Ownership Transfer in Universe Types

Peter Müller
Microsoft Research, USA
mueller@microsoft.com

Arsenii Rudich
ETH Zurich, Switzerland
arsenii.rudich@inf.ethz.ch

Abstract

Ownership simplifies reasoning about object-oriented programs by controlling aliasing and modifications of objects. Several type systems have been proposed to express and check ownership statically.

For ownership systems to be practical, they must allow objects to migrate from one owner to another. This *ownership transfer* is common and occurs, for instance, during the initialization of data structures and when data structures are merged. However, existing ownership type systems either do not support ownership transfer at all or they are too restrictive, give rather weak static guarantees, or require a high annotation overhead.

In this paper, we present UTT, 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.

Categories and Subject Descriptors D3.3 [*Programming Languages*]: Language Constructs

General Terms Languages, Verification

Keywords Universe Types, Ownership Transfer, Aliasing, Uniqueness

1. Introduction

Ownership allows programmers to structure the object store and to control aliasing and modifications of objects. The ownership structure makes programs easier to understand, to maintain, and to reason about. Ownership has been used to verify object invariants [29, 31, 33, 34], to show the absence of data races in multi-threaded programs [7, 26], to facilitate memory management for real-time programs [3, 9], to check object immutability [24], and to prove representation independence [4].

In the existing ownership systems, each object has at most one owner object. The ownership relation is a tree order. We call the set of all objects with the same owner a *context*. The *root context* is the set of objects with no owner. Ownership type systems allow programmers to express and check ownership properties statically. In these type systems, a type conveys information about the class of an object as well as the object's owner. Besides the ownership topology, ownership type systems also enforce restrictions on references between objects in different contexts.

The owner of an object is first determined when the object is created. In many common designs, the owner is not fixed throughout an object's lifetime, but changes dynamically. The need for this *ownership transfer* is illustrated by the following examples:

- *Merging data structures*: data structures such as lists are merged efficiently by transferring the internal representation of one structure to the context of the other [15].
- *Object initialization*: constructors often take an existing object as parameter and then capture this object, that is, transfer ownership to the object being constructed [17]. A special case of object initialization is the Factory pattern, where product objects are created in the context of a global factory and then transferred to the client [36].
- *Work flow systems*: tasks in work flow systems are transferred repeatedly from processor to processor.

The first example is illustrated in Fig. 1. A list consists of a main object of class `List` and a doubly-linked cyclic structure of `Node` objects. The node referenced by the `List`'s `first` field is a dummy node to simplify the handling of empty lists. In our example, a `List` object owns its nodes. Therefore, efficient merging of two lists requires ownership transfer. We assume here that all nodes of a context are transferred together. The resulting structure is shown in Fig. 2.

Static type safety of an ownership type system means essentially that the static type of an expression e correctly reflects the owner of the object e evaluates to. Consequently,

OOPSLA '07 October 21–25, Montréal, Québec, Canada.
Copyright © ACM 978-1-59593-786-5/07/0010. . . $5.00

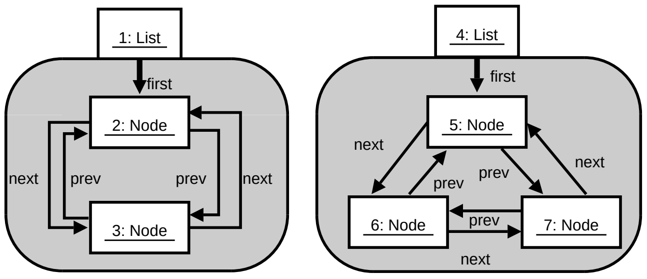

Figure 1. Ownership structure of two lists. Each `List` object owns its `Node` objects. Objects are depicted by boxes; references are depicted by arrows, which are labeled with the name of the field holding the reference. Rounded boxes delimit ownership contexts. The owner sits atop the context it owns.

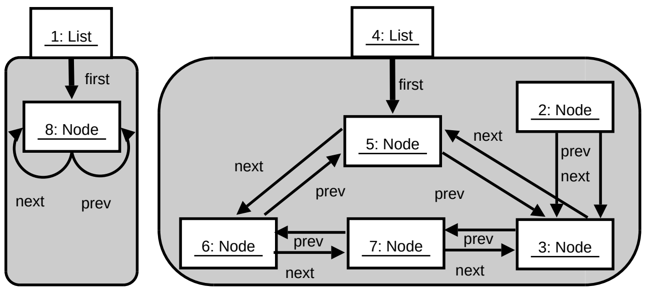

Figure 2. Ownership structure after the nodes of `List` object 1 have been transferred to `List` object 4 and the two node structures have been merged. To preserve type safety, the `first` field of `List` object 1 is set to a fresh node in its context (object 8). The dummy `Node` object 2 is not reachable after the merge operation and will be garbage collected.

in the presence of ownership transfer, objects may change their type dynamically. For instance, `Node` object 2 in Fig. 1 is owned by `List` object 1. This is reflected in the type of `List`'s field `first`. In Universe Types [20], this field has type **`rep`** `Node`, where the **`rep`** modifier indicates that the referenced object is owned by **`this`**. However, after the transfer (Fig. 2) `Node` object 2 is owned by `List` object 4. Therefore, the **`rep`** modifier of the `first` field of `List` 1 does no longer reflect the owner of the referenced `Node` correctly. This is type safe only if the transfer sets the `first` field of `List` object 1 to a different value, for instance, a new dummy node, before the field is accessed again.

In general, type safety requires that the well-typedness of each variable (local variable, method parameter, or field) that is affected by an ownership transfer is re-established before the next access to the variable. To enforce this condition, an ownership type system must be able to determine statically all stack and heap locations that reference a transferred object. In general, this requires knowledge of the whole call stack—because method executions in progress may have local variables that point to the transferred objects—and knowledge of all subclasses—because a subclass may contain fields that point to the transferred objects.

```
void retain(List list4) {
  rep Node node3 = first.next;
  list4.merge(this);
  // node3 points to object owned by list4
  node3.next = null;
}
```

Figure 3. Unsafe method of class `List`. The last statement destroys the node structure of `list4`.

For instance, consider an execution of method `retain` (Fig. 3) with `List` object 1 as receiver and `List` object 4 as parameter. In the state before calling `merge` (illustrated by Fig. 1), local variable `node3` holds a reference to `Node` object 3. The well-typedness of `node3` is violated when `merge` transfers the nodes of **`this`** to `list4`. In the state after the call (Fig. 2), `node3` points to an object owned by `list4`, even though its **`rep`** modifier indicates that the object is owned by **`this`**. Method `retain` exploits the ill-typedness of `node3` to destroy the link structure of `list4`'s nodes by setting the `next` field of `node3` to **`null`**. To avoid this problem, `node3`'s well-typedness must be re-established after the call to `merge`. However, this cannot be done by method `merge` because `node3` is not in the stack frame of `merge`.

To prevent the problem of ill-typed aliases on transferred objects, existing type systems for ownership transfer use uniqueness [2, 25, 32] to control the possible references to objects that may be transferred. A reference is *unique* if the referenced object is not aliased. Unique variables may only hold unique references and the **`null`** value. Therefore, transferring an object using a unique variable does not affect any other stack or heap locations, and it suffices to assign a new value to the unique variable. If all objects of a context are transferred together, it even suffices to guarantee that there is only one reference from the owner into the whole context, whereas aliasing within the context is not restricted. This notion of *external uniqueness* [15, 42] is far more expressive. For instance, the `first` references in Fig. 1 are not unique in the strict sense because the nodes are also referenced by their neighbor nodes. However, they are externally-unique because all other references to `Node` objects come from objects within the context.

The problem with method `retain` is caused by the fact that the `first` field of `List` object 1 is *not* externally-unique when the lists are merged because the local variable `node3` holds a second reference into the context of `List` 1.

Although (external) uniqueness enables type-safe ownership transfer, the existing techniques building on (external) uniqueness have severe shortcomings: (1) They impose a high annotation overhead on programmers. (2) Even external uniqueness is too restrictive for common implementations. For instance, when a list stores references to its first and last node, these references are not externally-unique. (3) Some of the techniques provide encapsulation that is too weak for certain applications of ownership such as the verification of object invariants. We explain these shortcomings in more detail in the next section.

In this paper, we present an extension of Universe Types [19, 20] that supports ownership transfer. Our system combines ownership type checking with a static analysis to enforce an alias invariant that is even less restrictive than external uniqueness. This alias invariant permits temporary aliases on transferable objects. In particular, a context need not be externally-unique while a method executes on an object inside this context. This provides flexibility that is needed in many implementations. On the other hand, our alias invariant is strong enough to obtain an externally-unique reference on a cluster of objects and to transfer the cluster in a type-safe way. The alias invariant is enforced by a modular, intraprocedural static analysis. This analysis makes each variable that is potentially affected by a transfer unusable and enforces that it is assigned a new value before it is accessed. For instance, our static analysis would make the variable `node3` in method `retain` (Fig. 3) unusable, thereby preventing its abuse after the transfer.

Our approach solves the shortcomings of the existing approaches based on uniqueness. The main contributions are:

- An alias invariant that is less restrictive than external uniqueness, but strong enough to enable type-safe ownership transfer. In particular, it permits several external references on transferable objects.
- A modular static analysis to enforce the alias invariant. This analysis requires only negligible annotation overhead.
- An extension of Universe Types to support ownership transfer. The extended type system can handle almost all common examples for ownership transfer. Like Universe Types, it enforces the owner-as-modifier discipline, which enables the verification of object invariants [34].

Outline. In the next section, we discuss existing work on (external) uniqueness. Sec. 3 provides the background on Universe Types that is needed in the rest of the paper. Our approach to ownership transfer in explained in Sec. 4 and formalized in Sec. 5. Sec. 6 illustrates our technique by examples. We present an informal soundness theorem in Sec. 7. We describe our implementation in Sec. 8, discuss related work in Sec. 9, and offer conclusions in Sec. 10.

2. Background on Uniqueness

A very strong notion of uniqueness is to enforce that unique variables hold unique references (or **`null`**) in all execution states. This is typically achieved using *destructive reads*, which assign **`null`** to a unique variable when it is read [25]. Enforcing uniqueness in all execution states is overly restrictive. For instance, calling a method on the unique field f creates a temporary alias on the stack and, therefore, nullifies f. This is cumbersome since the caller has to restore the value of f after the call. Moreover, destructive reads suffer from several problems [11]: (1) They require a change of the language semantics, which is unintuitive for programmers. (2) They make it difficult to query information about the unique object, especially in side-effect free methods. (3) They are not compatible with non-null type systems [12, 23].

A more practical approach is to permit unique objects to be temporarily aliased from stack locations. Temporary aliases enable method calls on unique variables without nullifying the receiver. Techniques that allow temporary aliases on unique objects have to address two problems:

1. *Callbacks*: If a method with receiver o calls another method m while o's unique fields are temporary aliased, m must be prevented from calling back into o under the false assumption that o's unique fields are actually unique.
2. *Capturing*: A method that receives a temporary alias of a unique object must be prevented from storing this reference in a field, which would create a permanent alias from a heap location.

In the following, we discuss how existing approaches solve these problems.

Clarke and Wrigstad [15, 42] enforce external uniqueness, but permit temporary aliases. They use *borrowing* to handle callbacks and capturing. `borrow` blocks permit temporary aliases on unique objects. When a unique field f is borrowed, its value is copied into a non-unique local variable l, which can then be used in method calls. The callback problem is prevented by nullifying f at the beginning of the `borrow` block. Therefore, callbacks do not find the external uniqueness invariant violated. Capturing is prevented by giving l a fresh owner such that storing the reference in a field is prevented by the ownership type system. At the end of a `borrow` block, the value of f is restored. Borrowing permits temporary aliases on unique objects in a safe way. However, it introduces annotation overhead and requires owner-polymorphic methods to support the fresh owner of a borrowed object. Moreover, assigning **`null`** to the borrowed variable leads to similar problems as destructive reads.

AliasJava [2] permits temporary aliases through lent references. A *lent reference* may point to unique objects in any ownership context. The types of lent references do not convey ownership information. Therefore, lent references do not compromise the type safety of ownership transfer. Lent references are the only temporary aliases to an otherwise unique object. Assigning a unique variable to a non-lent variable makes the unique variable unusable. To prevent the callback problem, a new unique value must be assigned to each unusable unique field before the next method call. Capturing is prevented by disallowing lent references to be stored in fields. Lent references permit type-safe ownership transfer. However, they weaken encapsulation drastically because they may point to arbitrary ownership contexts and may be used to modify objects. For instance, method `retain` (Fig. 3) type checks in AliasJava if `node3` is declared `lent`. This lent variable is used to break `list4`'s encapsulation and to destroy its link structure. It is unclear how to maintain object invariants in the presence of lent references [34].

Alias burying [10] relies on a static analysis to track temporary aliases on unique objects. Whenever a unique variable is read, all existing aliases become unusable. Therefore, a unique variable effectively behaves as if it was actually unique even though unusable aliases may exist. To prevent capturing, unique variables may be passed to methods only as borrowed parameters. Like lent variables in AliasJava, borrowed parameters cannot be stored in fields. To handle callbacks, methods are annotated with read effects. If a method potentially reads a unique field f, all temporary aliases of f are made unusable before the method is called. Therefore, every field read by a method is effectively unique, including fields that are read during callbacks. Alias burying permits temporary aliasing without destructive reads and borrowing. However, it has major drawbacks. Its expressiveness is limited by the underlying static analysis. Moreover, `borrowed` annotations and read effects cause a high annotation overhead.

Capabilities systems [11, 22] model uniqueness through universal capabilities, which permit arbitrary access to an object. Callbacks are handled by annotating methods with the capabilities they expect. Therefore, a method expecting a field f to be unique cannot be called while f is aliased because the caller cannot provide the expected capabilities. Capturing is handled by passing capabilities. If a caller of a method m passes the capabilities of an argument p to m then the caller will not consider p to be unique after the call unless m returns the capabilities back to the caller. If the capabilities are not retained by m, m can capture the argument, but not use it since it does not have the necessary capabilities. While capabilities systems are very flexible, they require a large overhead because methods have to be annotated with the capabilities they expect and return. Moreover, capabilities have not been integrated with ownership type systems and external uniqueness.

3. Background on Universe Types

Universe Types [19, 20] is an ownership type system that permits arbitrary aliasing, but restricts modifications of objects. In this section, we explain the main concepts of Universe Types by an example. A formalization of non-generic Universe Types in Isabelle is given in [27], and a formalization of Generic Universe Types is presented in [18].

```
class List {
  rep Node first = new rep Node();  // dummy

  void add(any Object o) {
    rep Node n;
    n = new rep Node(o, first, first.next);
    first.next.prev = n;
    first.next = n;
  }

  // constructors and other methods omitted.
}
```

Figure 4. Implementation of a list in Universe Types. `List` objects own their `Node` objects, as indicated by the **`rep`** modifier in all occurrences of class `Node`.

Ownership Modifiers. A type consists of an ownership modifier and a class name. The *ownership modifier* expresses object ownership relative to the current receiver object **`this`**[1]. Programs may contain the ownership modifiers **`peer`**, **`rep`**, and **`any`**. **`peer`** expresses that an object has the same owner as the **`this`** object, **`rep`** expresses that an object is owned by **`this`**, and **`any`** expresses that an object may have any owner. **`any`** types are supertypes of the **`rep`** and **`peer`** types with the same class because they convey less specific ownership information.

The use of ownership modifiers is illustrated by classes `List` (Fig. 4) and `Node` (Fig. 5), which implement a doubly-linked list of objects. For simplicity, we omit access modifiers from all examples. A `List` object owns its `Node` objects since they form the internal representation of the list and should, therefore, be protected from unwanted modifications. This ownership relation is expressed by the **`rep`** modifier of `List`'s field `first`, which points to the dummy node of the list. All nodes of a list have the same owner, therefore, the `prev` and `next` fields of `Node` have a **`peer`** modifier. Finally, the elements stored in the list may have any owner, which is indicated by the **`any`** modifier of `Node`'s `elem` field.

Owner-as-Modifier Discipline. Universe Types allow an object o to be referenced by any other object, but reference chains that do not pass through o's owner must not be used to modify o. This allows owner objects to control modifications of owned objects, for instance, to maintain invariants. This *owner-as-modifier* discipline is enforced by disallowing modifications of objects through **`any`** references. That is, an expression of an **`any`** type may be used as receiver of field reads and calls to side-effect free (*pure*) methods, but not of field updates or calls to non-pure methods. To check

[1] We ignore static methods in this paper, but an extension is possible [33].

```
class Node {
  peer Node next, prev;
  any Object elem;

  Node()  { next = this; prev = this; }

  Node(any Object e, peer Node p, peer Node n)
    { elem = e; prev = p; next = n; }

  void flip() {
    peer Node tmp = next;
    next = prev;
    prev = tmp;
  }

  // other methods omitted.
}
```

Figure 5. Nodes form the internal representation of lists. Method `flip` is used to reverse the list as we discuss later.

this property, Universe Types require side-effect free methods to be annotated with the keyword **`pure`**.

As a consequence of the restriction on **`any`** receivers, Universe Types prevent certain callbacks. When an object o calls a method m on a **`rep`** receiver, m cannot call back into o using a non-pure method. Since m's receiver is owned by o, it can reach o only by a reference chain that contains at least one **`any`** reference. Consequently, this reference chain cannot be used to call a non-pure method on o. Our technique for ownership transfer uses this property to address the callback problem described in Sec. 2.

Viewpoint Adaptation. Since ownership modifiers express ownership relative to **`this`**, they have to be adapted when this "viewpoint" changes. Consider the second parameter of `Node`'s second constructor. The **`peer`** modifier expresses that the parameter object must have the same owner as the receiver of the constructor. On the other hand, `List`'s method `add` calls the constructor on a **`rep`** receiver, that is, an object that is owned by **`this`**. Therefore, the second parameter of the constructor call also has to be owned by **`this`**. This means that from this particular call's viewpoint, the second parameter needs a **`rep`** modifier, although it is declared with a **`peer`** modifier. In the type system, this *viewpoint adaptation* is done by combining the modifier of the receiver of a call (here, **`rep`**) with the modifier of the formal parameter (here, **`peer`**). This combination yields the argument modifier from the caller's point of view (here, **`rep`**).

Field accesses and calls on receiver **`this`** do not require any viewpoint adaptation, because the viewpoint does not change. We model this behavior by an additional ownership modifier **`this`**, which is used internally by the type system for the **`this`** variable. Combining the **`this`** modifier with any modifier u yields u. In particular, for a **`rep`** field f, the field access **`this`**.f has a **`rep`** type, whereas for any other receiver p, the field access $p.f$ has an **`any`** type. This allows us to keep the contexts of different objects separate.

Runtime Model. Each object stores a reference to its owner. The owner of an object is determined by the creation expression. For instance, the **`rep`** modifier in the initialization of `List`'s `first` field (Fig. 4) indicates that the new object is owned by **`this`**. The runtime ownership information is used to check downcasts from **`any`** to **`rep`** or **`peer`** types as well as to evaluate **`instanceof`** expressions.

Universe Invariant. Universe Types guarantee the following properties [27]:

1. *Type safety*: The ownership modifier of the type of a well-typed expression e correctly reflects the owner of the object e evaluates to.
2. *Tree order*: In all execution states of a well-typed program, the ownership relation among the objects in the heap is a tree order.
3. *Owner-as-modifier*: The evaluation of a well-typed expression modifies only those objects that are (transitively) owned by the owner of **`this`**.

4. Ownership Transfer

Our solution to ownership transfer builds on external uniqueness [15, 42]. Like in Clarke and Wrigstad's work, we transfer whole groups of objects, for instance, all nodes of a doubly-linked list. We call such a group a *cluster*. We provide a **`release`** statement to obtain an externally-unique reference into a cluster. A **`capture`** statement uses an externally-unique reference to transfer the cluster.

In this section, we extend Universe Types to support clusters, present the **`release`** and **`capture`** statements, and discuss the alias invariant that enables type-safe ownership transfer. The type rules and static analysis to maintain this alias invariant are presented in Sec. 5.

4.1 Clusters

Clusters can be handled by a simple extension of Universe Types. We call the extended type system *Universe Types with Transfer* or *UTT* for short. We explain the extensions in the following.

Clusters are declared explicitly by a class member declaration of the form **`cluster`** cn, where cn is a globally unique *cluster name*. We illustrate the use of clusters by a revised version of the doubly-linked list (class `MList` in Fig. 6). This list implementation declares a cluster `R` to store the nodes.

In UTT, an object is owned by a pair consisting of an owner object and a cluster name. This pair uniquely identifies a cluster. Objects in the root context are owned by $<$**`null`**$, \texttt{root}>$. We use the phrase *the cluster cn of an object o* to refer to the cluster of objects owned by $<o, cn>$.

```
class MList {
  cluster R;
  rep<R> Node first = new rep<R> Node();

  void reverse() {
    rep<R> Node p = first;
    do {
      p.flip();
      p = p.prev;
    } while(p != first);
  }

  free Node getNodes() {
    free Node res = release(first);
    first = new rep<R> Node();
    return res;
  }

  void merge(peer MList l) {
    free Node un = l.getNodes();
    rep<R> Node rn = capture(un, rep<R>);

    first.prev.next = rn.next;
    rn.next.prev = first.prev;
    rn.prev.next = first;
    first.prev = rn.prev;
  }

  // constructors and other methods omitted.
}
```

Figure 6. A list implementation using the cluster `R` to store the nodes. Class `Node` is presented in Fig. 5.

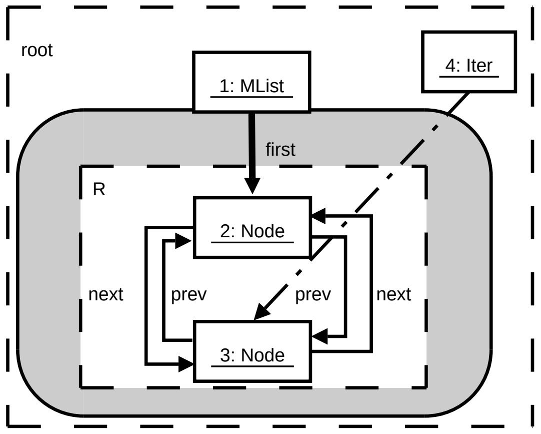

Figure 7. Object structure of an `MList`. Clusters are depicted by dashed boxes. The reference from the `Iter` object to `Node` object 3 is an **`any`** reference.

Definition 4.1 (External reference)**.** *A reference is an* external reference into the cluster cn of an object o *if and only if the following conditions hold:*

1. *The origin of the reference is external to the cluster, that is, the variable holding the reference is either a field of an object that is not (transitively) owned by $<o, cn>$ or a stack location of a stack frame whose receiver object is not (transitively) owned by $<o, cn>$.*
2. *The target of the reference is internal to the cluster, that is, the reference points to an object (transitively) owned by $<o, cn>$.*
3. *The reference is not an* **`any`** *reference.*

Since **`any`** references do not convey any ownership information, they are not affected by ownership transfer and, thus, are not considered external. Type safety guarantees that the variable holding the external reference into o's cn cluster is a field of o or a variable in a stack frame for a method execution on receiver o.

We use a static analysis to control external references into clusters. As explained in Sec. 5.4, this analysis makes certain variables *unusable*, that is, enforces that new values are assigned to these variables before they are accessed. We say that *a reference is unusable* if it is held by an unusable variable.

Definition 4.2 (External uniqueness)**.** *A reference to an object o is* externally-unique *if it is the only usable external reference into the cluster containing o.*

Fig. 7 shows an instance of class `MList` with two nodes. The reference held by the `first` field is an external reference into the cluster `R` of `MList` object 1. It is externally-unique provided that no stack variable points to one of the nodes by a non-**`any`** reference.

Type-safe ownership transfer requires restrictions on the external references into a cluster. Since these restrictions are not met by all implementations, it is useful to distinguish between transferable and non-transferable clusters. External references into non-transferable clusters need not be restricted beyond the encapsulation of Universe Types. To simplify the presentation, we do not consider non-transferable clusters in this paper. However, our implementation [41] and the formalization in our technical report [35] support them.

Ownership Modifiers. The **`peer`** modifier indicates that two objects belong to the same cluster, that is, have the same owner object and cluster name. Like in Universe Types, the **`any`** modifier does not provide any ownership information, neither about owner objects nor about cluster names.

We replace the **`rep`** modifier of Universe Types by a parametric version **`rep`** $\langle cn \rangle$ that specifies a cluster name. For instance, the modifier **`rep`** $\langle$`R`$\rangle$ in the declaration of `MList`'s field `first` indicates that the node is in the `R` cluster of **`this`**. The modifier **`rep`** $\langle cn \rangle$ may be used in the class C that declares cn and its subclasses. However, we impose an additional restriction on field declarations: Fields with the modifier **`rep`** $\langle cn \rangle$ may be declared only in class C, but not in C's subclasses. This restriction guarantees that the only fields that are affected by a transfer of cluster cn are declared in class C and, thus, can be found by a modular analysis.

The ownership modifier **free** indicates that a reference is externally-unique. In our system, external uniqueness is enforced at the time a cluster is transferred, but typically not maintained over many execution states because every access to a **free** variable destroys its external uniqueness. Consequently, we disallow the **free** modifier in the declaration of a field. However, a field of an object o can nevertheless point to transferable objects in o's clusters if it is declared with a **rep** modifier.

rep $\langle cn \rangle$ types and **free** types are subtypes of the **any** types with the same class. For different cluster names cn_1 and cn_2, **rep** $\langle cn_1 \rangle$ and **rep** $\langle cn_2 \rangle$ types are incomparable because they refer to different clusters.

Viewpoint Adaptation. The viewpoint adaptation operator $\triangleright$ of UTT is defined by the following table. The first argument (rows) is the ownership modifier of the receiver. The second argument (columns) is the ownership modifier of the field, method parameter, or method result to be viewpoint-adapted. For instance, the modifier of the access `p.prev` in `MList`'s method `reverse` is determined by combining the modifier of the receiver `p` (**rep** $\langle$R$\rangle$) with the modifier of field `prev` (**peer**), which yields **rep** $\langle$R$\rangle$.

$\triangleright$	peer	rep $\langle cn_2 \rangle$	any	free
this	peer	rep $\langle cn_2 \rangle$	any	free
peer	peer	any	any	free
rep $\langle cn_1 \rangle$	rep $\langle cn_1 \rangle$	any	any	free
any	any	any	any	free
free	any	any	any	free

Accessing a field or calling a method on a **free** receiver yields an **any** reference, unless the modifier of the method is **free**. This ensures that the receiver reference stays externally-unique. Calling a method with a **free** return type yields a **free** reference, independent of the modifier of the receiver. For instance, the call `l.getNodes()` in `MList`'s method `merge` has type **free** `Node`.

Runtime Model. Besides its owner object, each object stores the name of the cluster it belongs to. The cluster information is used to check downcasts from **any** to **rep** $\langle cn \rangle$ types. Like the owner object, the cluster name is first determined when the object is created. For instance, the dummy node created in the initializer of `first` (Fig. 6) is owned by <**this**, R>. A new object of type **free** T is put into a new free cluster. A *cluster is free* if the objects in the cluster are owned by <**null**, cf>, where cf is a fresh cluster name that is not used in the program. In the following, we use cf as cluster name for free clusters and cn for non-free clusters.

4.2 Release and Capture

Ownership transfer is performed by a combination of two polymorphic statements, **release** and **capture**. The **release** statement takes an argument of type **rep** $\langle cn \rangle$ T and transfers all objects in the cn cluster of **this** to a new free cluster. The **release** operation returns its argument, but with static type **free** T. Our static analysis ensures that the reference is actually externally-unique by making all other external references unusable.

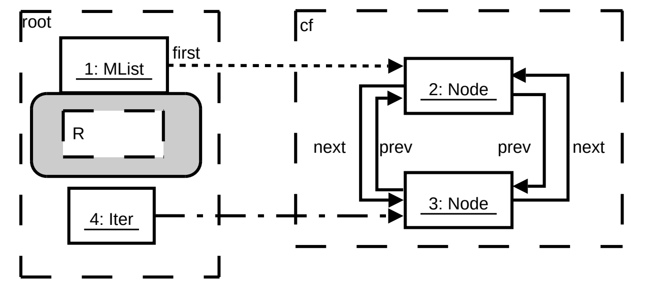

Figure 8. The object structure from Fig. 7 after releasing the `R` cluster of the `MList` object. The `first` field is unusable because it is ill-typed after the **release**. The well-typedness of the **any** reference from the `Iter` object is not affected by the **release**.

The **capture** statement takes an argument y of type **free** T and an ownership modifier u. It transfers the objects in the free cluster containing y to the owner indicated by u. The modifier u must be **peer** or **rep** $\langle cn \rangle$, because **any** and **free** do not indicate an owner. The **capture** operation returns y, but with static type u T.

`MList`'s method `getNodes` uses the **release** statement to obtain an externally-unique reference `res` to the dummy node. Fig. 8 shows the object structure after releasing the `R` cluster of `MList` object 1. The field `first` still references the dummy node. However, to enforce that `res` is externally-unique, our static analysis makes `first` unusable. Method `getNodes` returns an externally-unique reference into the free cluster after setting `first` to a new dummy node. Method `merge` calls `getNodes` to obtain a **free** reference to the first node of list `l`. It then uses **capture** to transfer `l`'s nodes to the `R` cluster of **this**. After the **capture**, the nodes of both lists are in the same cluster and can be merged.

Note that the **release** statement can be used to release clusters of **this**, but not of any other object. For instance, method `merge` cannot directly release `l`'s nodes because the field access `l.first` has an **any** type, whereas the **release** statement expects a **rep** $\langle cn \rangle$ argument. Therefore, `merge` must call method `getNodes` on `l` to release the nodes. This restriction of **release** improves encapsulation because it prevents objects from "stealing" another object's cluster.

release and **capture** change the owner of the objects in the released and captured clusters. Therefore, both statements have side-effects and must not be used in pure methods.

4.3 Alias Invariant

UTT restricts external references into clusters as described by the following confinement property.

Definition 4.3 (Confinement)**.** *An object o is* confined *if and only if each external reference into a cluster cn of o is held by a variable v that satisfies at least one of the following conditions:*

1. *v is a field of o.*
2. *v is a local variable or parameter of the current method execution, and o is the receiver of this method.*
3. *v is unusable.*

When the **this** object is confined, a static analysis can determine modularly all variables that point into a cluster of **this** and to make them unusable when the cluster is released. For Case 1, we know that v is declared in the class C that declares cn because field declarations may only mention cluster names declared in the enclosing class. Therefore, we can determine all variables affected by releasing cn by inspecting the fields of C and local variables and parameters of the enclosing method. Note that the confinement property does not restrict **any** references since we do not consider **any** references to be external references (see Def. 4.1).

The capturing problem described in Sec. 2 violates confinement by storing an external reference into o's cn cluster in a field of an object different from o. In our system, this is prevented by viewpoint adaptation. Consider a method call $x.m(p)$ where p is a temporary alias into a cluster of the current receiver o. We may assume that p has modifier **rep** $\langle cn \rangle$. If the receiver x is **this**, confinement of o is preserved because o and x are the same object. If x has modifier **rep** $\langle cn \rangle$, then x and p point into the same cluster, and storing p does not create an external reference. For all other modifiers of x, m receives an **any** reference to p, which cannot be used to create an external reference.

Since Universe Types prevent callbacks via non-pure methods into the (transitive) owners of **this** (see Sec. 3), not all objects need to be confined in all execution states. The following *alias invariant* defines when objects must be confined. It holds in all execution states of a well-typed program.

Definition 4.4 (Alias invariant)**.** *Each object o is confined unless the method currently executing is pure or o is one of the (transitive) owner objects of the current receiver object.*

This definition allows confinement to be violated in two situations. First, while a pure method is executed; since pure methods must not perform **release** statements, they do not rely on confinement. Second, for a (transitive) owner object o of the current receiver. This is possible because m can neither release a cluster of o (because o is different from **this**) nor call a non-pure method of o to perform the **release**. Therefore, it does not rely on o being confined.

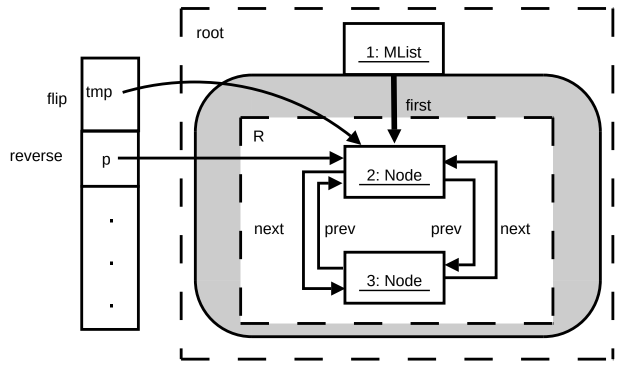

Figure 9. Object structure of an `MList` during the execution of `p.flip`. The box on the left-hand side depicts the call stack. `MList` object 1 is not confined since `p` holds a reference into the `R` cluster, but is not a local variable of the current method execution. This violation is permitted because the `MList` object 1 is the owner object of the current receiver, `Node` object 2.

Permitting (transitive) owners of **this** not to be confined enables a very natural programming style as we illustrate by method `reverse` of class `MList` (Fig. 6). Let's assume that `reverse` is executed on receiver x. The first assignment to `p` creates an additional external reference into the `R` cluster of x. During the execution of `flip`, x is not confined because `p` holds an external reference into the `R` cluster of x, but is not a variable of the current method execution (see Fig. 9). This violation is permitted by the alias invariant because x is the owner of `p`. Therefore, `p` need not be made unusable before the call to `flip`. When the call terminates, x is confined again and we know that `p` still points into the `R` cluster of x because this cluster cannot be transferred by `flip`.

The treatment of temporary aliases and calls on owned objects is one of the key virtues of our technique. We need neither borrowing [15, 42] nor read effects [10] to handle the call to `p.flip` because callbacks are prevented by the type system. We also do not have to prevent `flip` from storing its receiver in a field because this would not create an external reference. In contrast to alias burying [10], our static analysis does not make `p` unusable when the field `first` is read in the condition of the loop. This is because we do not enforce external uniqueness when a variable is read, but only at the time of a transfer.

In summary, our alias invariant:

- does not restrict references within a cluster (since we use external uniqueness),
- does not restrict references from **any** variables (which are not considered external references),

- does not require the owner objects of the current receiver to be confined (since our type system prevents callbacks to these objects via non-pure methods), and
- does not require confinedness while a pure method is executed (since pure methods must not transfer objects).

Each of these exceptions to the standard notion of uniqueness makes our system more flexible without losing static type safety.

5. Formalization

In this section, we present the UTT rules for a subset of Java including classes and inheritance, instance fields, dynamically-bound methods, and the usual operations on objects (allocation, field read, field update, casts). For simplicity, we omit several features of Java such as interfaces, exceptions, constructors, static fields and methods, inner classes, primitive types and the corresponding expressions, and all statements for control flow. We do not expect that any of these features is difficult to handle. We only show the static rules here. A full formalization including the runtime model, operational semantics, and proofs is presented in our technical report [35].

Judgments. Our formalization uses the following two main judgments. A type judgment has the form $\Gamma;\mathcal{U} \vdash s$ and expresses that statement s is well-typed in a declaration environment Γ. $\mathcal{U}$ is the set of unusable variables before the execution of s as defined by the static analysis in Sec. 5.4.

The judgment $\Gamma;\mathcal{U} \vdash s : \mathcal{U}'$ expresses that $\mathcal{U}'$ is the set of unusable variables after statement s if $\mathcal{U}$ is the set of unusable variables before s.

5.1 Programming Language

Fig. 10 summarizes the syntax of our language. We assume that all identifiers of a program are globally unique except for **this** as well as method and parameter names of overridden methods. This can be achieved easily by preceding each identifier with the class or method name of its declaration (but we omit this prefix in our examples). $\overline{T}$ denotes a sequence of Ts. In such a sequence, we denote the i-th element by T_i. We sometimes use sequences of tuples $S = \overline{X\ T}$ as maps and use a function-like notation to access an element $S(X_i) = T_i$.

A program P consists of a sequence of classes. We keep the current program implicit in the notations. Each class Cls has a class identifier C, a superclass C', a list of cluster declarations $\overline{cn}$, a list of field declarations $\overline{T\ f}$, and a list of method declarations $\overline{M}$. $FieldId$ is the set of all field identifiers. A type T consists of an ownership modifier u and a class identifier C.

A method M consists of a result type T, a method name m, exactly one formal parameter $T\ x$, a list of local variable declarations $\overline{T\ x}$, and a statement s. The method returns the value of the predefined local variable `res`. $VarId$ is the set of variable names containing **this**, the explicit formal method parameter, and all local variables including `res`. For simplicity, we do not support pure methods in our formalization, but we include them in the discussion. An extension to pure methods is straightforward and does not reveal any interesting aspects.

The set of statements includes assignment, field read, field update, method call, object creation, cast, sequential composition, **release**, and **capture**. We provide the usual expressions in the form of assignment statements because our static analysis depends on the modifier of the variable a value is assigned to. For instance, reading a **free** variable does not create an external reference if the value is assigned to an **any** variable. This form is obtained by introducing auxiliary variables for sub-expressions. These variables have the static types (including the ownership modifiers) of the corresponding sub-expressions. For simplicity, we omit all literals including **null**.

A declaration environment Γ maps **this**, the formal method parameter, and all local variables to their types.

$$
\begin{array}{rcl}
P & ::= & \overline{Cls} \\
Cls & ::= & \texttt{class}\ C\ \texttt{extends}\ C'\ \texttt{\{}\ \overline{cn}\texttt{;}\ \ \overline{T\,f}\texttt{;}\ \ \overline{M}\ \texttt{\}} \\
u & ::= & \texttt{this} \mid \texttt{peer} \mid \texttt{rep}\,\langle cn\rangle \mid \texttt{any} \mid \texttt{free} \\
T & ::= & u\,C \\
M & ::= & T\ m(T\ x)\texttt{\{}\ \overline{T\,x}\texttt{;}\ \ s\ \texttt{\}} \\
s & ::= & x\ \texttt{:=}\ x\texttt{;} \\
 & \mid & x\ \texttt{:=}\ x\texttt{.}f\texttt{;} \\
 & \mid & x\texttt{.}f\ \texttt{:=}\ x\texttt{;} \\
 & \mid & x\ \texttt{:=}\ x\texttt{.}m(x)\texttt{;} \\
 & \mid & x\ \texttt{:=}\ \texttt{new}\ T\texttt{;} \\
 & \mid & x\ \texttt{:=}\ (T)\ x\texttt{;} \\
 & \mid & s_1\ s_2 \\
 & \mid & x\ \texttt{:=}\ \texttt{release}(x)\texttt{;} \\
 & \mid & x\ \texttt{:=}\ \texttt{capture}(x,u)\texttt{;} \\
\Gamma & ::= & \overline{x\,T} \\
\mathcal{U} & \subseteq & FieldId \cup VarId
\end{array}
$$

Figure 10. Syntax and declaration environment.

5.2 Well-Formedness

The well-formedness rules for types, methods, and classes are presented in Fig. 11. If a type $u\ C$ has a **rep** $\langle cn\rangle$ modifier then $u\,C$ is well-formed in an environment Γ only if cn is declared in a (not necessarily proper) superclass of the enclosing class. The enclosing class is the class of **this** in Γ (WF-TYPE). The function `clusters`(C_0) yields the names of all clusters declared in class C_0 and its superclasses.

A method m is well-formed in a class C if the statement s constituting its body is well-typed in the environment Γ. The types of the parameter, result, and local variables must be well-formed in Γ, and m must respect the rule for overriding, see below. Γ maps m's formal parameter and declared local variables to their declared types, **this** to the type **this** C, and the result variable `res` to m's result type

$$\textsc{Wf-Type}\ \dfrac{\begin{array}{c} u = \texttt{rep}\,\langle cn\rangle \Rightarrow \Gamma(\texttt{this}) = \texttt{this}\; C_0 \wedge \\ \wedge cn \in \texttt{clusters}(C_0) \end{array}}{\Gamma \vdash_{wf} u\; C} \qquad \textsc{Wf-Override}\ \dfrac{\begin{array}{c} (\forall C' : C \leq C' \Rightarrow \\ \texttt{mType}(C', m) \text{ is undefined} \vee \\ \texttt{mType}(C, m) = \texttt{mType}(C', m)) \end{array}}{\texttt{override}(C, m)}$$

$$\textsc{Wf-Method}\ \dfrac{\begin{array}{c} \Gamma = p\; T_p, \overline{y\; T}, \texttt{this}\; (\texttt{this}\; C), \texttt{res}\; T_r \\ \Gamma \vdash_{wf} T_p, \overline{T}, T_r \qquad \texttt{override}(C, m) \\ \Gamma; \varnothing \vdash s \qquad\qquad \Gamma; \varnothing \vdash s : \mathcal{U} \\ (\mathit{FieldId} \cup \{\texttt{res}\}) \cap \mathcal{U} = \varnothing \end{array}}{C \vdash_{wf} T_r\; m(T_p\; p)\; \{\overline{T\; y};\; s\}} \qquad \textsc{Wf-Class}\ \dfrac{\begin{array}{c} C \vdash_{wf} \overline{M} \qquad T_i = u_i\; C_i \qquad u_i \neq \texttt{free} \\ (u_i = \texttt{rep}\,\langle cn'\rangle \Rightarrow cn' \in \overline{cn}) \end{array}}{\vdash_{wf} \texttt{class}\; C\; \texttt{extends}\; C'\; \{\overline{cn};\; \overline{T\; f};\; \overline{M}\}}$$

Figure 11. Well-formedness rules.

(Wf-Method). Moreover, after the method body s, neither the result variable nor any field is unusable. Method m respects the rule for overriding if it does not override a method or if all overridden methods have the identical signature (Wf-Override). Function `mType` yields the signature of a method m in a class C, and is undefined if C does not contain a method m.

A class C is well-formed if all of its methods are well-formed, none of the fields has ownership modifier **`free`**, and all cluster names used in field declarations are declared in C (Wf-Class). As discussed in Sec. 4.3, the last constraint allows us to determine all fields affected by a **`release`** statement by inspecting the fields that are declared in the same class as the released cluster. In particular, this analysis does not have to consider fields of subclasses, which is important for modularity.

Well-formed programs do not contain the ownership modifier **`this`**. We do not make this requirement explicit in our rules. The **`this`** modifier is used implicitly for the **`this`** variable as shown by the definition of Γ in Wf-Type and Wf-Method.

5.3 Type Rules

Two types $u_1\; C_1$ and $u_2\; C_2$ are subtypes (denoted by $u_1\; C_1 \leq u_2\; C_2$) if (1) C_1 is a subclass of C_2 according to the rules of Java and (2) the ownership modifiers are identical, u_2 is **`any`**, or u_1 is **`this`** and u_2 is **`peer`**.

The type rules are presented in Fig. 12. The judgment $\Gamma; \mathcal{U} \vdash s$ expresses that statement s is well-typed in environment Γ and unusable-set $\mathcal{U}$. Our type rules implicitly require types to be well-formed, that is, a type rule is applicable only if all types involved in the rule are well-formed in the respective environment. All rules require that no unusable parameter, local variable, or field of **`this`** is read. Our type rules implicitly require types to be well-formed in the respective environment.

The rules for assignment (T-Assign), object creation (T-New), and sequential composition (T-Comp) are straightforward. As explained in Sec. 3, the ownership modifier of a field access is determined by combining the modifier of the receiver and the modifier of the field (T-Read). The function `fType`(C, f) yields the declared type of a field f that is declared in or inherited by class C.

For a field update, the right-hand side variable must be a subtype of the viewpoint-adapted field type (T-Write). The rule is analogous to field read, but has two additional requirements. First, the modifier of the receiver variable must not be **`any`** or **`free`**. **`any`** is forbidden to enforce the owner-as-modifier discipline. It would be type safe to permit updates of **`any`** fields on **`free`** receivers, but we forbid this for simplicity. Second, a **`rep`** $\langle cn\rangle$ field f must be updated through receiver **`this`**. Otherwise, the viewpoint adaptation $u \triangleright u_f$ yields **`any`**, but it is obviously unsafe to update f with an object with an arbitrary owner.

The rule for method calls (T-Call) is in many ways similar to field reads (for result passing) and updates (for argument passing). The last antecedent of T-Call requires that the unusable set does not contain any fields. This is necessary to ensure that the invoked method may assume all fields to be usable.

T-Cast could be strengthened to prevent more cast errors statically, but we omit this check since it is not strictly needed. Casts from **`any`** types to **`free`** types are forbidden because we cannot ensure efficiently that the right-hand side reference is actually externally-unique.

T-Release requires the argument y to have a **`rep`** $\langle cn\rangle$ type. It yields a **`free`** reference. T-Capture requires the captured variable to have a **`free`** type. The ownership modifier u must determine an owner, that is, must be **`peer`** or **`rep`** $\langle cn\rangle$.

5.4 Static Analysis

In this subsection, we present a modular, intraprocedural static analysis to determine unusable variables. The static analysis makes the following variables unusable: First, **`free`** variables if it is not statically guaranteed that the reference held by the **`free`** variable is actually externally-unique. Second, variables of **`rep`** $\langle cn\rangle$ types if the cluster is potentially transferred.

$$\textsc{T-Assign}\;\frac{\begin{array}{c}\Gamma(y)\le\Gamma(x)\\ y\notin\mathcal{U}\end{array}}{\Gamma;\mathcal{U}\vdash x\texttt{ := }y}\qquad \textsc{T-New}\;\frac{u\ne\texttt{any}\qquad u\,C\le\Gamma(x)}{\Gamma;\mathcal{U}\vdash x\texttt{ := }\texttt{new}\;u\,C}\qquad \textsc{T-Comp}\;\frac{\begin{array}{c}\Gamma;\mathcal{U}\vdash s_1\qquad \Gamma;\mathcal{U}'\vdash s_2\\ \Gamma;\mathcal{U}\vdash s_1:\mathcal{U}'\end{array}}{\Gamma;\mathcal{U}\vdash s_1\;s_2}$$

$$\textsc{T-Read}\;\frac{\begin{array}{c}\Gamma(y)=u\,C\qquad u_f\,C_f=\texttt{fType}(C,\;f)\\ (u\rhd u_f)\,C_f\le\Gamma(x)\\ y\notin\mathcal{U}\qquad\qquad y=\texttt{this}\Rightarrow f\notin\mathcal{U}\end{array}}{\Gamma;\mathcal{U}\vdash x\texttt{ := }y\texttt{.}f}\qquad \textsc{T-Write}\;\frac{\begin{array}{c}\Gamma(x)=u\,C\qquad u_f\,C_f=\texttt{fType}(C,\;f)\\ \Gamma(y)\le(u\rhd u_f)\,C_f\qquad u\notin\{\texttt{any},\texttt{free}\}\\ u_f=\texttt{rep}\,\langle cn\rangle\Rightarrow x=\texttt{this}\\ x,y\notin\mathcal{U}\end{array}}{\Gamma;\mathcal{U}\vdash x\texttt{.}f\texttt{ := }y}$$

$$\textsc{T-Call}\;\frac{\begin{array}{c}\Gamma(y)=u\,C\qquad \texttt{mType}(C,m)=u_p\,C_p\to u_r\,C_r\\ \Gamma(z)\le(u\rhd u_p)\,C_p\qquad (u\rhd u_r)\,C_r\le\Gamma(x)\\ u\notin\{\texttt{any},\texttt{free}\}\\ u_p=\texttt{rep}\,\langle cn\rangle\Rightarrow y=\texttt{this}\\ y,z\notin\mathcal{U}\qquad\qquad \mathit{FieldId}\cap\mathcal{U}=\varnothing\end{array}}{\Gamma;\mathcal{U}\vdash x\texttt{ := }y\texttt{.}m\texttt{(}z\texttt{)}}\qquad \textsc{T-Cast}\;\frac{\begin{array}{c}\Gamma(y)=u_y\,C_y\qquad u\,C\le\Gamma(x)\qquad u\,C\le u_y\,C_y\\ u=\texttt{free}\Rightarrow u_y=\texttt{free}\\ y\notin\mathcal{U}\end{array}}{\Gamma;\mathcal{U}\vdash x\texttt{ := (}u\,C\texttt{)}\;y}$$

$$\textsc{T-Release}\;\frac{\begin{array}{c}\Gamma(x)=\texttt{free}\,C\qquad \Gamma(y)=\texttt{rep}\,\langle cn\rangle\;C\\ y\notin\mathcal{U}\end{array}}{\Gamma;\mathcal{U}\vdash x\texttt{ := }\texttt{release(}y\texttt{)}}\qquad \textsc{T-Capture}\;\frac{\begin{array}{c}\Gamma(x)=u\,C\qquad \Gamma(y)=\texttt{free}\,C\\ u\in\{\texttt{peer},\;\texttt{rep}\,\langle cn\rangle\}\\ y\notin\mathcal{U}\end{array}}{\Gamma;\mathcal{U}\vdash x\texttt{ := }\texttt{capture(}y\texttt{,}u\texttt{)}}$$

Figure 12. Type rules.

$$\textsc{U-Assign}\;\frac{\begin{array}{c}\Gamma(x)=u_x\,C_x\qquad \Gamma(y)=u_y\,C_y\\ \mathcal{U}'=\text{if }(u_x=\texttt{any}\vee u_y\ne\texttt{free})\\ \text{then }\mathcal{U}\text{ else }\mathcal{U}\cup\{y\}\end{array}}{\Gamma;\mathcal{U}\vdash x\texttt{ := }y:\mathcal{U}'\setminus\{x\}}\qquad \textsc{U-New}\;\frac{}{\Gamma;\mathcal{U}\vdash x\texttt{ := }\texttt{new}\;T:\mathcal{U}\setminus\{x\}}\qquad \textsc{U-Comp}\;\frac{\begin{array}{c}\Gamma;\mathcal{U}\vdash s_1:\mathcal{U}_1\\ \Gamma;\mathcal{U}_1\vdash s_2:\mathcal{U}_2\end{array}}{\Gamma;\mathcal{U}\vdash s_1;s_2:\mathcal{U}_2}$$

$$\textsc{U-Read}\;\frac{}{\Gamma;\mathcal{U}\vdash x\texttt{ := }y\texttt{.}f:\mathcal{U}\setminus\{x\}}\qquad \textsc{U-Write}\;\frac{\mathcal{U}'=\text{if }(x=\texttt{this})\text{ then }\mathcal{U}\setminus\{f\}\text{ else }\mathcal{U}}{\Gamma;\mathcal{U}\vdash x\texttt{.}f\texttt{ := }y:\mathcal{U}'}$$

$$\textsc{U-Call}\;\frac{\begin{array}{c}\texttt{mType}(C_y,m)=u_p\,C_p\to T_r\\ \Gamma(y)=u_y\,C_y\qquad\qquad \Gamma(z)=u_z\,C_z\\ \mathcal{U}_1=\text{if }(u_p=\texttt{any}\vee u_z\ne\texttt{free})\text{ then }\mathcal{U}\text{ else }\mathcal{U}\cup\{z\}\\ \mathcal{U}_2=\text{if }(u_y\notin\{\texttt{peer},\;\texttt{this}\})\text{ then }\mathcal{U}_1\text{ else}\\ \mathcal{U}_1\cup\{v\in\texttt{dom}(\Gamma)\mid\Gamma(v)=\texttt{rep}\,\langle\_\rangle\;C_v\}\end{array}}{\Gamma;\mathcal{U}\vdash x\texttt{ := }y\texttt{.}m\texttt{(}z\texttt{)}:\mathcal{U}_2\setminus\{x\}}\qquad \textsc{U-Cast}\;\frac{\begin{array}{c}\Gamma(x)=u_x\,C_x\qquad \Gamma(y)=u_y\,C_y\\ \mathcal{U}'=\text{if }(u_x=\texttt{any}\vee u_y\ne\texttt{free})\\ \text{then }\mathcal{U}\text{ else }\mathcal{U}\cup\{y\}\end{array}}{\Gamma;\mathcal{U}\vdash x\texttt{ := (}T\texttt{)}\;y:\mathcal{U}'\setminus\{x\}}$$

$$\textsc{U-Release}\;\frac{\begin{array}{c}\Gamma(y)=u_y\,C_y\qquad \Gamma(\texttt{this})=\texttt{this}\,C_0\\ \mathcal{U}'=\mathcal{U}\cup\{v\in\texttt{dom}(\Gamma)\mid\Gamma(v)=u_y\,C_v\}\cup\\ \{f\in\texttt{fields}(C_0)\;\mid\;\texttt{fType}(C_0,f)=u_y\,C_f\}\end{array}}{\Gamma;\mathcal{U}\vdash x\texttt{ := }\texttt{release(}y\texttt{)}:\mathcal{U}'\setminus\{x\}}\qquad \textsc{U-Capture}\;\frac{\mathcal{U}'=\mathcal{U}\cup\{y\}}{\Gamma;\mathcal{U}\vdash x\texttt{ := }\texttt{capture(}y\texttt{,}u\texttt{)}:\mathcal{U}'\setminus\{x\}}$$

Figure 13. Rules of static analysis.

Our static analysis computes for each program point a set $\mathcal{U}$ of unusable variables. $\mathcal{U}$ is a subset of the fields declared in or inherited by the enclosing class as well as the parameters and local variables of the enclosing method. The rules of the static analysis are presented in Fig. 13. The judgment $\Gamma;\mathcal{U}\vdash s:\mathcal{U}'$ expresses that in an environment Γ, $\mathcal{U}'$ is the unusable-set after the statement s if $\mathcal{U}$ is the unusable-set before the statement.

Assigning to a variable x removes x from the unusable set (U-Assign). The right-hand side variable y becomes unusable if it has a **free** type and x does not have an **any** type. Under this condition, the assignment creates an external reference into the cluster into which y points such that y is no longer externally-unique and, thus, must become unusable.

Creating a new object makes the left-hand side variable x usable (U-New). The rule for sequential composition is straightforward (U-Comp).

Field read (U-Read) is similar to assignment, but simpler because the field f and, therefore, the right-hand side cannot have a **free** type. Updating a field removes it from the unusable set if the receiver is **this**. Since the static analysis only tracks fields of **this**, there is no effect for other receivers.

The most interesting rule handles method calls (U-CALL). Analogously to assignments, the actual argument z becomes unusable if it has modifier **free** and is passed to a non-**any** parameter. If the receiver has a **peer** or **this** modifier then all parameters and local variables with a **rep** $\langle \_ \rangle$ modifier become unusable. These variables hold external references into clusters that are potentially transferred by the called method if this method is executed on receiver **this** or calls back into **this**. Therefore, we conservatively make these variables unusable. Note that this is only done for **peer** or **this** receivers. For **rep** $\langle cn \rangle$ receivers, UTT prevents callbacks into **this** via non-pure methods. Even in the presence of pure methods, this rule is sufficient because pure methods must not release any cluster.

Even though our treatment of calls on receivers with a **peer** or **this** modifier is conservative, it is not a severe restriction in practice. Local variables with a **peer** or **any** modifier remain usable after the call (as do variables pointing into non-transferable clusters [35]). Moreover, the value of a variable with a **rep** $\langle cn \rangle$ modifier can be carried over by assigning it to an **any** variable before the call and casting it back to a **rep** $\langle cn \rangle$ modifier after the call. The runtime check associated with the cast fails if the cluster has been transferred by the call. This allows programmers to by-pass our conservative type rules in cases they know that a certain cluster is not transferred by a method call.

Casts are completely analogous to assignments (U-CAST). A **release**(y) operation makes all variables unusable that point into the released cluster. These are the parameters and local variables with the same ownership modifier as y as well as all fields declared in or inherited by the enclosing class with this modifier. This guarantees that the reference returned by the **release** is externally-unique. **capture** makes the captured variables unusable because it is no longer externally-unique.

Note that the static analysis only tracks parameters and locals of the enclosing method as well as fields of the enclosing class and its superclasses. Therefore, the analysis is fully modular and intraprocedural. An interprocedural analysis would permit a less conservative call rule for receivers with modifier **peer** and **this**. However, in the presence of dynamic method binding, interprocedural analyses are inherently non-modular because they require knowledge of all method overrides.

6. Examples

In this section, we illustrate UTT by four examples: the merging of data structures, object initialization, the Factory pattern, and a work flow system. In the examples, we use a Java-like syntax with constructors and expressions. A translation into the language subset supported by UTT is straightforward: constructors can be replaced by initialization methods and expressions can be eliminated using temporary variables.

```
class Lexer {
  cluster S;
  rep<S> InputStream stream;

  Lexer(free InputStream s) {
    stream = capture(s, rep<S>);
  }
}

class Client{
  cluster T;

  void main(any String file) {
    rep<T> InputStream s;
    s = new rep<T> InputStream(file);
    free InputStream fs = release(s);

    rep<T> Lexer lexer = new rep<T> Lexer(fs);
    // ...
  }
}
```

Figure 14. Example of an object initialization. Lexer's constructor captures its parameter.

6.1 List Merging

In this subsection, we revisit the implementation of class MList (Fig. 6). It is easy to see that reverse is well-formed (WF-METHOD, Fig. 11). First, throughout the method body, the set of unusable variables $\mathcal{U}$ is empty. Second, the method body type checks because all expressions except for the loop condition have type **rep**<R> Node and no unusable variables are accessed.

Method getNodes illustrates the **release** statement. Releasing first makes all local variables and fields with modifier **rep**<R> unusable. WF-METHOD requires that upon termination, all fields are usable. This is achieved by setting first to a new node. Without this field update, getNodes would not be well-formed.

Method merge performs the actual ownership transfer. By capturing un, this variable becomes unusable because it is no longer externally-unique. However, we need not assign a new value to un because it dies when the method terminates. Like in alias burying [10], we defer the update of a variable until it is used, and the update is not necessary for local variables that are never used again.

The MList example illustrates that our technique requires very little overhead beyond the Universe annotations. In particular, methods that do not perform transfers, such as reverse, can be written like in standard Universe Types.

```
class Product { /* ... */ }

class Factory{
  cluster T;

  pure free Product create() {
    rep<T> Product t = new rep<T> Product();
    return release(t);
  }
}

class Client{
  cluster P;

  rep<P> Product getProduct(any Factory f) {
    free Product p = f.create();
    return capture(p, rep<P>);
  }
}
```

Figure 15. Implementation of the Factory pattern.

6.2 Object Initialization

The example in Fig. 14 is adapted from [17]. It illustrates how our system supports object initialization. The constructor of class `Lexer` expects a **`free`** `InputStream`, which is captured by the lexer. That is, it is transferred to the `S` cluster of **`this`** and then stored in a field.

Method `main` of class `Client` creates a `Lexer`. It first creates an `InputStream` in cluster `T` and then releases it. We do not create a **`free`** `InputStream` here, because a constructor call corresponds to a method call, and our system does not permit calls on **`free`** receivers. The input stream is then passed to the constructor of the `Lexer`, where it is captured. The **`release`**`(s)` operation makes all variables with modifier **`rep`** $\langle$`T`$\rangle$ unusable. This prevents the following statements from using `s`.

6.3 Factory Pattern

Fig. 15 shows an implementation of the Factory pattern. The `create` method of `Factory` creates a new product, releases it, and returns it. Like in `Client`'s `main` method (Fig. 14), we do not create a **`free`** object to be able to call a constructor. Since `create` does not modify any existing objects, we declared it **`pure`**. This allows `getProduct` to call it on the **`any`** receiver `f`. After obtaining a **`free`** `Product` from the factory, `getProduct` transfers it to cluster `P`.

6.4 Work Flow

The examples we considered so far transfer either the whole internal representation of a data structure (`MList`) or newly created objects (`Lexer` and `Factory`). The next example repeatedly transfers ownership of a single object. This pattern occurs for instance for packets in a communication system or tasks in a work flow application. It can be implemented in UTT using multiple clusters.

Fig. 16 shows the implementation of a simple work flow system, where a dispatcher (class `Dispatcher`) sends orders (class `Order`) through a pipeline of processors (abstract class `Processor`). As illustrated by method `process` of `Processor`, each processor captures the order in cluster `O`, stores it in the field `current`, performs its operations, and releases the order again.

Note that our implementation does not maintain external uniqueness of `Order` objects throughout the work flow. Orders are temporarily stored in the field `current` to reduce parameter passing for more complex implementations of `doWork`. Capturing the order also allows processors to modify it. Nevertheless, we can release the `Order` object again for the transfer to the next processor.

Concrete implementations of a processor (such as class `Pricer`) may declare additional clusters to store local data. For instance, `Pricer` maintains a collection of special offers in its `P` cluster. The local data could also contain **`any`** references to orders in cluster `O`, for instance, to maintain a cache or statistical information. Using a separate cluster for local data allows method `process` to release the order without making the local data unusable.

7. Soundness

In this section, we summarize the properties guaranteed by UTT. The formalization and proof of these properties is beyond the scope of this paper, but see our technical report [35]. The static guarantees provided by UTT are summarized by the following soundness theorem:

Theorem 7.1 (Soundness)**.** *In each execution state of a well-formed program, the following properties hold:*

1. Type safety*: The ownership modifier of the type of a* usable *variable v reflects the owner of the object referenced by v.*
2. Tree order*: The ownership relation among the objects in the heap is a tree order.*
3. External uniqueness*: There is at most one external reference into each free cluster.*
4. Alias invariant*: The alias invariant (Def. 4.4) holds.*

Moreover, the following property holds for each well-typed statement s:

5. Owner-as-modifier*: The execution of s modifies only fields and ownership of those objects that, in the state before executing s, are (transitively) owned by the owner of* **`this`** *or by an object in a free cluster.*

Type safety is restricted to usable variables. It includes, in particular, that a usable **`free`** variable references an object in a free cluster. In combination with external uniqueness, this implies that **`free`** variables hold externally-unique references. The owner-as-modifier property permits modifications of objects in a free cluster. These modifications occur

```
class Order {
  // order data:
  int clientId;
  rep List items;

  // fields to be filled during work flow:
  int total;

  // other fields and methods omitted
}


abstract class Processor {
  cluster O;
  rep<O> Order current;

  free Order process(free Order order) {
    current = capture(order, rep<O>);
    doWork();
    free Order res = release(current);
    current = null;
    return res;
  }

  abstract void doWork();
}


class Pricer extends Processor {
  cluster P;
  rep<P> HashMap offers;

  void doWork() {
    int price = /* determine price */
    current.total = price;
  }

  // other methods omitted
}


class Dispatcher {
  Processor[] pipeline;

  void handleOrder(free Order o) {
    for (int i = 0; i < pipeline.length; i++)
      o = pipeline[i].process(o);
  }

  // other methods omitted
}
```

Figure 16. Work Flow Example.

if the cluster is captured during the execution of the statement s.

We proved the conjunction of all properties of Theorem 7.1 by rule induction on an operational semantics [35]. The base case covers all primitive statements; the induction step covers sequential composition and method calls. The execution of each statement preserves properties 1–4 and satisfies property 5. We only sketch the proofs of the most interesting cases here.

***External uniqueness for* `release`.** Consider the statement x `:=` **`release`**`(`y`)`. By T-RELEASE, y has ownership modifier **`rep`** $\langle cn \rangle$. The statement transfers all objects owned by <**`this`**, cn> to a new free cluster. By type safety before the **`release`**, we know that no usable **`free`** variable v references one of the transferred objects. Since these objects are transferred to a new cluster, v remains externally-unique. It remains to show that x is externally-unique after the **`release`**.

Since the alias invariant holds for **`this`** before the **`release`**, we know that all usable external references to objects owned by <**`this`**, cn> are held by fields of **`this`** that are declared in the same class as cn (WF-CLASS and type safety) or by local variables or parameters of the current method execution. According to U-RELEASE, these variables are unusable after the **`release`**. Therefore, x holds the only usable external reference into the released cluster.

***Type safety for* `capture`.** Consider the statement x `:=` **`capture`**`(`y`,`u`)`. The statement transfers all objects in the cluster referenced by y to a non-free cluster described by u.

By T-CAPTURE, y is usable and has ownership modifier **`free`**. By external-uniqueness, y holds the only usable external reference to a transferred object. Consequently, y is the only variable whose type safety is potentially affected by the capturing. In particular, the type safety of **`peer`** variables is not affected because all objects in the captured cluster are transferred together, that is, they remain peers. Since y is unusable after the **`capture`** (U-CAPTURE), type safety is preserved. Variable x is well-typed because it has the ownership modifier u (T-CAPTURE).

Alias invariant for method calls. Consider the call x `:=` y`.`m`(`z`)`. By T-CALL, we know that y has modifier **`this`**, **`peer`**, or **`rep`** $\langle cn \rangle$. We continue by case distinction.

Case (1): y has ownership modifier **`this`** or **`peer`**. By type safety, **`this`** and y have the same owner. Therefore, the same set of objects S must be confined in the caller and the callee method.

First, we show that passing control to the callee preserves the alias invariant. For all objects $o \in S$ other than the objects referenced by **`this`** and y, confinement is preserved by passing control because fields are unchanged and unusable variables remain unusable.

For **this**, the local variables and parameters with ownership modifier $\texttt{rep}\,\langle\_\rangle$ for any cluster name become unusable (U-CALL). Because of type safety, these are the only variables that hold external references into a cluster of **this**.

There are no usable external references into a cluster of y before control is passed. If y and **this** hold different references, this property follows from the alias invariant before the call. If y and **this** reference the same object, this is the case because the local variables with ownership modifier $\texttt{rep}\,\langle\_\rangle$ are unusable. Therefore, after parameter passing, the only local variables and parameters that hold external references into a cluster of y belong to the execution of the callee method.

In summary, the alias invariant holds after control has been passed to the callee method. By the induction hypothesis, it is preserved by the method body. When control returns to the caller, the top stack frame is removed, which trivially preserves the alias invariant by reducing the number of variables. Assigning the result value to x preserves the alias invariant because x is a local variable of the caller.

Case (2): y has ownership modifier $\texttt{rep}\,\langle cn\rangle$. By type safety, we know that the owner object of y is **this**. Therefore, **this** need not be confined while the callee method executes. By the alias invariant before the call, we know that the alias invariant holds after passing control to the callee.

By the induction hypothesis, the alias invariant is preserved by the method body. By the owner-as-modifier property, the callee method does not change fields of **this**. Because of type safety, these are the only fields that hold external references into a cluster of **this**. Moreover, if the callee transfers objects into a cluster of **this**, then we know that these objects are not referenced from the stack. This is the case because the callee can transfer either free clusters (in this case, we use external uniqueness) or clusters released by the callee (in this case, the clusters are transitively owned by y; therefore, their owner object is confined before the call). Consequently, the method body also preserves confinement of **this**.

Removing the stack frame for the callee does not affect the local variables and parameters of the caller. Finally, assigning the result value to x preserves the alias invariant because x is a local variable of the caller. $\square$

8. Implementation

We implemented UTT as part of the MultiJava compiler [41]. In this section, we highlight the most interesting aspects of this implementation, namely runtime support, inference of **release** and **capture** statements, and inference of cluster information for local variables.

Runtime Support. As explained in Sec. 4.1, conceptually each object stores a reference to its owner object and the name of the cluster it belongs to. In the implementation, we represent the owner of an object by a designated *pseudo owner* object. The objects in one cluster can have different pseudo owners, which are organized in a union find structure.

The indirection through pseudo owners enables an efficient implementation of **release** and **capture**. Capturing a cluster is implemented by connecting the union find structures of pseudo owners for the involved clusters. Releasing a cluster is implemented by setting the owner reference in the union find structure of pseudo owners for that cluster to `null`. Both operations require amortized constant time.

Inference of Release and Capture. To reduce the annotation overhead, our implementation infers **release** and **capture** statements through the following simple rules. Whenever a program attempts to assign a $\texttt{rep}\,\langle cn\rangle$ expression to a **free** variable, a **release** is performed. Whenever a program attempts to assign a **free** expression to a $\texttt{rep}\,\langle cn\rangle$ or **peer** variable, a **capture** is performed.

For instance in method `getNodes` of class `MList` (Fig. 6), our implementation permits the assignment **free** `Node res = first`, and infers the **release**. Similarly, the **capture** in method `merge` is inferred.

Inference of Cluster Information. To further reduce the annotation overhead, we also infer the clusters of local **rep** variables from the clusters declared for fields, method parameters, and method results. This inference is done by a data flow analysis that keeps track of all possible clusters a variable may point to at each program point.

For most of our examples, the inference seems trivial because there is only one cluster in each class. However, our implementation also supports an implicitly declared non-transferable cluster. Therefore, the inference has to choose at least among two clusters. For instance in method `reverse` (Fig. 6), our implementation allows programmers to omit the cluster information for the local variable `p`. The data flow analysis determines from the declarations of the fields `first` and `prev` that `p` points to the `R` cluster throughout the method body. In method `merge`, we infer from the various field updates that local variable `rn` points to cluster `R`. This cluster is then used for the inferred **capture** statement.

The inference of **release** and **capture** statements as well as of clusters for local variables eliminates almost all annotation overhead of UTT over Universe Types. Programmers merely have to declare clusters and annotate field declarations and method signatures with cluster information.

9. Related Work

We discussed uniqueness and capabilities in Sec. 2. In this section, we discuss related work on ownership.

The existing ownership type systems enforce different forms of encapsulation. Ownership type systems following the *owner-as-dominator* discipline [6, 8, 13, 14, 16, 15, 38, 42] require that all reference chains from an object in the root context to an object o in a different context go through o's owner. This restriction on aliasing allows owners to control all accesses to owned objects, for instance, to guarantee representation independence. UTT enforces the owner-as-modifier discipline. Since this discipline does not restrict aliasing, it can handle some patterns that are not supported by the owner-as-dominator discipline such as collections with iterators or the Flyweight pattern [20, 36]. The owner-as-modifier discipline is also used in Universe Types [20], Generic Universe Types [19], and Lu and Potter's work [30].

Most existing ownership type systems do not support transfer [8, 13, 14, 16, 19, 20, 30, 38], but could adopt the technique presented in this paper. Our technique requires that a method cannot call non-pure methods on (transitive) owners of its receiver. This requirement is not enforced by the existing owner-as-dominator systems, but an adaptation is straightforward.

Hogg's islands [25] use (strictly) unique variables with destructive reads to permit transfer. We have discussed the problems of this approach in Sec. 2. Our system solves these problems by building on external uniqueness and by permitting temporary aliases. Flexible alias protection [37] offers a **free** mode for (strictly) unique references, but does not provide a technique to enforce uniqueness.

Clarke and Wrigstad [15, 42] permit ownership transfer using external uniqueness, destructive reads, and borrowing. While our system builds on their idea of external uniqueness, it requires less annotation overhead. Clarke and Wrigstad's type system is owner-parametric. Therefore, they have to enforce that an object *o* is transferred only to contexts where *o*'s ownership parameters are available. In our system, objects can be transferred to any context. SafeJava [6] adopted Clarke and Wrigstad's approach. A notion similar to external uniqueness was also proposed by Banerjee and Naumann [5], but without details of how to enforce it.

AliasJava [2] supports ownership transfer using destructive field reads and lent variables. As discussed in Sec. 2, lent variables compromise encapsulation, whereas our **any** references preserve encapsulation because they cannot be used for modifications. Making lent variables in AliasJava read-only would be too restrictive since they allow the only temporary aliases on unique objects in AliasJava. Therefore, passing a unique reference to a non-pure method would be as cumbersome as without temporary aliases. Ownership domains [1] adopt ideas from AliasJava, but use external uniqueness, although the paper does not describe how it is enforced. Our clusters are a restricted version of ownership domains. In particular, we do not provide public clusters and links between clusters.

Our confinement property was inspired by our verification methodology for ownership-based object invariants [34]. The treatment of callbacks is analogous: Both confinement and object invariants need not hold for the (transitive) owners of the current receiver object because callbacks via non-pure methods are prevented by Universe Types. The ownership-based invariant of **this** is checked before peer calls to avoid problems with callbacks, just like confinement is enforced before peer calls by making local variables and parameters unusable.

Spec# [29] uses a dynamic encoding of ownership via a ghost field `owner` and object invariants. With dynamic ownership, transfer amounts to an update of the `owner` field. Like in our system, the invariants of the transitive owners of an object *o* may be temporarily violated when *o* is transferred. While dynamic ownership is very flexible, it requires program verification to check ownership properties, whereas our system permits syntactic checking.

In Sing# [21], processes can own data in a designated exchange heap. Ownership is transferred when the data is sent to another process. A static data-flow analysis enforces that a process only accesses data it owns. To track local aliases, Sing# builds on capabilities [22], which we discussed in Sec. 2. Sing# does not support deep (hierarchic) ownership, and the use of a designated exchange heap is too restrictive for general object-oriented programming.

Shape analysis [40] can be used to infer alias structures. However, most shape analyses are whole-program analyses, whereas our technique is fully modular. Rinetzky et al. [39] present a technique to realize modular shape analyses based on dynamic ownership. Their system supports ownership transfer using constraints and annotations similar to Clarke and Wrigstad's work [15, 42]. We expect that our work can be combined with the modular shape analysis to overcome the shortcomings of destructive reads and to reduce the annotation overhead.

Role analysis [28] is a general technique to describe aliasing relationships between objects via types. It requires programmers to provide role descriptions, role specifications for method signatures, as well as read and write effects for methods. UTT expresses simpler properties, but requires significantly less annotation overhead. Role analysis does not provide encapsulation, which is one of the main motivations of our work. In both role analysis and UTT, the reconfiguration of object structures changes the types of objects to reflect a change of roles or ownership, respectively. Role analysis permits objects referenced from the stack to violate their type; dangerous callbacks are detected by an interprocedural, non-modular analysis. UTT enforces modularly that all potential receivers of calls to non-pure methods are confined; therefore, ownership transfer cannot violate type safety.

10. Conclusions

We presented UTT, an extension of Universe Types that supports clusters and ownership transfer. Even though many references may point into a cluster, our modular static analysis enforces that a cluster is externally-unique at the time of transfer. UTT is very flexible because is permits temporary aliases, several fields pointing into one cluster, and **any** references. UTT can handle most ownership transfer examples. An example it cannot handle is the splitting of lists because we have no operation to split a cluster into two. Such an operation in general requires a whole-program analysis or reference counting to ensure that the alias invariant is preserved by the split.

As future work, we plan to use our implementation to assess the expressiveness of UTT in case studies. Two extensions seem particularly useful. First, Universe Types provide very limited support for static fields; type-safety requires that all static fields have **`any`** modifiers, which prevents modification of global data. We will investigate an extension of UTT where static fields are **`free`** such that methods can capture the global data, modify it, and release it again.

Second, when releasing a cluster, UTT makes all external references except for one unusable. To retain several usable references, for instance, to the first and last node of a list, programmers have to introduce artificial bridge objects. A bridge object sits in the same cluster as the nodes and holds peer references to the first and last node. Releasing the cluster yields a **`free`** reference to the bridge object, which can be captured and then accessed to obtain usable references to the first and last node. We are working on an extension of UTT that can release a cluster while retaining several usable external references, without requiring bridge objects.

Finally, we are working on an extension of the type inference for local variables to infer all ownership modifiers for locals. This will further reduce the annotation overhead.

Acknowledgments

Thanks to Werner Dietl, James Noble, and Joseph Ruskiewicz for insightful discussions and suggestions, and to the anonymous reviewers for their helpful comments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. Müller's work was carried out at ETH Zurich.

References

[1] 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 *LNCS*, pages 1–25. Springer-Verlag, 2004.

[2] 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.

[3] C. Andrea, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time systems. In D. Thomas, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *LNCS*. Springer-Verlag, 2006.

[4] A. Banerjee and D. Naumann. Representation independence, confinement, and access control. In *Principles of Programming Languages (POPL)*, pages 166–177. ACM, 2002.

[5] A. Banerjee and D. Naumann. Ownership: transfer, sharing, and encapsulation. In S. Eisenbach, G. T. Leavens, P. Müller, A. Poetzsch-Heffter, and E. Poll, editors, *Formal Techniques for Java-like Programs*, 2003.

[6] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230. ACM Press, 2002.

[8] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, pages 213–223. ACM Press, 2003.

[9] C. Boyapati, A. Salcianu, J. W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Programming language design and implementation (PLDI)*, pages 324–337. ACM Press, 2003.

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

[11] J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *Principles of programming languages (POPL)*, pages 283–295. ACM Press, 2005.

[12] P. Chalin and P. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, LNCS. Springer-Verlag, 2007. To appear.

[13] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.

[14] 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.

[15] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In L. Cardelli, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 176–200. Springer-Verlag, 2003.

[16] 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.

[17] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Research Report 156, Digital Systems Research Center, 1998.

[18] W. Dietl, S. Drossopoulou, and P. Müller. Formalization of Generic Universe Types. Technical Report 532, ETH Zurich, 2006. `sct.inf.ethz.ch/publications`.

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

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

[21] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys '06*, pages 177–190. ACM Press, 2006.

[22] M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *Programming language design and implementation (PLDI)*, pages 13–24. ACM Press, 2002.

[23] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Object-oriented programing, systems, languages, and applications (OOPSLA)*, pages 302–312. ACM Press, 2003.

[24] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In R. D. Nicola, editor, *European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*. Springer-Verlag, 2007.

[25] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 271–285. ACM Press, 1991.

[26] B. Jacobs, F. Piessens, K. R. M. Leino, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Software Engineering and Formal Methods (SEFM)*, pages 137–147. IEEE Computer Society, 2005.

[27] M. Klebermaß. An Isabelle formalization of the Universe Type System. Master's thesis, Technische Universität München, 2007. `sct.inf.ethz.ch/projects/student_docs/Martin_Klebermass`.

[28] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Principles of programming languages (POPL)*, pages 17–32. ACM Press, 2002.

[29] 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.

[30] Y. Lu and J. Potter. Protecting representation with effect encapsulation. In *Principles of programming languages (POPL)*, pages 359–371. ACM Press, 2006.

[31] Y. Lu, J. Potter, and J. Xue. Object Invariants and Effects. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS. Springer-Verlag, 2007. To appear.

[32] N. H. Minsky. Towards alias-free pointers. In P. Cointe, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 1098 of *LNCS*, pages 189–209. Springer-Verlag, 1996.

[33] P. Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

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

[35] P. Müller and A. Rudich. Formalization of ownership transfer in Universe Types. Technical Report 556, ETH Zurich, 2007. `sct.inf.ethz.ch/publications`.

[36] S. Nägeli. Ownership in design patterns. Master's thesis, ETH Zurich, 2006. `sct.inf.ethz.ch/projects/student_docs/Stefan_Naegeli`.

[37] J. Noble, J. Vitek, and J. M. Potter. Flexible alias protection. In E. Jul, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *LNCS*. Springer-Verlag, 1998.

[38] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, pages 311–324. ACM Press, 2006.

[39] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In R. D. Nicola, editor, *European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*. Springer-Verlag, 2007.

[40] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[41] Y. Takano. Implementing uniqueness and ownership transfer in the Universe Type System. Master's thesis, ETH Zurich, 2007. `sct.inf.ethz.ch/projects/student_docs/Yoshimi_Takano`.

[42] T. Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Royal Institute of Technology Stockholm, 2006.