, we describe the concrete annotation syntax. For maximum flexibility and to work around some of the limitations of Java 1.5 annotations, all annotation values are strings. Annotations that are plural take values that are arrays of strings.

The annotations are illustrated using snippets from a canonical `Sequence` abstract data type, a common benchmark for ownership type systems. Within the `Sequence`, the `iters` ownership domain is used to hold `Iterator` objects that clients use to traverse the `Sequence`, and the default *private* `owned` ownership domain is used to hold the `Cons` cells in the linked list that is used to represent the `Sequence`. The full example is shown in Figure 1.

@Domains: declare public or private domains on a type.

- **Format**: *identifier*
- **Applies to**: type (class or interface).
- **Examples**: the following declares a private `owned` domain (`owned` is private by naming convention), and a public domain `iters` to store the `Iterator` objects of the `Sequence`.

```
@Domains({"owned","iters"})
class Sequence<T> {
...
}
```

@DomainParams: declare ordered domain parameters on a type or method domain parameters on a method.

- **Format**: *identifier*
- **Applies to**: type or method.
- **Examples**: `Sequence` declares a domain parameter `Towner` to hold its elements.

```
@DomainParams({"Towner"})
class Sequence<T> {
...
}
```

@DomainInherits: pass parameters to superclass or implemented interfaces.

- **Format**: $typename < parameter, \ldots >$
- **Applies to**: type (class or interface).
- **Examples**: the `Iterator` interface is also parameterized by the `Towner` domain parameter. Class `SeqIterator` inherits domain parameter `Towner` from interface `Iterator`, and adds the `list` parameter to access the `Cons` cells.

```
@DomainParams({"list", "Towner"})
@DomainInherits({"Iterator <Towner>"})
class SeqIterator<T> implements Iterator<T> {
...
}
```

@DomainLinks: declare domain links.

- **Format**: $fromDomainId$ `->` $toDomainId$
- **Applies to**: type (class or interface).
- **Examples**: the `Sequence` gives `Iterator` objects in the `iters` domain permission to access objects in the `owned` domain, including the `Cons` cells.

```
@DomainLinks({..., "iters -> owned", ...})
class Sequence<T> {
...
}
```

@DomainAssumes: declare domain link assumptions.

- **Format**: $fromDomainId$ `->` $toDomainIds$
- **Applies to**: type (class or interface).
- **Examples**: the `Sequence` assumes that the `owner` of the `Sequence` has access to the `Towner` domain containing the sequence elements.

```
@DomainAssumes("owner -> Towner") /* default */
class Sequence<T> {
...
}
```

@Domain: declare the domain, actual parameters and actual array parameters.

- **Format**: `annotation<domParams,...>[arrayParams,...]`
  - `annotation`: indicate a domain name (e.g., `owned`), one of the special alias types (e.g., `unique`), or a public domain of an object using a field access syntax (e.g., `seq.iters`);
  - `<domParams,...>`: specify actual domain parameters by order of formal domain parameters, at object creation and access sites;

- `[arrayParams,...]`: in ownership domains, arrays have two ownership modifiers, one for the array object itself and one for the objects stored in the array. For variables of array type, this argument specifies the actual array parameters by order of array dimension (for multi-dimensional arrays).
- **Applies to**: local variable declaration, field declaration, method formal parameter and method return value.
- **Examples**: the following declares a `unique Iterator` object and binds the `list` domain parameter on `SeqIterator` to `owned` domain on `Sequence`, and the `Towner` domain parameter on `SeqIterator` to the parameter by the same name on `Sequence`.

```
@Domain("unique<owned,Towner>")
SeqIterator<T> it = new SeqIterator<T>(head);
```

- **Examples**: a `lent` array of `shared Strings`:

```
@Domain("lent[shared]")String args[];
```

@DomainReceiver: declare the domain of the receiver of a constructor or a method.

- **Format**: *identifier*
- **Applies to**: constructor or method.
- **Examples**:

```
@DomainReceiver("state")
void run() { ... }
```

4. Tool Design and Implementation

Ownership domain annotations are typechecked using two visitors on the Eclipse Abstract Syntax Tree (AST).

4.1 Ownership Domains Typechecking

A first-pass visitor performs the following:

- **Identify Problematic Patterns:** these will need to be replaced with equivalent constructs, e.g., by declaring a local variable and adding the appropriate annotations to it;[3]
- **Read Annotations from AST:** the Java 1.5 annotations added to a program are part of the AST. The visitor locates the annotations nodes in the AST and parses their contents using a JavaCC [26] parser. The visitor also locates special block comments on method invocation expressions as described later. In addition, the visitor infers default annotations for some AST nodes that cannot be annotated, e.g., it implicitly defaults the `NullLiteral` AST node to `unique`. The visitor maps each AST node to an annotation structure in preparation for the second pass visitor which will typecheck the annotations;
- **Propagate Local Annotations:** the visitor propagates the explicit annotations from the AST nodes (for types, variables, and methods) to all the expression nodes in the AST, including translating formals to actuals.

A second-pass visitor checks the annotations on each expression based on the static semantics of Ownership Domains. Checking the assignment rule requires a value flow analysis. A Live Variables Analysis (LVA) from a lightweight data flow analysis framework [5] — that also uses the Eclipse AST, is invoked intra-procedurally at each method boundary using a separate visitor. The LVA analysis verifies that a `unique` pointer only has one non-`lent` read.

4.2 Additional Features

The tool offers the following additional features:

[3] Using the Eclipse built-in refactoring ("Extract Local Variable"), this operation can be performed with very little effort.

```
@Domains({"owned","iters"})
@DomainParams({"Towner"})
@DomainAssumes("owner -> Towner")
@DomainLinks({"owned->Towner", "iters->Towner",
              "iters->owned"})
class Sequence<T> {
  @Domain("owned<Towner>") Cons<T> head;
  void add(@Domain("Towner")T o) {
    @Domain("owned<Towner>")
    Cons<T> cons = new Cons<T>(o,head);
    head = cons;
  }
  @Domain("iters<Towner>") Iterator<T> getIter() {
    @Domain("iters<owned, Towner>")
    SeqIterator<T> it = new SeqIterator<T>(head);
    return it;
  }
}

@DomainParams({"Towner"})
@DomainAssumes("owner -> Towner")
class Cons<T> {
 @Domain("Towner") T obj;
 @Domain("owner<Towner>")Cons<T>  next;

 Cons(@Domain("Towner")T obj,
      @Domain("owner<Towner>")Cons<T> next) {
   this.obj=obj;
   this.next=next;
 }
}

@DomainParams({"Towner"})
interface Iterator<T> {
  @Domain("Towner")T next();
  boolean hasNext();
}

@DomainParams({"list", "Towner"})
@DomainAssumes({"list -> Towner"})
@DomainInherits({"Iterator<Towner>"})
class SeqIterator<T> implements Iterator<T> {
  @Domain("list<Towner>")Cons<T> current;
  ...
  SeqIterator(@Domain("list<Towner>")Cons<T> head) {
    current = head;
  }
  public @Domain("Towner") T next() {
    @Domain("Towner")T obj2 = current.obj;
     current = current.next;
     return obj2;
   }
}

@Domains({"owned","state"})
class SequenceClient {
  final @Domain("owned<state>")
        Sequence<Integer> seq = new Sequence<Integer>();

  void run() {
    @Domain("state")Integer int5 = new Integer(5);
    seq.add(int5);
    @Domain("seq.iters<state>")
    Iterator<Integer> it = this.seq.getIter();
    while (it.hasNext()) {
      @Domain("state")Integer cur = it.next();
      ...
    }
  }
  ...
}
```

Figure 1. A `Sequence` Abstract Data Type with ownership domain annotations.

```
@DomainParams({"state"})
class Student {
...
}
@DomainParams({"state"})
class Data ... {
  final @Domain("state<state<state>>")
  Sequence<Student> vStudent;

  @Domain("state<state>")Student
  getStudentRecord(@Domain("shared")String sSID) {
    @Domain("vStudent.iters<state<state>>")
    Iterator<Student> i = vStudent.getIter();
    while (i.hasNext()) {
      @Domain("state<state>")
      Student objStudent = i.next();
      ...
      }
    ...
    }
  }
```

Figure 2. Adding annotations to generic code.

External Libraries. There are two approaches to support adding annotations to the standard Java libraries and other third-party libraries, one that involves annotating the library and pointing the tool to the annotated library and one that involves placing the annotations in external files. The earlier tool used the former approach [46], but we adopted the latter approach this time since it does not require changing library or third-party code — which may not be available and when it is, tends to evolve separately. Other annotation based systems adopted the same strategy [42]. The tool supports associating ownership domain annotations with any Java bytecode `.class` file using an external XML file, following the same annotation constructs described in Section 3.

Generics. Our annotation system currently treat generic types as orthogonal to ownership domain parameters, so generic type parameters and arguments are added separately from ownership domain annotations — except that nested actual domains may need to be provided where applicable. Proponents of Generic Ownership [41] argue that this leads to awkward syntax, which may be true. However, in our case studies annotating two 15,000-line Java programs including using generic types, we did not observe this to be a serious problem. Figure 2 illustrates the interaction between generics and ownership domains: the `Student` class is parameterized by the `state` domain parameter. The `Data` class maintains a `Sequence` of `Student` objects and is also parameterized by `state`.

Method Domain Parameters. Java 1.5 annotations cannot be added at method invocation expressions. So we used block comments to specify the actual domains for a parameterized method (See Figure 3 for an example). Unfortunately, even proposals to improve the Java 1.5 annotation facilities [19] do not yet address adding annotations to such expressions.

Defaulting Tool. To reduce the annotation burden, we implemented a separate tool to add default annotations such as marking private fields as `owned`, method parameters as `lent`, and `Strings` as `shared`. However, an annotation added by the defaulting tool (e.g., `owned`) may need to be modified manually to supply actual domains for domain parameters (e.g., `owned<owned>`).

Annotation 'owner'. We also added the special `owner` annotation, similar to `peer` in Universes [17]. Using `owner` can often eliminate a domain parameter: e.g., in Figure 1, `Cons.owner` is `Sequence`, `SeqIterator.owner` is `Sequence`.

4.3 Tool Limitations and Future Work

Java 1.5 annotations suffer from the following limitations: (1) A declaration cannot have multiple annotations of the same annotation type; (2) Annotation types cannot have members of the their

```
class Sequence<T> {
...
  @DomainParams("TTowner") /* Method domain parameter */
  @Domain("shared") /* Domain for return value */
  static <TT> String
    toString(@Domain("lent<TTowner>")Sequence<TT> seq) {
    ...
  }
  void dump() {
    @Domain("owned<shared>")
    Sequence<String> seq = ...;

    @Domain("shared")
    /* Provide <actuals...> using block comment */
    String str = Sequence.toString/*<state>*/(seq);
  }
}
```

Figure 3. Declaring and binding method domain parameters.

```
while (objCourseFile.ready()) {
  this.vCourse.add(new Course(courseFile.readLine()));
}
/*  ABOVE MUST BE REWRITTEN AS  ....  */
while (objCourseFile.ready()) {
 @Domain("shared")String line = courseFile.readLine();
 @Domain("state<state>")Course crs = new Course(line);
 this.vCourse.add(crs);
}
```

Figure 4. Re-writing a new expression using local variables.

own type; (3) It is only legal to use single-member annotations for annotation types with multiple members, as long as one member is named `value`, and all other members have default values. Otherwise, the more verbose syntax is required, e.g., `@Name(first = "Joe", last = "Hacker")`; (4) Annotation types cannot extend any entity (class, interface or annotation); and (5) Annotations are allowed on type, field, variable and method declarations and not allowed on type parameters or method invocations.

The fist restriction prevented us from using the `@Domain` annotation to specify both the annotation on the receiver and on the return type of a method. The second restriction prevented us from having shorthand constant annotations for the special alias types, e.g., `@owned` instead of `@Domain("owned")`: such constants cannot be used inside other annotations as in `@Domain(annotation = @owned, parameters = {@owned})`.

To avoid having multiple ways of indicating the same meaning, we use strings for all the annotations and require annotations of the form `@Domain("owned<owned>")`. Although developers may be more likely to introduce spelling mistakes in string annotations, the typechecker will catch these problems early enough. The third restriction, i.e., the lack of positional arguments, required the use of the verbose syntax `@Domains(publicDomains = {"d1", "d2"}, privateDomains = {"pda", "pdb"})`.

The final restriction and the current lack of annotation inference require converting some expressions to more verbose constructs by declaring local variables and annotating them. The most common such expressions were new expressions (See Figure 4) and cast expressions (See Figure 5).

We plan to address some of the following limitations:

- **Infer method domains:** just as actual type arguments do not have to be passed to a generic method in Java, it may be possible to infer, in most cases, the actuals for method domain parameters based on the types of the actual arguments;
- **Allow suppressing messages:** since reflective code cannot be annotated successfully using ownership domains [6];

```
ArrayList vCourse = student.getRegisteredCourses();
for (int i=0; i<vCourse.size(); i++) {
 if (((Course) vCourse.get(i)).conflicts(course)) {
   ...
 }
}
/* ABOVE MUST BE REWRITTEN AS ... */
@Domain("lent<state>")
ArrayList vCourse = student.getRegisteredCourses();
for (int i=0; i<vCourse.size(); i++) {
  @Domain("lent<state>")
  Course crs = (Course) vCourse.get(i);
  if (crs.conflicts(course)) {
    ...
  }
}
```

Figure 5. Re-writing a cast expression using local variables.

- **Display annotations more elegantly:** an Eclipse plug-in by Eisenberg and Kiczales [18] can beautify Java 1.5 annotations for interactive editing while the analysis uses the same AST.

5. Ownership Domains Case Studies

The annotation-based system is mostly complete — the domain link checks are still being implemented as of this writing. We used the tools to add and check ownership domain annotations on two real Java programs with around 15,000 lines of code each.

JHotDraw. The subject system for the first case study is JHotDraw [23]. Version 5.3 has around 200 classes and 15,000 lines of Java. JHotDraw is rich with design patterns [22], uses both composition and inheritance heavily and has evolved through several versions. We first used the defaulting tool then manually modified the annotations as needed. Adding annotations was iterative. For instance, over several iterations, we made more use of the `owned` annotation. JHotDraw was annotated without making any structural refactoring such as extracting interfaces, etc. Some code changes were needed however to use our annotation system, e.g., extract a local variable from a new expression and add an annotation on the local variable, convert an anonymous class to a nested class to add domain parameters to it, etc. JHotDraw Version 5.3 did not use generic types, so we used Eclipse refactorings [21] to infer generic types of containers.

HillClimber. By many accounts, JHotDraw is considered the brainchild of experts in object-oriented design and programming. In comparison, the subject system for this case study, HillClimber, is another 15,000 line application that was mainly developed and maintained by undergraduates [2]. In previous work, we re-engineered the original Java program to an ArchJava [4] implementation with ownership domain annotations, but using language extensions instead of Java 1.5 annotations [2]. The re-engineering case study also produced a version that refactored the original code by making most class fields as `private` [2]. For this case study, we started from the refactored Java code and added ownership domain annotations to it.

Unlike JHotDraw, adding annotations to HillClimber involved refactoring to decouple the code as discussed below. We also refactored the code to use generics, mostly automatically using Eclipse. However, Eclipse cannot infer the generic type of a variable of type `Vector` storing arrays of `Node` objects. Such code was manually refactored to use `Vector<Vector<Node>>`.

Compared to the earlier case study with language extensions [2], the annotation-based system allowed using Eclipse refactoring tools to extract interfaces and infer generic types while adding the ownership domain annotations. Comparing the number of hours would not be meaningful since the annotation-based system was still under development while the case study was under way, and that would not account for the learning effect of annotating the same program twice. Anecdotally, we were more productive with the annotation-based system than with the earlier tool using language extensions. The overall process changed around 40% of the lines of code in HillClimber. The 40% code changes included boilerplate `imports` to use our Java 1.5 annotations, and code changes to support adding annotations to some expressions. To more accurately gauge the manual annotation overhead, an AST-visitor was used to count the instances where the current annotation is the same as the one generated by the defaulting tool: over 40% of the annotations were exactly the same as the default ones for HillClimber; that number was around 30% for JHotDraw. There are 60 type errors remaining in JHotDraw and 42 errors remaining in HillClimber.

In this following discussion, we illustrate using actual examples from JHotDraw and HillClimber, how ownership domains can express and enforce design intent related to object encapsulation and communication, using code snippets from the subject systems. The code was slightly edited for presentation by removing trivial modifiers. Some code is highlighted using underlining.

5.1 Ownership domains enforce instance encapsulation

All ownership type systems can express and enforce instance encapsulation which is stronger than the module visibility mechanism of making a field `private`. In ownership domains, placing a field in the private `owned` domain means that the object can be reached only by going through its owner; as a result, no aliases to that object can leak to the outside. Consider `CompositeFigure` in JHotDraw:

```
@Domains({"owned"})
@DomainParams({"M"})...
abstract class CompositeFigure ... {
 // The figures that comprise this figure
 @Domain("owned<M<M>>")Vector<Figure> fFigures;

 /**
  * Adds a vector of figures.
  */
 void addAll(@Domain("M<M<M>>")Vector<Figure> figs) {
  // Cannot assign object in "M<M>" to "owned<M>"
  // this.fFigures = figs;

  // This is correct however
  fFigures.addAll(figs);
 }
...
}
```

Annotating field `fFigures` with `owned` encapsulates the list of composite `Figures` (`fFigures`) to prevent objects that only have access to the composite object from modifying the list directly. If a developer tries to subvert the language visibility mechanisms by exposing a `private` or `protected` field using a `public` accessor method, the ownership domains type system prohibits a `public` method from having an `owned` parameter or return value. Letting Eclipse generate a setter for the `fFigures` field produces the following code — without annotations:

```
void setFFigures(Vector<Figure> figs) {
  this.fFigures = figs;
}
```

Upon adding the annotations, a developer can realize that the setter is overwriting the existing field since the parameter `figs` cannot be marked as `owned` and any other annotation would fail the assignment check when overwriting the `fFigures` field.

When manually adding annotations, it is possible to miss many opportunities for making objects `owned`. Indeed, we initially annotated `fFigures` with the domain parameter `M` instead of `owned`. In some cases, objects should be `owned` but are not, and making them `owned` may require code changes, e.g., to return a copy of an object instead of an alias to a private field.

Visualizing the annotations encouraged us to make more use of the `owned` annotation since `owned` avoids cluttering the top-level domains [1]. Perhaps better tool support can prompt a developer to encapsulate a field that could be annotated with `owned` but is not, e.g., a lightweight compile-time ownership inference algorithm [33] could suggest possible Eclipse "quickfixes".

5.2 Ownership domains specify architectural tiers

A tiered architecture is often used to organize an application into a User Interface tier, a Business Logic tier, and a Data tier. Ownership domains can express and enforce such a tiered runtime architecture by representing a tier as an ownership domain [3], and a permission between tiers as a domain link to allow objects in the User Interface tier to refer to objects in the Business Logic tier but not vice versa. Such an architectural structure and constraints cannot be easily expressed in plain Java code.

We organized the core JHotDraw types in Figure 6 according to the Model-View-Controller design pattern as follows:

- `Model`: consists of `Drawing`, `Figure`, `Connector`, etc. A `Drawing` is composed of `Figures` which know their containing `Drawing`. A `Figure` has a list of `Handles` to allow user interactions. A `Drawing` also extends `FigureChangeListener` (not shown) to listen to changes to its `Figures`.
- `View`: consists of `DrawingEditor`, `DrawingView` and associated types. `DrawingView` extends `DrawingChangeListener` (not shown) to listen to changes to `Drawing` objects.
- `Controller`: includes interfaces such as `Handle`, `Tool` and `Command`. A `Tool` is used by a `DrawingView` to manipulate a `Drawing`. A `Command` encapsulates an action to be executed — a simple instance of the Command design pattern [22, p. 233] without undo support.

Once we defined the three top-level ownership domains, `Model`, `View` and `Controller`, we passed the corresponding domain parameters `M`, `V` and `C` to various types as discussed below. A visualization of the JHotDraw execution structure based on these ownership domain annotations is available [1].

In HillClimber, the application *window* uses a *canvas* to display *nodes* and *edges* of a *graph* in order to demonstrate algorithms for constraint satisfaction problems provided by the *engine*. So we organized the HillClimber types in Figure 12 as follows. The `data` ownership domain stores the graph objects (instances of `Graph`, `Node`, etc., and those of their subclasses, `HillGraph`, `HillNode`, etc.). The `ui` domain holds user interface objects. The `logic` domain holds instances of `HillEngine`, `Search` (and subclasses thereof) objects, and associated objects. A visualization of the HillClimber execution structure based on these ownership domain annotations is available [1].

5.3 Ownership domains expose implicit communication

Design patterns — such as Observer [22, p. 293], used to decouple object-oriented code also tend to make the communication between objects implicit. Adding ownership domain annotations helps make that communication more explicit.

We initially wanted to parameterize `Drawing` (See Figure 7) with only the `M` domain parameter, but `DrawingChangeListener` is implemented by `DrawingView`. So `DrawingChangeListener` needed to be annotated with the `V` domain parameter corresponding to the `View`. By making implicit communication explicit, annotations seem to prematurely constrain `DrawingChangeListener` objects to be in the `View` domain. Since `Drawing` was a core interface referenced by other interfaces in the core `framework` package, this led to passing all three domain parameters to many additional interfaces and classes.

It is true that if `Drawing` had to be parameterized by domain parameter `V` for some other reason, the implicit communication in the observer would not have been discovered this way. Ownership domain annotations help make implicit communication explicit when a reference requires permission to access a new part of the program for the first time.

In HillClimber, adding ownership domain annotations exposed covert object communication through base classes from two parallel inheritance hierarchies. During an early iteration, we parameterized the base class `GraphCanvas` by the `ui` and `data` domain parameters. We then realized that `Graph`, the base class for `HillGraph`, required the `ui` domain parameter (See Figure 12). Class `Graph` only needed the `ui` domain parameter to properly annotate a `GraphCanvas` field reference that we did not expect. This in turn revealed that `HillGraph` and `HillCanvas` were communicating through their base classes `Graph` and `GraphCanvas`. In the end, the reference to `GraphCanvas` was moved from `Graph` to `HillGraph` and generalized as an `IHillCanvas` reference by extracting an interface `IHillGraph` from `HillGraph`.

```
/**
 * Drawing is a container for Figures. Drawing sends
 * out DrawingChanged events to DrawingChangeListeners
 * whenever a part of its area was invalidated.
 * The Observer pattern is used to decouple the Drawing
 * from its views and to enable multiple views.
 */
@DomainParams({"M", "V"})
@DomainInherits({"FigureChangeListener<M>",...})
interface Drawing extends FigureChangeListener... {
  // Adds a listener for this drawing.
  void addDrawingChangeListener(
      @Domain("V<M,V>")DrawingChangeListener l);

  // Adds a figure and sets its container
  // to refer to this drawing.
  @Domain("M<M>")
  Figure add(@Domain("M<M>") Figure figure);
...
}
```

Figure 7. Adding annotations to `Drawing`.

```
@DomainParams({"M","V","C"})
interface Handle {
 void invokeStart(@Domain("V<M,V,C>")DrawingView v);
 ...
 @Domain("M<M,V,C>")Undoable getUndoActivity();
}
```

Figure 8. `Handle` with `M`, `V` and `C` domain parameters.

5.4 Ownership domains expose tight coupling

Let us temporarily ignore the earlier limitation with adding annotations to the listeners and assume that `Drawing` could be parameterized by only the `M` domain parameter. Let us consider whether it would be possible to parameterize interface `Handle` (See Figure 8) with domains `M` and `C`. A `Handle` would be in the `C` domain and would access objects in that domain and in `M` domain, i.e., it should not access objects in the `V` domain parameter. Note that even if the explicit parameter `C` was not provided, that domain would still be accessible to `Handle` using the `owner` annotation.

A comment in the code indicated that Version 4.1 deprecated the original `invokeStart` method which took a `Drawing` object as one of its parameters, in favor of an `invokeStart` method that takes instead a formal parameter `DrawingView` parameterized by `M`,`V`, and `C`. This required passing to `Handle` the additional domain parameter `V`. Since `Handle` is a core interface referenced by other interfaces in the core `framework` package, this also led to passing all three domain parameters to many additional types.

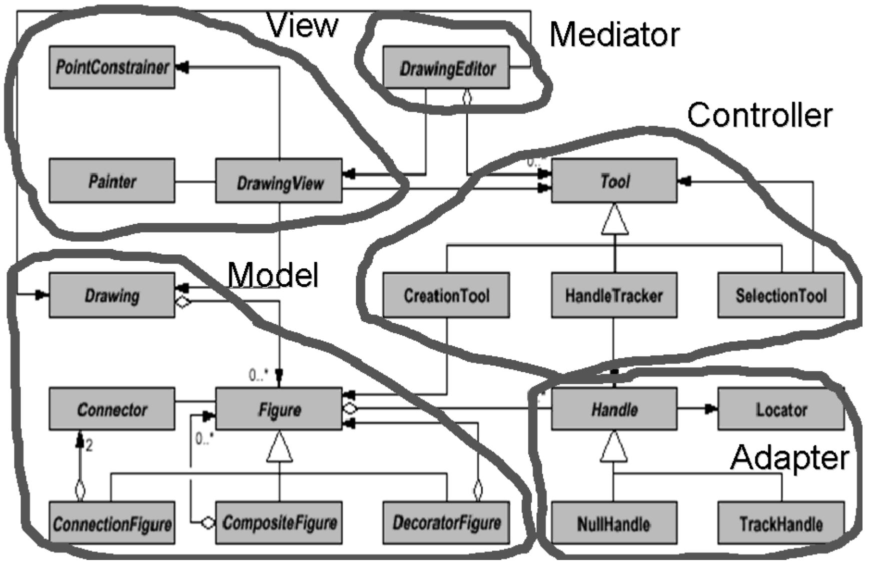


Figure 6. Simplified class diagram for JHotDraw (Adapted from manual class diagram by Riehle [43, 12]).

```
@DomainParams({"M","C"})
interface Handle {
 @DomainParams({"V"})
 void invokeStart(@Domain("V<M,V,C>")DrawingView v);
 ...
 @Domain("M<M>")Undoable getUndoActivity();
}
```

Figure 9. `Handle` with only `M` and `C` domain parameters.

```
@DomainParams({"M","C"})
@DomainInherits({"Handle<M,C>"})
abstract class AbstractHandle implements Handle {
 // Will not typecheck since 'V' unbound
 @Domain("V<M,V,C>")DrawingView view;
 ...
 @DomainParams({"V"})
 void invokeStart(@Domain("V<M,V,C>")DrawingView v) {
   // Cannot store argument in field 'this.view'
 }
}
```

Figure 10. Method domain parameters can enforce lifetime.

5.5 Ownership domains expose and enforce object lifetime

Let us assume in this section that the refactoring which introduced the tighter coupling was never performed, i.e., `Handle` still needed a `Drawing` instead of a `DrawingView`. Undo support was added to JHotDraw for the first time in Version 5.3. In particular, `Handle` now had a reference to `Undoable` — which in turn required domain parameters `M`,`V` and `C` because `Undoable`'s `getDrawingView()` method returned a `DrawingView`.

Now, let us see if it would be possible to annotate `Undoable` and `Handle` with only the domain parameters `M` and `C` (See Figure 9) — the domain parameter `V` can then be supplied to `invokeStart()` as a method domain parameter.

Using a method domain parameter to annotate the formal parameter `v` could enforce the constraint that a developer should not store in a field the `DrawingView` object passed as an argument to `invokeStart()`, as in Figure 10. Of course, a developer could store the `DrawingView` object in a field of type `Object`, but that field would have to be cast to a `DrawingView` to be of any use.

Instead of a method domain parameter, the `lent` annotation could also be used to allow a temporary alias to an object within a method boundary. We found a few such examples in JHotDraw. Method `setAffectedFigures` in Figure 11 makes a copy of the `lent` argument so it cannot just hold on to it.

In fact, `lent` can be formally modeled as a method domain parameter. However, the type system does not allow a method to return a `lent` value but it allows a method to return an object in a method domain parameter. In the case of `DrawingView`, `lent` cannot be used because implementations of `invokeStart()` construct `Undoable` objects that maintain aliases to the `DrawingView` and thus require the `V` domain parameter.

For that same reason, the `Undoable` interface requires the `V` domain parameter because `Undoable` stores the `DrawingView` where the activity to be undone was performed in order to undo the changes to that view only. This may slightly violate the Model-View-Controller design, where model objects should not hold on to view objects, because there might be multiple views that need to be updated in response to changes in the model. At the same time, it would be counter-intuitive for a user to undo a change in one view and observe changes in some other view. Thus, ownership domain annotations expose the tighter coupling that the Undo feature introduced. Figure 11 shows in more detail the interaction between `Handle`, `Undoable` and `DrawingView`.

An earlier empirical study of JHotDraw mentioned that "a common architectural mistake [. . .] was to provide `Figures` with a reference to the `Drawing` or the `DrawingView`. `Figures` do not by default have any access to either the `Drawing` or the `DrawingView` in which they are contained. This prevents them from accessing information such as the size of the `Drawing`. However, it is possible to overcome this problem by passing the view into the constructor of a figure, which can then store and access this as required" [28]. Starting with Version 5.3, one could get to the `Figure`'s `Handles` through its `handles()` method then get a `DrawingView` through a `Handle`'s `UndoActivity` objects.

5.6 Ownership domains promote decoupling code

Ownership domain annotations highlight tight coupling and promote programming practices that decouple code.

Programming to an Interface. It is recommended to "refer to objects by their interfaces" [7, Item #34] since interfaces can reduce

```
@DomainParams({"M", "V", "C"})
@DomainInherits({"LocatorHandle<M,V,C>"})
class ResizeHandle extends LocatorHandle {
  @Override
  void invokeStart(int x, int y,
  @Domain("V<M,V,C>") DrawingView view) {
        setUndoActivity(createUndoActivity(view));
      ...
  }
  /**
   * Factory method for undo activity.
   * To be overriden by subclasses.
   */
  protected @Domain("M<M,V,C>")Undoable
  createUndoActivity(
                    @Domain("V<M,V,C>")DrawingView view) {
    @Domain("unique<M,V,C>")
    ResizeHandle.UndoActivity
    undoable = new ResizeHandle.UndoActivity(view);
    return undoable;
  }

    @DomainParams({"M", "V", "C"})
    @DomainInherits("UndoableAdapter<M,V,C>")
    static class UndoActivity extends UndoableAdapter {
    ...
    }
}
/**
 * Basic implementation for an Undoable activity
 */
@DomainParams({"M", "V", "C"})
@DomainInherits("Undoable<M,V,C>")
public class UndoableAdapter implements Undoable {
  @Domain("V<M,V,C>")DrawingView myDrawingView;

  UndoableAdapter(@Domain("V<M,V,C>")DrawingView dv) {
    setDrawingView(dv);
  }
 @Domain("V<M,V,C>") DrawingView getDrawingView() {
   return myDrawingView;
 }
 void setDrawingView(@Domain("V<M,V,C>")DrawingView dv) {
   myDrawingView = dv;
 }
 void setAffectedFigures(@Domain("lent<M>")FigureEnumeration fe) {
   // the enumeration is not reusable therefore a copy is made
   // to be able to undo-redo the command several time
   rememberFigures(fe);
 }
}
```

Figure 11. Concrete implementation class of `Handle`.

coupling between classes by splitting intent from implementation. When fewer domain parameters are needed to annotate an interface (as compared to the corresponding class), ownership domain annotations can enforce this idiom.

In particular, an implementation class can require a private ownership domain to be passed as an actual value for one its parameters. Since a private ownership domain cannot be named by an outside client, the client is then forced to use the interface which does not require these parameters.

For instance, in the earlier `Sequence` example (Figure 1), the `SeqIterator` class receives the `Sequence`'s private domain `owned` and hides the extra parameterization behind the `Iterator` interface. This forces a client of the `Sequence` to access the iterator objects only through the `Iterator` interface. A client may not even cast the `Iterator` reference to a `SeqIterator` class.

We used a similar technique to decouple the code in HillClimber (See Figure 12 for the inheritance hierarchy). The original implementation for class `HillNode` had a field reference of type `HillGraph`. However, `HillGraph` took the three domain parameters `ui`, `logic` and `data`, which required passing all those parameters to `HillNode`.

```
@DomainParams({"ui","logic","data"})
@DomainInherits({"Node<data>"})
class HillNode extends Node {
  @Domain("data<ui,logic,data>")HillGraph graph;
...
}
```

When adding annotations, an unexpected domain parameter often indicates unnecessary coupling, e.g., why should `HillNode` have access to the `ui` domain? Thus a lengthy domain parameter list can be an objective measure of a code smell [2]. Furthermore, ownership domain annotations can help a developer lower the coupling by suggesting which specific type declarations need to be generalized to shorten the list of domain parameters on the enclosing type.

In HillClimber, one solution was to extract an `IHillGraph` interface from class `HillGraph` that only requires the `data` domain parameter and make a `HillNode` object reference the `HillGraph` object through the `IHillGraph` interface. We decided against carrying this refactoring further and eliminating the `ui` and `logic` domain parameters on `HillGraph` itself.

Since the `HillGraph`, `HillNode`, etc., form a parallel inheritance hierarchy to `Graph`, `Node`, etc., a similar refactoring was performed on `Graph` by extracting a `IGraph` interface – even though `Graph` and `IGraph` are both parameterized by `data`.

```
@DomainParams({"ui","logic","data"})
@DomainInherits({"Graph<data>",
                 "IHillGraph<data>"})
class HillGraph extends Graph
                implements IHillGraph {
  ...
}
@DomainParams({"data"})
@DomainInherits({"IGraph<data>"})
interface IHillGraph extends IGraph {
...
}
@DomainParams({"data"})
@DomainInherits({"Node<data>"})
class HillNode extends Node {
  @Domain("data<data>")IHillGraph graph;
...
}
```

Tightly coupled code was observed throughout HillClimber. Similarly, we were surprised that a dialog class `FontDialog` required the `data` domain parameter. It turned out that `FontDialog` had a field reference declared with its most specific type `GraphCanvas`. In some cases, it is possible to generalize the type of the reference, e.g., use `java.awt.Frame` to eliminate the need for the domain parameter. However, `FontDialog` needed access to some of the `GraphCanvas` functionality, so a different solution was needed.

Mediator Pattern. Defining an interface is sometimes insufficient to decouple code since referring to an object through its interface still requires access to the domain the object is in. One solution is to use the Mediator design pattern [22, p. 273], as shown here.

In the original HillClimber implementation, `Node` obtained a reference to `GraphCanvas`, which violates the Law of Demeter [32], i.e., objects should only talk to their immediate neighbors:

```
@DomainParams({"data"})
abstract class Entity {
  @Domain("data<data>") Graph graph; // parent graph
...
}
@DomainParams({"data"})
@DomainInherits({"Entity<data>"})
class Node extends Entity
 ...
 int getHeight() {
   return graph.getCanvas().getFontMetrics()...;
  }
}
```

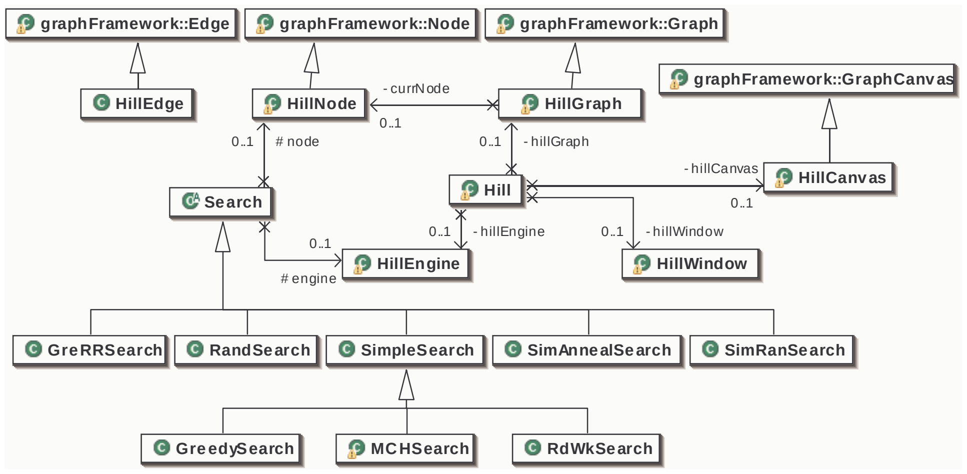


Figure 12. Partial UML Class Diagram for HillClimber obtained from the original implementation using Eclipse UML [39]. This diagram does not reflect some of the types introduced during refactoring, such as `IGraph`, `IHillGraph` and `ICanvasMediator`.

Extracting an interface from `GraphCanvas` would not work, as that reference would still need the `ui` domain parameter. Moreover, the implementation of `getFontMetrics()` could not be moved to `Graph` as it required access to objects in the `ui` domain.

```
@DomainParams({"data"})
abstract class Entity {
  @Domain("ui")IGraphCanvas canvas; // 'ui' unbound
...
}
```

A mediator was defined as follows:

```
/**
 * Mediator interface
 */
interface ICanvasMediator {
 @Domain("shared")FontMetrics getFontMetrics();
...
}
/**
 * Mediator implementation class
 */
@DomainParams({"ui","data"})
class MediatorImpl implements ICanvasMediator {
 @Domain("ui<ui,data>")GraphCanvas canvas;

 MediatorImpl(@Domain("ui<ui,data>")GraphCanvas c) {
  this.canvas = c;
 }
 @Domain("shared")FontMetrics getFontMetrics() {
   return canvas.getFontMetrics();
 }
...
}
```

`GraphCanvas` initializes the mediator:

```
@DomainParams({"ui","data"})
class GraphCanvas extends ... {
 @Domain("data<ui,data>")MediatorImpl mediator;
 ...
 @Domain("data")ICanvasMediator getMediator() {
     return mediator;
 }
}
```

`Entity` and `Node` can then use the mediator as follows:

```
@DomainParams({"data"})
abstract class Entity {
  @Domain("data") ICanvasMediator mediator;
...
}
```

```
/**
 * DrawApplication defines a standard presentation
 * for standalone drawing editors
 */
@DomainParams({"M", "V", "C"})
@DomainInherits({"DrawingEditor<M,V,C>", ...)
class DrawApplication implements DrawingEditor... {
  // Opens a new window with a drawing view.
  @DomainReceiver("unique")
  protected void open(...) {
    fIconkit = new Iconkit(this);
    ...
  }
}
class Iconkit {
  static @Domain("unique")Iconkit fgIconkit = null;

  // Constructs an Iconkit that uses the given editor
  // to resolve image path names.
  @DomainReceiver("unique")
  public Iconkit(@Domain("unique")Component comp){
    fgIconkit = this;
    ...
  }
}
```

Figure 13. Annotating a singleton using `unique`.

```
@DomainParams({"data"})
@DomainInherits({"Entity<data>"})
class Node extends Entity {
  int getHeight() {
    return getMediator().getFontMetrics()...;
  }
```

5.7 Ownership domains can help identify singletons

While adding ownership domain annotations, we discovered a curious instance of the Singleton design pattern: `IconKit`'s constructor was not private, although it had a static `instance()` method. Indeed, there is a `unique` instance of `DrawingEditor` (the application itself) and a `unique IconKit` (See Figure 13) at runtime.

6. Expressiveness Challenges

In this section, we discuss some of the expressiveness gaps that we encountered, some of which had been previously mentioned.

```
class DrawApplication implements DrawingEditor ... {
...
class MDI_DrawApplication extends DrawApplication ...{
...
@DomainParams({"M", "V", "C"})
@DomainInherits({"MDI_DrawApplication<M,V,C>"})
class JavaDrawApp extends MDI_DrawApplication {
...
@Domains({"Model", "View", "Controller"})
class Main {
  @Domain("View<Model,View,Controller>")
  JavaDrawApp app = new JavaDrawApp();

  public static void main(
      @Domain("lent[shared]")String args[]) {
      @Domain("lent")Main system = new Main();
  }
}
```

Figure 14. Defining the top-level domains in a separate class.

6.1 An object cannot be in more than one ownership domain

Ownership domains, as most other ownership type systems, support only *single ownership*, i.e., an object cannot be part of more than one ownership hierarchy. Proposals for *multiple ownership* [11] lift this restriction in other type systems. Ownership domains do not support *ownership transfer* [31] either, i.e., an object's owner does not change — only `unique` objects can flow between any two domains. As a result, many fine-grained ownership domains cannot be defined to represent multiple roles in design patterns: e.g., if an object is both a mediator in the Mediator pattern and a view in the Model-View-Controller pattern, it cannot be in both a `Mediator` ownership domain and a `View` ownership domain at the same time.

For instance, creating top-level ownership domains to correspond to the design in Figure 6 would have been more challenging than creating the three top-level domains for `Model`, `View` and `Controller`: placing a `DrawingEditor` object in a `Mediator` domain would have prohibited it from also being in the `View` domain.

6.2 An object cannot place itself in a domain it declares

An object cannot place itself in an ownership domain that it declares. This is problematic for the root application object, i.e., the `JavaDrawApp` instance (`JavaDrawApp` extends `DrawApplication` which in turn extends `DrawingEditor`). True to form, we solve this problem with an extra level of indirection by creating a fake top-level class `Main` to declare the `Model`, `View` and `Controller` top-level ownership domains and declare the `JavaDrawApp` object in the `View` domain (See Figure 14).

6.3 Public domains are hard to use

Public domains make the ownership domains type system more flexible than *owner-as-dominator* type systems [15]. Also, public domains are ideal for visualization because placing an object inside a public domain of another object relates these objects without cluttering the top-level domains [1]. However, public domains are typically hard to use without refactoring the code. We started using them in a few cases but quickly abandoned those attempts.

Since the Observer design pattern tends to make communication between objects implicit, we attempted to represent listeners more explicitly using ownership domain annotations. For instance, it might make sense to create a public domain `LISTENERS` as a domain to hold the `Listener` objects that an `Observer` will notify — a `Listener` often needs special access to the `Observer`, but usually does not need special access to the `Subject`.

JHotDraw uses a delegation-based event model: for instance, a `DrawingView` calls method `figureSelectionChanged` to notify a `FigureSelectionListener` observer of selection changes. So it might make sense to declare a `FIGURESELECTIONLISTENERS`

```
/**
 * DrawingView renders a Drawing and listens to its
 * changes. It receives user input and delegates
 * it to the current Tool.
 */
@DomainParams({"M", "V", "C"})
@DomainInherits({"DrawingChangeListener<M,V>"})
interface DrawingView extends DrawingChangeListener... {
  // Add a listener for selection changes
  void addFigureSelectionListener(
   @Domain("?<M,V,C>")FigureSelectionListener fsl);
...
}
@Domains({"owned"})
@DomainParams({"M", "V", "C"})
@DomainInherits({"DrawingView<M,V,C>"})
class StandardDrawingView implements DrawingView... {
  // Registered list of listeners for selection changes
  private @Domain("owned<?<M,V,C>>")
  Vector<FigureSelectionListener> fSelectionListeners;

  StandardDrawingView(
    @Domain("V<M,V,C>")DrawingEditor editor, ...) {
    // editor is in 'V' domain parameter, not 'C'!
    addFigureSelectionListener(editor);
    ...
  }
  // Add a listener for selection changes.
  // Command implements FigureSelectionListener
  // but Command is in the 'C' domain parameter!
  void addFigureSelectionListener(
    @Domain("?<M,V,C>")FigureSelectionListener fsl) {
    fSelectionListeners.add(fsl);
  }
```

Figure 15. How to annotate `addFigureSelectionListener`?

public domain on `Command` to hold the `FigureSelectionListener` objects. But `Command` implements `FigureSelectionListener`, so a `Command` *is-a* `FigureSelectionListener`. Thus a `Command` object cannot split a part of itself and place it in the public domain `FIGURESELECTIONLISTENERS` that it declares.

6.4 Listener objects are particularly challenging

There were additional complications when trying to highlight the event subsystem in JHotDraw using ownership domain annotations. `Command`, which is in the `Controller` domain, implements `FigureSelectionListener`, and so does `DrawingEditor`, which is in the `View` domain.

Consider method `addFigureSelectionListener` in (See Figure 15). How would one annotate the formal parameter `FigureSelectionList`
The parameter should support both annotations `C<M,V,C>` and `V<M,V,C>`. Existential ownership [13, 29, 34] may be the answer to increase the expressiveness, e.g., by annotating the parameter with "`any`" [34]. Other problems of adding ownership domains annotations to listeners had been previously identified [44].

6.5 Static code can be challenging

Even in such a well-designed program as JHotDraw, we found a few instances where ownership annotations cannot be made to typecheck. In particular, in Figure 16, the static `Hashtable` cannot have the `M`, `V`, and `C` domain parameters because the domain parameters declared on the class `NullDrawingView` are not in scope for static members. Static members can only be annotated with `shared` or `unique`, and these values cannot flow to the `Mx`, `Vx` or `Cx` method domain parameters.

Annotating the generic `Hashtable` also requires nested parameters: `Hashtable` has three domain parameters for its keys, values and entries. Both `DrawingView` and `DrawingEditor` take `M`, `V`, and `C` as parameters. Although the number of annotations seems excessive and maybe argues in favor of generic ownership [41], the

```
@DomainParams({"M", "V", "C"})
@DomainInherits({"DrawingView<M,V,C>"})
class NullDrawingView implements DrawingView ... {
  static @Domain("unique<?<?,?,?>,?<?,?,?>,?>")
  Hashtable<DrawingEditor,DrawingView> dvMgr = ...;

  @DomainParams({"Mx","Vx","Cx"})
  public synchronized static @Domain("Vx<Mx,Vx,Cx>")
  DrawingView getManagedDrawingView(
          @Domain("Vx<Mx,Vx,Cx>")DrawingEditor ed) {
    if (dvMgr.containsKey(ed)) {
      @Domain("Vx<Mx,Vx,Cx>")
      DrawingView dv = dvMgr.get(ed);
      return dv;
    }
    ...
  }
```

Figure 16. How to annotate objects that are stored in static fields?

ownership domains for the Hashtable key, value and entries need not correspond to the M, V and C ownership domains.

A solution that is not type-safe would be to store the Hashtable as Object, then cast down to a Hashtable upon use — the equivalent of raw types but without re-implementing them in the ownership domains type system. Another solution would be to refactor the program to eliminate this static field since it gives any object access to all the DrawingView and DrawingEditor objects. Since it is often unrealistic to perform such a significant refactoring, maybe the best solution would be to support package-level static ownership domains, similar to confined types [9].

6.6 Annotations may be unnecessarily verbose

Ownership domain annotations tend to be verbose: e.g., formal method parameters need to be fully annotated even if they are not used in the method body or used in a restricted way. This produces particularly unwieldy annotations for containers of generic types.

In Figure 17, method clearStackVerbose indicates the current level of annotations needed. It should be possible to leave out domain parameters when they are not really needed. This may involve using implicit existential ownership types as in clearStackAny: i.e., there exists some domain parameters d1, d2, d3, d4, such that the formal method parameter s could be annotated with lent<d1<d2,d3,d4>>. Using appropriate defaults, the annotations could probably be reduced to the level needed to annotate a raw type, as shown in clearStack.

6.7 Manifest ownership can reduce the annotation burden

The current defaulting tool only adds the shared annotation to String objects. However, during the annotation process, we found ourselves adding the shared annotation to many other types such as Font, FontMetrics, Color, etc. Specifying a per-type default globally and not for every instance, as in *manifest ownership* [13], would have reduced the annotation burden.

6.8 Reflective code cannot be annotated

JHotDraw uses reflective code to serialize and deserialize its state and such code cannot be annotated using ownership domains [6].

6.9 Annotate Exceptions as lent

We annotated exceptions with lent since we were not particularly interested in reasoning about them. However, richer annotations are possible [45].

7. Related Work

Case studies applying ownership type systems on real code are few and far between. Hächler [25] documented a case study applying

```
@Domains({"owned"})
@DomainParams({"M","V","C"})
public class UndoManager {
  /**
   * Collection of undo activities
   */
  @Domain("owned<M<M,V,C>>")Vector<Undoable> undoStack;

  void clearStackVerbose(
    @Domain("lent<M<M,V,C>>")Vector<Undoable> s) {
    s.removeAllElements();
  }

  void clearStackAny(
    @Domain("lent<?<?,?,?>>")Vector<Undoable> s) {
    s.removeAllElements();
  }

  void clearStack(
    @Domain("lent")Vector<Undoable> s) {
    s.removeAllElements();
  }
}
```

Figure 17. Reducing annotations when they are not really needed.

Universes [36, 17] on an industrial software application and refactoring the code in the process. Although the subject system in the case study is larger than JHotDraw (around 55,000 lines of code), the author annotated only a portion of the system. The author manually generated visualizations of the ownership structure whereas we had access to tool support to visualize the ownership structure and adjust the annotations accordingly [1].

Nägeli [38] evaluated how the Universes and Ownership Domains type systems express the standard object-oriented design patterns [22]. However, in real world complex object-oriented code, design patterns rarely occur in isolation [43]. As we discussed earlier, these subtle interactions, combined with the single ownership constraint of the type system, make the annotations difficult.

In a previous case study, we re-engineered HillClimber using ArchJava [4] to specify a component-and-connector architecture in code and ownership domain annotations to specify the data sharing [2]. In the earlier case study, we performed refactorings similar to the ones described here. However, adding ownership domain annotations to the ArchJava program seemed easier. Indeed, ArchJava's port construct effectively reduces coupling; in the plain Java implementation, the same effect had to be achieved using programming to interfaces, using mediators, etc.

ArchJava's properties are available at the expense of various restrictions on object-oriented implementations. The previous case study also identified that adding ownership domain annotations required less effort than encoding the architectural structure in ArchJava [2, 6]. Fewer defects are introduced since code that passes object references need not be changed and the ownership annotations need not affect the runtime semantics of the program. Moreover, the ownership domain annotations, while tedious to add manually, are relatively straightforward once the top-level domains are decided, compared to re-engineering to use ArchJava.

Adding ownership domains annotations manually still required significant effort, and researchers are still looking at scalable inference of ownership domain annotations [6, 16]. Current inference techniques [35, 33] however only infer the equivalent of owned, shared, lent and unique annotations, i.e., they assume a strict owner-as-dominator hierarchy which is not flexible enough to represent many design patterns. Some approaches do not map the results of the analysis back to an ownership type system [35, 33]. A fully automated inference cannot create multiple public domains in one object and meaningful domain parameters, which are critical for representing the abstract design intent, as in the three top-level

`Model`, `View`, and `Controller` domains in JHotDraw. Existing inference algorithms often generate imprecise annotations, producing for each class a long list of domain parameters, often placing each field in a separate domain, and annotating many more objects as `shared` or `lent` than necessary [6, 16].

8. Conclusion

We presented an annotation-based system that re-implements the ownership domains type system as a set of Java 1.5 annotations, using the Eclipse infrastructure. Using annotations imposes many restrictions and requires changing the code slightly to add annotations to it, and the annotation language does take some getting used to. Still, the annotation-based system is an improvement over custom infrastructure, language extensions, and the resulting limited tool support: it enabled us to annotate larger object-oriented programs "in the wild" to study how ownership domains can express and enforce design intent related to object encapsulation and communication and to identify expressiveness limitations.

In future work, we plan on making the type system more flexible and extending the annotation language in a non-breaking way.

Acknowledgments

This work was supported in part by NSF grant CCF-0546550, DARPA contract HR00110710019, the Department of Defense, and the Software Industry Center at Carnegie Mellon University and its sponsors, especially the Alfred P. Sloan Foundation.

References

[1] M. Abi-Antoun and J. Aldrich. Compile-Time Views of Execution Structure Based on Ownership. In *Intl. Workshop on Aliasing, Confinement and Ownership*, 2007.

[2] M. Abi-Antoun, J. Aldrich, and W. Coelho. A Case Study in Re-engineering to Enforce Architectural Control Flow and Data Sharing. *J. Systems and Software*, 80(2), 2007.

[3] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.

[4] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE*, 2002.

[5] J. Aldrich and D. Dickey. The Crystal Data Flow Analysis Framework 2.0. `http://www.cs.cmu.edu/~aldrich/courses/654/`, 2006.

[6] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, 2002.

[7] J. Bloch. *Effective Java*. Addison-Wesley, 2001.

[8] B. Bokowski and A. Spiegel. Barat — A Front–End for Java. Technical Report B-98-09, Freie Universität Berlin, 1998.

[9] B. Bokowski and J. Vitek. Confined Types. In *OOPSLA*, 1999.

[10] C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In *POPL*, 2003.

[11] N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple Ownership. In *OOPSLA*, 2007. To appear.

[12] H. B. Christensen. Frameworks: Putting Design Patterns into Perspective. In *SIGCSE Innov. & Tech. in Comp. Sci. Ed.*, 2004.

[13] D. Clarke. *Object Ownership & Containment*. PhD thesis, University of New South Wales, 2001.

[14] D. Clarke and T. Wrigstad. External Uniqueness is Unique Enough. In *ECOOP*, 2003.

[15] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.

[16] W. Cooper. Interactive Ownership Type Inference. Senior Thesis, Carnegie Mellon University, 2005.

[17] W. Dietl and P. Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8), 2005.

[18] A. D. Eisenberg and G. Kiczales. Expressive Programs through Presentation Extension. In *AOSD*, 2007.

[19] M. D. Ernst and D. Coward. JSR 308: Annotations on Java types. `http://pag.csail.mit.edu/jsr308/`, 2006.

[20] Universes Tools. `www.sct.ethz.ch/research/universes/tools/`, 2007.

[21] R. M. Fuhrer, F. Tip, A. Kiezun, J. Dolby, and M. Keller. Efficiently Refactoring Java Applications to Use Generic Libraries. In *ECOOP*, 2005.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[23] Gamma, E. et al. JHotDraw. `http://www.jhotdraw.org/`, 1996.

[24] G. Goth. Beware the march of this IDE: Eclipse is overshadowing other tool techniques. *IEEE Software*, 22(4), 2005.

[25] T. Hächler. Applying the Universe Type System to an Industrial Application: Case Study. Master's thesis, ETH Zurich, 2005.

[26] JavaCC. `https://javacc.dev.java.net/`, 2006.

[27] JSR 175: A Metadata Facility for the Java Programming Language. `http://jcp.org/en/jsr/detail?id=175`, 2006.

[28] D. Kirk, M. Roper, and M. Wood. Identifying and Addressing Problems in Object-Oriented Framework Reuse. *Empirical Software Engineering*, 2006.

[29] N. Krishnaswami and J. Aldrich. Permission-Based Ownership: Encapsulating State in Higher-Order Typed Languages. In *PLDI*, 2005.

[30] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and Verification Challenges for Sequential Object-Oriented Programs. *Formal Aspects of Computing*, 2007. Submitted.

[31] K. R. M. Leino and P. Müller. Object Invariants in Dynamic Contexts. In *ECOOP*, 2004.

[32] K. J. Lieberherr and I. M. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Software*, 6(5), 1989.

[33] Y. Liu and A. Milanova. Ownership and Immutability Inference for UML-based Object Access Control. In *ICSE*, 2007.

[34] Y. Lu and J. Potter. Protecting Representation with Effect Encapsulation. In *POPL*, 2006.

[35] K.-K. Ma and J. S. Foster. Inferring Aliasing and Encapsulation Properties for Java. In *OOPSLA*, 2007. To appear.

[36] P. Müller and A. Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, 1999.

[37] G. C. Murphy, M. Kersten, and L. Findlater. How are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4), 2006.

[38] S. Nägeli. Ownership in Design Patterns. Master's thesis, Department of Computer Science, Federal Institute of Technology Zurich, 2006.

[39] Omondo. EclipseUML. `http://www.omondo.com/`, 2006.

[40] A. Potanin. Ownership Generic Java. `www.mcs.vuw.ac.nz/~alex/ogj/`, 2005.

[41] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic Ownership for Generic Java. In *OOPSLA*, 2006.

[42] Annotation File Utilities. `http://pag.csail.mit.edu/jsr308/annotation-fil` 2006.

[43] D. Riehle. *Framework Design: a Role Modeling Approach*. PhD thesis, Federal Institute of Technology Zurich, 2000.

[44] J. Schäfer and A. Poetzsch-Heffter. Simple Loose Ownership Domains. In *FTfJP*, 2006.

[45] D. Werner, , and P. Müller. Exceptions in Ownership Type Systems. In *FTfJP*, 2004.

[46] ArchJava. `http://www.archjava.org/`, 2007.

Formalizing Composite State Encapsulation

Adrian Fiech
Memorial University, St. John's, NL, A1B 3X5, Canada
afiech@cs.mun.ca

Ulf Schünemann
3S-Smart Software Solutions GmbH, Kempten, Germany
u.schuenemann@3s-software.com

Abstract

Representation exposure is a well documented and studied problem in object-oriented systems. We introduce the *Potential Access Path* methodology as a tool to reason about composite objects and protection of their representation. Our system enforces the owner-as-modifier disciplin, which does not restrict aliasing but requires that all modifications to an encapsulated aggregate are initiated by the aggregate's owner. A novel design choice in our system is the `free` mode that allows read-only aliases. This new weak uniqueness property provides us with additional flexibility to transfer subcomponents from one aggregate to another.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.1.5 [*Programming Techniques*]: Object-oriented Programming

General Terms Languages, Security, Theory

Keywords Alias protection, Representation exposure, Ownership types, State encapsulation, Java

1. Introduction

The recursive combination of smaller objects to one composite object (object composition) is a central technique in the construction of object-oriented software. The encapsulation of such composite objects is an important criterion for the quality of object-oriented designs. A lack of encapsulation makes the composite object's correct functioning depend on its context – its implementation cannot be verified in a modular way and cannot be safely reused in new contexts.

When discussing object-oriented software systems one often considers three related notions of the object concept. At the base-level, the system is a flat "sea" of elementary ***implementation objects*** – instances of concrete classes. At the top level we have ***abstract objects***, which are defined solely by their operations' externally visible behaviour (e.g described with the concept of interfaces in Java along with some behavioural specification). The implementation of this behaviour is delegated to the in-between level – structures of collaborating implementation objects rooted in a ***representative***. That representative, aided by its collaborators, provides the desired functionality specified in the interface. This cluster of cooperating objects will be referred henceforth as the ***composite object*** (or the ***aggregate***).

An abstract object's invariant – specified in the behavioural component of the interface – may depend on the internal structure of the composite object. In general, object-oriented languages do not prevent "outsiders" from obtaining references to the internal structure. Such exposure of the internal representation can lead to mutation of the structure while the representative object remains completely oblivious to the changes. The invariant may be violated and the implementation of the abstract object might behave inconsistent with its specification.

Among the first attempts to address the perils of representation exposure are [11] and [3]. Here the composite object is fully encapsulated and neither incoming nor outgoing references are allowed. The absence of incoming references guarantees that any modifications to the internal structure of the composite object must be triggered through the representative object's interface. Unfortunately the full encapsulation is well too restrictive and many common object-oriented idioms are impossible to implement in such systems.

The ownership type (***OT***) system introduced in [8] relaxes the restriction on the outgoing references. Each object is treated as a representative of a certain composite object, which is owned by the representative. Which objects constitute the owned composite object is specified by program annotations (ownership contexts). A tree-like ownership structure is established among the run-time objects and the system satisfies the owner-as-dominator property: All reference chains from the root object to any other object o (thus any reference chain leading inside the composite object to which o belongs) must pass through o object's owner (the representative). The owner-as-dominator property guarantees again that any modifications to the internal structure of the composite object must be triggered through the representative object's interface. Outgoing references are permitted, but only upwards in the ownership tree structure. Incoming references are still prevented by the type system. Although more flexible than full encapsulation, many popular design patterns cannot be implemented using ownership types. An often cited example is that of the Iterator pattern. A composite object – collection – often provides clients with an external iterator that allows the client to travers the elements stored in the collection. To move from one element to another, the iterator must be able to access the internal structure of the collection composite object. But because the ownership type system does not permit incoming references, the iterator must itself be part of the encapsulated composite object. This again prevents the external client to acces the iterator.

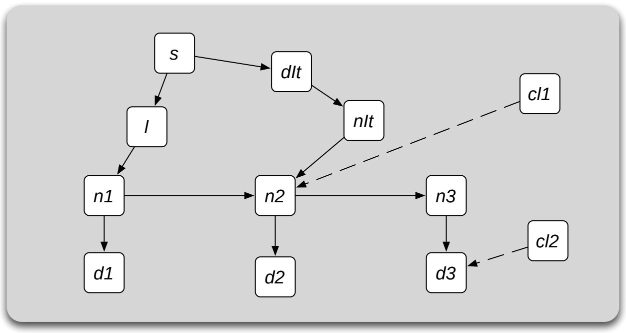
Systems that enforce the owner-as-modifier discipline [14], [17], [13] and [10] constitute a natural evolution of the ownership type system. We still have a tree-like hierarchy of object ownership and the mode annotations determine membership in the composite object. But unlike in the OT systems, the ownership information determines the legality of method calls. Arbitrary incoming and outgoing references are permitted. At the same time the owner-as-modifier property is satisfied: If an object o is modified (the composite object changes) then the change has been triggered through a sequence of method calls originating in the o's owner (the representative).

In this workshop paper we describe the ***Potential Access Pass*** methodology introduced in [17] as a tool to reason about composite objects and representation protection. We also put forward a novel weak uniqueness property for reference paths that generalizes the standard notion of free or unique references by allowing read-only aliases. This property provides us with an additional flexibility to transfer sub-components from one composite object to another.

Outline In section 2 we present an example – the *Set* composite object. Section 3 introduces the ***Potential Access Path*** methodology. In section 4 we present our language ***JaM*** and develop the formal mode-system. Section 5 provides the operational semantics for our language. Next we formally verify the CSE property (section 6). In section 7 we revisit the *Set* example – guiding the reader through the design and implementation process and and providing detailed *JaM* code. Related Work follows in section 8. We briefly conclude the paper with section 9.

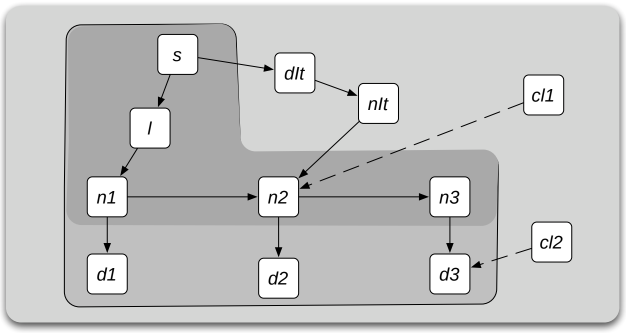
2. The Set Example

We will consider now the implementation of an abstract object *Set*. Our *Set* shall provide the standard methods *add*, *remove* and *contains*. We will also supply an external iterator that provides clients with the possibility to traverse all the *Set* elements. How would we go about implementing the *Set*? Let's assume that we have available a pre-existing *List* component (implemented as a composite object with a single linked list of nodes *ni* with data objects *di*, node iterator *nIt* and the representative *l*). We can use the *List* component to implement our *Set*. The node iterator can be used to implement an iterator over the data elements. Assuming three elements in our set, we end up with the following (run-time) object structure – *s* is the representative of the composite *Set* object and *dIt* an iterator over the set elements:



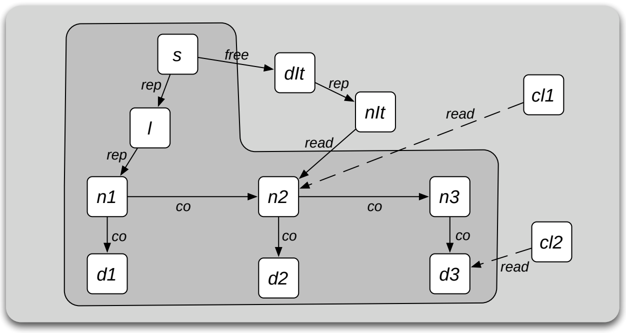
In general, it is possible that other objects obtain references into the above structure. Through these references the *Set* composite object might be modified. For example, an object *cl1* with a reference to *n2*, could send the message *setNext* to *n2* with `nil` as parameter, destroying the integrity of the *Set* composite object. What's worse, the representative object *s* will not even be aware about the changes. Such situation certainly must be prevented. In a different setting, an outsider, *cl2*, might obtain a reference to *d3*. Here it is not so clear, if *cl2* should be allowed to modify *d3*. It all depends on what elements are stored in the set. If the set is used by an online lottery to maintain a viewable list of winning numbers, it would be undesirable to allow some dishonest players to make modifications to the element *d3* (replacing legitimate number with one selected by the devious player). Only the owners of the set (e.g. lottery providers) should be allowed to make modifications to *d3*. On the other hand, if the set is used to keep track of players registrations, the participants should be permitted to make modifications to their registration data (update e-mail address, telephone, etc.). From the set (or lottery organizers) perspective, what matters is that the created registrations are preserved, not their contents. In both scenarios we want the representative object *s* to protect the integrity of the composite object *Set*. What differs is the extent of the protection.

What does it mean that *s* protects certain objects? We take the viewpoint that any changes to the state of these objects should be initiated by *s* itself. We will be referring to the set of objects protected by s as the ***composite*** of *s*. In our *Set* example, depending on the context, we have two different composites. (The dark grey area represents composite that corresponds to the set of player registrations; the larger boundary corresponds to the set of winning numbers.):

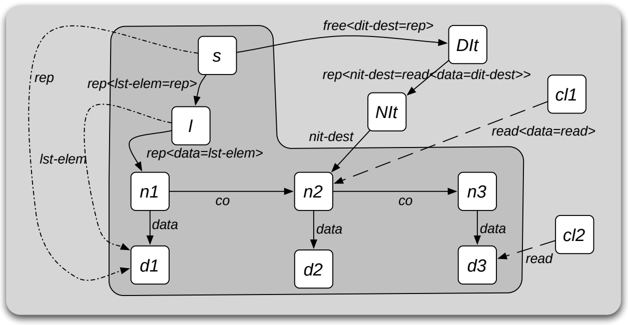


We still need to specify what it means that changes to these objects are initiated by *s*. To this effect we separate the methods of *s* into two kinds: ***observers*** and ***mutators***. When executing an observer on *s* we are guaranteed of no changes to its instance variables, whereas mutators have the right to modify them. But this is not enough. Observers shall never affect changes to instance variables of any object in the composite of *s*. As a consequence, for an object in *s*' composite to change its state, *s* must be executing a mutator – *s* is aware of the fact that it's composite state might change.

The next issue we have to address is: How can programmers specify membership inside a composite object? We borrow here from work of others [11], [15], [8] and use ***mode annotations*** on object's references. To begin with, we will use three kinds of modes: `rep`, `co` and `read`. The most important mode is `rep` - if a reference from an object o to an object ω is designated by programmer as `rep`, we put ω into o's composite. The `co` mode states that the two objects linked by a `co`-reference belong to the same composite object. The third mode, `read`, tells us that based on that particular reference no statement about composite membership of both objects can be determined. In our original example we could assign the following modes (if we want to model the winning numbers set – in the other case replace `co`-references from n to d with `read`-references):



Unfortunately, to address both *Set* scenarios, we would need to define the class *Node* twice - once with a `co`-reference to its data and in the second case with a `read`-reference. If we want to reuse components, the modes `rep`, `co` and `read` are not enough. We need more flexibility. To this effect we introduce an additional class of modes $\alpha \in \mathbb{A}$ and parameterize the base modes with correlations. Now our complete modes are of the form $\mu < \alpha = \mu' >$. The intuition is as follows: if *o* has a reference of mode $\mu < \alpha = \mu' >$ to ω and ω has a reference of mode α to another object ω', then *o* can potentially obtain a μ' reference to ω (via a series of method calls). An α-reference $o \xrightarrow{\alpha} \omega$ does not tell us anything about *o'* and ω's membership in a composite object. We need some external reference to *o*, to possibly determine ω's membership. The described ***association modes***[1] and ***correlations*** are crucial for the structural flexibility of the mode technique. They allow a class to fix the modes of references in its instances without fixing the reference targets' assignment to a composite object. This decision is postponed to each instance's clients. Hence the same class can be reused in many different structural contexts. In our example we obtain the following situation:



The above diagram tells us, that *l* could potentially obtain an `lst-elem`-reference to *d1*, and therefore *s* could obtain a `rep`-reference to *d1*. *s* could also obtain a `rep`-reference to *d2* (via *l*, *n1* and *n2*). Additionally we can infer that *dIt* (the data iterator) can obtain a `read`-reference to *n2* and `dit-dest`-reference to *d2*. Hence with the help of *dIt*, *s* can avail of a `rep`-reference to *d2* – this time via a different path.

To allow a safe transfer of sub-components, we introduce one additional mode: `free`. By assigning a `free`-reference from *o* to ω we state that ω belongs to a special part of *o*'s composite – its ***movables***. *o* has the right to transfer the sub-component represented by ω to another aggregate. Note, that ω can be aliased by other `read` references.

In section 7 we revisit this example and elaborate more on transfer of sub-components.

3. Potential Access Paths and Composite Objects

During the execution of object-oriented programs, new objects are created, old ones are destroyed and links between objects (through which messages can be exchanged) are established. The run-time system constitutes a graph with objects as nodes and references as edges. If the programmer has the option of annotating references with the previously introduced modes, these annotations will be reflected in the graph.

Our tool in reasoning about composite object protection are paths of references between two (not necessarily directly) connected objects.

Paths in a graph are non-empty sequences $\pi = h_1, \ldots, h_n$ of object references $h_i = o_i \xrightarrow{\mu_i} \omega_i$ with $o_{i+1} = \omega_i$ (also written $\pi = o_1 \xrightarrow{\mu_1} o_2 \ldots o_n \xrightarrow{\mu_n} o_{n+1}$). Among all the possible paths in a given graph, we are only interested in certain kinds, namely those that allow us to make judgements about membership in composite objects. We already discussed in section 2 how to arrive at the judgment for immediate paths of length one. Now let's turn our attention to paths that emerge from combination of two adjacent edges. We first look at the following path: $o \xrightarrow{rep} \omega \xrightarrow{co} \varphi$. ω belongs to *o*'s composite and therefore φ must also belong to *o*'s composite (remember the intuition behind *co*). We could as well imagine an inferred edge $o \overset{rep}{\dashrightarrow} \varphi$ in our graph. This inferred edge tells us directly, that φ belongs to *o*'s composite. In contrast to $o \xrightarrow{rep} \omega$, $o \overset{rep}{\dashrightarrow} \varphi$ does not tell us if *o* has direct access to φ. But potentially, *o* could obtain a direct access to φ if there is a method of φ that returns `this` and if φ is propagated then to *o* as a result of some ω method. Therefore we will refer to the inferred edge $o \overset{rep}{\dashrightarrow} \varphi$ as a ***potential access path***. As mentioned before, many of the paths in the graph will convey no meaningful information about composite object membership. From our perspective interesting access paths are defined as follows:[2]

$$\frac{< o, \mu, \omega > \in \mathfrak{g}}{\mathfrak{g} \vdash < o, \mu, \omega > \in PAP(o, \mu, \omega)}$$

$$\frac{\mathfrak{g} \vdash \pi_1 \in PAP(o, \mu, q) \quad \mathfrak{g} \vdash \pi_2 \in PAP(q, co, \omega)}{\mathfrak{g} \vdash \pi_1 \bullet \pi_2 \in PAP(o, \mu, \omega)}$$

For association modes with correlations we have the additional rule:

$$\frac{\begin{array}{c}\mathfrak{g} \vdash \pi_1 \in PAP(o, \mu\langle \ldots, \alpha = \mu', \ldots \rangle, q) \\ \mathfrak{g} \vdash \pi_2 \in PAP(q, \alpha <>, \omega)\end{array}}{\mathfrak{g} \vdash \pi_1 \bullet \pi_2 \in PAP(o, \mu, \omega)}$$

The modes `rep` (and `free`) not only decide about the composite object membership, but also allow us to make statements about the yet to be defined object ownership. If there is a reference $o \xrightarrow{rep} \omega$, then *o* is considered the owner of object ω (having complete control about the changes to ω's state). But the ownership property can extend beyond objects reachable directly via *rep*-references (consider the extensions via *co*- or α-references). We formalize these concepts with the following definition.

DEFINITION 1. *In a graph $\mathfrak{g}$ we call an object o the **owner** of an object ω iff there exists an ownership path from* o *to ω. The set of ownership paths from* o *to ω is defined as follows:*

$$Osh_{\mathfrak{g}}(o, \omega) = PAP_{\mathfrak{g}}(o, rep, \omega) \cup PAP_{\mathfrak{g}}(o, free, \omega)$$

*For each object o in $\mathfrak{g}$, the corresponding **composite object** is defined as:*

$$composite_{\mathfrak{g}}(o) = \{o\} \cup \bigcup_{Osh_{\mathfrak{g}}(o,\omega) \neq \emptyset} composite_{\mathfrak{g}}(\omega).$$

o is called the **representative** *of the composite object. If $\omega \in composite_{\mathfrak{g}}(o)$ then o **dominates** ω.*

In general it is possible that an object has more than one owner in a given object graph. This is counterintuitive to our understanding that the owner controls the changes to the state of objects in its composite. We lose this exclusive control right, if an object has more than one owner. It is desirable (and for our composite state encapsulation property essential) that all objects in a given object graph have a unique owner.

DEFINITION 2. *We say that an object graph $\mathfrak{g}$ has the **Unique Owner (UO)** property, $\mathfrak{g} \vdash UO$, iff $\forall o, \tilde{o}, \omega$. $(Osh_{\mathfrak{g}}(o, \omega) \neq \emptyset \wedge Osh_{\mathfrak{g}}(\tilde{o}, \omega) \neq \emptyset) \Rightarrow (\tilde{o} = o)$.*

[1] They are related to, but nevertheless different from ownership parameters.

[2] For technical reasons, the *PAP*'s are determined in the object graph to which inverses of all `co`-labeled edges are added

There might be more than one ownership path from o to ω, and in case of rep-paths, two of these paths may start with different references outgoing from o, e.g $o \xrightarrow{\texttt{rep}} \psi \xrightarrow{\texttt{co}} \omega$ and $o \xrightarrow{\texttt{rep}} \chi \xrightarrow{\texttt{co}} \varphi \xrightarrow{\texttt{co}} \omega$. This is fine, as long as the owner is unique. The situation changes in case of free paths. free-references were introduced to allow safe transfer of sub-components (after destructive read). In the previous situation when replacing rep with free we arrive at following paths: $o \xrightarrow{\texttt{free}} \psi \xrightarrow{\texttt{co}} \omega$ and $o \xrightarrow{\texttt{free}} \chi \xrightarrow{\texttt{co}} \varphi \xrightarrow{\texttt{co}} \omega$. We still might have a unique owner of ω, but this is not enough. Even if we read destructively the free-reference from o to χ, o will retain another free access path to ω (via ψ) and it would not be safe to pass this sub-component to another composite object. Here we need a stronger property.

DEFINITION 3. *We say that an object graph* $\mathfrak{g}$ *has the* ***Unique Head (UH)*** *property,* $\mathfrak{g} \vDash UH$, *iff* $\forall o, \tilde{o}, \omega, h, \pi, \tilde{h}, \tilde{\pi}\ .(\ h{\bullet}\pi \in PAP_{\mathfrak{g}}(o, \text{free}, \omega) \ \wedge\ \tilde{h}{\bullet}\tilde{\pi} \in Osh_{\mathfrak{g}}(\tilde{o}, \tilde{\omega})) \Rightarrow (\tilde{h} = h \ \wedge\ mult(h, \mathfrak{g}) = 1)$.

UH tells us that if we have multiple free paths to an object ω, the initial free-reference must be unique (thereafter we can have multiple co-paths leading to ω). Under such circumstances, after destructive read of the initial reference we are guaranteed that o does not own ω anymore.

Both UO and UH are properties of the object graph at a frozen point in time. When the graph changes, so might its compliance with UO and UH.

When an object ω executes a method f, we can find in the object graph a path of references through which a sequence of method calls leading to the call of f took place. When the method f is a mutator, the state of ω (and therefore the composite state of any object o to whose composite ω belongs) might change. We expect that o actually initiates the change. The next two properties help us with it.

DEFINITION 4. *The* ***Representative Control (RC)*** *property ensures that if ω (belonging to the composite object o) executes a mutator, then this mutator execution is nested inside a mutator execution on o. The* ***Mutator Control Path (MCP)*** *property ensures that a mutator on an object ω is always invoked via a sequence of calls along the edges of an ownership path to ω.* [3]

Notice, that o does not necessarily control the membership in its composite object – through temporary rep, co or free references in the execution of observers new paths can be established that add an object to $composite_{\mathfrak{g}}(o)$. Even though this addition is only temporary, it is a change of $composite_{\mathfrak{g}}(o)$ not necessarily controlled by o. The desired state encapsulation property does not require us to impose control on *temporary* additions since temporary members cannot be used to represent the composite's state. To represent state, only a core of composite object's members can be used that remains in the composite between method invocations.

DEFINITION 5. *For an object o, its* ***state representation*** *is defined as:*

$$strep_{\mathfrak{g}}(o) = composite_{\widehat{\mathfrak{g}}}(o),$$

where $\widehat{\mathfrak{g}} \subseteq \mathfrak{g}$ *is a subgraph containing references stored only in the instance variables of objects. The composite state of an object o is defined as*

$$compState_{\mathfrak{g}}(o) = \bigcup_{\omega \in strep_{\mathfrak{g}}(o)} state(\omega)$$

$strep_{\mathfrak{g}}(o)$ is a set of implementation objects which collectively represent the composite object's state by virtue of their shallow states.

[3] A formal definition of these properties can be found in [17]

At this point we state the main property of our system:

DEFINITION 6. *The* ***Composite State Encapsulation (CSE)*** *property ensures that if an execution step of a* JaM *program transforms an object graph* $\mathfrak{g}$ *into* $\mathfrak{g}'$ *such that* $compState_{\mathfrak{g}}(o) \neq compState'_{\mathfrak{g}}(o)$ *then o is executing a mutator.*[4]

One of the features of our system is the ability to move sub-components from one aggregate to another. The set of all objects belonging to a composite object can be divided into two parts, depending on the ability of the composite object to transfer these objects.

DEFINITION 7. ***Fixtures*** *is the subset of* $composite_{\mathfrak{g}}(o)$ *that is reachable via* rep *path sequences only:*

$$fixtures_{\mathfrak{g}}(o) = \bigcup_{PAP_{\mathfrak{g}}(o,rep,\omega)\neq\emptyset} \Big(\{\omega\} \cup fixtures_{\mathfrak{g}}(\omega) \Big)$$

Movables*, another subset of* $composite_{\mathfrak{g}}(o)$ *is defined as:*

$$movables_{\mathfrak{g}}(o) = composite_{\mathfrak{g}}(o) \setminus (fixtures_{\mathfrak{g}}(o) \cup \{o\})$$

Objects in $fixtures_{\mathfrak{g}}(o)$ can never be removed from the composite object. $movables_{\mathfrak{g}}(o)$ on the other hand contains objects, which can be safely transferred between different agregates (via destructive read of free references). However, the transfer can happen only as sub-components and not as single objects.

DEFINITION 8. *Let* $\omega \in composite_{\mathfrak{g}}(o)$ *and let* φ *be an object such that* $\mathfrak{g} \vdash \pi \in PAP(\varphi, \texttt{free}, \omega)$ *for some path* π. *Then* $composite_{\mathfrak{g}}(\omega)$ *is called a (movable)* ***sub-component*** *of o.*

4. Mode Checking in JaM

So far we did not introduce the syntax of our language ***JaM*** (Java with Modes). It is a fully orthogonal extension of a Java subset that classifies object references by mode annotations. To reduce the complexity of the formal treatment, we omit several nonessential features (e.g. static methods, subclassing). The grammar of *JaM* is defined below:

$$\begin{array}{rcl}
p \in P & ::= & D^* \\
d \in D & ::= & \texttt{class}\ \mathbb{C}\ \{(\mathcal{T}\ Id;)^*\ Meth^*\} \\
f \in Meth & ::= & \mathcal{K}\ \mathcal{T}\ Id((\mathcal{T}\ Id)^*)\ \{(\mathcal{T}\ Id;)^*\ S;\ \texttt{return}\ E\} \\
\kappa \in \mathcal{K} & ::= & \texttt{mut} \mid \texttt{obs} \\
t \in \mathcal{T} & ::= & \mathcal{M}\,\mathbb{C} \\
\mu \in M & ::= & (\texttt{rep} \mid \texttt{co} \mid \texttt{read} \mid \texttt{free} \mid \mathbb{A}) < \Delta > \\
\delta \in \Delta & ::= & (\mathbb{A} = M)^* \\
s \in S & ::= & S; S \mid V = E \mid \texttt{if}(E \odot E)\{S\} \\
 & & \mid \texttt{while}(E \odot E)\{S\} \\
o \in \odot & ::= & \texttt{==} \mid \texttt{!=} \\
e \in E & ::= & \texttt{val}(V) \mid \texttt{destval}(V) \mid \texttt{null} < \Delta > \\
 & & \mid \texttt{new} < \Delta >\ C() \mid E \Leftarrow Id(E^*) \\
v \in V & ::= & v \mid \texttt{this}.v
\end{array}$$

Notable difference from Java is the introduction of modes, marking of methods as mutator or observer and the explicit read operations (val and destval). All *JaM* programs stripped of their annotations are legal Java programs. Java programs can be translated into legal JaM programs by annotating all variable/parameter declarations with co, declaring all methods as mut and introducing the explicit non-destructive read operator val.

[4] A more formal definition of CSE can be found in section 6

In general, when executing *JaM* programs, the properties listed in section 3 will not hold. We rule out illegal *JaM* programs with the help of a mode-system, which is orthogonal to the standard Java type-system.[5] For space reasons we concentrate here on deriving the correct modes for expressions. When checking method definitions in the class c, we must verify that the method body is well typed (moded) and the result is of the same type as declared in the method signature. The verification of the method body happens with respect to a type/mode assignment Γ. It is constructed by assigning to `this` the mode `ref co`, and by assigning to each local variable/parameter with the type τ in the signature, the type `ref` τ.[6]

The typing judgment $\Gamma, \kappa \vdash e : \tau$ expresses that term e is legal inside a method of kind κ (`mut` or `obs`) and has static type τ in the context of type assumptions Γ for local variables and parameters. Selected rules for deriving types/modes in *base-JaM* are given below:

$$\frac{\Gamma, \kappa \vdash \nu : \texttt{ref}\ \tau \qquad \tau^* = \tau[\texttt{free} \mapsto \texttt{read}]}{\Gamma, \kappa \vdash \texttt{val}(\nu) : \tau^*}$$

$$\frac{\Gamma, \kappa \vdash \nu : \texttt{ref free}\ c \qquad \nu = \texttt{this}.y \Rightarrow \kappa = \texttt{mut}}{\Gamma, \kappa \vdash \texttt{destval}(\nu) : \texttt{free}\ c}$$

$$\frac{\vdash c\ \texttt{ok}}{\Gamma, \kappa \vdash \texttt{new}<\delta>\ c() : \texttt{free}<\delta>\ c}$$

$$\frac{\begin{array}{c}\Gamma, \kappa \vdash \nu : \texttt{ref}\ \tau \qquad \Gamma, \kappa \vdash e : \tau' \qquad \vdash \tau' \leq_m \tau \\ \nu \neq \texttt{this} \qquad \nu = \texttt{this}.y \Rightarrow \kappa = \texttt{mut}\end{array}}{\Gamma, \kappa \vdash \nu = e : \texttt{Cmd}}$$

$$\frac{\begin{array}{c}\Gamma, \kappa \vdash e : \mu\ c \qquad \vdash (f : \overline{\tau_i} \xrightarrow{\kappa^*} \tau) \in \Sigma(\mu\ c) \\ \kappa^* = \texttt{mut} \Rightarrow \kappa = \texttt{mut} \wedge \mu \neq \texttt{read} \\ \overline{\Gamma, \kappa \vdash e_i : \tau_i' \qquad \vdash \tau_i' \leq_m \tau_i}\end{array}}{\Gamma, \kappa \vdash e \Leftarrow f(\overline{e_i}) : \tau}$$

Non-destructive read of a variable ν is assigned the type τ of the reference stored in that variable. But this works only if the mode of the reference is `rep`, `co` or `read`. If the mode is `free`, we cannot do it. The copy of that reference could be then stored in another variable with the mode `rep` (as `free` can be converted to any other mode) and the UH property would be violated. We do not want to disallow a non-destructive read of `free` variables and therefore change in such cases the resulting mode to `read` (which is always safe).

We can destructively read only `free` variables (we would not gain anything by allowing it for `rep`, `co` or `read`). There is no restriction on reading local variables and parameters, but if the read variable is an instance variable, we can do it only inside mutators (we are setting the instance variable to `nil`, therefore modifying the state of the object executing the method).

In the creation expression we decided to specify the correlation set δ to be added to the `free` mode of its value. Although not necessary, it simplifies the formal treatment.

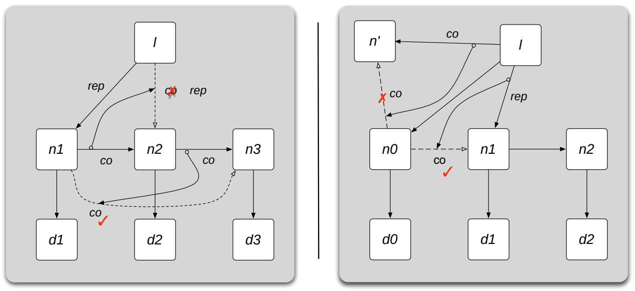
Assignment to instance variables is legal only inside mutators. Also, the mode of the value must be compatible with the mode of the variable.[7]

[5] We also disallow certain class of modes – see section 6

[6] In the full formal system the set of types is extended to include `ref` τ, so we can distinguish between values of type τ and variables that hold values of that type. As `ref` types are not visible to the programmer, we excluded them from the JaM syntax.

[7] We have μ<δ> $\leq_m$ `read`<δ>, `free`<δ> $\leq_\texttt{m}$ `rep`<δ>, `read`<$\delta, \alpha = \mu, \delta'$> $\leq_\texttt{m}$ `read`<δ, δ'> and `read`<$\alpha = \mu$> $\leq_\texttt{m}$ `read`<$\alpha = \mu'$> if $\mu \leq_m \mu'$. We elaborate more in section 6.

Method invocations $e \Leftarrow f(e_1, \ldots, e_n)$ are rather tricky. Assuming that f has the (mode) signature $\mu \longrightarrow \tilde{\mu}$ it is tempting to return to the sender of a message f as a result reference with the mode specified in the signature of f, namely $\tilde{\mu}$. But what the signature of f tells us is merely the mode of the temporary reference to the result that the receiving objects has. When this reference is passed from the receiver to the sender, that mode might have to be adjusted. Let's consider the following situation:



Object *l* invokes `next()` on `Node` *n1* which returns the `co` reference *n1* $\longrightarrow$ *n2*. The reference *l* $\longrightarrow$ *n2* which *l* obtains must not be a `co` reference, since *n2* must be owned by *l*. The return of the `co` reference can be better understood as the mode-preserving shortening of two-reference path *l* $\xrightarrow{\texttt{rep}}$ *n1* $\xrightarrow{\texttt{co}}$ *n2* to a one-reference path *l* $\xrightarrow{\texttt{rep}}$ *n2*. Should, on the other hand, the node *n1* call `next()` on its `co`-object *n2*, then the returned reference's mode is not adapted, since the return simply shortens `co` path *n1* $\xrightarrow{\texttt{co}}$ *n2* $\xrightarrow{\texttt{co}}$ *n3* to *n1* $\xrightarrow{\texttt{co}}$ *n3*. Analogously, the mode of references passed as parameters has to be adapted: If *l* has created a new `Node` object *n0* in its composite (to be included in the list structure), then it should supply to *n0*'s `setNext` operation (expecting a `co` reference) one of its `rep` references, namely *l* $\xrightarrow{\texttt{rep}}$ *n1*, and not a reference *l* $\xrightarrow{\texttt{co}}$ *n'* to a node that is a `co`-object in the same composite as *l*.

Consequently, two notions of signatures have to be distinguished:

Exported signatures The interfaces which all instances of a class c export have a signature $\Sigma(c)$ defined by the class. Its entries $f : \overline{\mu_i\ d_i} \longrightarrow \mu\ d$ specify the types of the parameter values which implementations of operation f expect to receive, and the type of the result values which they ensure to produce. Against this signature, the operations' implementations in class c are type-checked.

Imported signatures The interfaces which senders import through μ_r-references to c-objects have the signature $\Sigma(\mu_r c)$ with modes from c-objects' signature $\Sigma(c)$ adapted relative to call-link mode μ_r. Its entries $f : \overline{\mu_r \circ \mu_i\ d_i} \longrightarrow \mu_r \circ \mu\ d$ specify the types of the parameter values which the sender must ensure to supply, and the type of the result values which the sender can expect to obtain.

The method invokation rule type checks against the imported signature. The following rule provides us with the imported signature of a class c relative to the call link μ_r (here we show the simplified rule for modes without correlations):

$$\frac{\begin{array}{c}\vdash (f : \overline{\mu_i\ d_i} \xrightarrow{\kappa} \mu\ d) \in \Sigma(c) \\ \mu_i \neq \texttt{rep} \qquad \mu_i = \texttt{co} \Rightarrow \mu_r \neq \texttt{read}\end{array}}{\vdash (f : \overline{\mu_r \circ \mu_i\ d_i} \xrightarrow{\kappa} \mu_r \circ \mu\ d) \in \Sigma(\mu_r\ c)}$$

How do we read this rule? Our system disallows `rep` as parameter mode and permits calling of methods with `co` parameters only via references that are not `read`. If the call over the μ_r link is legal, the sender must supply an argument of mode $\mu_r \circ \mu_i$ (not just μ_i). The returned reference is viewed by the sender as $\mu_r \circ \mu$ (and not μ). The adaptation, called the ***import*** of μ through μ_r and written $\mu_r \circ \mu$, is defined as follows:

$$\mu_r \circ \texttt{read}{<}\alpha = \mu{>} = \texttt{read}{<}\alpha = \mu_r \circ \mu{>}$$
$$\mu_r \circ \texttt{free}{<}\alpha = \mu{>} = \texttt{free}{<}\alpha = \mu_r \circ \mu{>}$$
$$\mu_r \circ \texttt{rep}{<}\alpha = \mu{>} = \texttt{read}{<}\alpha = \mu_r \circ \mu{>}$$
$$\mu_r \circ \alpha{<>} = \mu' \quad \text{if} \quad \mu_r = \mu{<}\alpha = \mu'{>} \; .$$

Let's verify the plausibility of these definitions with respect to the result $\hat{\mu}$-reference returned via a μ_r-reference (we leave out correlations for now):

- $\hat{\mu} = \texttt{read}$: The receiver returns a `read` reference and therefore does not know anything about the targets owner. Without this information, the sender can accept it only as a `read` reference – any other choice would be unsafe. This is what $\mu_r \circ \texttt{read} = \texttt{read}$ gives us.
- $\hat{\mu} = \texttt{free}$: The sender can safely accept a `free` reference from the receiver as `free`, since it was the unique initial segment of ownership paths to all `co`-objects reachable through it, and all these old ownership paths are destroyed by the removal of the receiver's `free` handle from the graph (via destructive read). $\mu_r \circ \texttt{free} = \texttt{free}$ does it.
- $\hat{\mu} = \texttt{rep}$: If the receiver returns a `rep` reference, the receiver may still possess further `rep` handles with the same target, and thus remain the target's owner. Hence the sender cannot accept the handle as `free` or `rep` without risking a violation of unique ownership. Accepting it as `co` would make the sender a `co`-object of the target, and thus also owned by the receiver. This might raise a uniqueness conflict with any old owner of the sender. Only `read` is safe and $\mu_r \circ \texttt{rep} = \texttt{read}$ gives us the right mode.
- $\hat{\mu} = \texttt{co}$: If the returned reference is `co`, i.e., points to the receiver's `co`-object, the sender best accepts it with the mode μ_r of the call-link: If μ_r is `rep` or `free`, then the sender already had an ownership path to the target by concatenation of the call-link and the receiver's `co` handle. Hence it is reasonable to shorten it to a direct μ_r handle. In case of `free`, the accepted reference will replace the unstored `free` call-link as the unique initial edge of `free` ownership paths to the receiver and all its `co`-objects. If μ_r is `co` then sender and target were already `co`-objects through the call-link and the handle of the receiver, so that a direct `co`-handle is safe. And if μ_r is `read` then the accepted handle can only be `read`, since a `read` call-link gives the sender no information about the receiver's owner. Again $\mu_r \circ \texttt{co} = \mu_r$ is the right choice.

Similarly we consider the parameter passing mechanism. The receiver of a method expects a parameter of mode $\hat{\mu}$. The sender passes $\tilde{\mu}$ argument over μ_r-reference. The call rule tells us that $\tilde{\mu}$ must be compatible with $\mu_r \circ \hat{\mu}$. Is it sensible?

- $\hat{\mu} = \texttt{read}$: A parameter of mode `read` means that the receiver makes no assumptions about the target's place in the object graph. Hence the sender can supply references of any mode and any mode is compatible with $\texttt{read} = \mu_r \circ \texttt{read}$.
- $\hat{\mu} = \texttt{free}$: If the receiver expects a `free` parameter then only `free` references of the sender (which are destroyed in the call step) can guarantee the necessary uniqueness of the initial ownership path segments. $\mu_r \circ \texttt{free} = \texttt{free}$ does the trick.
- $\hat{\mu} = \texttt{rep}$: If the receiver expects a `rep`-reference then a reference to an object in the receiver's composite object must be passed. However, no mode on sender's reference can guarantee that the target is in the receiver's composite. Hence methods with `rep` parameters are disallowed.
- $\hat{\mu} = \texttt{co}$: A parameter of mode `co` means that the receiver expects a handle to an object with the same owner as itself. If the call-link is of mode $\mu_r = \texttt{read}$ then the sender has no information about the receiver's owner and thus cannot know which handle's target would have the same status. This situation is disallowed by the typing rule. The other call-links provide us with enough information about the owner of the receiver. If the call-link is of mode $\mu_r = \texttt{co}$, then sender and receiver have both the same owner. Therefore it is safe to pass a co-reference (the object at its end has again the same owner). And if the call-link is of mode $\mu_r = \texttt{rep}$ or `free` then respectively, `rep` or `free` handle of the sender guarantees that receiver and target have the same owner, namely the sender. In all three cases $\mu_r \circ \texttt{co} = \mu_r$ does the trick.

Notice that we are also required to recursively import the modes "hidden" in the correlations. A related discussion can be found in [17].

5. Operational Semantics

The formalization of the execution of *JaM* programs is provided in the style of *small-step semantics*. We take the standard approach, where transformation of program terms $(e, \vec{\eta}, s, om) \rightarrow (e', \vec{\eta}', s', om')$ is defined in the following three contexts: a dynamic stack $\vec{\eta}$ of ***environments*** $\eta_i \in Env$ that maps local identifiers to locations, a changing ***store*** $s \in \mathfrak{Store}$ that maps locations $\ell \in \mathcal{L}oc$ to values currently at these locations and a growing ***object map*** $om \in Omap$ that maps identifiers $o \in \mathbb{O}$ of created objects to object "values": tuples of field environments ρ_o (mapping field names to locations), and method suites F_o (mapping operation names to methods).

The reduction steps are the expected ones, but we include three non-standard features specifically for accommodating reasoning about composite objects:

- We formalize reference values as so-called ***handles***: A handle is not just the object-identifier ω of the target object, but a triple $h = < o, \mu, \omega >$ which also includes the identifier o of the source object and the mode μ of o's reference to ω.
- We record *call-links* (references through which method invocations are made) in the computational state (as annotations to the stack environments)
- Object graph is included (and manipulated) as an explicit fourth context $\mathfrak{g}$ of the term reduction rules.

The meaning of program p as a computational process is formalized now as a sequence of reduction steps $(e, \vec{\eta}, s, om, \mathfrak{g}) \Longrightarrow (e', \vec{\eta}', s', om', \mathfrak{g})$. The transformations starts with the expression $\texttt{new}{<>}\ c_n() {\Leftarrow} main()$ in the initial context $(\eta_0, \mathbf{s_0}, om_0, \mathbf{g_0}) = (\emptyset^{obs}_{<nil,read,nil>}, \emptyset, \emptyset, \emptyset)$. Each reduction step replaces in the term e one subterm, the *redex*, by another term. In particular, locations $\ell \in \mathcal{L}oc$ are substituted for identifiers x (using η) and for field names $\texttt{this}.x$ (using $\rho_{\texttt{this}}$). Values of variables are substituted for read access expressions (using s) and method bodies are substituted for operation call expressions (using om). Through these substitutions, the transformed terms are not just the statements and expressions of the program syntax, but belong to the larger category $\mathcal{R}$ of ***runtime terms***.[8]

The sources in all handles in the store and the runtime term should coincide with the object to which the corresponding store location or method nesting level belongs (***source consistency***). At

[8] Detailed syntax for $\mathcal{R}$ can be found in [17]

locations $\ell = \rho_o(x)$ of fields x of object o, we expect to find only handles $s(\ell) = h$ whose source is o. Then the object-map is source consistent, in symbols, $\models_s om$. Analogously, at locations $\ell = \eta_i(x)$ of local variables and parameters x in environments η_i of invocations with receiver $\mathtt{r}$, we expect to find only handles $s(\ell) = h$ whose source is $\mathtt{r}$. Then the environment is source consistent, $\models_s \eta$. And at all method nesting levels in the runtime term e with corresponding receiver $\mathtt{r}$, we expect to find only handles h with source $\mathtt{r}$, and locations ℓ containing handles $s(\ell) = h$ with source $\mathtt{r}$. If this is the case the runtime term is source consistent, in symbols, $\models_{s,\eta} e$.

We split the definition of reduction steps into two complementary aspects. On one side are sub-terms that can be completely substituted in one step to a new term. This substitution will be captured in redex replacement rules $(e, \vec{\eta}, s, om, \mathfrak{g}) \longrightarrow (e', \vec{\eta'}, s', om', \mathfrak{g})$. On the other side we must select a suitable sub-term for the next substitution. This selection can be conveniently specified with the help of a ***reduction context***. A reduction context $\mathcal{E}$ is a runtime term "with a hole" symbolized by '$\Box$'. A complete runtime term $\hat{e} = \mathcal{E}[e]$ is obtained by filling a term e into the hole. Reduction steps are then written $(\mathcal{E}[e], \vec{\eta}, \mathbf{s}, om, \mathbf{g}) \Longrightarrow (\mathcal{E}[e'], \vec{\eta'}, \mathbf{s}', om', \mathbf{g}')$ and performed according to the following reduction rule:

$$\frac{\mathcal{E} \in \mathcal{R}^{\Box} \qquad (e, \vec{\eta}, s, om, \mathfrak{g}) \longrightarrow (e', \vec{\eta'}, s', om', \mathfrak{g})}{(\mathcal{E}[e], \vec{\eta}, \mathbf{s}, om, \mathbf{g}) \Longrightarrow (\mathcal{E}[e'], \vec{\eta'}, \mathbf{s}', om', \mathbf{g}')}$$

A selection of redex replacement rules is given below.

$$\frac{s(l) =< o, \mu, \omega > \qquad \mu' = \mu[\mathtt{free} \mapsto \mathtt{read}]}{\begin{array}{l}(\mathtt{val}(l), \vec{\eta}, s, om, \mathfrak{g}) \\ \longrightarrow (< o, \mu', \omega >, \vec{\eta}, s, om, \mathfrak{g} \oplus o \xrightarrow{\mu'} \omega)\end{array}}$$

$$\frac{s(l) =< o, \mu, \omega >}{\begin{array}{l}(\mathtt{destval}(l), \vec{\eta}, s, om, \mathfrak{g}) \\ \longrightarrow (< o, \mu, \omega >, \vec{\eta}, s[l \mapsto < o, \mu, \mathtt{nil} >], om, \mathfrak{g})\end{array}}$$

$$\frac{\begin{array}{l}h =< s, \mu, r > \quad \vdash VarMths(c) = \langle \{\overline{x_i : \mathtt{ref}\ t_i}\}, F\rangle \\ \mathtt{fresh}\, o \in \mathbb{O}_c \quad \mathtt{fresh}\, l_i \in [\![\mathtt{ref}\ \mu_i\ c_i]\!] \\ \rho = \{\overline{x_i \mapsto l_i}\} \quad h_i =< o, \mu_i, \mathtt{nil} >\end{array}}{\begin{array}{l}(\mathtt{new}{<}\delta{>}\ \ c(), \vec{\eta} \bullet \eta_h^{\kappa}, s, om, \mathfrak{g}) \longrightarrow \\ (< r, \mathtt{free}{<}\delta{>}, o >, \vec{\eta} \bullet \eta_h^{\kappa}, s[l_i \mapsto h_i], \\ om[o \mapsto < \rho, F >], \mathfrak{g} \oplus r \xrightarrow{free} o)\end{array}}$$

$$\frac{\begin{array}{l}r \in \mathbb{O}_c \quad om(r) =< \ldots, F > \\ F(f) = \kappa^*\ \tau\ f(\overline{\mu_i\ c_i\ y_i})\{\overline{\mu'_j\ c'_j\ z'_j}; s; \mathtt{return}\, e\} \\ \mathtt{fresh}\, l \in [\![\mathtt{ref\ co}\ c]\!] \quad \mathtt{fresh}\, l_i^y \in [\![\mathtt{ref}\ \mu_i\ c_i]\!] \\ \mathtt{fresh}\, l_i^z \in [\![\mathtt{ref}\ \mu'_i\ c'_i]\!] \\ \hat{\eta} = \{\mathtt{this} \mapsto l, y_i \mapsto l_i^y, z_j \mapsto l_j^z\} \\ s' = s[l \mapsto < r, \mathtt{co}, r >, l_i^y \mapsto < r, \mu_i, o_i >, \\ \qquad l_j^z \mapsto < r, \mu'_j, \mathtt{nil} >] \\ \mathfrak{g}' = \mathfrak{g} \ominus s \xrightarrow{\mu"} o_i \oplus r \xrightarrow{co} r \oplus r \xrightarrow{\mu_i} o_i\end{array}}{\begin{array}{l}({<}s, \mu_r, r{>} \Leftarrow f(< s, \mu_i", o_i >), \vec{\eta}, s, om, \mathfrak{g}) \longrightarrow \\ (\ll s; \mathtt{return}\, e \gg, \vec{\eta} \bullet \hat{\eta}^{\kappa^*}_{<s,\mu_r,r>}, s', om, \mathfrak{g}')\end{array}}$$

$$\frac{l \in Loc_{\mu\ c}}{\begin{array}{l}(l =< o, \tilde{\mu}, \tilde{\omega} >, \vec{\eta}, s, om, \mathfrak{g}) \longrightarrow \\ (\varepsilon, \vec{\eta}, s[l \mapsto < o, \mu, \tilde{\omega} >], om, \mathfrak{g} \ominus o \xrightarrow{\tilde{\mu}} \tilde{\omega} \ominus s(l) \oplus o \xrightarrow{\mu} \tilde{\omega})\end{array}}$$

$$\frac{\begin{array}{l}(s' = s[l \mapsto \bot \mid l \in im(\hat{\eta})] \\ \mathfrak{g}' = \mathfrak{g} \oplus s \xrightarrow{\mu_r \circ \mu} \omega \ominus s \xrightarrow{\mu_r} r \ominus r \xrightarrow{\mu} \omega \ominus s(im(\hat{\eta}))\end{array}}{\begin{array}{l}\ll \mathtt{return}\, {<}r, \mu, \omega{>} \gg, \vec{\eta} \bullet \hat{\eta}^{\kappa^*}_{<s,\mu_r,r>}, s, om, \mathfrak{g}) \longrightarrow \\ ({<}s, \mu_r \circ \mu, \omega{>}, \vec{\eta}, s', om, \mathfrak{g}')\end{array}}$$

Below we provide the rationale for the given semantics rules (explanation for graph modifications is provided separately):

- Non-destructive read access $\mathtt{val}(l)$ copies the value from the store (at location ℓ) to the runtime term (at the redex position). This value is always a handle $< o, \mu, \omega >$. In case of a `free` handle, an exact copy would immediately violate UH. The copy is safe if its mode is weakened to `read`.
- Destructive read access $\mathtt{destval}(l)$ evaluates to the value at location ℓ, but resets the store at ℓ to a `nil`-handle.
- An object creation expression instantiates the class c to a new object with a fresh object-identifier $\mathtt{o}$. It evaluates to a `free` handle from the current (creator) object $\mathtt{r}$ to the new object $\mathtt{o}$. Instantiating c also involves the initialization of fresh locations ℓ_i of respective types $\mathtt{ref}\ \mu_i\ c_i$, to `nil`-handles with source $\mathtt{o}$ and modes μ_i. Furthermore $\mathtt{o}$ is mapped to an object value $< \{\overline{x_i \mapsto l_i}\}, F >$.
- A method invocation is executed after all its subexpressions, arguments and the receiver have evalueted. The execution continues with the body $\ll \mathtt{s;\ return\ e} \gg$. The newly created environment contains `this`, parameters and local variables. They are bound to fresh locations of corresponding `ref`-types. These locations are initialized to: a handle to the receiver (of mode `co`), argument expression values adapted to the parameters' modes, and `nil`-handles of the local variables' modes.
- An assignment statement is executed after the left-hand side has reduced to a location ℓ and the right-hand side to a value $< o, \mu', \omega' >$. It updates the store at ℓ to the handle with the mode adapted according to the location's store partition.
- A `return` statement is executed after its return expression has evaluated to a result handle. Evaluation continues in the environment $\vec{\eta}$ with the result handle adapted to the calling context, i.e., with the sender as the new source and with a mode adapted to the sender's perspective. The current top-level environment is removed from the stack and the locations of the names in it (parameters, locals, and `this`) are removed from the store.

The object graph is formalized as a *multiset* $\mathfrak{g} \in \mathbb{N}^{\mathbb{O} \times \mathcal{M} \times \mathbb{O}}$ of edges whose multiplicity represents the *number* of the corresponding handlesoccurrences in s, η, or e. The multiplicity of edges is increased and decreased in accordance with the addition and removal of handles to/from e, η and s.

We will examine now some of the graph transformations during different reduction steps.

- Non-destructive read increases the multiplicity of the handle $h =< o, \mu', \omega >$. This models the redex's substitution to h, which increases the number of h's occurences in the term.
- Destructive read of a (free) variable leaves the object graph unchanged: The new occurence of handle $h =< o, \mu, \omega >$ in the term is balanced by removing one occurence from the store.
- New object creation adds creator object $\mathtt{r}$'s initial reference to the new object ω to the object graph: $\mathfrak{g}' = \mathfrak{g} \oplus < \mathtt{r}, \mathtt{free}, \omega >$. This models the redex's substitution to $< \mathtt{r}, \mathtt{free}, \omega >$.
- Method invocation equips the receiver with a new `this` reference $< \mathtt{r}, \mathtt{co}, \mathtt{r} >$ and with a parameter handle $< \mathtt{r}, \mu_i, \omega_i >$ for every argument handle $< \mathtt{s}, \mu''_i, \omega_i >$ supplied by the sender. That is, the multiplicity of $< \mathtt{r}, \mathtt{co}, \mathtt{r} >$ and edges $< \mathtt{r}, \mu_i, \omega_i >$ increases, while that of edges $< \mathtt{s}, \mu''_i, \omega_i >$ decreases. This matches the arguments' disappearance from the term and the parameters' and the `this`-reference's appearance at fresh locations in the store. The call-link $< \mathtt{s}, \mu_r, \mathtt{r} >$ is

not changed. Its disappearance from the term is balanced by its occurence in the new top-level environment.

- Variable update converts a handle $< o, \tilde{\mu}, \tilde{\omega} >$ to $< o, \mu, \tilde{\omega} >$, i.e., decreases the multiplicity of the first handle and increases that of the second one. This matches, respectively, the disappearence of the right-hand side handle $< o, \tilde{\mu}, \tilde{\omega} >$ from the term and the appearance of the handle $< o, \mu, \tilde{\omega} >$ at location ℓ in the store. Additionally, the multiplicity of the old handle $< o, \mu, \omega >$ at location ℓ decreases since the update at location ℓ overwrites it.
- Method return combines the call-link $< \mathtt{s}, \mu_r, \mathtt{r} >$ and the return handle $< \mathtt{r}, \mu, \omega >$ to the edge $< \mathtt{s}, \mu_r \circ \mu, \omega >$ in the sender, i.e., the former two edge's multiplicity decreases while the latter one's multiplicity increases. This matches the appearence of $< s, \mu_r \circ \mu, r >$ in the runtime term and the disappearence of handle $< \mathtt{r}, \mu, \omega >$ from the term and of call-link $< s, \mu_r, \mathtt{r} >$ (together with the finished invocation) from the environment stack. Additionally, since the locations of the finished invocation's variables in the store are reset, the multiplicities of all (non-`nil`) handles lost by this are decreased to keep the object graph in sync.

For the implementations of the *JaM* language, no representation of the object graph at runtime is needed. The graph has no impact on the computation and is invisible from outside of the program. It can actually be calculated from the other run-time contexts. We included it in the rules to make the nature of transformations more obvious.

6. Verifying Run-Time Properties

The reduction rules are a tool that enables us to establish the properties that we expect to hold during execution of legal *JaM* programs. (In the following, $e_0 = \mathtt{new}<> \ c_n() \Leftarrow main()$, is the initial expression of a legal program p.)

The ownership paths in all object graphs reachable in the execution of legal *JaM* programs satisfy the Unique Owner and Unique Head integrity invariants.

THEOREM 1. *If* $(e_0, \vec{\eta_0}, \mathbf{s}_0, om_0, \mathbf{g}_0) \Longrightarrow^* (e, \vec{\eta}, \mathbf{s}, om, \mathbf{g})$ *then*

$$\mathfrak{g} \models UH, UO$$

The structure of mutator access as recorded in the environment stack during the execution of legal *JaM* programs is always consistent with ownership paths as captured in the integrity invariants Representative Control and Mutator Control Path.

THEOREM 2. *If* $(e_0, \vec{\eta_0}, \mathbf{s}_0, om_0, \mathbf{g}_0) \Longrightarrow^* (e, \vec{\eta}, \mathbf{s}, om, \mathbf{g})$ *then*

$$\mathfrak{g}, \vec{\eta} \models \mathsf{RC} \ \text{ and } \ \mathfrak{g}, \vec{\eta} \models \mathsf{MCP}$$

The following theorem is the main result, which establishes the *Composite State Encapsulation* property.

THEOREM 3. *Let* $(e_0, \vec{\eta_0}, \mathbf{s}_0, om_0, \mathbf{g}_0) \Longrightarrow^* (e, \vec{\eta}, \mathbf{s}, om, \mathbf{g}) \Longrightarrow (e', \vec{\eta'}, \mathbf{s}', om', \mathbf{g}')$. *Then for all* $o \in \mathrm{dom}(om)$:

$$compState_{\mathfrak{g}}(o) \neq compState_{\mathfrak{g}'}(o)$$
$$\Rightarrow \exists i \leq n \bullet \ \mathtt{r}_i = o \ \wedge \ \kappa_i = \mathbf{mut},$$

where $\vec{\eta} = {\eta_1}_{h_1}^{\kappa_1}, \ldots, {\eta_n}_{h_n}^{\kappa_n}$ *with* $h_i =< \mathtt{s}_i, \mu_i, \mathtt{r}_i >$.

The theorem simply states that if a state of a composite object (represented by o) changes, then the representative o is executing a mutator.

The proofs for the first two theorems are by induction on the number of reduction steps from e_0 to e. Once we establish these properties (plus several helpful lemmas), the proof for Composite State Encapsulation is straightforward.

Complete set of proofs for *JaM* and *base-JaM* (without association modes and correlations) can be found in [17]. Although often tedious and lengthy, proving these results in *base-JaM* is fairly manageable. Things change, when introducing association modes with correlations. While the formal treatment of many *JaM* properties is a simple forward adaptation from *base-JaM*, the proofs of the unique owner and unique head invariants must be redone completely. Potential access paths in *JaM* have much more complicated structure than in base-*JaM*. The possibility of extending μ-paths to non-μ paths is the culprit. We were forced to restrict permissible association modes and their correlations.

- We don't consider extensions $o \overset{\mathtt{co}}{\dashrightarrow} q \overset{\alpha}{\dashrightarrow} \omega$ and $o \overset{\alpha'}{\dashrightarrow} q \overset{\alpha}{\dashrightarrow} \omega$ of `co`- and association paths by association paths. Also the extension $o \overset{\mu}{\dashrightarrow} q \overset{\alpha}{\dashrightarrow} \omega$ of potential access paths by association paths to `co`- and `free` paths $o \overset{\mathtt{co}}{\dashrightarrow} \omega$ and $o \overset{\mathtt{free}}{\dashrightarrow} \omega$ is disallowed. This simplification is reflected in constraints on the nesting structure of mode-terms. Only modes `free`, `rep` and `read` are parameterized by correlations ($\mathtt{free}{<}\delta{>}$, $\mathtt{rep}{<}\delta{>}$ and $\mathtt{read}<\delta>$ are legal, but $\mathtt{co}<\delta>$ and $\alpha<\delta>$ are not). Also, only correlations to `rep`, `read` and association modes are permitted ($\mu < \alpha = \mathtt{rep} < \delta >>$, $\mu < \alpha = \mathtt{read} < \delta >>$ and $\mu{<}\alpha = \gamma{<}{>}{>}$).
- Implicit mode-conversions from $\mathtt{free}<>$ to $\mathtt{co}<>$ or $\alpha<>$ caused by assignment or parameter passing is disallowed. (Tedious invariants about all sequences of association paths starting from targets of `free<>` paths are needed in order to show that such conversions preserve the uniquness of ownership.) This simplification is reflected in the definition of the mode compatibility relation $\leq_m$.

At this point we want to comment on the sub-mode rules introduced in section 4. There we constrain the width- (more or fewer correlations) and depth- (correlations with compatible modes) compatibility between modes. Without this restriction it would be possible for an object to convert `read` references to `rep`. We could weaken a `rep<data=rep<>>` reference to ω to a `rep<data=read<>>` reference. Through this reference the source could store a `read` reference in ω as a `data` reference and read it back through the original reference as a `rep`. The same scenario can be set up using width-compatibility. Two distinct `rep<data=rep<>>` and `rep<data=read<>>` references to ω could be converted to the same mode `rep<>` and then linked by a `co`-reference. By reading it back through the original references the source can obtain, as with depth-compatibility, a `rep<data=rep<>>` reference and a `rep<data=read<>>` reference to the same object. Depth- and width- compatibility in *JaM*'s type system exists only between `read` modes. The `read` modes are compatible because through converted `read` references nothing can be stored in the target (since only observers can be called on the target).

7. The Set Example Revisited

We take a look now at concrete implementation of our *Set* composite object. The relevant interfaces are: `Iterator<T>`, `List<T>` and `Set<T>`. The types of variables, parameters and results are prefixed by our mode annotations (e.g. `rep Node<T>`).[9] We don't show `val`

[9] Although generic types are not part of our syntax, we use them in the example. This does not really affect our system, as the mode annotations are completely orthogonal to the standard Java types.

and destval in our code.[10] We also use void methods, which are not declared in our syntax.[11]

```
interface Iterator<T>{
   void step();
   dest T current();
}

interface List<T>{
   void add(lst-elem T e);
   void remove(read T e);
   lst-elem T contains(read T e);
   free<nit-dest=lst-elem> Iterator<Node<T>>
      getNodeIter();
}

interface Set<T>{
   void add(set-elem T e);
   void remove(read T e);
   set-elem T contains(read T e);
   free<dit-dest=set-elem> Iterator<T>
      getDataIter();
}
```

Inspecting the add method of the List<T> interface we see, that to add an element to a List, we must supply an element that has the same composite membership as the other list elements. Notice, that we are not saying anything about what the membership will be (e.g in the composite of the list object itself or in some other composite). This will depend on the context in which List<T> instantiations are used. The contains method tells us, that when we look for an element in the list, we can supply any element without worrying about its composite membership. But if we find this object, we return it with the information that it belongs to the same composite as all other list elements. The getNodeIter method returns an iterator that can be passed to other components (this is the meaning of free). At the same time we specify that this iterator's destination is in the same composite as the list elements (this is what the correlation <nit-dest=lst-elem> tells us). In the case of the Set<T> interface we can extract similar information: we can only add elements that belong to the same composite as all the other set elements. The set iterator can again be passed to other components, and the iterator's destination is in the same composite as all the set elements.

The *List* abstract object will be implemented with objects of the class Node<T>:

```
class Node<T>{
  co Node<T> next;
  data T value;

  void setNext(co Node<T> n){
    this.next = n;
  }

  void setValue(data T p){
    this.value = p;
  }

  co Node<T> getNext(){
    return this.next;
  }

  data T getValue(){
    return this.value;
  }
}
```

Here we notice that the next link points to an object in the same composite as the referring node (co Node<T> next). The values stored in our nodes belong to some yet unspecified composite data. The signature of the setNext method tells us, that we must provide an object belonging to the same composite as the node executing that method.

Next we implement a *node iterator*, that allows us to traverse nodes contained in some structure:

```
class NodeIt<T> implements Iterator<Node<T>>{
   nit-dest Node<T> curnode;

   void startAt(nit-dest Node<T> n){
      this.curnode = n;
   }

   void step(){
      this.curnode = this.curnode<=getNext();
   }

   nit-dest Node<T> current(){
      return this.curnode;
   }
}
```

Here the iterrator points to a current node that belongs to some composite nit-dest. To set up the initial point of the node traversal, we need to supply a node that belongs to that nit-dest composite.

We use the *node iterator* to implement an iterator DataIt<T> that traverses not the nodes themselves, but the values stored in the nodes:

```
class DataIt<T> implements Iterator<T>{
   rep<nit-dest=read<data=dit-dest>>
      Iterator<Node<T>> nIt;

   void wrap(free<nit-dest=read<data=dit-dest>>
                     Iterator<Node<T>> newnIt){
      this.nIt = newnIt;
   }

   void step(){
      this.nIt<=step();
   }

   dit-dest T current(){
      dit-dest T res;
      if (this.nIt<=current() != null){
         res = this.nIt<=current()<=getValue();
      }
      return res;
   }
}
```

This iterator "wraps" the *node iterator*. It puts the internal node iterator nIt into the composite controlled by the *data iterator* (the base mode rep does it). Any changes to the state of nIt can only be initiated on *data iterator*'s instigation. At the same time we specify that we don't expect any information about nit-dest composite

[10] The compiler can deduce, based on the context, which read operation should be used.

[11] We can view them as syntactic sugar for methods that return this as result and assign it right back to the variable trough wich the method was invoked. This is particularly helpful, if we try to send mutators over free references without losing them.

(setting `<nit-dest=read<...>>` says exactly this). The method `current` returns objects in `dit-dest` composite.

Now we are ready to direct our attention to the *List* implementation:

```
class ListImp<T> implements List<T>{
   rep<data=lst-elem> Node<T>  anchor;

   void add(lst-elem T e){
      rep<data=lst-elem> temp;

      temp = anchor;
      this.anchor = new<data=lst-elem> Node<T>();
      this.anchor<=setData( e );
      this.anchor<=setNext( temp );
   }

  void remove(read T e){
     ...
  }

  lst-elem T contains(read T e){
     ...
  }

  free<nit-dest=rep<data=lst-elem>> Iterator<T>
  getNIt(){
     free<nit-dest=rep<data=lst-elem>> NodeIt<T>
        nIt;

     nIt = new<nit-dest=rep<data=lst-elem>>
              NodeIt<T>()<=startAt(this.anchor);
     return nIt;
  }
}
```

The `anchor` points to the initial node that is put into the list's composite (via `rep`). Concurrently we specify that objects in that node's `data` composite belong to list's `lst-elem` composite. `getNIt` provides a *node iterator* over the node structure. The iterator belongs to *List*'s movables (`free` mode) and therefore can be safely passed to other composite objects (e.g the `SetImp<T>`). Also, the iterator's destination objects (the nodes) belong to list's composite (via `rep`).

The remaining part of the puzzle is the `SetImp<T>` class implementation:

```
class SetImp<T> implements Set<T> {
   rep<lst-elem=set-elem> T  entryList;

   void add(set-elem T e){
      if (entryList<=contains(e) == null){
         entryList<=add(e)
      }
   }

   set-elem contains (read T e){
      return entryList<=contains(e)
   }


   void remove(read T e){
      entryList<=remove(e)
   }


   free<dit-dest=set-elem> Iterator<T> getDIt(){
      free<nit-dest=read<data=set-elem>>
         NodeIt<T> nIt;
      free<dit-dest = set-elem>  dIt;

      nIt = entryList<=getNIt();
      dIt = new<dit-dest=set-elem> DataIt();
      return dIt<=wrap(nIt);
   }
}
```

The *Set* is implemented with the help of a *List* component. The list is put into set's composite, so only the set can make changes to list's structure. Even if some other objects have references to the nodes, they cannot send `setNext()` to them. The list elements (in the composite `lst-elem`) end up in the composite `set-elem`.

In the following code we define the class `OnlineLottery` that holds both, the winning numbers and the players' registrations. `winningNumbers` are in `OnlineLottery`'s composite and therefore cannot ever be transferred to another composite object. `playersReg` on the other hand is part of `OnlineLottery`'s movables and can at any point be "sold" to another "lottery enterprise".

```
class OnlineLottery {
   rep<set-elem=rep> SetImp<NUM>  winningNumbers;

   free<set-elem=reg> SetImp<REG>  playersReg;

   free<dit-dest=rep> Iterator<NUM> getWinIt(){
      return winningNumbers<=getDIt();
   }

   free<dit-dest=reg> Iterator<REG> getRegIt(){
      return playersReg<=getDIt();
   }

   void newDraw() {
      rep<dit-dest=rep> Iterator<T> internalIt
         = this<=getWinIt();

      while (internalIt<=current() != null) {
         internalIt<=current()<=setNum(random);
         internalIt<=step()
      }
   }

   free<set-elem=reg> SetImp<REG> sellRegSet() {
      return destval(playersReg);
   }

   read<set-elem=reg> SetImp<REG> exposeRegSet() {
      return val(playersReg);
   }
}
```

`OnlineLottery` class provides two iterators `getWinIt()` and `getRegIt()`, which either iterate other the set of winning numbers or the set of registrations. The `getWinIt()` iterator, when used internally (e.g. in the `newDraw` method), returns `rep` references that can be modified. When requested by an external client, the exported mode is adapted to `free<dit-dest=read>` allowing only read acces to the numbers.[12] Player's registrations belong to some composite `reg` and therefore `getRegIt()` iterator returns immutable references to an `OnlineLottery` object. On the other hand, clients that own `reg`, will receive mutable references from `getRegIt()`.

[12] See the import operation in section 4

Our `OnlineLottery` objects can sell the registration set, `playersReg`, to another gambling provider. Notice the `destval` in `sellRegSet`. The new owner can modify the `sellRegSet` (adding and removing registrations), but cannot change the content of any registrations (only the owner of the reg composite can). We can also expose the `playersReg` set, but the value of such operation is not clear here (the set cannot be modified - we can only obtain an iterator from such set, which is identical to the iterator obtained via `getRegIt()`).

If we want the `OnlineLottery` to own both, the list of registrations and the registrations themselves and later on transfer the ownership of the list and the registrations to another provider, we might be tempted to declare `playersReg` with the mode `free<set-elem=free>`. Unfortunately we cannot do it – our system does not allow it. We simply cannot guarantee the uniqueness of the references to the set elements (e.g. we could repeatedly call observers that return "`free`" references to the same set element and every-time store them in a different `free` variable of `OnlineLottery`). To make the described transfer feasible, we would need to modify the `Node` class. The mode of `value` needs to be change to `free`. The `OnlineLottery` (or `playersReg` set) owns now the registrations indirectly through the nodes, which are the direct owners of the registrations). The `DataIt` must be changed now as well, returning a `read` reference (via `val` – not `destval`).

8. Related Work

Blake and Cook were the first to characterize the problem of composite object encapsulation [4]. They warned that the common handing out of references to part objects enables clients to modify them in a way violating the integrity of the whole.

The Islands approach [11] proposes three techniques for making object interaction more predictable: the observer/mutator distinction, the uniqueness of certain references and the isolation of specific regions in the object graph (Islands). The work also contributes a system of access mode annotations with `read`, `unique` and `free`. `read` references cannot be assigned to variables but they can be bound to parameters. Island's `free` indicates references to whose target no other reference exist. `unique` is a variation on `free` with temporary aliases. Only un-captured references are allowed in or out of Islands. They must be either `read` or aliases of `unique`.

Flexible Alias Protection (FAP) [15] is another approach towards encapsulation of composite objects. FAP addresses the coupling caused through sharing of mutable state by a two-pronged strategy: the absence of all inbound references into composite objects representation and the independence of container objects from their contents' state. It is the first system to introduce the `rep` mode, which describes references from an object to its state-representing components. The ability to specify `rep` references by some kind of annotation is fundamental to nearly all typing disciplines for composite object encapsulation. FAP also introduces association roles α, for a user-defined classification of object references according to different semantic roles.

Ownership Types (OT) [8] was the first system of composite object encapsulation presented with complete formal definitions (typing rules, interpretation of annotations, encapsulation property) and a proof sketch. The authors introduced the graph-theoretical notion of dominator to define the relaxed hiding policy of representation containment. The concept of `co` references was introduced in OT (under the name `owner`). The authors also observed the importance of `co` for the proper typing of `this`. α roles from FAP reappear here as *context parameters* to the class. Like any hiding policy, OT excludes iterators and other common patterns. Some of the OT descendants and variations are [6], [5], [1], [12] and [16].

The Calculus in [9] is an ambitious foundational work on the isolation of regions in the object graph with several technical innovations. The OT system is generalized to cover the missing language features and make it more flexible. The formalization is done with the help of an object calculus. The decisive step towards more flexibility was to loosen the connection between the structure of object composition and the nesting of protection domains, the ownership contexts.

Universes [14] is the first technique that enforces a policy of encapsulation without hiding. Universes simplify OT by replacing OTs problematic context parameters by runtime ownership checks. Universes prevent flexible object creation and composition by fixing new objects owner always to their creator.

AliasJava [2] is characterized as a capability-based system. It combines aliasing annotations with ownership annotations. It makes aliasing patterns explicit and enforces a relaxed hiding policy. The authors are the first to develop a constraint-based algorithm for inferring the new annotations, and the first to report on the usability of their system for real-world software like Java's standard library. A drawback of AliasJava is its need to represent ownership parameters at runtime.

The work most closely related to ours is [13] and [10] (the later evolved from [14]). [13] introduces a novel type system "Effective Ownership Types" (EOT). Each method definition is provided with effective owners. A method owned by o can only update objects with an owner that dominates o. The ownership tree is established in the same manner as in OT. The static type system tracks down the unsafe mutations. As in our system, object references and non-mutating access are unrestricted. OT system is a special case of EOT, where all methods belong to the owner of the defining object. Unlike in our system, mutator calls via inside-out references are permitted. EOT also can express mutating iterators. Such iterator carries a reference to its collection object and can therefore add and delete elements by making calls on the collection's interface. This is not possible in our system. Our system allows the safe transfer of sub-components (inside the *movables*) from one composite object to another. In EOT the owner of an object is fixed for its lifetime and transfer of sub-components is not possible.

Transfer of ownership has been first described in [7]. The authors introduce the concept of "external uniqueness". Here `unique` describes the only reference into an aggregate from outside the aggregate. Internal aliases to a unique reference are permitted. The authors work in the owner-as dominator setting. In *JaM*, `free` references can have arbitrary `read` aliases and the `free` reference as well as its aliases can be captured in variables.

Ownership transfer is not possible in the Universe Types system [14], [10]. As in our system `read-only` references (or `any` in [10]) are allowed to cross the boundary of encapsulation. In both cases modification of objects through such references is disallowed. [14] cannot produce iterators that deliver mutable objects (a dynamic downcast from `read` to `rep` is required). This has been rectified in [10]. The Viewpoint Adaptation in [10] is closely related to our Signature Import. In our system we have to deal additionally with `free` modes. Without them Viewpoint Adaptation and Signature Import appear almost identical.

9. Conclusion

We presented the *Potential Access Path* methodology as a toll to reason about composite objects and their state. The main technical result of this paper is the *Composite State Encapsulation* property – a guarantee that modifications to composite object's state are controlled solely by its representative.

Our system enables the definition of nested composite objects with a complex internal structure, their observation through external iterator objects, their incremental construction, and their transfer across abstraction boundaries. It is a purely static system in which container objects and their iterator objects can each be encapsulated individually (state-protected from one another).

The flexibility of our system results from a novel weak uniqueness property for reference paths. That property generalizes the standard notion of free or unique references (which are not aliased by any references). We believe that our system is one of the first to combine object-as-modifier discipline with transfer of ownership.

In our system all ownership information is removed from objects. This should mitigate the loss of ownership information problem in subclassing. The association modes (or roles) are not placeholders for reference target's owner, but uninterpreted type tags on object's references. The available roles are not limited by a parameter list, nor by the references targeting it. Our system allows the bottom-up construction process, in which sub-objects are created before their owners.

10. Acknowledgments

We would like to thank Gary Leavens for his encouragement and insightful comments on Schünemann's thesis.

Fiech's and Schünemann's research was partially supported by NSERC grant OGP0170497.

References

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP '04: Proceedings of the 18th European Conference on Object-Oriented Programming*, 2004.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 311–330, New York, NY, USA, 2002. ACM Press.

[3] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECCOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 32–59. Springer-Verlag, 1997.

[4] Edwin H. Blake and Steve Cook. On including part hierarchies in object-oriented languages with an implementation in smalltalk. In *ECOOP '87: Proceedings of the European Conference on Object-Oriented Programming*, pages 41–50, London, UK, 1987. Springer-Verlag.

[5] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–223, New York, NY, USA, 2003. ACM Press.

[6] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 292–310, New York, NY, USA, 2002. ACM Press.

[7] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP '03: Proceedings of the 17th European Conference on Object-Oriented Programming*, 2003.

[8] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64, New York, NY, USA, 1998. ACM Press.

[9] David Gerard Clarke. *Object ownership and containment*. PhD thesis, University of New South Wales, 2002.

[10] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP '07: Proceedings of the 21st European Conference on Object-Oriented Programming*, 2007.

[11] John Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA '91: Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, pages 271–285, New York, NY, USA, 1991. ACM Press.

[12] Neel Krishnaswami and Jonathan Aldrich. Permission-based ownership: encapsulating state in higher-order typed languages. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 96–106, New York, NY, USA, 2005. ACM Press.

[13] Yi Lu and John Potter. Protecting representation with effect encapsulation. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 359–371, New York, NY, USA, 2006. ACM Press.

[14] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical report, Fernuniversität Hagen, 2001.

[15] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 158–185, London, UK, 1998. Springer-Verlag.

[16] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic java. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 311–324, New York, NY, USA, 2006. ACM Press.

[17] Ulf Schünemann. *Composite objects: dynamic representation and encapsulation by static classification of object references*. PhD thesis, Memorial University of Newfoundland, 2005.