Separating Ownership Topology and Encapsulation with Generic Universe Types

WERNER DIETL, University of Washington
SOPHIA DROSSOPOULOU, Imperial College London
PETER MÜLLER, ETH Zurich

20

Ownership is a powerful concept to structure the object store and to control aliasing and modifications of objects. This article presents an ownership type system for a Java-like programming language with generic types.

Like our earlier Universe type system, Generic Universe Types structure the heap hierarchically. In contrast to earlier work, we separate the enforcement of an ownership topology from an encapsulation system. The topological system uses an existential modifier to express that no ownership information is available statically. On top of the topological system, we build an encapsulation system that enforces the owner-as-modifier discipline. This discipline does not restrict aliasing, but requires modifications of an object to be initiated by its owner. This allows owner objects to control state changes of owned objects—for instance, to maintain invariants. Separating the topological system from the encapsulation system allows for a cleaner formalization, separation of concerns, and simpler reuse of the individual systems in different contexts.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Encapsulation, generic, ownership types, owner-as-modifier, topology, universe types

ACM Reference Format:
Dietl, W., Drossopoulou, S., and Müller, P. 2011. Separating ownership topology and encapsulation with generic universe types. ACM Trans. Program. Lang. Syst. 33, 6, Article 20 (December 2011), 62 pages.
DOI = 10.1145/2049706.2049709 http://doi.acm.org/10.1145/2049706.2049709

1. INTRODUCTION

The concept of object ownership allows programmers to structure the object store hierarchically and to control aliasing and access between objects. Ownership has been applied successfully to various problems—for instance, program verification [Drossopoulou et al. 2008; Leino and Müller 2004; Müller 2002; Müller et al. 2006]; thread synchronization [Boyapati et al. 2002; Jacobs et al. 2005]; memory management [Andrea et al. 2006; Boyapati et al. 2003]; and representation independence [Banerjee and Naumann 2002].

Most of this work was finished while W. Dietl was at ETH Zurich.
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, and the EPSRC grant Practical Ownership Types for Objects and Aspect Programs, EP/D061644/1.
Author contact address: wmdietl@cs.washington.edu.

DOI 10.1145/2049706.2049709 http://doi.acm.org/10.1145/2049706.2049709

Most ownership models share a fundamental topology, whereby each object has at most one owner object. The set of all objects with the same owner is called a *context*. The *root context* is the set of objects with no owner. The ownership relation is a tree order.

However, existing models differ in the encapsulation system they enforce. The original ownership types [Clarke et al. 1998] and their descendants [Boyapati 2004; Clarke 2001; Clarke and Drossopoulou 2002; Potanin et al. 2006] restrict aliasing and enforce the *owner-as-dominator* discipline: All reference chains from an object in the root context to an object o in a different context go through o's owner. In the original ownership types and Clarke's thesis [Clarke 2001], this restriction applies to references stored in fields as well as to references stored in local variables. This severe restriction of aliasing is necessary for some of the applications of ownership, for instance, memory management [Boyapati 2004; Boyapati et al. 2003] and representation independence [Banerjee and Naumann 2002]. Various variants of ownership types enforce a weaker form of the owner-as-dominator discipline by allowing certain references to bypass owners, for instance, references in some local variables [Aldrich et al. 2002] or references from inner-class objects [Boyapati 2004].

For some other applications such as program verification, restricting aliasing is not necessary. Instead, it suffices to restrict the set of references which can be used to modify an object and to enforce the *owner-as-modifier* discipline: An object o may be referenced by any other object, but reference chains that do not pass through o's owner must not be used to modify o. Analogously to the original ownership types, reference chains may include references stored in fields as well as in local variables. This allows owner objects to control state changes of owned objects and thus maintain invariants. The owner-as-modifier discipline was inspired by Flexible Alias Protection [Noble et al. 1998]. Variations of it are enforced by the Universe type system [Dietl and Müller 2005; Cunningham et al. 2008]; in Spec♯'s dynamic ownership model [Leino and Müller 2004]; and Effective Ownership Types [Lu and Potter 2006b].

By restricting modifications rather than the existence of references, the owner-as-modifier discipline supports common implementations where objects are shared between objects, such as collections with iterators, shared buffers, or the Flyweight pattern [Dietl and Müller 2005; Nägeli 2006]. Some implementations can be slightly adapted to satisfy the owner-as-modifier discipline, for example an iterator can delegate modifications to the corresponding collection which owns the internal representation.

According to the above definition (formalized in our recent work on nongeneric Universe types [Cunningham et al. 2008]), owner-as-modifier imposes strictly weaker restrictions than the owner-as-dominator discipline of the original ownership types. However, in this article, as in our earlier work on Universe types, we use a slightly more restrictive definition of owner-as-modifier, which forbids some modifications that owner-as-dominator allows, namely, modifications of objects in ancestor contexts of the context containing the current receiver. For illustration, consider the objects in Figure 1, in a UML-like notation, which describe a simple representation of a map. In this object graph, the reference from the `Node` instance 5 to the `Data` instance 2 could be used for modifications in the owner-as-dominator discipline and the above definition of the owner-as-modifier discipline; however, the owner-as-modifier discipline we enforce in this article forbids modifications through such references. Therefore, the encapsulation guarantee from Generic Universe Types is not comparable to the owner-as-dominator discipline, in the sense that neither implies the other one, even though for most practical examples the owner-as-modifier discipline provides the weaker guarantees. We also do not consider the relaxations (for instance, regarding local variables and inner-class objects) that have been used in variations of ownership types. The main

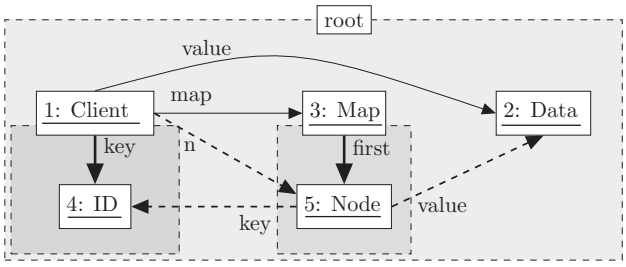


Fig. 1. Object structure of a map from `ID` to `Data` objects. The map is represented by `Node` objects. The client has a direct reference to a node. Objects, references, and contexts are depicted by rectangles, arrows, and dashed rectangles, respectively. Owner objects sit atop the context of objects they own. Arrows are labeled with the name of the variable that stores the reference. Dashed arrows depict references that cross context boundaries without going through the owner. If the owner-as-modifier discipline is enforced, such references must not be used to modify the state of the referenced objects. The source code for this example is shown in Figures 2, 3, and 4.

choice, however, is in the intended application domain, with the owner-as-dominator and owner-as-modifier disciplines geared towards different applications.

Because Universe types have very simple annotations, they are, in some cases, unable to express all necessary ownership information for programs to type-check. Rather than make such programs type-incorrect, we introduce casts into the language. For this reason, we also represent the ownership structure at runtime. In contrast, original ownership types and their descendants rely exclusively on static checks.

Although ownership type systems have covered all features of Java-like languages (including, for example, exceptions, inner classes, and static class members) there are only a few proposals of ownership type systems that support generic types. The goal of the combination is, for example, to type a collection of `Book` objects as "my collection of library books," expressing that the collection object belongs to the current `this` object, whereas the `Book` objects in the collection belong to an object "library".

SafeJava [Boyapati 2004] supports type parameters and ownership parameters independently, but does not integrate both forms of parametricity. This leads to significant annotation overhead. Ownership Domains [Aldrich and Chambers 2004] combine type parameters and domain parameters into a single parameter space, and thereby reduce the annotation overhead. However, type parameters are not covered by their formalization. Ownership Generic Java (OGJ) [Potanin et al. 2006] allows programmers to attach ownership information through type parameters. OGJ piggybacks ownership information on type parameters—in particular, each class C has a type parameter to encode the owner of a C object. OGJ enforces the owner-as-dominator discipline. A later extension combines OGJ with immutability [Zibin et al. 2010]. Jo∃ [Cameron and Drossopoulou 2009; Cameron 2009] combines the theory on existential types with a parametric ownership type system. Ownership information is passed as additional type parameters, and existential types can be used to allow subtype variance. Jo$\exists_{deep}$ provides optional enforcement of the owner-as-dominator discipline.

In this article, we present Generic Universe Types (GUT), an ownership type system for a programming language with generic types similar to Java 5 and C♯ 2.0. A key feature of GUT is that it separates the ownership topology from the encapsulation system. Cleanly separating the ownership topology from the encapsulation system improves the formalization and presentation of ownership type systems and increases their flexibility.

The topological system describes and enforces the hierarchical ownership structure of the object store that we described above, but does not impose any restrictions on

references or operations; any object can reference and access any other object in the heap. Imposing restrictions is the task of the separate encapsulation system, which is built on top of the topological system. The encapsulation system of GUT enforces the owner-as-modifier discipline; but it is possible to define other encapsulation systems. For instance, we could use Spec♯'s verification-based encapsulation system [Leino and Müller 2004] on top of our topological system.

Enforcing only the topological system already provides strong guarantees. Suppose there was a second map object in Figure 1. The topological system guarantees that the `first` fields of the two maps will reference two different node objects. Mixing nodes from different maps is prevented. Even without an encapsulation system, this guarantee is sufficient for some applications of ownership—for instance, to show the absence of data races [Cunningham et al. 2007] or to prove termination [Ádám et al. 2007]. The encapsulation discipline additionally ensures that the owner-as-modifier discipline is adhered to. For example, reference `n` from the client (object 1) to the node (object 5) cannot be used to modify object 5, and reference `value` from the node (object 5) to the data (object 2) cannot be used to modify object 2, as these references bypass the owners of the referenced objects. The topological system alone permits these modifications, and we could design alternative encapsulation systems that would permit them as well (for instance, the owner-as-dominator discipline would allow object 5 to modify object 2 through the `value` reference).

The separation of topology and encapsulation is simplified by distinguishing between a reference that can refer to an arbitrary object and a reference that points to a specific context, but where this specific context is not expressible in the type system. We distinguish between the "don't care" modifier `any` that can reference an arbitrary object and the "don't know" modifier `lost` that references an object for which the precise ownership information cannot be expressed statically. Updates of `any` variables are always possible, since the owner of their value is not of interest. Updates of `lost` variables must be forbidden, since the ownership information required for type-safe updates is not statically known.

The idea of separating topological information from further guarantees is not new. In the following systems, the topological properties are described through one set of annotations, while further annotations restrict accesses between objects [Aldrich and Chambers 2004; Lu and Potter 2006a]; or restrict the write effects [Lu and Potter 2006b]; or restrict invalidation of invariants [Lu et al. 2007]. Our work is novel, in that it is the first to separate the topological aspects from the encapsulation aspects in the setting of Generic Universe Types. Furthermore, while in the aforementioned works the extra guarantees can only be given through extra type annotations, in our work, the extra guarantees (i.e., the encapsulation system) do not require additional annotations; they are simply enforced through a stronger version of the type system.

Contributions and Comparison to our Earlier Work. Our work on Generic Universe Types makes the following contributions: (1) a formal integration of type parameters and ownership; (2) a clean separation of the topological system from the encapsulation system; (3) more static ownership information; (4) a formalization using Ott [Sewell et al. 2007], a system that provides support for defining languages by having a simple input language, sort-checking input, and producing LaTeX output (we did not use Ott's theorem prover output, however); and (5) a type checker implemented in the Checker Framework [Papi et al. 2008; Ernst 2008]. The Ott formalization and the type checker implementation are available from http://www.cs.washington.edu/homes/wmdietl/ownership/.

In order to be able to separate the topological system from the encapsulation system, we adopted the modifier `lost`, which we had introduced in our earlier work on

nongeneric Universe Types [Cunningham et al. 2008], into the Generic Universe Types work. Since the presence of the `lost` modifier is pervasive in the type system and the generic type system is much more powerful than the nongeneric one, this adoption required a complete revisit and rethinking of our original Generic Universe Types work [Dietl et al. 2007], an extension of the definition of dynamization, and the introduction of the new concepts of strict subtypes and strict well-formedness. The `lost` modifier allows us to support ownership covariance in a way that retains more static knowledge about the ownership topology than the original Generic Universe Types. The additional static ownership information allows us to express examples that were previously not typable. Our current work makes GUT formally cleaner and more expressive. See Section 6.2 for a more detailed comparison to our previous work.

Outline. Section 2 of this article illustrates the main concepts of Generic Universe Types by examples. Section 3 defines the programming language, the runtime system, and the operational semantics, without specifying how the ownership topology or encapsulation discipline are enforced. The topological system is given in Section 4, and Section 5 presents the encapsulation system that builds on top of the topological system. In Section 6 we discuss related work, and, finally, Section 7 concludes.

An electronic appendix available in the ACM Digital Library accompanies this article.

Appendix A presents additional properties of GUT that are needed in the proofs, and Appendix B contains the proofs of the properties.

2. MAIN CONCEPTS

In this section we explain the main concepts of Generic Universe Types (GUT) informally by two examples: a generic map and an implementation of the decorator pattern. We chose these examples for clarity and to illustrate the main concepts of our system. A realistic map implementation would, for instance, use hash tables; however, such an implementation would exhibit the same distinction between internal nodes and the external data elements as in our example. This section focuses on the topological system of GUT, which will be discussed in detail in Section 4. We highlight the impact of the encapsulation system in key places; it will be discussed in turn in Section 5.

Class `Map` (Figure 2) implements a generic map from keys to values. Key-value pairs are stored in singly-linked `Node` objects (Figure 3). The `main` method of class `Client` (Figure 4) builds up the map structure shown in Figure 1. In the second example, class `Decorator` can be used to decorate arbitrary objects (Figure 5) as shown in class `Demo` (Figure 6). For simplicity, we omit access modifiers from all examples.

2.1. Ownership Modifiers

A type in GUT is either a type variable or consists of an ownership modifier, a class name, and possibly type arguments. The *ownership modifier* expresses object ownership relative to the current receiver object `this`.[1] Programs may contain the ownership modifiers `peer`, `rep`, and `any`, which have the following meanings.

—`peer` expresses that an object has the same owner as the `this` object; that is, that the current object and the referenced object share the same owner and are therefore in the same context.
—`rep` expresses that an object is owned by `this`; that is, the current object is the owner of the referenced object.
—`any` expresses that an object may have an arbitrary owner. The `any` modifier is a "don't care" modifier and expresses that the ownership of the referenced object is deliberately unspecified for this reference; `any` types, therefore, are supertypes of the

[1]We ignore static fields and methods here, but an extension is possible [Dietl 2009].

```
class Map<K, V> {
  rep Node<K, V> first;

  void put(K key, V value) {
    rep Node<K, V> newfirst = new rep Node<K, V>();
    newfirst.init(key, value, first);
    first = newfirst;
  }

  pure V get(any Object key) {
    rep Node<K, V> n = getNode(key);
    return n != null ? n.value : null;
  }

  pure rep Node<K, V> getNode(any Object key) {
    rep Node<K, V> n = first;
    while (n != null) {
      if (n.key.equals(key)) return n;
      n = n.next;
    }
    return null;
  }
}
```

Fig. 2. An implementation of a generic map. `Map` objects own their `Node` objects, as indicated by the `rep` modifier in all occurrences of class `Node`.

```
class Node<K, V> {
  K key; V value;
  peer Node<K, V> next;

  void init(K k, V v, peer Node<K, V> n) { key = k; value = v; next = n; }
}
```

Fig. 3. Nodes form the internal representation of maps. Class `Node` implements nodes for singly-linked lists of keys and values.

```
class ID { /* ... */ }
class Data { /* ... */ }

class Client {
  peer Map<rep ID, any Data> map;

  void main() {
    map = new peer Map<rep ID, any Data>();
    peer Data value = new peer Data();
    rep ID key = new rep ID();
    map.put(key, value);

    any Node<rep ID, any Data> n = map.getNode(key);
    n.key = new rep ID(); // OK
    n.next = new rep Node<rep ID, any Data>(); // Error
  }
}
```

Fig. 4. Main program for our map example. The execution of method `main` creates the object structure in Figure 1.

rep and peer types with the same class and type arguments, as any types convey less specific ownership information.

The use of ownership modifiers is illustrated by the class Map (Figure 2). A Map object owns its Node objects, since they form the internal representation of the map. This ownership relation is expressed by the rep modifier of Map's field first, which points to the first node of the map.

Internally, the type system uses two additional ownership modifiers, self and lost.

—self is only used as the modifier for the current object this and distinguishes the current object from other objects that have the same owner. Therefore, a type with the self modifier is a subtype of a type with the peer modifier with the same class and type arguments. The use of a separate self modifier highlights the special role that the current object plays in ownership systems and simplifies the overall system by removing special cases for accesses on this.
—lost signifies that the ownership information cannot be expressed statically with one of the other ownership modifiers. It is a "don't know" modifier that indicates that ownership information was "lost" in the type-checking process; in contrast to the any modifier, concrete ownership information might be needed for the reference. lost types are subtypes of corresponding any types, since we want to be able to use any references to refer to arbitrary objects, including objects with lost ownership; on the other hand, lost types are supertypes of the corresponding peer and rep types, since lost types provide less detailed information.

Our encapsulation system enforces the owner-as-modifier discipline by restricting modifications of objects to self, peer, and rep receivers. That is, an expression of a lost or 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 nonpure methods. To check this property, the encapsulation system requires side-effect-free methods to be annotated with the keyword pure. This distinction between pure and nonpure methods is not relevant for the topological system.

2.2. Viewpoint Adaptation

Since ownership modifiers express ownership relative to this, they have to be adapted when this "viewpoint" changes. Consider Node's method init (Figure 3). The third parameter has type peer Node<K,V> and is used to initialize the next field. The peer modifier expresses that the parameter object must have the same owner as the receiver of the method. On the other hand, Map's method put calls init on a rep Node receiver, that is, an object that is owned by this. Therefore, the third parameter of the call to init also has to be owned by this, which means that from this particular call's viewpoint, the third 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 type of the receiver of a call (here, rep Node<K,V>) with the type of the formal parameter (here, peer Node<K,V>). This combination yields the argument type from the caller's point of view (here, rep Node<K,V>).

Viewpoint adaptation results in lost ownership information if the ownership is not expressible from the new viewpoint. For instance, imagine there was a field access map.first in Figure 4; the viewpoint adaptation of the field type, rep Node<K,V>, yields a lost type because there is no ownership modifier to more precisely express a reference into the representation of object map. As a consequence, soundness of the topological system requires that methods cannot directly modify a rep field of an object other than this.

If only the topological system is enforced, a reference containing lost or arbitrary ownership information can still be used as the receiver of field updates and modifying method calls. Consider the main program in Figure 4. Local variable `n` stores a reference into the representation of another object, since method `getNode` returns a reference to the internal nodes of the peer map. The update of `n.key` is valid, as it preserves the topology of the heap. We have full knowledge of the type of the field after viewpoint adaptation, and no ownership information is lost. On the other hand, the update of `n.next` has to be forbidden. After the viewpoint adaptation, the type of the left-hand side is `lost Node<rep ID, any Data>`; this type contains a `lost` ownership modifier and, therefore, the heap topology cannot be ensured statically. If the owner-as-modifier discipline is also enforced, the receiver type of field updates and modifying method calls needs to have a `self`, `peer`, or `rep` modifier. In the example from Figure 4, the update of `n.key` is illegal in the owner-as-modifier discipline because it would modify an object in a statically-unknown context; thus, the modification might bypass the owner of the modified object.

Viewpoint adaptation and the owner-as-modifier discipline provide encapsulation of internal representation objects. Again, let us consider method `getNode` from class `Map`. By viewpoint adaptation of the return type, `rep Node<K,V>`, clients of the map can only obtain a `lost` reference to the nodes. The owner-as-modifier discipline requires a `self`, `peer`, or `rep` receiver type for modifications, and thus, guarantees that clients cannot directly modify the node structure. This allows the map to maintain invariants over the nodes, for instance, that the node structure is acyclic.

2.3. Type Parameters

Ownership modifiers are also used in actual type arguments—for instance, `Client`'s method `main` instantiates `Map` with the type arguments `rep ID` and `any Data`. Thus, field `map` has type `peer Map<rep ID, any Data>`, which has three ownership modifiers. The *main modifier* `peer` expresses that the `Map` object has the same owner as `this`, whereas the *argument modifiers* `rep` and `any` express ownership of the keys and values relative to the `this` object, in this case that the keys are `ID` objects owned by `this` and that the values are `Data` objects in an arbitrary context. It is important to understand that the argument modifiers again express ownership relative to the current `this` object (here, the `Client` object), and not relative to the instance of the generic class that contains the argument modifier (here, the `Map` object `map`).

Type variables are not subject to the viewpoint adaptation that is performed for non-variable types. When type variables are used, for instance, in field declarations, the ownership information they carry stays implicit, and hence does not have to be adapted. The substitution of type variables by their actual type arguments happens in the scope in which the type variables were instantiated. Therefore, the viewpoint is the same as for the instantiation, and no viewpoint adaptation is required. For instance, imagine there was a field read `n.key` in method `main` (Figure 4). The declared type of the field is the type variable `K`. Reading the field through the `n` reference substitutes the type variable by the actual type argument `rep ID`, and does not perform a viewpoint adaptation.

Thus, even though the `Map` class does not know the owner of the keys and values (due to the implicit `any` upper bound for `K` and `V`, see below), clients of the map can recover the exact ownership information from the type arguments. This illustrates that Generic Universe Types provide strong static guarantees similar to those of owner-parametric systems [Clarke et al. 1998], even in the presence of `any` types. The corresponding implementation in nongeneric Universe Types requires a downcast from the `any` type to a `rep` type and the corresponding runtime check [Dietl and Müller 2005]. As in Java, the implementation of a generic class is checked against the declared upper bounds

```
class Decoration<V> {
  void set(V val) {}
}

class Decorator {
  <V, O extends peer Decoration<V>>
  O decorate(V in) {
    O res = new O();
    res.set(in);
    return res;
  }

  <V, O extends peer Decoration<V>>
  peer List<O> decorateList(any List<V> inlist) {
    peer List<O> res = new peer List<O>();
    for( V in : inlist ) {
      res.add( decorate<V, O>(in) );
    }
    return res;
  }
}
```

Fig. 5. A decorator for arbitrary objects. As shown in method `decorate`, type variables with `peer` or `rep` upper bounds can be instantiated.

```
class MyDecoration extends Decoration<peer Data> {
  peer Data f;

  void set(peer Data d) { f = d; }
}

class Demo {
   void main() {
      rep Data d = new rep Data();
      rep MyDecoration dd =
         new rep Decorator().decorate<rep Data, rep MyDecoration>(d);
   }
}
```

Fig. 6. Class `MyDecoration` decorates `peer Data` objects and is then used by class `Demo`.

of type parameters. In GUT, the upper bounds as well as the actual type arguments specify ownership relative to the current object. In the example from Figure 4, the keys are references into the representation of the current `Client` object. The `Map` and `Node` classes, however, are type-checked against the upper bound of the keys, which defaults to `any`. Therefore, the map does not get privileged access to the representation of the client object.

Type variables have upper bounds, which default to `any Object`. In a class C, the ownership modifiers of an upper bound express ownership relative to the C instance `this`. However, when C's type variables are instantiated, the modifiers of the actual type arguments are relative to the receiver of the method that contains the instantiation. Therefore, checking the conformance of a type argument to its upper bound requires a viewpoint adaptation. Equally, method type variables have upper bounds that are relative to the current instance of the declaring class.

As an example, consider the implementation of the decorator pattern presented in Figures 5 and 6. Class `Decorator` (Figure 5) can be used to decorate arbitrary

```
class ClientUser {
  void useMap( peer Client client ) {
    client.map.put(new rep ID(), new peer MyData()); // Error
  }
}
```

Fig. 7. Viewpoint adaptation of the map results in lost ownership.

objects with a type provided as method type variable `O`. The upper bound of `O` has the type `peer Decoration<V>`. Method `decorateList` decorates a list of elements. Class `Demo` (Figure 6) presents a use of the decorator: the call of method `decorate` is type-correct, because the upper bound for type variable `O` after viewpoint adaptation is `rep Decoration<rep Data>`, which is a supertype of the actual type argument, `rep MyDecoration`. This subtype relation can be derived from the superclass declaration of class `MyDecoration` and adapting to a `rep` viewpoint.

In all our examples, for clarity, we provide explicit method type arguments. Java implements method type argument inference, which is extensible to the inference of GUT annotations. Similarly, the new Java 7 diamond syntax should be extensible to also infer GUT annotations.

2.4. The `lost` and `any` Modifiers and Limited Covariance

There is a fundamental difference between a generic type that uses an arbitrary owner as type argument and a generic type that uses an unknown owner as type argument.

For example, the map from Figure 4 has type `peer Map<rep ID, any Data>` and specifies that: the map object has the same owner as the current object; the keys are `ID` objects owned by the current object, and the values are Data objects that have arbitrary owners. The type specifies that an arbitrary owner is allowed for the values and that it is legal to use peers, reps, or objects with any other kind of ownership.

The adaptation of a type argument might yield a `lost` type, signifying that ownership information could not be expressed from the new viewpoint, as illustrated in Figure 7. The type of the field access `client.map` is `peer Map<lost ID, any Data>`. We can statically still express that the map object itself is in the same context as the current object, and we still know that the values are in an arbitrary context. But from this new viewpoint, we cannot express that the keys have to be owned by the client instance `client`; there is no specific ownership modifier for this relation, and therefore the `lost` modifier is used. It would not be type-safe to allow the call of method `put` on the receiver of this type. The signature for method `put` after viewpoint adaptation contains `lost`, and the topological system cannot express the precise ownership required for the first argument. On the other hand, the signature of method `get` does not contain `lost` and can still be called. Note that methods `get` and `getNode` use `any Object` as parameter types and not the type variable `K`. Using the upper bound of a type variable instead of the type variable allows us to call a method, even if the actual type argument loses ownership information. This is particularly useful for pure methods that do not modify the heap. Note that the same design is used in the Java 5 interface `java.util.List`, for example, by methods `contains`, `indexOf`, and `remove`. These methods use `Object` as parameter type instead of the corresponding type variable, which allows them to be called on receivers that contain wildcards, and thus increases the applicability of these methods.

Subtyping with covariant type arguments is in general not statically type-safe. For instance, if `List<String>` were a subtype of `List<Object>`, then clients that view a string list through type `List<Object>` could store `Object` instances in the string list, which breaks type safety. The same problem occurs for the ownership information encoded in types. If `peer Map<rep ID, any Data>` were a subtype of `peer Map<any ID, any Data>`, then clients that view the map through the latter type could use method

```
class Cast {
  void m( any Object obj ) {
    peer Map<rep ID, any Data> map = (peer Map<rep ID, any Data>) obj;
    map.put(new rep ID(), new peer Data());
  }
}
```

Fig. 8. Demonstration of a cast.

put (Figure 2) to add a new Node object where the key has an arbitrary owner, even though the map instance requires a specific owner. The covariance problem can be prevented by disallowing covariant type arguments (as in Java and C♯), using use-site or declaration-site variance annotations (i.e., wildcards as found in Java or variance annotations as found in Scala [Odersky 2008]), by runtime checks (as done for arrays in Java), or by elaborate syntactic support [Emir et al. 2006].

Our topological system supports a limited form of covariance without requiring additional checks. Covariance is permitted if the corresponding modifier of the supertype is lost. For example, peer Map<rep ID, any Data> is a subtype of peer Map<lost ID, any Data>. This is safe because the topological system already prevents updates of variables that contain lost. In particular, it is not possible to call method put because the signature after substitution contains lost, which prevents the unsound addition of an arbitrary object shown above.

2.5. Runtime Representation

We store the ownership information and the runtime type arguments, including their associated ownership information, explicitly in the heap because this information is needed in the runtime checks for casts and for instantiating type variables. In this respect, our runtime model is similar to that of the .NET CLR [Kennedy and Syme 2001], where runtime information about generics is present and "new constraints" can be used to allow the instantiation of type variables.

For example, method m in Figure 8 takes an object with an arbitrary owner as argument and uses a cast to retrieve ownership information for the main modifier of the map reference and also for the type arguments. To check this cast at runtime, the object needs to store a reference to its owner and the ownership and type information for the type arguments.

Storing the ownership information at runtime also enables us to create instances of type variables if the main modifier of the corresponding upper bound is peer or rep. In our language, every class can be instantiated using a uniform new expression, which initializes all fields to null. Type variables with any as upper bound cannot be instantiated, as we could not ensure that the actual type argument provides concrete ownership information, which is necessary for the correct placement in the ownership topology. In the implementation of the decorator pattern, presented in Figures 5 and 6, we want to instantiate the type variable O; its upper bound is peer (rep would also be possible, but would limit the possible callers of the method), and we therefore know that the new object will have the same owner as the current object.

There are alternatives to storing the genericity information at runtime: erasure of genericity as found in Java 5 and expansion of generic class declarations as found in C++ templates. It is possible to erase a GUT program into a Universe Types program without generics [Cunningham et al. 2008], using casts. The interpretation of casts and type arguments is the same: both are from the viewpoint of the current object. Therefore, the casts inserted into the erased program use the same types that are used as type arguments. In contrast, expanding the type arguments into the declaring class does not work in general, as the viewpoint for the type argument and the expanded

$$
\begin{array}{rcl}
P & ::= & \overline{Cls},\ C,\ e \\
Cls & ::= & \texttt{class}\ Cid\texttt{<}\overline{TP}\texttt{>}\ \texttt{extends}\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\ \{\ \overline{fd}\ \overline{md}\ \} \\
C & ::= & Cid \mid \texttt{Object} \\
TP & ::= & X\ \texttt{extends}\ {}^{s}N \\
fd & ::= & {}^{s}T\ f\texttt{;} \\
md & ::= & p\ \texttt{<}\overline{TP}\texttt{>}\ {}^{s}T\ m\texttt{(}\overline{mpd}\texttt{)}\ \{\ e\ \} \\
p & ::= & \texttt{pure} \mid \texttt{impure} \\
mpd & ::= & {}^{s}T\ pid \\
e & ::= & \texttt{null} \mid x \mid \texttt{new}\ {}^{s}T\texttt{()} \mid e\texttt{.}f \mid e_0\texttt{.}f\ \texttt{=}\ e_1 \mid \\
 & & \quad e_0\texttt{.}m\texttt{<}\overline{{}^{s}T}\texttt{>(}\overline{e}\texttt{)} \mid \texttt{(}{}^{s}T\texttt{)}\ e \\
{}^{s}T & ::= & {}^{s}N \mid X \\
{}^{s}N & ::= & u\ C\texttt{<}\overline{{}^{s}T}\texttt{>} \\
u & ::= & \texttt{self} \mid \texttt{peer} \mid \texttt{rep} \mid \texttt{lost} \mid \texttt{any} \\
x & ::= & pid \mid \texttt{this}
\end{array}
$$

pid	parameter identifier
f	field identifier
m	method identifier
X	type variable identifier
Cid	class identifier

Fig. 9. Syntax of our programming language.

type differ and a viewpoint adaptation is not always possible; see Section 6.2 for an example.

This concludes our informal introduction to Generic Universe Types. In the next section we present the programming language and semantics on which we build.

3. PROGRAMMING LANGUAGE SYNTAX AND SEMANTICS

In this section we define the syntax and operational semantics of the programming language. It presents a standard model for a class-based object-oriented language that is independent of the topological and the encapsulation system, which is presented in Sections 4 and 5. Alternative systems to enforce a topology or encapsulation discipline, for example using dynamic ownership, could be built for this programming language.

3.1. Programming Language

We formalize Generic Universe Types for a sequential subset of Java 5 and C♯ 2.0 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 and C♯, such as interfaces, enum types, 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 (see, e.g., Boyapati [2004], Dietl and Müller [2004], Leavens et al. [2008], and Müller [2002]). The language we use is similar to Featherweight Generic Java [Igarashi et al. 2001]. We added field updates because the treatment of side effects is essential for ownership type systems, and especially the owner-as-modifier discipline.

Figure 9 summarizes the syntax of our language and our naming conventions for variables. 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).

The superscript ^{s} distinguishes the sorts for static-checking from the corresponding sorts used to describe the runtime behavior.

A sequence of A's is denoted as $\overline{A}$. In such a sequence, we denote the ith element by A_i. We denote sequences of a certain length k by $\overline{A_k}$. A sequence $\overline{A}$ can be empty; the empty sequence is denoted by $\emptyset$. We use sequences of "maplets" $S = \overline{a \mapsto b}$ as maps and use a function-like notation to access an element $S(a_i) = b_i$. We use dom to denote the domain of a sequence of maplets, for example, $\mathrm{dom}(S) = \overline{a}$.

A program P consists of a sequence of classes $\overline{Cls}$, the identifier of a main class C, and a main expression e. A program is executed by creating an instance o of C and then evaluating e with o as `this` object. We assume that we always have access to the current program P, and keep P implicit in the notation. Each class *Cls* has a class identifier, type variables with upper bounds, a superclass with type arguments, a sequence of field declarations, and a sequence of method declarations. f is used for field identifiers. As in Java, each class directly or transitively extends the predefined class `Object`.

A method declaration *md* consists of the purity annotation, the method type variables with their upper bounds, the return type, the method identifier m, the formal method parameters *pid* with their types, and an expression as body. The result of evaluating the expression is returned by the method. Method parameters x include the explicit method parameters *pid* and the implicit method parameter `this`.

To be able to enforce the owner-as-modifier discipline, we have to distinguish statically between side-effect-free (*pure*) methods and methods that potentially have side effects. Pure methods are marked by the keyword `pure`. In our syntax, we mark all other methods by `impure`, although we omit this keyword in our examples. Method purity is not relevant for the discussions in the current section and for the topological system in Section 4; it will be used by the encapsulation system in Section 5.

An expression e can be the `null` literal, a method parameter access, object creation, field read, field update, method call, or cast.

A type ^sT is either a nonvariable type or a type variable identifier X. A non-variable type ^sN consists of an ownership modifier, a class identifier, and a sequence of type arguments.

An ownership modifier u can be `self`, `peer`, `rep`, `lost`, or `any`. Note that we restrict the use of `self` and `lost` in the formalization only as much as is needed for the soundness of the system. For example, we allow the use of `lost` in the declared field type, even though such a field can never be assigned a value.

The following sections define subclassing and look-up functions that make accessing different parts of the program simpler.

3.1.1. Subclassing. We use the term *subclassing* (symbol $\sqsubseteq$) to refer to the reflexive and transitive relations on classes declared in a program by the `extends` keyword, irrespective of main modifiers. It is defined on instantiated classes $C\texttt{<}\overline{^sT}\texttt{>}$, which are denoted ^sCT. The subclass relation is the smallest relation satisfying the rules in Definition 3.1. Each class that is instantiated with its type variables is a subclass of the class and type arguments it is declared to extend (SC1). Subclassing is reflexive (SC2) and transitive (SC3). In all three rules, the subclass is the class instantiated with its declared type variables, whereas the superclass is instantiated with type arguments that depend on the relationship between sub- and superclass; this makes substitutions in later rules simpler. The substitution of the type arguments $\overline{^sT}$ for the type variables $\overline{X}$ in ^sT is denoted $^sT[\overline{^sT}/\overline{X}]$. For the substitution to be defined, the two sequences have to have the same length.

Definition 3.1 (*Subclassing*).

$\boxed{{}^{s}CT \sqsubseteq {}^{s}CT'}$ subclassing

$$\frac{\texttt{class } Cid\texttt{<}\overline{X_k \texttt{ extends } \_}\texttt{> extends } C'\texttt{<}\overline{{}^{s}T}\texttt{> } \{\ \_\ \_\ \} \in P}{Cid\texttt{<}\overline{X_k}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>}}\ \text{SC1}$$

$$\frac{\texttt{class } C\texttt{<}\overline{X_k \texttt{ extends } \_}\texttt{>} \ldots \in P}{C\texttt{<}\overline{X_k}\texttt{>} \sqsubseteq C\texttt{<}\overline{X_k}\texttt{>}}\ \text{SC2} \qquad \frac{\begin{array}{c} C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C_1\texttt{<}\overline{{}^{s}T}_1\texttt{>} \\ C_1\texttt{<}\overline{X}_1\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}'\texttt{>} \end{array}}{C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}'[\overline{{}^{s}T}_1/\overline{X}_1]\texttt{>}}\ \text{SC3}$$

Consider the declaration of class `MyDecoration` in Figure 6. Using rule SC1, we can derive `MyDecoration` $\sqsubseteq$ `Decoration<peer Data>`.

3.1.2. Field Type Look-up. Function FType is used to look up the declared type of a field in a class. Note that the function is only defined if the field is declared in the given class; superclasses are not considered. Class `Object` has no fields, and therefore the function is undefined in this case.

Definition 3.2 (*Field Type Look-up*).

$\boxed{\text{FType}(C, f) = {}^{s}T}$ look up field f in class C

$$\frac{\texttt{class } Cid\texttt{<}\_\texttt{> extends } \_\texttt{<}\_\texttt{> } \{\ \_\ {}^{s}T\ f\texttt{;}\ \_\ \_\ \} \in P}{\text{FType}(Cid, f) = {}^{s}T}\ \text{SFTC\_DEF}$$

In the preceding definition, the part $\_\ {}^{s}T\ f\texttt{;}\ \_\ \_$ is read as "some sequence of field declarations, then a field declaration with static type ${}^{s}T$ and identifier f, followed by another sequence of field declarations, and finally an arbitrary sequence of method declarations."

3.1.3. Method Signature Look-up. The look-up of a method signature in a class works like field look-up and yields the method signature of a method with the given name in class C.

Definition 3.3 (*Method Signature Look-up*).

$\boxed{\text{MSig}(C, m) = ms_o}$ look up signature of method m in class C

$$\frac{\begin{array}{l}\texttt{class } Cid\texttt{<}\_\texttt{> extends } \_\texttt{<}\_\texttt{> } \{\ \_\ \_\ ms\ \{\ e\ \}\ \_\ \} \in P \\ \text{MName}(ms)\texttt{=}m\end{array}}{\text{MSig}(Cid, m) = ms}\ \text{SMSC\_DEF}$$

MName yields the method name of a method signature. Note that we do not support method overloading, so the method name is sufficient to uniquely identify a method. In the definition of MSig we use ms_o as a result to signify an *optional* method signature. In the definition of the method overriding rules (in Definition 4.17), we need to explicitly distinguish between an undefined method signature (using the notation *None*) and a defined method signature ms.

As in FGJ [Igarashi et al. 2001], in a method signature

$\_\ \texttt{<}\overline{X_l \texttt{ extends } {}^{s}N_l}\texttt{>}\ {}^{s}T\ \ m\texttt{(}\overline{{}^{s}T_q\ pid}\texttt{)}$

the method type variables $\overline{X_l}$ are bound in the types $\overline{{}^{s}N_l}$, ${}^{s}T$, and $\overline{{}^{s}T_q}$ and α-convertible signatures are equivalent.

3.1.4. Class Domain Look-up. The domain of a class is the sequence of type variables that it declares. The predefined class `Object` does not declare any type variables.

$$
\begin{array}{rcl}
h & ::= & \overline{(\iota \mapsto o)} \\
\iota & & \text{Address} \\
{}^{o}\iota & ::= & \iota \mid \texttt{any}_a \mid \texttt{root}_a \\
v & ::= & \iota \mid \texttt{null}_a \\
o & ::= & \left({}^{r}T, \overline{fv}\right) \\
{}^{r}T & ::= & {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>} \\
\overline{fv} & ::= & \overline{f \mapsto v} \\
{}^{r}\Gamma & ::= & \left\{\overline{X \mapsto {}^{r}T};\ \overline{x \mapsto v}\right\}
\end{array}
$$

Fig. 10. Definitions for the runtime model.

Definition 3.4 (*Class Domain Look-up*).

$\boxed{\text{ClassDom}(C) = \overline{X}}$ look up type variables of class C

$$\frac{\texttt{class}\ Cid\texttt{<}\overline{X_k\ \texttt{extends}\ \_}\texttt{>}\ \texttt{extends}\ \_\texttt{<}\_\texttt{>}\ \{\ \_\ \_\ \} \in P}{\text{ClassDom}(Cid) = \overline{X_k}}\ \text{SCD\_NVAR}$$

$$\frac{}{\text{ClassDom}(\texttt{Object}) = \emptyset}\ \text{SCD\_OBJECT}$$

3.1.5. Upper Bounds Look-up. The bounds of a class is the sequence of upper bounds of the type variables that the class declares.

Definition 3.5 (*Upper Bounds Look-up*).

$\boxed{\text{ClassBnds}(C) = \overline{{}^{s}N}}$ look up bounds of class C

$$\frac{\texttt{class}\ Cid\texttt{<}\overline{X_k\ \texttt{extends}\ {}^{s}N_k}\texttt{>}\ \texttt{extends}\ \_\texttt{<}\_\texttt{>}\ \{\ \_\ \_\ \} \in P}{\text{ClassBnds}(Cid) = \overline{{}^{s}N_k}}\ \text{SCBC\_NVAR}$$

$$\frac{}{\text{ClassBnds}(\texttt{Object}) = \emptyset}\ \text{SCBC\_OBJECT}$$

3.2. Runtime Model

Figure 10 defines our model of the runtime system. The prefix ^{r} distinguishes sorts of the runtime model from their static counterparts.

A heap h maps addresses to objects. An address ι is an element of a countable, infinite set of addresses. The domain of a heap h, written $\text{dom}(h)$, is the set of all addresses that are mapped to an object in the heap h. A value v can be an address ι or the special null-address $\texttt{null}_a$. An object o consists of its runtime type and a mapping from field identifiers to the values stored in the fields. The notation $h(\iota)\downarrow_1$ and $h(\iota)\downarrow_2$ is used to access the first and second component of the object at address ι in heap h.

The runtime type ${}^{r}T$ of an object o consists of o's owner address, o's class, and of the runtime types for the type arguments of this class. An owner address ${}^{o}\iota$ can be the address ι of the owning object, the root owner $\texttt{root}_a$, or the $\texttt{any}_a$ owner. The owner address of objects in the root context is $\texttt{root}_a$. The special owner address $\texttt{any}_a$ is used when the corresponding static type has the `any` modifier. Consider for instance an execution of method `main` (Figure 4), where the address of `this` is 1 and the owner of 1 is $\texttt{root}_a$. The runtime type of the object stored in `map` is $\texttt{root}_a$ `Map<1 ID,` $\texttt{any}_a$ `Data>`.

The first component of a runtime environment ${}^{r}\Gamma$ maps method type variables to their runtime types. The second component is the current stack frame, which maps method parameters to the values they store. Since the domains of these mappings are disjoint, we overload the notation and use ${}^{r}\Gamma(X)$ to access the runtime type for type variable X and ${}^{r}\Gamma(x)$ to access the value for method parameter x.

The following sections again define various functions to simplify the notation.

3.2.1. Heap Operations. Our heap model is very simple: we can create an empty heap $\emptyset$ and add to or update in an existing heap h a mapping from address ι to an object o, written as $h + (\iota \mapsto o)$. If address ι is already mapped to an object, this mapping is overwritten. We use the shorthand notation $h(\iota\,.\,f)$ for reading field f of the object at address ι, that is, $h(\iota)\downarrow_2(f)$.

For convenience, we provide two additional operations, which can be modeled on top of the basic operations: (1) creation of a new object and (2) updating a field value in a heap.

(1) For the addition of an object o as a new object to heap h, resulting in a new heap h' and address ι, we use the notation $h + o \;=\; (h', \iota)$. We ensure that ι is a fresh address and that the only modification to the heap is the addition of the new object.

Definition 3.6 (*Object Addition*).

$\boxed{h + o \;=\; (h', \iota)}$ add object o to heap h resulting in heap h' and fresh address ι

$$\frac{\iota \notin \mathrm{dom}(h) \qquad h'{=}h + (\iota \mapsto o)}{h + o \;=\; (h', \iota)}\ \textsc{hnew\_def}$$

(2) We write $h[\iota\,.\,f \,{=}\, v] \,{=}\, h'$ for the update of field f of the object at address ι in heap h to the new value v, resulting in new heap h'. We ensure that the new field value is valid, that is, v is either $\mathtt{null}_a$ or the address of an object in the heap. We also ensure that there is already an object at address ι and that the field identifier f is already in the set of fields of the object, since we do not want to add fields that would not be defined in the corresponding class. In the set of field values $\overline{fv}$, we overwrite the existing mapping for f to arrive at $\overline{fv}'$ and update the heap with the object that consists of the old runtime type and the new field values $\overline{fv}'$. Note that we only change the field value of the single object at address ι, and in particular that the runtime types in the heap remain unchanged.

Definition 3.7 (*Field Update*).

$\boxed{h[\iota\,.\,f \,{=}\, v] \,{=}\, h'}$ field update in heap

$$\frac{\begin{array}{l} v{=}\mathtt{null}_a \;\vee\; (v{=}\iota' \;\wedge\; \iota' \in \mathrm{dom}(h)) \\ h(\iota){=}({}^{r}T, \overline{fv}) \qquad f \in \mathrm{dom}(\overline{fv}) \qquad \overline{fv}'{=}\overline{fv}[f \mapsto v] \\ h'{=}h + (\iota \mapsto ({}^{r}T, \overline{fv}')) \end{array}}{h[\iota\,.\,f \,{=}\, v] \,{=}\, h'}\ \textsc{hup\_def}$$

3.2.2. Runtime Method Signature and Body Look-up. The following function is used to look up the method signature for an object at a particular address. The type-to-value assignment judgment (Definition 3.13) determines a class C, a superclass of the runtime class of ι, for which the static method signature function MSig yields a method signature. All overriding methods have the same method name and the overriding rule (Definition 4.17) ensures that the different signatures are consistent. $\mathrm{MSig}(P, h, \iota, m)$ yields an arbitrary possible signature, not necessarily the one declared in the smallest supertype of the runtime class of ι—as the different signatures are consistent, this suffices. Note that we do not support method overloading, so the method name is sufficient to uniquely identify a method.

Definition 3.8 (*Runtime Method Signature Look-up*).

$\boxed{\text{MSig}(h, \iota, m) = ms_o}$ look up method signature of method m at ι

$$\frac{h \vdash \iota : \_\ C\texttt{<\_>} \qquad \text{MSig}(C, m) = ms}{\text{MSig}(h, \iota, m) = ms}\ \text{RMS\_DEF}$$

At runtime, we need the ability to look up the implementation of a method that is declared in the smallest supertype of the runtime type of the receiver. We first define a look-up function that determines the method body e from the smallest superclass of class C that implements method m.

Definition 3.9 (*Static Method Body Look-up*).

$\boxed{\text{MBody}(C, m) = e}$ look up most-concrete body of m in class C or a superclass

$$\frac{\texttt{class}\ Cid\texttt{<\_>}\ \texttt{extends}\ \_\texttt{<\_>}\ \{\ \_\ \_\ ms\ \{\ e\ \}\ \_\ \} \in P \qquad \text{MName}(ms) = m}{\text{MBody}(Cid, m) = e}\ \text{SMBC\_FOUND}$$

$$\frac{\texttt{class}\ Cid\texttt{<\_>}\ \texttt{extends}\ C_1\texttt{<\_>}\ \{\ \_\ \overline{ms_n\ \{\ e_n\ \}}\ \} \in P \qquad \overline{\text{MName}(ms_n) \neq m} \qquad \text{MBody}(C_1, m) = e}{\text{MBody}(Cid, m) = e}\ \text{SMBC\_INH}$$

The following function uses the most concrete runtime type of the object at address ι to determine the corresponding method body.

Definition 3.10 (*Runtime Method Body Look-up*).

$\boxed{\text{MBody}(h, \iota, m) = e}$ look up most-concrete body of method m at ι

$$\frac{h(\iota)\downarrow_1 =\_\ C\texttt{<\_>} \qquad \text{MBody}(C, m) = e}{\text{MBody}(h, \iota, m) = e}\ \text{RMB\_DEF}$$

3.3. Static Types, Runtime Types, and Values

In this section we discuss two functions that convert static types to corresponding runtime types, the subtyping of runtime types, and what runtime and static types can be assigned to a value. The discussion gives precise semantics to the static types, in particular the meaning of the ownership modifiers.

3.3.1. Simple Dynamization of Static Types. We need to put static and runtime types into a relation, for example, when we evaluate an object creation, we need to convert the static type from the expression into the corresponding runtime type that is stored with the newly created object.

We define two dynamization functions: first, a simple dynamization function, sdyn, which puts strong requirements on the static types that it can convert to runtime types; second, a more general dynamization function, dyn, which is less restrictive. Defining two distinct dynamization functions allows us to avoid a cyclic dependency between runtime subtyping and the dynamization of static types. To determine runtime subtypes, we need to dynamize the instantiation of a superclass into a runtime type, and, in general, to determine the dynamization of a static type, we need to find a runtime supertype of the type of the current object. We use sdyn only to dynamize static types where we do not need runtime subtyping to determine the runtime type and use dyn in the general case where we can use runtime subtyping.

The simple dynamization function sdyn (Definition 3.11) relates a sequence of static types to a corresponding sequence of runtime types. The dynamization is relative to

a heap, a viewpoint object ι, a runtime type ${}^{r}T$, and substitutions for `lost` ownership modifiers. The viewpoint object is needed to dynamize types with a `rep` modifier; the heap is used to look up the owner of the viewpoint object in order to dynamize types with a `peer` modifier. This simple version of the dynamization uses only the type information available from its arguments and does not determine additional runtime types from the heap. Its sole purpose is to convert static types that appear in the upper bounds declaration and the superclass instantiation of a class.

To be defined, the sdyn function requires consistency between its arguments. A `peer` modifier in a static type expresses that the referenced object has the same owner as the current object, which is the owner ${}^{o}\iota$ of the runtime type ${}^{r}T$. If the owner ${}^{o}\iota$ of the runtime type is not the $\texttt{any}_a$ address, then we use that owner for the substitution of `peer`. However, if ${}^{o}\iota$ is the $\texttt{any}_a$ address, we cannot simply substitute `peer` by $\texttt{any}_a$. In this case we use an address ${}^{o}\iota_1$ that is either an address in the heap or the root owner to substitute `peer` modifiers. If a `rep` modifier appears in the static type, we need to ensure that the substitution for `rep` and for `peer` are consistent, that is, that the owner of the object that is used to substitute `rep` is the owner address that is used to substitute `peer`. owner(h, ι) yields the owner address of the object at address ι in heap h. Note that the viewpoint address ι is only used if the static types contain a `rep` modifier. If there is no `rep` modifier, the viewpoint address is ignored, and ι can be an arbitrary address that is not necessarily in the domain of the heap h.

Definition 3.11 (*Simple Dynamization of Static Types*).

$\boxed{\text{sdyn}(\overline{{}^{s}T}, h, \iota, {}^{r}T, \overline{{}^{o}\iota}) = \overline{{}^{r}T}}$ simple dynamization of types $\overline{{}^{s}T}$

$$\frac{\begin{array}{c} {}^{o}\iota' \in \text{dom}(h) \cup \{\texttt{root}_a\} \qquad {}^{o}\iota \neq \texttt{any}_a \Longrightarrow {}^{o}\iota' = {}^{o}\iota \\ \text{ClassDom}(C) = \overline{X} \qquad \texttt{rep} \in \overline{{}^{s}T} \Longrightarrow \text{owner}(h, \iota) = {}^{o}\iota' \\ \overline{{}^{s}T}[{}^{o}\iota'/\texttt{peer}, \iota/\texttt{rep}, \texttt{any}_a/\texttt{any}, \overline{{}^{r}T}/\overline{X}, \overline{{}^{o}\iota_i/\texttt{lost}}] = \overline{{}^{r}T'} \end{array}}{\text{sdyn}(\overline{{}^{s}T}, h, \iota, {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}, \overline{{}^{o}\iota_i}) = \overline{{}^{r}T'}}\ \text{SDYN}$$

We use the notation ${}^{s}T[{}^{o}\iota/u, \overline{{}^{r}T}/\overline{X}]$ for the substitution of owner address ${}^{o}\iota$ for occurrences of ownership modifier u and of runtime types $\overline{{}^{r}T}$ for type variables $\overline{X}$ in the static type ${}^{s}T$. The substitution yields a runtime type if all ownership modifiers and type variables in ${}^{s}T$ are replaced by their runtime counterparts, that is, owner addresses and runtime types. In the above definition, this might not be the case if the `self` modifier or type variables that are not declared in class C appear in $\overline{{}^{s}T}$. Also, the `lost` modifiers are substituted by the corresponding owners from the sequence $\overline{{}^{o}\iota}$. The number of `lost` modifiers in $\overline{{}^{s}T}$ has to correspond to the length of $\overline{{}^{o}\iota}$ and the ith occurrence of `lost` is substituted by the ith owner in $\overline{{}^{o}\iota}$. If the lengths do not match, the substitution is undefined. Our type system only applies sdyn within its domain.

As a first example, Figure 11 gives a simple heap, using the decorator from Figure 6. Then, sdyn(`peer Data`, h, 2, $\texttt{root}_a$ `MyDecoration`, $\emptyset$) results in $\texttt{root}_a$ `Data`.

A more complicated example is given in Figure 12. We can deduce that

sdyn(`rep Data`, h, 2, 1 `D<1 Object>`, $\emptyset$) = 2 `Data`

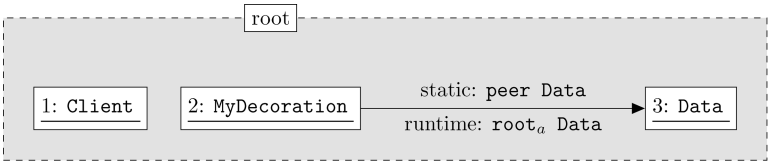
and

sdyn(`Z`, h, 2, 1 `D<1 Object>`, $\emptyset$) = 1 `Object`.

However,

sdyn(`X`, h, 2, 1 `D<1 Object>`, $\emptyset$)

is not defined. Type variable `X` is defined in a supertype of the given runtime type. sdyn does not apply subtyping to find a correct substitution for type variables that are defined in supertypes. The more complicated dyn is used for such purposes.



$h = (1 \mapsto (\texttt{root}_a\ \texttt{Client}, \_), 2 \mapsto (\texttt{root}_a\ \texttt{MyDecoration}, \_), 3 \mapsto (\texttt{root}_a\ \texttt{Data}, \_)$

Fig. 11. Example heap for the program from Figure 6.

```
class C<X extends rep Data, Y extends any Object> {}
class D<Z extends peer Object> extends C<rep Data, Z> {}
```

$$h = (1 \mapsto (\texttt{root}_a\ \texttt{Client}, \_), 2 \mapsto (1\ \texttt{D<1 Object>}, \_))$$

Fig. 12. Example program and heap.

3.3.2. Subtyping of Runtime Types. Subtyping of runtime types follows subclassing (see Definition 3.1). The owner of the supertype $^o\iota'$ is either the same as the owner of the subtype or $\texttt{any}_a$. The static superclass instantiation $\overline{^s T}$ is converted into the runtime types $\overline{^r T}'$ using the runtime subtype $^o\iota\ C\texttt{<}\overline{^r T}\texttt{>}$ as argument to sdyn. This ensures that type variables in $\overline{^s T}$ are substituted by the runtime type arguments $\overline{^r T}$ that are used in the subtype. There is no variance in the type arguments of a runtime type.

Definition 3.12 (*Runtime Subtyping*).

$\boxed{h \vdash {}^r T <: {}^r T'}$ type $^r T$ is a subtype of $^r T'$

$$\frac{C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{^s T}\texttt{>} \qquad {}^o\iota' \in \{{}^o\iota, \texttt{any}_a\} \qquad \text{sdyn}(\overline{^s T}, h, \_, {}^o\iota\ C\texttt{<}\overline{^r T}\texttt{>}, \overline{o_i}) = \overline{^r T}'}{h \vdash {}^o\iota\ C\texttt{<}\overline{^r T}\texttt{>} <:\ {}^o\iota'\ C'\texttt{<}\overline{^r T}'\texttt{>}}\ \text{RT\_DEF}$$

Note how an arbitrary address is used as a viewpoint object for sdyn. A `rep` modifier in $\overline{^s T}$ can be substituted with an arbitrary address that creates a consistent result (note that sdyn checks for consistency between the viewpoint address and the owner of the runtime type $^o\iota$). Also note that there is no equivalent to the `lost` modifier at runtime and that the substitution for `lost` modifiers $\overline{o_i}$ can be chosen arbitrarily. This reflects the interpretation of `lost` as an existential modifier that is substituted by whichever owner address fits the current context. The definition of a well-formed class in the topological system (see Definition 4.14) enforces that subclassing never introduces `lost` in the instantiation of a superclass.

For the decorator example (Figures 6, and 11) we deduced in Section 3.1.1 that `MyDecoration` $\sqsubseteq$ `Decoration<peer Data>`. In the previous section, we deduced the result of applying sdyn to `peer Data`. Combining these results we can deduce

$h \vdash \texttt{root}_a\ \texttt{MyDecoration} <: \texttt{root}_a\ \texttt{Decoration<root}_a\ \texttt{Data>}$

and

$h \vdash \texttt{root}_a\ \texttt{MyDecoration} <: \texttt{any}_a\ \texttt{Decoration<root}_a\ \texttt{Data>}$.

For the example in Figure 12, we can use subclassing rule SC1 to deduce `D<Z>` $\sqsubseteq$ `C<rep Data, Z>`. Using the results of the application of sdyn from the previous section, we can deduce that

$h \vdash 1\ \texttt{D<1 Object>} <: 1\ \texttt{C<2 Data, 1 Object>}$. Instead of address 2, any other address in the heap whose owner is address 1 could be chosen by the subtyping judgment.

We also have

$h \vdash 1$ D<1 Object> <: $\texttt{any}_a$ D<1 Object>.

What supertypes of $\texttt{any}_a$ D<1 Object> exist? We can deduce that

$h \vdash \texttt{any}_a$ D<1 Object> <: $\texttt{any}_a$ C<2 Data, 1 Object>.

But note that, instead of address 2, any other address in the heap h could be chosen. sdyn can choose an arbitrary viewpoint address, and since there are no peer modifiers in the superclass instantiation, the only restriction is that the owner look-up in the heap is defined.

3.3.3. Assigning a Runtime Type to a Value. To assign a runtime type ${}^r T$ to an address ι, we determine the most concrete runtime type ${}^r T_1$ from the heap and check whether ${}^r T_1$ is a runtime subtype of ${}^r T$. An arbitrary runtime type can be assigned to the $\texttt{null}_a$ value.

Definition 3.13 (*Assigning a Runtime Type to a Value*).

$\boxed{h \vdash v : {}^r T}$ runtime type ${}^r T$ assignable to value v

$$\frac{h(\iota)\downarrow_1 = {}^r T_1 \qquad h \vdash {}^r T_1 <: {}^r T}{h \vdash \iota : {}^r T}\ \text{RTT\_ADDR}$$

$$\frac{}{h \vdash \texttt{null}_a : {}^r T}\ \text{RTT\_NULL}$$

In the decorator example (Figure 11), we can deduce that

$h \vdash 2 : \texttt{root}_a$ MyDecoration

and

$h \vdash 2 : \texttt{root}_a$ Decoration<$\texttt{root}_a$ Data>.

For the example from Figure 12, we can deduce that

$h \vdash 2 : 1$ D<1 Object>

and

$h \vdash 2 : 1$ C<2 Data, 1 Object>.

3.3.4. Dynamization of a Static Type. This version of the dynamization function is applicable in a more general setting than the simple dynamization function sdyn introduced before. A static type is converted into a runtime type by using a heap, a runtime environment, and substitutions for lost modifiers. The runtime environment replaces the viewpoint argument of sdyn; the current receiver is extracted from the runtime environment and used as viewpoint object. dyn builds on runtime subtyping, and therefore on sdyn, to determine all necessary type information. This allows dyn to find the runtime equivalents to types which use type variables that are declared in a superclass of the type of the current object—types which cannot be dynamized using sdyn.

Function dyn is given in Definition 3.14. We determine the runtime supertype with class C that can be assigned to the current object ι, for which the domain of class C together with the method type variables from the runtime environment define all type variables that appear in the static type. If such a type does not exist, then the dynamization is not defined. The topological type rules (presented in Section 4) ensure that dyn is never used on such a static type.

The owner ${}^o\iota$ of the current object has to be another object in the heap or the root owner $\texttt{root}_a$. We need an owner address other than $\texttt{any}_a$ for the substitution of peer and self modifiers and prevent the runtime type-to-value judgment from using $\texttt{any}_a$ in the type by the constraint on the value of ${}^o\iota$. For a well-formed heap (defined in Definition 4.24), we know that the owner of each object in the heap is not $\texttt{any}_a$ and that we can determine such an ${}^o\iota$. Note that the runtime type arguments are invariant, and

that we do not need an additional constraint to ensure that the $\overline{{}^{r}T}$ are the most precise types possible.

We use a substitution that is very similar to the one from the simple dynamization to convert the static type ${}^{s}T$ into the corresponding runtime type ${}^{r}T'$. The owner ${}^{o}\iota$ is used to substitute `self` and `peer` modifiers, the current object ι is used to substitute `rep` modifiers, the class type variables are substituted by the runtime types $\overline{{}^{r}T}$ we determined from the heap for the current object, the method type variables are substituted by the runtime types $\overline{{}^{r}T_l}$ from the runtime environment, and the `lost` modifiers are substituted by the corresponding owners from the sequence $\overline{{}^{o}\iota}$. At runtime, we do not distinguish between `self` and `peer` modifiers, and substitute both with ${}^{o}\iota$. In Definition 3.15 we separately check for uses of `self` as the main modifier.

Definition 3.14 (*Dynamization of a Static Type*).

$\boxed{\mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{{}^{o}\iota}) = {}^{r}T}$ dynamization of static type (relative to ${}^{r}\Gamma$)

$$\frac{\begin{array}{cc} {}^{r}\Gamma = \{\overline{X_l \mapsto {}^{r}T_l}\ ;\ \texttt{this} \mapsto \iota, \_\} & h \vdash \iota :\ {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>} \\ {}^{o}\iota \in \mathrm{dom}(h) \cup \{\texttt{root}_a\} & \mathrm{ClassDom}(C) = \overline{X} \end{array} \quad {}^{s}T\,[{}^{o}\iota/\texttt{self}, {}^{o}\iota/\texttt{peer}, \iota/\texttt{rep}, \texttt{any}_a/\texttt{any}, \overline{{}^{r}T}/\overline{X}, \overline{{}^{r}T_l/X_l}, \overline{{}^{o}\iota_i/\texttt{lost}}] = {}^{r}T'}{\mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{{}^{o}\iota_i}) = {}^{r}T'}\ \text{DYNE}$$

Note that the outcome of *dyn* depends on finding ${}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}$, an appropriate supertype of the runtime type of the current object ι, which contains substitutions for all type variables not mapped by the environment. Thus, we may wonder whether there is more than one such appropriate superclass. However, because type variables are globally unique, if the free variables of ${}^{s}T$ are in the domain of a class, then they are not in the domain of any other class.

In the decorator example (Figure 11), using an ${}^{r}\Gamma$ where ${}^{r}\Gamma(\texttt{this}) = 1$, we can deduce that

$$\mathrm{dyn}(\texttt{peer MyDecoration}, h, {}^{r}\Gamma, \emptyset) = \texttt{root}_a\ \texttt{MyDecoration}$$

and

$$\mathrm{dyn}(\texttt{peer Decoration<peer Data>}, h, {}^{r}\Gamma, \emptyset) = \texttt{root}_a\ \texttt{Decoration<root}_a\ \texttt{Data>}.$$

Using the example from Figure 12, and again ${}^{r}\Gamma$ where ${}^{r}\Gamma(\texttt{this}) = 1$, we can deduce

$$\mathrm{dyn}(\texttt{rep D<rep Object>}, h, {}^{r}\Gamma, \emptyset) = 1\ \texttt{D<1 Object>}$$

and

$$\mathrm{dyn}(\texttt{rep C<lost Data, rep Object>}, h, {}^{r}\Gamma, 2) = 1\ \texttt{C<2 Data, 1 Object>}.$$

Earlier, we explained that

$$\mathrm{sdyn}(\texttt{X}, h, 2, 1\ \texttt{D<1 Object>}, \emptyset)$$

is not defined. Let us now consider ${}^{r}\Gamma'(\texttt{this}) = 2$ and $\mathrm{dyn}(\texttt{X}, h, {}^{r}\Gamma', \emptyset)$. Above, we deduced that

$$h \vdash 2 : 1\ \texttt{C<2 Data, 1 Object>}$$

and class `C` defines a type variable `X`. Therefore, we have that

$$\mathrm{dyn}(\texttt{X}, h, {}^{r}\Gamma', \emptyset) = 2\ \texttt{Data}.$$

3.3.5. Assigning a Static Type to a Value. To assign a static type to a value, we convert the static type into a runtime type, using the heap and runtime environment provided, and check whether this runtime type can be assigned to the value. If the main modifier of the static type is `self`, we also have to ensure that the value corresponds to the current object in the runtime environment. Note that we use an arbitrary substitution $\overline{{}^{o}\iota}$ for

`lost` modifiers which might appear in the static type. This expresses the meaning of `lost` as an existential quantifier that chooses a suitable owner to fulfill the runtime type-to-value judgment.

Definition 3.15 (*Assigning a Static Type to a Value* (*relative to* ${}^r\Gamma$)).

$\boxed{h, {}^r\Gamma \vdash v : {}^sT}$ static type ${}^sT$ assignable to value v (relative to ${}^r\Gamma$)

$$\frac{\mathrm{dyn}({}^sT, h, {}^r\Gamma, \overline{v_i}) \;=\; {}^rT \qquad h \vdash v : {}^rT \qquad {}^sT = \texttt{self \_<\_>} \Longrightarrow v = {}^r\Gamma(\texttt{this})}{h, {}^r\Gamma \vdash v : {}^sT}\ \text{RTSTE\_DEF}$$

For the well-formed heap judgment (defined in Definition 4.24), it is convenient to define a second version of this judgment which only takes the address of the current object instead of a complete runtime environment.

Definition 3.16 (*Assigning a Static Type to a Value (relative to* ι*)*).

$\boxed{h, \iota \vdash v : {}^sT}$ static type ${}^sT$ assignable to value v (relative to ι)

$$\frac{{}^r\Gamma = \{\emptyset\ ;\ \texttt{this} \mapsto \iota\} \qquad h, {}^r\Gamma \vdash v : {}^sT}{h, \iota \vdash v : {}^sT}\ \text{RTSTA\_DEF}$$

For the decorator example (Figure 11), using the previous results, we can now deduce

$h, 1 \vdash 2 :$ `peer MyDecoration`

and

$h, 1 \vdash 2 :$ `peer Decoration<peer Data>`.

Finally, combining all the results we deduced for the example from Figure 12, we can deduce

$h, 1 \vdash 2 :$ `rep D<rep Object>`

and

$h, 1 \vdash 2 :$ `rep C<lost Data, rep Object>`.

Note that there is no ownership modifier that could be used instead of `lost`, and would fulfill the judgment, since there is no modifier to express that an object is the representation of an object other than `this`.

3.4. Operational Semantics

We describe program execution by a big-step operational semantics for expressions and programs.

3.4.1. Evaluation of an Expression. The transition ${}^r\Gamma \vdash h, e \rightsquigarrow h', v$ expresses that the evaluation of an expression e in heap h and runtime environment ${}^r\Gamma$ results in value v and successor heap h'. The rules for evaluating expressions are presented and explained in the following.

Definition 3.17 (*Evaluation of an Expression*).

$\boxed{{}^r\Gamma \vdash h, e \rightsquigarrow h', v}$ big-step operational semantics

$$\frac{}{{}^r\Gamma \vdash h, \texttt{null} \rightsquigarrow h, \texttt{null}_a}\ \text{OS\_NULL} \qquad \frac{{}^r\Gamma(x) = v}{{}^r\Gamma \vdash h, x \rightsquigarrow h, v}\ \text{OS\_VAR}$$

$$\frac{\begin{array}{cc}\text{dyn}({}^{s}T, h, {}^{r}\Gamma, \emptyset) = {}^{r}T & \text{ClassOf}({}^{r}T) = C \\ (\forall f \in \text{fields}(C).\ \overline{fv}(f) = \texttt{null}_a) & h + ({}^{r}T, \overline{fv}) = (h', \iota)\end{array}}{{}^{r}\Gamma \vdash h, \texttt{new}\ {}^{s}T\,() \rightsquigarrow h', \iota}\ \text{OS\_NEW}$$

$$\frac{{}^{r}\Gamma \vdash h, e \rightsquigarrow h', v \qquad h', {}^{r}\Gamma \vdash v : {}^{s}T}{{}^{r}\Gamma \vdash h, (\,{}^{s}T\,)\ e \rightsquigarrow h', v}\ \text{OS\_CAST}$$

$$\frac{{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h', \iota_0 \qquad h'(\iota_0.f) = v}{{}^{r}\Gamma \vdash h, e_0.f \rightsquigarrow h', v}\ \text{OS\_READ}$$

$$\frac{{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \qquad {}^{r}\Gamma \vdash h_0, e_1 \rightsquigarrow h_1, v \qquad h_1[\iota_0.f = v] = h'}{{}^{r}\Gamma \vdash h, e_0.f = e_1 \rightsquigarrow h', v}\ \text{OS\_WRITE}$$

$$\frac{\begin{array}{cc}{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h_0, \iota_0 & {}^{r}\Gamma \vdash h_0, \overline{e_q} \rightsquigarrow h_1, \overline{v_q} \\ \text{MBody}(h_0, \iota_0, m) = e & \text{MSig}(h_0, \iota_0, m) = \_ \texttt{<}\overline{X_l\ \texttt{extends}\ \_}\texttt{>}\ \_\ m(\_\overline{pid}) \\ \text{dyn}({}^{s}\overline{T_l}, h, {}^{r}\Gamma, \emptyset) = \overline{{}^{r}T_l} & {}^{r}\Gamma' = \{\overline{X_l \mapsto {}^{r}T_l}\ ;\ \texttt{this} \mapsto \iota_0,\ \overline{pid \mapsto v_q}\} \\ {}^{r}\Gamma' \vdash h_1, e \rightsquigarrow h', v & \end{array}}{{}^{r}\Gamma \vdash h, e_0.m\texttt{<}\overline{{}^{s}T_l}\texttt{>}(\overline{e_q}) \rightsquigarrow h', v}\ \text{OS\_CALL}$$

The `null` expression always evaluates to the $\texttt{null}_a$ value (OS_NULL). Parameters, including `this`, are evaluated by looking up the stored value in the stack frame, which is part of the runtime environment ${}^{r}\Gamma$ (OS_VAR). Object creation determines the runtime type for the object from the static type using the heap h and the runtime environment ${}^{r}\Gamma$, and initializes all field values to $\texttt{null}_a$. (ClassOf$({}^{r}T)$ yields the class of the runtime type ${}^{r}T$; note that ${}^{s}T$ could be a type variable and cannot be used to determine the class. fields(C) yields all fields declared in or inherited by C.) We construct the new object using this runtime type and the field values and add it to the original heap h, resulting in an updated heap h' and the address ι of the new object in the new heap (OS_NEW). For cast expressions, we evaluate the expression e and check that the resulting value is well-typed with the static type given in the cast expression w.r.t. the current environment (OS_CAST).

For field reads (OS_READ), we evaluate the receiver expression e_0 and then look up the field in the heap. We require that the receiver expression evaluates to an address ι_0 and not to the $\texttt{null}_a$ value. For the update of a field f, we evaluate the receiver expression e_0 to address ι_0 and the right-hand side expression to value v, and update the heap h_1 with the new field value (OS_WRITE).

For method calls (OS_CALL), we evaluate the receiver expression e_0 and actual method arguments $\overline{e_q}$ in the usual order. The receiver object is used to look up the most concrete method body and the method signature (from which we extract the names of the method type variables $\overline{X_l}$ and the parameter names $\overline{pid}$ used to construct the new runtime environment ${}^{r}\Gamma'$; the static types in the method signature are irrelevant here). The method body expression e is then evaluated in the runtime environment that maps m's type variables to actual type arguments, as well as m's formal method parameters (including `this`) to the actual method arguments. (Note that the method type arguments are dynamized using an empty substitution for `lost` modifiers, which forbids occurrences of `lost` in the type arguments; our type rules enforce this constraint and, therefore, ensure that the dynamization is defined.) The resulting heap and address are the result of the call.

3.4.2. Evaluation of a Program. A program with main class C is executed by evaluating the main expression e in a heap h_0 that contains exactly one C instance in the root context where all fields of C are initialized to $\texttt{null}_a$ and a runtime environment ${}^{r}\Gamma_0$ which maps `this` to this C instance.

Definition 3.18 (*Evaluation of a Program*).

$\boxed{\vdash P \rightsquigarrow h, v}$ big-step operational semantics of a program

$$\frac{\begin{array}{l}\forall f \in \text{fields}(C).\ \overline{fv}(f) = \texttt{null}_a \\ \emptyset + (\texttt{root}_a\ C\texttt{<>}, \overline{fv}) = (h_0, \iota_0) \\ {}^r\Gamma_0 = \{\emptyset\ ;\ \texttt{this} \mapsto \iota_0\} \qquad {}^r\Gamma_0 \vdash h_0, e \rightsquigarrow h, v\end{array}}{\vdash \overline{Cls},\ C,\ e \rightsquigarrow h, v}\ \text{OSP\_DEF}$$

In the example from Section 2, we would have the classes from Figures 2, 3, and 4 as the sequence of classes, we use class identifier `Client` as the main class, and the expression `this.main()` as the main expression.

This concludes our discussion of the programming language syntax and semantics. The following section presents the topological system of GUT.

4. TOPOLOGICAL SYSTEM

In this section we formalize the topological system of GUT. We first formalize viewpoint adaptation and define the ordering of ownership modifiers and subtyping. We then present the static well-formedness conditions, including the type rules and the runtime well-formedness conditions. We conclude this section by discussing the properties of the topological system, most importantly type safety. The encapsulation system is discussed in Section 5.

Note that the formalization presents the rules that are necessary for type safety; it allows programs which are not meaningful for programmers to write, for example, it allows the declared field type to contain the `lost` ownership modifier, even though such a field can never be updated. This design choice keeps the formalization minimal and highlights what is necessary for type safety.

Type checking is performed in a type environment ${}^s\Gamma$, which maps the type variables of the enclosing class and method to their upper bounds, and the method parameters to their types:

$${}^s\Gamma ::= \{\overline{X \mapsto {}^sN};\ \overline{x \mapsto {}^sT}\}$$

As with the runtime environment ${}^r\Gamma$, we overload the notation; ${}^s\Gamma(X)$ refers to the upper bound of type variable X, and ${}^s\Gamma(x)$ refers to the type of method parameter x.

Note that the static environment stores the upper bounds for class and method type variables in its first component; the runtime environment only needs to store the actual type arguments for the method type variables, as the arguments for the class type variables are stored in the heap.

4.1. Viewpoint Adaptation

As we informally discussed in Section 2.2, ownership modifiers express ownership relative to an object. Therefore, they have to be adapted whenever the viewpoint changes. In the type rules, we need to *adapt a type* ${}^sT$ *from a viewpoint* that is described by another type ${}^sT'$ *to the viewpoint* `this`. In the following, we omit the phrase "to the viewpoint `this`". To perform the viewpoint adaptation, we define an overloaded operator $\triangleright$, whose remit is: (1) Adapt an ownership modifier from a viewpoint that is described by another ownership modifier; (2) Adapt a type from a viewpoint that is described by an ownership modifier; (3) Adapt a type from a viewpoint that is described by another type.

4.1.1. Adapting an Ownership Modifier w.r.t. an Ownership Modifier. We explain viewpoint adaptation using a field access $e_1.f$. Analogous adaptations occur for method

parameters and results, as well as upper bounds of type parameters. Let u be the main modifier of e_1's type, which expresses ownership relative to `this`. Let u' be the main modifier of f's type, which expresses ownership relative to the object that contains f. Then relative to `this`, the type of the field access $e_1.f$ has main modifier $u \triangleright u'$.

Definition 4.1 (*Adapting Ownership Modifiers*).

$\boxed{u \triangleright u' = u''}$ combining two ownership modifiers

$$\frac{}{\texttt{self} \triangleright u = u}\ \text{UCU\_SELF} \qquad \frac{}{\texttt{peer} \triangleright \texttt{peer} = \texttt{peer}}\ \text{UCU\_PEER}$$

$$\frac{}{\texttt{rep} \triangleright \texttt{peer} = \texttt{rep}}\ \text{UCU\_REP} \qquad \frac{}{u \triangleright \texttt{any} = \texttt{any}}\ \text{UCU\_ANY}$$

$$\frac{\text{otherwise}}{u \triangleright u' = \texttt{lost}}\ \text{UCU\_LOST}$$

The field access $e_1.f$ illustrates the motivation for this definition.

(1) Accesses via the current object `this` (that is, e_1 is the variable `this`) do not require a viewpoint adaptation, since the ownership modifier of the field is already relative to `this`. The `self` modifier is used to distinguish accesses through `this` from other accesses.
(2) If the main modifiers of both e_1 and f are `peer`, then the object referenced by e_1 has the same owner as `this`, and the object referenced by $e_1.f$ has the same owner as e_1, and thus the same owner as `this`. Consequently, the main modifier of $e_1.f$ is also `peer`.
(3) If the main modifier of e_1 is `rep` and the main modifier of f is `peer`, then the main modifier of $e_1.f$ is `rep`, since the object referenced by e_1 is owned by `this` and the object referenced by $e_1.f$ has the same owner as e_1, that is, `this`.
(4) If the object referenced by f can have an arbitrary owner, then the object referenced by $e_1.f$ can also have an arbitrary owner, regardless of the owner of e_1. That is, if the main modifier of f is `any`, then the main modifier of $e_1.f$ is also `any`, regardless of the modifier of e_1.
(5) In all other cases, we cannot determine statically that the object referenced by $e_1.f$ has the same owner as `this`, is owned by `this`, or that it can be an object with an arbitrary owner. Therefore, in these cases, the main modifier of $e_1.f$ is `lost`.

4.1.2. Adapting a Type w.r.t. an Ownership Modifier. As explained in Section 2, type variables are not subject to viewpoint adaptation. For nonvariable types, we determine the adapted main modifier and adapt the type arguments recursively.

Definition 4.2 (*Adapting a Type w.r.t. an Ownership Modifier*).

$\boxed{u \triangleright {}^{s}T = {}^{s}T'}$ ownership modifier - type adaptation

$$\frac{}{u \triangleright X = X}\ \text{UCT\_VAR} \qquad \frac{u \triangleright u' = u'' \quad u \triangleright \overline{{}^{s}T} = \overline{{}^{s}T}'}{u \triangleright u'\ C\texttt{<}\overline{{}^{s}T}\texttt{>} = u''\ C\texttt{<}\overline{{}^{s}T}'\texttt{>}}\ \text{UCT\_NVAR}$$

4.1.3. Adapting a Type w.r.t. a Type. We adapt a type ${}^{s}T$ from the viewpoint described by another type, $u\ C\texttt{<}\overline{{}^{s}T}\texttt{>}$.

Definition 4.3 (*Adapting a Type w.r.t. a Type*).

$\boxed{{}^sN \triangleright {}^sT \;=\; {}^sT'}$ type - type combination

$$\frac{u \triangleright {}^sT \;=\; {}^sT_1 \qquad {}^sT_1[\overline{{}^sT}/\overline{X}] = {}^sT' \qquad \text{ClassDom}(C) = \overline{X}}{u\ C\texttt{<}\overline{{}^sT}\texttt{>} \triangleright {}^sT \;=\; {}^sT'}\ \text{TCT\_DEF}$$

The operator $\triangleright$ adapts the ownership modifiers of ${}^sT$ and substitutes the type arguments $\overline{{}^sT}$ for the type variables $\overline{X}$ of C. Since the type arguments are already relative to `this`, they are not subject to viewpoint adaptation. Therefore, the substitution of type variables happens after the viewpoint adaptation of ${}^sT$'s ownership modifiers.

Note that the first parameter is a nonvariable type, since concrete ownership information u is needed to adapt the viewpoint and the actual type arguments $\overline{{}^sT}$ are needed to substitute the type variables $\overline{X}$. In the type rules, subsumption will be used to replace type variables by their upper bounds and thereby obtain a concrete type as first argument of $\triangleright$. The adaptation is undefined, if the look-up of the domain of class C is undefined, that is, C is not a valid class name in the program, or if the number of type arguments does not correspond to the number of type variables.

As an example, consider the call `map.getNode(key)` in Figure 4. The receiver `map` has type `peer Map<rep ID, any Data>`. The return type of the method is `rep Node<K, V>`. This type is first adapted from the viewpoint `peer`, resulting in the type `lost Node<K, V>`; then the substitution of the type arguments for the type variables results in the type `lost Node<rep ID, any Data>`.

If the order of the viewpoint adaptation and the substitution were the other way around, we would first substitute `rep Data` for `K` and `any Data` for `V`, resulting in `rep Node<rep ID, any Data>`. Then, adapting this type from the viewpoint `peer` would result in `lost Node<lost ID, any Data>`. This order of operations would not correctly represent the ownership information of the first type argument.

It is convenient to define look-up functions that determine the declared type(s) of a field, method signature, or upper bound, and adapt it from the viewpoint given by a type. These functions are defined in the following sections.

4.1.4. Adapted Field Type Look-up. To look up the viewpoint-adapted type of a field, we look up the declared type of the field and apply viewpoint adaptation. ClassOf(${}^sN$) yields the class of the nonvariable type ${}^sN$. The FType function is again only defined if the field is declared in the class of the given type.

Definition 4.4 (*Adapted Field Type Look-up*).

$\boxed{\text{FType}({}^sN, f) = {}^sT}$ look up field f in type ${}^sN$

$$\frac{\text{FType}(\text{ClassOf}({}^sN), f) = {}^sT_1 \qquad {}^sN \triangleright {}^sT_1 \;=\; {}^sT}{\text{FType}({}^sN, f) = {}^sT}\ \text{SFTN\_DEF}$$

4.1.5. Adapted Method Signature Look-up. To look up the viewpoint-adapted method signature, we look up the signature in the class of the type and then viewpoint-adapt the upper bounds, the return type, and the parameter types. The method type arguments $\overline{{}^sT_l}$ are substituted for the method type variables $\overline{X_l}$ in all types.

Definition 4.5 (*Adapted Method Signature Look-up*).

$\boxed{\text{MSig}({}^s N, m, \overline{{}^s T}) = ms_o}$ m in ${}^s N$ with method type arguments $\overline{{}^s T}$ substituted

$$\frac{\begin{array}{l}\text{MSig}(\text{ClassOf}({}^s N), m) = p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^s N_l}\texttt{>}\ {}^s T\ \ m\,(\overline{{}^s T'_q\ pid}) \\ ({}^s N \rhd \overline{{}^s N_l})[\overline{{}^s T_l}/\overline{X_l}] \ =\ \overline{{}^s N'_l} \qquad ({}^s N \rhd {}^s T)[\overline{{}^s T_l}/\overline{X_l}] \ =\ {}^s T' \\ ({}^s N \rhd \overline{{}^s T'_q})[\overline{{}^s T_l}/\overline{X_l}] \ =\ \overline{{}^s T''_q}\end{array}}{\text{MSig}({}^s N, m, \overline{{}^s T_l}) = p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^s N'_l}\texttt{>}\ {}^s T'\ m\,(\overline{{}^s T''_q\ pid})}\ \text{SMSN\_DEF}$$

Note that we have to perform capture-avoiding substitutions, that is, free type variables in ${}^s N$ must not be captured by the $\overline{X_l}$. If necessary, the $\overline{X_l}$ can be α-renamed in the declared method signature.

4.1.6. Adapted Upper Bounds Look-up. The bounds of a type are the upper bounds of the class of the type after viewpoint adaptation.

Definition 4.6 (*Adapted Upper Bounds Look-up*).

$\boxed{\text{ClassBnds}({}^s N) = \overline{{}^s N}}$ look up bounds of type ${}^s N$

$$\frac{\begin{array}{l}\text{ClassBnds}(\text{ClassOf}({}^s N)) = \overline{{}^s N}_1 \\ {}^s N \rhd \overline{{}^s N}_1 \ =\ \overline{{}^s N}\end{array}}{\text{ClassBnds}({}^s N) = \overline{{}^s N}}\ \text{SCBN\_DEF}$$

4.2. Static Ordering Relations

We first define an ordering relation $<:_u$ for ownership modifiers. Recall the definition of subclassing (symbol $\sqsubseteq$) from Definition 3.1, which is the reflexive and transitive relation on classes declared in a program. Building on the ordering of ownership modifiers and subclassing, we define *subtyping* (symbol $<:$), which additionally takes main modifiers into account.

4.2.1. Ordering of Ownership Modifiers. The ordering of ownership modifiers $<:_u$ relates more concrete modifiers to less concrete ones. Both `self` and `peer` express that an object has the same owner as `this`, where `self` is only used for the object `this`, and is therefore more specific than `peer` (OMO_TP). Both `peer` and `rep` are more specific than `lost` (OMO_PL and OMO_RL). All ownership modifiers are below `any` (OMO_UA), and the ordering of ownership modifiers is reflexive (OMO_REFL).

Definition 4.7 (*Ordering of Ownership Modifiers*).

$\boxed{u <:_u u'}$ ordering of ownership modifiers

$$\frac{}{\texttt{self} <:_u \texttt{peer}}\ \text{OMO\_TP} \qquad \frac{}{\texttt{peer} <:_u \texttt{lost}}\ \text{OMO\_PL}$$

$$\frac{}{\texttt{rep} <:_u \texttt{lost}}\ \text{OMO\_RL} \qquad \frac{}{u <:_u \texttt{any}}\ \text{OMO\_UA}$$

$$\frac{}{u <:_u u}\ \text{OMO\_REFL}$$

Note that the ordering of ownership modifiers is not transitive, as $\texttt{self} <:_u \texttt{lost}$ is not included; this could be added, but only transitivity of subtyping is needed.

4.2.2. Static Subtyping. The subtype relation $<:$ is defined on static types. The judgment ${}^s\Gamma \vdash {}^s T <: {}^s T'$ expresses that type ${}^s T$ is a subtype of type ${}^s T'$ in type environment ${}^s\Gamma$. The environment is needed, since static types may contain type variables. The rules for this subtyping judgment are given in Definition 4.8.

Two types with the same main modifier are subtypes if the corresponding classes are subclasses. Ownership modifiers in the superclass instantiation ($\overline{{}^sT}_1$) are relative to the instance of class C, whereas the modifiers in a type are relative to this. In particular, from the subclass relation $C\langle\overline{X}\rangle \sqsubseteq C'\langle\overline{{}^sT}_1\rangle$ we cannot simply derive ${}^s\Gamma \vdash u\ C\langle\overline{X}\rangle <:\ u\ C'\langle\overline{{}^sT}_1\rangle$. Instead, $\overline{{}^sT}_1$ has to be adapted from the viewpoint of the C instance to this (ST1). For types with the same class, according to ST2, the main modifiers have to follow the ordering of ownership modifiers and the type arguments have to follow the type argument subtyping $<:_l$, explained below. A type variable is a subtype of itself and of its upper bound in the type environment (ST3). Subtyping is transitive (ST4).

Definition 4.8 (*Static Subtyping*).

$\boxed{{}^s\Gamma \vdash {}^sT <:\ {}^sT'}$ static subtyping

$$\frac{C\langle\overline{X}\rangle \sqsubseteq C'\langle\overline{{}^sT}_1\rangle \quad u\ C\langle\overline{{}^sT}\rangle \rhd \overline{{}^sT}_1 = \overline{{}^sT}'}{{}^s\Gamma \vdash u\ C\langle\overline{{}^sT}\rangle <:\ u\ C'\langle\overline{{}^sT}'\rangle}\ \text{ST1} \qquad \frac{u <:_u u' \qquad \vdash \overline{{}^sT} <:_l \overline{{}^sT}'}{{}^s\Gamma \vdash u\ C\langle\overline{{}^sT}\rangle <:\ u'\ C\langle\overline{{}^sT}'\rangle}\ \text{ST2}$$

$$\frac{{}^sT = X\ \vee\ {}^s\Gamma(X) = {}^sT}{{}^s\Gamma \vdash X <:\ {}^sT}\ \text{ST3} \qquad \frac{{}^s\Gamma \vdash {}^sT <:\ {}^sT_1 \quad {}^s\Gamma \vdash {}^sT_1 <:\ {}^sT'}{{}^s\Gamma \vdash {}^sT <:\ {}^sT'}\ \text{ST4}$$

Reflexivity of nonvariable types can be deduced from the reflexivity of ownership modifier ordering, type argument subtyping, and rule ST2. For type variables, rule ST3 gives reflexivity.

The type `any Object` is at the root of the type hierarchy. Every other type is a subtype of it.

In Section 3.1.1 we derived

`MyDecoration` $\sqsubseteq$ `Decoration<peer Data>`.

Using rule ST1 we can derive the two subtype relationships

`rep MyDecoration` <: `rep Decoration<rep Data>`

and

`any MyDecoration` <: `any Decoration<lost Data>`.

Note that in the second example, we cannot give concrete ownership information for the type argument in the supertype, since we do not know the location of the object and cannot express the relationship between the object and the type argument from an arbitrary viewpoint.

By rule ST2 we can derive `rep MyDecoration` <: `any MyDecoration`.

We illustrate rule ST3 using the method call `res.set(in)` from class `Decorator` (Figure 5). The variable `res` has the type variable `O` as its type. To type-check the method call, we need a concrete class to look up the method signature for method `set`. We use rule ST3 to go from the type variable `O` to its upper bound `peer Decoration<V>`, and can then successfully type-check the call.

Type Argument Subtyping. We use type argument subtyping (symbol $<:_l$) only for subtyping in type argument positions. Two nonvariable type arguments either have the same main modifier or the supertype has the `lost` main modifier. The type arguments can recursively vary by the type argument relation. A type variable is only a type argument subtype of itself.

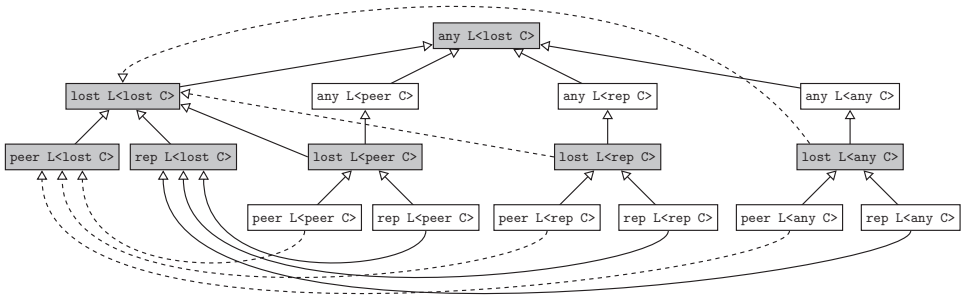


Fig. 13. Static subtyping illustrated on an example. Both dashed and solid arrows denote subtyping; we use dashed lines only to make the paths clearer. Gray types contain `lost` and cannot be supertypes for strict subtyping (Definition 4.10).

Definition 4.9 (*Type Argument Subtyping*).

$\boxed{\vdash {}^{s}T <:_{l} {}^{s}T'}$ type argument subtyping

$$\frac{u' \in \{u, \texttt{lost}\} \quad \vdash \overline{{}^{s}T} <:_{l} \overline{{}^{s}T}'}{\vdash u\ C\texttt{<}\overline{{}^{s}T}\texttt{>} <:_{l} u'\ C\texttt{<}\overline{{}^{s}T}'\texttt{>}}\ \text{AST1} \qquad \frac{}{\vdash X <:_{l} X}\ \text{AST2}$$

This relation allows us to abstract away ownership information in type arguments. Note that we do not use the same subtyping relation that is used on "top-level" types. It would not be type-safe to allow `peer List<peer Data>` as a subtype of `peer List<any Object>`, as the latter type allows storing objects of an arbitrary ownership and class information, whereas the former is restricted to `Data` objects that share the same owner as the current object. Also note that we impose restrictions on the uses of types that contain the `lost` modifier. Nevertheless, this abstraction is helpful, as it allows us to reference and modify objects with partially unknown ownership.

Consider the following class declaration:

```
class L<X extends any Object> {...}
```

Figure 13 shows the relation between instantiations of class `L`. Note that an arbitrary ${}^{s}\Gamma$ can be used, as we do not consider type variables here.

As another example, consider the code from Figure 12. We can derive the subclass relation `D<Z>` $\sqsubseteq$ `C<rep Data, Z>` and the subtype relation ${}^{s}\Gamma \vdash$ `self D<Z>` $<:$ `self C<rep Data, Z>`. However, deriving ${}^{s}\Gamma \vdash$ `peer D<Z>` $<:$ `peer C<rep Data, Z>` is not possible because it would interpret the instantiation of type variable `X` as representation of the current object, even though it is meant to be the representation of the `D` object. The correct subtype relation is ${}^{s}\Gamma \vdash$ `peer D<Z>` $<:$ `peer C<lost Data, Z>`.

Strict Subtyping. In certain judgments it is convenient to express that a type ${}^{s}T$ is a subtype of type ${}^{s}T'$ and that ${}^{s}T'$ does not contain the `lost` ownership modifier. We define a strict subtyping judgment to express this concisely.

Definition 4.10 (*Strict Subtyping*).

$\boxed{{}^{s}\Gamma \vdash {}^{s}T <:_{s} {}^{s}T'}$ strict static subtyping

$$\frac{{}^{s}\Gamma \vdash {}^{s}T <: {}^{s}T' \quad \texttt{lost} \notin {}^{s}T'}{{}^{s}\Gamma \vdash {}^{s}T <:_{s} {}^{s}T'}\ \text{SSTDEF}$$

In Figure 13, types that contain `lost` are marked with a gray background. These types cannot appear as a strict supertype. For the example in Figure 12, `peer D<Z>` is a subtype, but not a strict subtype of `peer C<lost Data, Z>`.

4.3. Static Well Formedness

This section defines well-formedness judgments for the static system, including the topological type rules.

4.3.1. Well-formed Static Type. The judgment ${}^s\Gamma \vdash {}^sT$ OK expresses that type ${}^sT$ is well formed in type environment ${}^s\Gamma$. Type variables are well formed if they are contained in the type environment (WFT_VAR). We use the overloaded notation $X \in {}^s\Gamma$ to denote that ${}^s\Gamma(X)$ is defined, and similarly $x \in {}^s\Gamma$, to denote that ${}^s\Gamma(x)$ is defined; note that ${}^s\Gamma$ is a pair consisting of two separate domains. Well-formedness of the upper bounds is checked by the well-formed static environment judgment (Definition 4.18) that checks well-formedness of the types in the environment. A nonvariable type $u\ C\texttt{<}\overline{{}^sT}\texttt{>}$ is well formed (WFT_NVAR) if its type arguments $\overline{{}^sT}$ are well formed, do not contain `self` (denoted by $\texttt{self} \notin \overline{{}^sT}$), and each actual type argument is a subtype of the upper bound adapted to the current viewpoint.

Definition 4.11 (*Well-formed Static Type*).

$\boxed{{}^s\Gamma \vdash {}^sT\ \ \mathsf{OK}}$ well-formed static type

$$\frac{X \in {}^s\Gamma}{{}^s\Gamma \vdash X\ \ \mathsf{OK}}\ \text{WFT\_VAR}$$

$$\frac{\begin{array}{ll} {}^s\Gamma \vdash \overline{{}^sT}\ \ \mathsf{OK} & \texttt{self} \notin \overline{{}^sT} \\ \mathrm{ClassBnds}\left(u\ C\texttt{<}\overline{{}^sT}\texttt{>}\right) = \overline{{}^sN} & {}^s\Gamma \vdash \overline{{}^sT} <: \overline{{}^sN} \end{array}}{{}^s\Gamma \vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>}\ \ \mathsf{OK}}\ \text{WFT\_NVAR}$$

We restrict the `self` modifier to the main modifier of a nonvariable type. Allowing the use of `self` in a type argument position would complicate the runtime system without adding expressiveness.

Note how the look-up of the upper bounds can result in lost ownership information. This well-formed type judgment is intentionally weak and forbids only what is needed for the soundness of the system; this simplifies the formalization, but complicates the proofs. It allows static types that will never reference a valid object at runtime, and could therefore be forbidden without limiting the expressiveness of the system and providing earlier detection of likely errors.

As an example, consider class C from Figure 12:

```
class C<X extends rep Data, Y extends any Object> {}
```

The type `peer C<peer Data, peer Object>` is well formed. The viewpoint-adapted upper bound of type variable `X` is `lost Data`, which is a supertype of `peer Data`. However, this type will never reference an object at runtime because the ownership of the type argument is not consistent with the upper bound, and the type system forbids the creation of such objects, as we explain next.

4.3.2. Strictly Well-formed Static Type. To guarantee well-formedness of the heap, we also define a stricter form of well-formed types that is used for types that can be used for object creations.

A type variable is strictly well formed if it is contained in the type environment (SWFT_VAR). We do not need to put additional constraints on the upper bound of the type variable. A nonvariable type is strictly well formed if its type arguments are also

strictly well formed, the modifiers `self` and `lost` do not appear in the type, and the type arguments are strict subtypes of the adapted upper bounds (SWFT_NVAR).

Definition 4.12 (*Strictly Well-formed Static Type*).

$\boxed{{}^s\Gamma \vdash {}^sT \ \text{strictly OK}}$ strictly well-formed static type

$$\frac{X \in {}^s\Gamma}{{}^s\Gamma \vdash X \ \text{strictly OK}}\ \text{SWFT\_VAR}$$

$$\frac{{}^s\Gamma \vdash \overline{{}^sT} \ \text{strictly OK} \qquad \{\texttt{self}, \texttt{lost}\} \notin u\ C\texttt{<}\overline{{}^sT}\texttt{>} \qquad \text{ClassBnds}(u\ C\texttt{<}\overline{{}^sT}\texttt{>}) = \overline{{}^sN} \qquad {}^s\Gamma \vdash \overline{{}^sT} <:_s \overline{{}^sN}}{{}^s\Gamma \vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>} \ \text{strictly OK}}\ \text{SWFT\_NVAR}$$

The use of strict subtyping ensures that `lost` does not appear in the viewpoint-adapted upper bounds, and thus that no ownership information was lost by the viewpoint adaptation.

Continuing the example the type `peer C<peer Data, peer Object>` is not strictly well formed. The viewpoint-adapted upper bound of type variable `X` contains `lost`, and therefore the type cannot be used in `new` expressions. This ensures that a well-formed heap will never contain such an ill-formed object.

We do need both nonstrict and strict well-formed type judgments to allow flexible access to objects with lost ownership information. We will present an example after the topological type rules.

4.3.3. Topological Type Rules. We are now ready to present the topological type rules. The judgment ${}^s\Gamma \vdash e : {}^sT$ expresses that expression e is well typed with type ${}^sT$ in environment ${}^s\Gamma$. The definition also uses the strict typing judgment ${}^s\Gamma \vdash e :_s {}^sT$ to expresses that expression e is well typed with type ${}^sT$ in environment ${}^s\Gamma$ and that ${}^sT$ does not contain `lost` ownership modifiers; this definition simplifies the type rules TR_WRITE and TR_CALL.

Definition 4.13 (*Topological Type Rules*).

$\boxed{{}^s\Gamma \vdash e : {}^sT}$ expression typing

$$\frac{{}^s\Gamma \vdash e : {}^sT_1 \qquad {}^s\Gamma \vdash {}^sT_1 <: {}^sT \qquad {}^s\Gamma \vdash {}^sT \ \text{OK}}{{}^s\Gamma \vdash e : {}^sT}\ \text{TR\_SUBSUM} \qquad \frac{\texttt{self} \notin {}^sT \qquad {}^s\Gamma \vdash {}^sT \ \text{OK}}{{}^s\Gamma \vdash \texttt{null} : {}^sT}\ \text{TR\_NULL} \qquad \frac{{}^s\Gamma(x) = {}^sT}{{}^s\Gamma \vdash x : {}^sT}\ \text{TR\_VAR}$$

$$\frac{{}^s\Gamma \vdash {}^sT \ \text{strictly OK} \qquad \text{om}({}^sT, {}^s\Gamma) \in \{\texttt{peer}, \texttt{rep}\}}{{}^s\Gamma \vdash \texttt{new}\ {}^sT\ \texttt{()} : {}^sT}\ \text{TR\_NEW} \qquad \frac{{}^s\Gamma \vdash e : \_ \qquad {}^s\Gamma \vdash {}^sT \ \text{OK}}{{}^s\Gamma \vdash \texttt{(}{}^sT\texttt{)}\ e : {}^sT}\ \text{TR\_CAST}$$

$$\frac{{}^s\Gamma \vdash e_0 : {}^sN_0 \qquad \text{FType}({}^sN_0, f) = {}^sT}{{}^s\Gamma \vdash e_0.f : {}^sT}\ \text{TR\_READ} \qquad \frac{{}^s\Gamma \vdash e_0 : {}^sN_0 \qquad \text{FType}({}^sN_0, f) = {}^sT \qquad {}^s\Gamma \vdash e_1 :_s {}^sT}{{}^s\Gamma \vdash e_0.f = e_1 : {}^sT}\ \text{TR\_WRITE}$$

$$\frac{{}^s\Gamma \vdash e_0 : {}^sN_0 \qquad \text{MSig}({}^sN_0, m, \overline{{}^sT_l}) = \_ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^sN_l}\texttt{>}\ {}^sT\ m\texttt{(}\overline{{}^sT'_q\ pid}\texttt{)} \qquad {}^s\Gamma \vdash \overline{e_q} :_s \overline{{}^sT'_q} \qquad {}^s\Gamma \vdash \overline{{}^sT_l} \ \text{strictly OK} \qquad {}^s\Gamma \vdash \overline{{}^sT_l} <:_s \overline{{}^sN_l}}{{}^s\Gamma \vdash e_0.m\texttt{<}\overline{{}^sT_l}\texttt{>(}\overline{e_q}\texttt{)} : {}^sT}\ \text{TR\_CALL}$$

$\boxed{{}^s\Gamma \vdash e :_s {}^sT}$ strict expression typing

$$\frac{{}^s\Gamma \vdash e : {}^sT \quad \texttt{lost} \notin {}^sT}{{}^s\Gamma \vdash e :_s {}^sT}\ \text{STR\_DEF}$$

An expression of type ${}^sT_1$ can also be typed with ${}^sT_1$'s well-formed supertypes (TR_SUBSUM). The `null`-reference can have any well-formed type that does not contain the `self` modifier (TR_NULL). The type of method parameters (including `this`) is determined by a look-up in the type environment (TR_VAR). Objects must be created in a specific context. Therefore, only types with main modifiers `peer` and `rep` are allowed for object creations. Also, the type must be a strictly well-formed static type (TR_NEW).

Function om is used to determine the main ownership modifier of a type. For a nonvariable type, om yields the main ownership modifier, ignoring the static environment; for a type variable, om yields the main ownership modifier of the upper bound of the type, as determined using the given static environment.

The rule for casts (TR_CAST) is straightforward; it could be strengthened to prevent more cast errors statically, but we omit these checks since they are not strictly needed.

As explained in Section 4.1, the type of a field access is determined by adapting the declared type of the field from the viewpoint described by the type of the receiver (TR_READ). If the receiver type is a type variable, subsumption is used to determine its upper bound because FType is defined on nonvariable types only. Subsumption is also used for inherited fields to ensure that f is actually declared in ${}^sN_0$.

For a field update, the right-hand side expression must be typable as the viewpoint-adapted field type, which is also the type of the whole field update expression (TR_WRITE). The rule is analogous to field read, but has the additional requirement that the adapted field type does not contain `lost`, which is enforced by using strict expression typing. In this case, we cannot enforce statically that the right-hand side has the required owner, and therefore must forbid the update.

The rule for method calls (TR_CALL) is in many ways similar to field reads (for result-passing) and updates (for argument-passing). The method signature is determined using the receiver type ${}^sN_0$ and the actual type arguments $\overline{{}^sT_l}$ substituted for the method's type variables $\overline{X_l}$. Again, subsumption is used to find a type for the receiver that declares the method. The type of the invocation expression is determined by the return type ${}^sT$. The method type arguments must be subtypes of the upper bounds and, modulo subsumption, the actual method argument expressions $\overline{e_q}$ must have the formal parameter types. For these checks to be precise, we have to forbid `lost` in the upper bounds and the parameter types, which is achieved by using strict subtyping and strict expression typing, respectively. Note that the return type may contain `lost`.

The method type arguments must be strictly well-formed types. Like the static type that is used in an object creation, the static types that are supplied as method type arguments are dynamized to the corresponding runtime types. They are used in the operational semantics to construct the runtime environment. We need to show that the method type arguments are well formed from the viewpoint of the receiver, and therefore need to enforce strict well-formed types as method type arguments.

Note that the topological type system treats pure and nonpure methods identically. For type soundness we always need to forbid method calls where the viewpoint-adapted upper bounds or parameter types contain `lost`. The purity information is only used for the encapsulation system, which is presented in Section 5.

Deterministic Object Creation. We forbid creation of objects that contain the `lost` modifier, statically by enforcing that `lost` is not contained in the type and at runtime by using dyn with an empty substitution for `lost`. Also, the `any` modifier is forbidden as

the main modifier of the static type. A design alternative would be to allow the creation of types that contain `lost` anywhere in the type and also `any` as the main modifier, and then at runtime choose an arbitrary owner that fulfills the constraints imposed by the upper bounds. Even though this is not a soundness issue, we prefer the more stringent rules that ensure deterministic behavior of object creation.

We ensure that all subexpressions are well formed, not strictly well formed, to allow flexible access to objects with lost ownership information. Imagine that class `C` from Figure 12 has a field `f` of type `Y`. The type `peer D<peer Object>` is both nonstrictly and strictly well formed. Imagine a variable `x` of type `peer D<peer Object>` and a field read `x.f`. By rule TR_READ, we need to use subsumption to find the supertype that declares field `f`. This supertype is `peer C<lost Data, peer Object>`, since from the current viewpoint we cannot express the ownership of type argument `X`. This type is not strictly well formed because it contains `lost` in a type argument and also in an upper bound. However, we can still consider this type well formed and can determine `peer Object` as type of the field access. Similarly, an update of field `f` would be valid. If we required strict well-formedness for all static types, we would lose this significant expressiveness.

4.3.4. Well-formed Class Declaration. The judgment Cls OK expresses that class declaration *Cls* is well formed. According to rule WFC_DEF, this is the case if: (1) the upper bounds of *Cls*'s type variables are well formed in the type environment that maps *Cls*'s type variables to their upper bounds; (2) the `self` modifier is not used in *Cls*'s upper bounds; (3) the type arguments to the superclass are strictly well formed, and they are strict subtypes of the upper bounds of the superclass; (4) *Cls*'s fields are well formed; and (5) *Cls*'s methods are well formed.

Definition 4.14 (*Well-formed Class Declaration*).

$\boxed{Cls\ \mathsf{OK}}$ well-formed class declaration

$$\frac{{}^s\Gamma{=}\{\overline{X_k \mapsto {}^sN_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ Cid\texttt{<}\overline{X_k}\texttt{>}, \_\} \quad {}^s\Gamma \vdash \overline{{}^sN_k}\ \mathsf{OK} \quad \texttt{self} \notin \overline{{}^sN_k} \quad {}^s\Gamma \vdash \overline{{}^sT}\ \text{strictly OK} \quad \text{ClassBnds}(\texttt{self}\ C\texttt{<}\overline{{}^sT}\texttt{>}) = \overline{{}^sN'} \quad {}^s\Gamma \vdash \overline{{}^sT} <:_s \overline{{}^sN'} \quad {}^s\Gamma \vdash \overline{fd}\ \mathsf{OK} \quad {}^s\Gamma, Cid \vdash \overline{md}\ \mathsf{OK}}{\texttt{class}\ Cid\texttt{<}\overline{X_k\ \texttt{extends}\ {}^sN_k}\texttt{>}\ \texttt{extends}\ C\texttt{<}\overline{{}^sT}\texttt{>}\ \{\ \overline{fd}\ \overline{md}\ \}\ \mathsf{OK}}\ \text{WFC\_DEF}$$

$$\frac{}{\texttt{class Object}\ \{\}\ \mathsf{OK}}\ \text{WFC\_OBJECT}$$

This definition allows the use of `lost` modifiers in the declaration of the upper bounds of type variables. However, note that such a class can never be instantiated (type rule TR_NEW requires a strictly well-formed type) and also never subclassed (because the instantiation of a superclass does not allow `lost` in the upper bounds).

The `self` modifier has the special meaning of referring to the current object only. Using the `self` modifier in an upper bound type would result in the undesired situation that the supertype of the corresponding type variable contains the `self` modifier, even though the type variable obviously does not contain the `self` modifier. For the soundness of the static-type-to-value judgment (see Definition 3.15) this has to be forbidden.

In Figure 12, we introduced class `C` with a type variable `X` that has `rep Data` as an upper bound. Class C can never be instantiated because the viewpoint-adapted upper bound always results in lost ownership information. However, the subclass `D` can be instantiated. We therefore consider class C well formed, even though the class can never be instantiated.

4.3.5. Well-formed Field Declaration. Field declarations are well formed if their corresponding types are well formed.

Definition 4.15 (*Well-formed Field Declaration*).

$\boxed{{}^s\Gamma \vdash {}^sT\ f;\ \mathsf{OK}}$ well-formed field declaration

$$\frac{{}^s\Gamma \vdash {}^sT\ \ \mathsf{OK}}{{}^s\Gamma \vdash {}^sT\ f;\ \mathsf{OK}}\ \text{WFFD\_DEF}$$

For soundness, the field types only need to be well formed; they do not need to be strictly well formed.

4.3.6. Well-formed Method Declaration. The judgment ${}^s\Gamma, C \vdash md\ \ \mathsf{OK}$ expresses that method md is well formed in type environment ${}^s\Gamma$ and class C. A method declaration md is well formed if (1) the return type, the upper bounds of md's type variables, and md's parameter types are well formed in the type environment that maps md's and ${}^s\Gamma$'s type variables to their upper bounds, as well as `this` and the explicit method parameters to their types. The type of `this` is the enclosing class instantiated with its type variables, $C\texttt{<}\overline{X'_k}\texttt{>}$, with main modifier `self`; (2) the upper bounds must not contain the `self` modifier; (3) the method body, expression e, is well typed with md's return type; and (4) md respects the rules for overriding; see below.

Definition 4.16 (*Well-formed Method Declaration*).

$\boxed{{}^s\Gamma, C \vdash md\ \ \mathsf{OK}}$ well-formed method declaration

$$\frac{\begin{array}{c}{}^s\Gamma = \{\overline{X'_k \mapsto {}^sN'_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \_\} \\ {}^s\Gamma' = \{\overline{X'_k \mapsto {}^sN'_k},\ \overline{X_l \mapsto {}^sN_l}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \overline{pid \mapsto {}^sT_q}\} \\ {}^s\Gamma' \vdash \overline{{}^sN_l}, {}^sT, \overline{{}^sT_q}\ \ \mathsf{OK} \qquad \texttt{self} \notin \overline{{}^sN_l} \\ {}^s\Gamma' \vdash e : {}^sT \qquad C\texttt{<}\overline{X'_k}\texttt{>} \vdash m\ \ \mathsf{OK}\end{array}}{{}^s\Gamma, C \vdash \_\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^sN_l}\texttt{>}\ {}^sT\ m(\overline{{}^sT_q\ pid})\ \{\ e\ \}\ \ \mathsf{OK}}\ \text{WFMD\_DEF}$$

We allow `lost` in the parameter types and the upper bounds of the method type variables; such a method is never callable, as the type rule for method calls TR_CALL forbids the occurrence of `lost` in these types. Also note that `self` is forbidden only in the upper bound types, but is allowed as the main modifier of the return and parameter types. A method with `self` as main modifier of a parameter type is only callable on receiver `this`; `self` as main modifier of the return type will result in `lost` if the method is not called on receiver `this`.

Method m respects the rules for overriding if it does not override a method or if all overridden methods have the identical signatures after substituting type variables of the superclasses by the instantiations given in the subclass (the notation $ms'[\overline{{}^sT}/\overline{X'}]$ is used to apply the substitution to the upper bounds, the return type, and the parameter types in the method signature ms'). Consistent renaming of method type variable identifiers and parameter identifiers is allowed.

For simplicity, we require that overrides do not change the purity of a method, although overriding nonpure methods by pure methods would be safe for the encapsulation system in Section 5. Moreover, parameter and return types are invariant, although contravariant, respectively, covariant changes could be allowed [Cunningham et al. 2008].

Definition 4.17 (*Method Overriding*).

$\boxed{{}^{s}CT \vdash m \ \mathsf{OK}}$ method overriding OK

$$\frac{\forall C'\langle\overline{X'}\rangle.\ \forall\, {}^{s}\overline{T}.\ (C\langle\overline{X}\rangle \sqsubseteq C'\langle{}^{s}\overline{T}\rangle \Longrightarrow C\langle\overline{X}\rangle,\ C'\langle{}^{s}\overline{T},\ \overline{X'}\rangle \vdash m\ \mathsf{OK})}{C\langle\overline{X}\rangle \vdash m\ \mathsf{OK}}\ \textsc{ovr\_def}$$

$\boxed{{}^{s}CT,\ C\langle{}^{s}\overline{T},\ \overline{X}\rangle \vdash m\ \mathsf{OK}}$ method overriding OK auxiliary

$$\frac{\begin{array}{c}\mathrm{MSig}(C,\ m) = ms \qquad \mathrm{MSig}(C',\ m) = ms'_o \\ ms'_o{=}None\ \vee\ (ms'_o{=}ms'\ \wedge\ ms'[{}^{s}\overline{T}/\overline{X'}]{=}ms)\end{array}}{C\langle\overline{X}\rangle,\ C'\langle{}^{s}\overline{T},\ \overline{X'}\rangle \vdash m\ \mathsf{OK}}\ \textsc{ovra\_def}$$

The requirement is expressed by two rules. Rule OVRA_DEF determines the method signatures in classes C and C'. The method signature in the superclass must either be undefined, signified by the value *None*, or must be identical after substitution of the type arguments and possibly renaming of method type variables and parameter identifiers. Rule OVR_DEF quantifies over all superclass instantiations and checks that the methods are consistent by using rule OVRA_DEF.

4.3.7. Well-formed Type Environment. The judgment ${}^{s}\Gamma\ \mathsf{OK}$ expresses that type environment ${}^{s}\Gamma$ is well formed. This is the case if all upper bounds of type variables and the types of method parameters are well formed and `self` does not appear in the upper bounds. Moreover, `this` must be mapped to a nonvariable type with main modifier `self` and using the declared type variables of the class as type arguments.

Definition 4.18 (*Well-formed Type Environment*).

$\boxed{{}^{s}\Gamma\ \mathsf{OK}}$ well-formed static environment

$$\frac{\begin{array}{c}{}^{s}\Gamma{=}\{\overline{X_k \mapsto {}^{s}N_k},\ \overline{X'_l \mapsto {}^{s}N'_l}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\langle\overline{X_k}\rangle,\ \overline{pid \mapsto {}^{s}T_q}\} \\ \mathrm{ClassDom}(C) = \overline{X_k} \qquad \mathrm{ClassBnds}(C) = \overline{{}^{s}N_k} \\ {}^{s}\Gamma \vdash \overline{{}^{s}T_q},\ \overline{{}^{s}N_k},\ \overline{{}^{s}N'_l}\ \mathsf{OK} \qquad \texttt{self} \notin \overline{{}^{s}N_k},\ \overline{{}^{s}N'_l}\end{array}}{{}^{s}\Gamma\ \mathsf{OK}}\ \textsc{swfe\_def}$$

Note that `self` $C\langle\overline{X_k}\rangle$ is well formed, since we check that class C is instantiated with its type variables, that is, $\mathrm{ClassDom}(C) = \overline{X_k}$.

4.3.8. Well-formed Program Declaration. The judgment $\vdash P\ \mathsf{OK}$ expresses that program P is well formed. This holds if all classes in P are well formed, the main class C is a nongeneric class in P, the main expression e is well typed in an environment with `this` as an instance of class C, and where subclassing does not contain cycles.

Definition 4.19 (*Well-formed Program Declaration*).

$\boxed{\vdash P\ \mathsf{OK}}$ well-formed program

$$\frac{\begin{array}{l}\overline{Cls_i\ \mathsf{OK}} \\ \{\emptyset\ ;\ \texttt{this} \mapsto \texttt{self}\ C\langle\rangle\} \vdash \texttt{self}\ C\langle\rangle\ \mathsf{OK} \\ \{\emptyset\ ;\ \texttt{this} \mapsto \texttt{self}\ C\langle\rangle\} \vdash e : \_ \\ \forall C',\ C''.\ \big((C'\langle\_\rangle \sqsubseteq C''\langle\_\rangle\ \wedge\ C''\langle\_\rangle \sqsubseteq C'\langle\_\rangle) \Longrightarrow C'{=}C''\big)\end{array}}{\vdash \overline{Cls_i},\ C,\ e\ \mathsf{OK}}\ \textsc{wfp\_def}$$

4.4. Runtime Well Formedness

This section defines the well-formedness conditions of the runtime system.

4.4.1. Runtime Field Type Look-up. It is convenient to look up the declared type of a field for an object. We determine the supertype that can be assigned to the object at address ι, whose class C declares the field f. Note that there is at most one such class in which the field can be declared.

Definition 4.20 (*Runtime Field Type Look-up*).

$\boxed{\text{FType}(h, \iota, f) = {}^{s}T}$ look up type of field in heap

$$\frac{h \vdash \iota : \_\ C\text{<}\_\text{>} \qquad \text{FType}(C, f) = {}^{s}T}{\text{FType}(h, \iota, f) = {}^{s}T}\ \text{RFT\_DEF}$$

4.4.2. Runtime Upper Bounds Look-up. To look up the runtime types of the upper bounds of a runtime type ${}^{r}T$ from the viewpoint ι, we first determine the static upper bound types and then use simple dynamization to determine the runtime types.

Definition 4.21 (*Runtime Upper Bounds Look-up*).

$\boxed{\text{ClassBnds}(h, \iota, {}^{r}T, \overline{o_\iota}) = \overline{{}^{r}T}}$ upper bounds of type ${}^{r}T$ from viewpoint ι

$$\frac{\text{ClassBnds}(\text{ClassOf}({}^{r}T)) = \overline{{}^{s}N} \qquad \text{sdyn}(\overline{{}^{s}N}, h, \iota, {}^{r}T, \overline{o_\iota}) = \overline{{}^{r}T}}{\text{ClassBnds}(h, \iota, {}^{r}T, \overline{o_\iota}) = \overline{{}^{r}T}}\ \text{RCB\_DEF}$$

We provide a sequence of owner addresses $\overline{o_\iota}$ that is used to substitute `lost` modifiers that might appear in the static upper bounds look-up.

The simple dynamization requires that `self` does not appear in the static type and that all type variables can be substituted by the runtime type; this is always the case for a well-formed class (see Definition 4.14).

4.4.3. Strictly Well-formed Runtime Type. By $h,\ \iota \vdash {}^{o_\iota}\ C\text{<}\overline{{}^{r}T}\text{>}$ strictly OK we denote that the runtime type ${}^{o_\iota}\ C\text{<}\overline{{}^{r}T}\text{>}$ is strictly well formed in heap h with viewpoint address ι. The owner ${}^{o_\iota}$ has to be an address in the heap or one of the special addresses $\texttt{any}_a$ or $\texttt{root}_a$. The type arguments have to be strictly well formed and must be runtime subtypes of the corresponding upper bounds.

Definition 4.22 (*Strictly Well-formed Runtime Type*).

$\boxed{h,\ \iota \vdash {}^{r}T\ \text{ strictly OK}}$ strictly well-formed runtime type ${}^{r}T$

$$\frac{{}^{o_\iota} \in \text{dom}(h) \cup \{\texttt{any}_a, \texttt{root}_a\} \qquad \text{ClassBnds}(h, \iota, {}^{o_\iota}\ C\text{<}\overline{{}^{r}T_k}\text{>}, \emptyset) = \overline{{}^{r}T'_k}}{h,\ \_ \vdash \overline{{}^{r}T_k}\ \text{ strictly OK} \qquad h \vdash \overline{{}^{r}T_k} <: \overline{{}^{r}T'_k}}{h,\ \iota \vdash {}^{o_\iota}\ C\text{<}\overline{{}^{r}T_k}\text{>}\ \text{ strictly OK}}\ \text{SWFRT\_DEF}$$

We call this judgment a strictly well-formed runtime type because conceptually it corresponds to the strictly well-formed static type judgment. We did not find a need to define a weak form of well-formed runtime type.

This judgment uses a viewpoint address to express that the runtime type is well formed for a specific viewpoint address ι. The address ι is used to determine the runtime upper bound types. If the declared upper bounds contain the `rep` modifier, then ι will be used in the runtime upper bounds. This ensures that the `rep` upper bound is interpreted correctly. It is interesting to note that a class with a `rep` upper bound can never be instantiated, only a subclass of it could be instantiated. We never need to check strict runtime well-formedness of such a type, and in our uses the viewpoint address ι can be arbitrary.

The type arguments are also checked to be strictly well formed, but we use different viewpoint addresses $\bar{\iota}$. The type arguments are types that potentially were created in

a different viewpoint, for example, they are the result of substituting type arguments for a type variable. Using different viewpoints for the different type arguments allows for each type argument to be relative to a different point of instantiation.

Note that we use an empty sequence of substitutions for `lost` modifiers in ClassBnds. This forbids occurrences of `lost` in the declared upper bounds of a class. Our type rules ensure that we never use strictly well-formed runtime types for classes that use `lost` in upper bounds, since all corresponding static types are checked for strict well-formedness.

4.4.4. Well-formed Object at an Address. An object at an address ι is well formed in a heap h, denoted $h \vdash \iota$ OK, if the runtime type of the object in the heap is strictly well formed; the root owner $\texttt{root}_a$ is in the set of transitive owners of the object (owners(h, ι) yields the set containing the owner address for ι, owner(h, ι); and all the transitive owners), and for all the fields that are declared in the corresponding class, the field type can be assigned to the field value. By mandating that all objects are (transitively) owned by $\texttt{root}_a$ and because each runtime type has one unique owner address, we ensure that ownership is a tree structure.

Definition 4.23 (*Well-formed Object at an Address*).

$\boxed{h \vdash \iota \ \mathsf{OK}}$ well-formed object at an address

$$\frac{h(\iota)\downarrow_1 =_{\_} C\texttt{<\_>} \qquad h,\ \iota \vdash h(\iota)\downarrow_1 \ \text{strictly OK} \qquad \texttt{root}_a \in \text{owners}(h, \iota) \\ \forall f \in \text{fields}(C).\ \exists\, {}^sT.\ (\text{FType}(h, \iota, f) = {}^sT \ \wedge\ h, \iota \vdash h(\iota.f) :\ {}^sT)}{h \vdash \iota \ \mathsf{OK}}\ \text{WFA\_DEF}$$

This definition allows a field type with the `self` main modifier. The address is well formed, if the corresponding field value is the same address again. Also, field types can contain `lost` modifiers and can reference objects of a suitable type because the static-type-to-value judgment chooses a suitable owner addresses. However, the static type rules forbid that such fields are used in an update, and therefore a well-formed program will never create such a heap.

4.4.5. Well-formed Heap. A heap h is well formed, denoted h OK, if all the addresses in the heap are well formed.

Definition 4.24 (*Well-formed Heap*).

$\boxed{h \ \mathsf{OK}}$ well-formed heap

$$\frac{\forall \iota \in \text{dom}(h).\ h \vdash \iota \ \mathsf{OK}}{h \ \mathsf{OK}}\ \text{WFH\_DEF}$$

4.4.6. Well-formed Environments. We need to express that the runtime information consisting of a heap h and a runtime environment ${}^r\Gamma$ are consistent with the static environment ${}^s\Gamma$, written as $h, {}^r\Gamma : {}^s\Gamma$ OK.

Definition 4.25 (*Well-formed Environments*).

$\boxed{h, {}^r\Gamma : {}^s\Gamma \ \mathsf{OK}}$ runtime and static environments correspond

$$\frac{\begin{array}{l} {}^r\Gamma = \{\overline{X_l \mapsto {}^rT_l}\ ;\ \texttt{this} \mapsto \iota,\ \overline{pid \mapsto v_q}\} \\ {}^s\Gamma = \{\overline{X_l \mapsto {}^sN_l},\ \overline{X'_k \mapsto \_}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>},\ \overline{pid \mapsto {}^sT_q}\} \\ h \ \mathsf{OK} \qquad {}^s\Gamma \ \mathsf{OK} \qquad h,\ \iota \vdash \overline{{}^rT_l} \ \text{strictly OK} \\ \text{dyn}(\overline{{}^sN_l}, h, {}^r\Gamma, \emptyset) = \overline{{}^rT'_l} \qquad h \vdash \overline{{}^rT_l} <: \overline{{}^rT'_l} \\ h, {}^r\Gamma \vdash \iota :\ \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>} \qquad h, {}^r\Gamma \vdash \overline{v_q} :\ \overline{{}^sT_q} \end{array}}{h, {}^r\Gamma : {}^s\Gamma \ \mathsf{OK}}\ \text{WFRSE\_DEF}$$

The runtime environment only contains the method type variables $\overline{X_l}$ with their runtime type arguments $\overline{{}^r T_l}$, whereas the static environment contains the method type variables $\overline{X_l}$ and the class type variables $\overline{X'_k}$ with their respective static upper bound types. The type of the current object `this` has to match with the class type variables $\overline{X'_k}$ and the type of `this` must be assignable to the current object ι. The formal parameter types $\overline{{}^s T_q}$ must be assignable to the argument values $\overline{v_q}$.

The method type arguments $\overline{{}^r T_l}$ must be strictly well-formed runtime types and must be subtypes of the dynamization of the corresponding upper bounds. Note that the static upper bounds of the method type variables $\overline{{}^s N_l}$ are dynamized using dyn, since the $\overline{{}^s N_l}$ can contain other method type variables which need a runtime environment ${}^r\Gamma$ for dynamization and could also come from a supertype of the runtime type of the current object ι; sdyn would not be defined for such upper bounds. Note that we do not need to ensure that the ownership structure induced by the method type arguments and method arguments is correct—in particular, we do not need to ensure that $\texttt{root}_a$ is contained, as we do in Definition 4.24. We ensure that all addresses are contained in the heap, and therefore ensure the well formedness of the ownership structure using Definition 4.24.

An empty substitution for `lost` modifiers is also used; our type rules ensure that the upper bounds of method type variables do not contain `lost` modifiers (see TR_CALL, Definition 4.13; note that a well-formed method might contain `lost` modifiers in upper bounds and parameter types, but such a method will never be callable, and therefore we never need to show a correspondence between the static and runtime environments).

The heap h and the static environment ${}^s\Gamma$ have to be well formed according to their respective well-formedness judgments, see Definition 4.24 and Definition 4.18.

4.5. Properties of the Topological System

This final section presents the main properties of the topological system and outlines their proofs. Additional properties can be found in Appendix A, ACM Digital Library (online only) and the detailed proofs in Section B.1.1 in Appendix B.

4.5.1. Type Safety. Type safety of Generic Universe Types in particular ensures that the static ownership information is correctly reflected in the runtime system, which is expressed by the following theorem. If a well-typed expression e is evaluated in a well-formed environment (including a well-formed heap), then the resulting heap is well formed and e's static type can be assigned to the result of e's evaluation.

THEOREM 4.26 (TYPE SAFETY).

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \qquad h, {}^r\Gamma : {}^s\Gamma \ \ \mathsf{OK} \\ {}^s\Gamma \vdash e : {}^s\boldsymbol{T} \\ {}^r\Gamma \vdash h, e \rightsquigarrow h', v \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} h', {}^r\Gamma : {}^s\Gamma \ \ \mathsf{OK} \\ h', {}^r\Gamma \vdash v : {}^s\boldsymbol{T} \end{array}\right.$$

The proof of Theorem 4.26 runs by rule induction on the operational semantics. Lemma 4.28 is used to prove field reads and method results, whereas Lemma 4.29 is used to prove field updates and method argument-passing.

We omit a proof of progress, since this property is not affected by adding ownership to a Java-like language. The basic proof can easily be adapted from FGJ [Igarashi et al. 2001]. Extensions to include field updates and casts have also been done before [Bierman et al. 2003; Flatt et al. 1999]. Only the additional check of the ownership information in a cast is different from these previous approaches; its treatment is analogous to a standard Java cast.

4.5.2. Validation and Creation of a New Viewpoint. The following judgment is convenient for the formulation of the viewpoint adaptation lemmas. It checks that all type variables that appear in the static type ${}^sT$ are either from the class of ${}^sN$ or from the method type variables $\overline{X_l}$. Also, in some static environment, ${}^sN$ needs to be well formed. This ensures that all type variables are substituted, and therefore that the type after substitution of the class and method type variables is consistent with the new environment. Note that the types ${}^sN$ and ${}^sT$ are not part of the new viewpoint ${}^r\Gamma'$, but for brevity we want to include the check of this common side-condition here. The judgment determines the runtime types $\overline{{}^rT_l}$ for the method type arguments using the original environment ${}^r\Gamma$ and an empty substitution for `lost` modifiers. The new viewpoint ${}^r\Gamma'$ is constructed using the method type arguments, the new current object ι, and arbitrary method arguments.

Definition 4.27 (*Validate and Create a New Viewpoint*).

$$\boxed{h, {}^r\Gamma \vdash {}^sN, {}^sT; (\overline{{}^sT}/\overline{X}, \iota) = {}^r\Gamma'} \quad \text{validate and create new viewpoint } {}^r\Gamma'$$

$$\frac{\begin{array}{c} {}^s\Gamma \vdash {}^sN \ \mathsf{OK} \qquad \text{ClassDom}(\text{ClassOf}({}^sN)) = \overline{X} \qquad \text{free}({}^sT) \subseteq \overline{X}, \ \overline{X_l} \\ \text{dyn}(\overline{{}^sT_l}, h, {}^r\Gamma, \emptyset) = \overline{{}^rT_l} \qquad {}^r\Gamma' = \{\overline{X_l \mapsto {}^rT_l} \ ; \ \texttt{this} \mapsto \iota, \_\} \end{array}}{h, {}^r\Gamma \vdash {}^sN, {}^sT; (\overline{{}^sT_l}/\overline{X_l}, \iota) = {}^r\Gamma'} \ \text{NVP\_DEF}$$

4.5.3. Adaptation from a Viewpoint. The following lemma expresses that viewpoint adaptation from a viewpoint to `this` is correct. Consider the `this` object of a runtime environment ${}^r\Gamma$ and two objects o_1 and o_2. If from the viewpoint `this`, o_1 has the static type ${}^sN$, and from viewpoint o_1, o_2 has the static type ${}^sT$, then from the viewpoint `this`, o_2 has the static type ${}^sT$ adapted from ${}^sN$, ${}^sN \triangleright {}^sT$. The following lemma expresses this property using the address ι and value v of the objects o_1 and o_2, respectively. (Note that v can be the $\texttt{null}_a$ value because every static type (that does not contain `self`) can be assigned to the $\texttt{null}_a$ value.)

LEMMA 4.28 (ADAPTATION FROM A VIEWPOINT).

$$\left.\begin{array}{l} h, {}^r\Gamma \vdash \iota : {}^sN \\ h, {}^r\Gamma \vdash v : {}^sT \\ h, {}^r\Gamma \vdash {}^sN, {}^sT; (\overline{{}^sT}/\overline{X}, \iota) = {}^r\Gamma' \end{array}\right\} \Longrightarrow \exists {}^sT'. \ ({}^sN \triangleright {}^sT)[\overline{{}^sT}/\overline{X}] \ = {}^sT' \wedge h, {}^r\Gamma \vdash v : {}^sT'$$

This lemma justifies the type rule TR_READ and the method result in TR_CALL. Note how we can choose suitable substitutions for `lost` modifiers in the static types, that is, the static type after viewpoint adaptation might contain more `lost` ownership information and suitable runtime types are chosen. The proof runs by induction on the shape of static type ${}^sT$.

4.5.4. Adaptation to a Viewpoint. The following lemma is the converse of the preceding Lemma 4.28. It expresses that viewpoint adaptation from `this` to an object o_1 is correct. If from the viewpoint `this`, o_1 has the static type ${}^sN$, and o_2 has the static type ${}^sN \triangleright {}^sT$, then from viewpoint o_1, o_2 has the static type ${}^sT$. The lemma requires that the adaptation of ${}^sT$ from viewpoint ${}^sN$ does not contain `lost` ownership modifiers, since the lost ownership information cannot be recovered.

LEMMA 4.29 (ADAPTATION TO A VIEWPOINT).

$$\left.\begin{array}{l} h, {}^{r}\Gamma \vdash \iota : {}^{s}N \\ ({}^{s}N \rhd {}^{s}T)[\overline{{}^{s}T}/\overline{X}] \;=\; {}^{s}T' \qquad \texttt{lost} \notin {}^{s}T' \\ h, {}^{r}\Gamma \vdash v : {}^{s}T' \\ h, {}^{r}\Gamma \vdash {}^{s}N, {}^{s}T; (\overline{{}^{s}T}/\overline{X}, \iota) = {}^{r}\Gamma' \end{array}\right\} \Longrightarrow h, {}^{r}\Gamma' \vdash v : {}^{s}T$$

This lemma justifies the type rule TR_WRITE and the requirements for the types of the parameters in TR_CALL. The proof again runs by induction on the shape of static type ${}^{s}T$.

This concludes our discussion of the topological system. In summary, we presented the static and dynamic semantics of the topological system and proved type safety. The next section presents an encapsulation system that builds on top of the topological system.

5. ENCAPSULATION SYSTEM

On top of the topological system that we defined in the previous section, we now define an encapsulation system that enforces the owner-as-modifier discipline. Encapsulation is first defined for expressions and then for methods, classes, and programs. We conclude this section by stating the owner-as-modifier property formally and outlining its proof; the detailed proof can be found in Section B.1.2 in Appendix B (online only).

The owner-as-modifier discipline allows an object o to be referenced from anywhere, but reference chains that do not pass through o's owner must not be used to modify o. This allows owner objects to control state changes of owned objects and supports program verification [Drossopoulou et al. 2008; Müller et al. 2006; Leino and Müller 2004; Müller 2002]. It is also enforced in Spec♯'s dynamic ownership model [Leino and Müller 2004] and Effective Ownership Types [Lu and Potter 2006b]. Note that all references, in particular method parameters and local variables, are subjected to the same restrictions and cannot be used to circumvent the owner-as-modifier discipline.

5.1. Encapsulated Expression

The judgment ${}^{s}\Gamma \vdash e$ enc, given below, expresses that expression e is an encapsulated expression; that is, it is a topologically well-typed expression that constrains the possible field updates and method calls.

Definition 5.1 (*Encapsulated Expression*).

$\boxed{{}^{s}\Gamma \vdash e \text{ enc}}$ encapsulated expression

$$\frac{}{{}^{s}\Gamma \vdash \texttt{null} \text{ enc}}\ \text{E\_NULL} \qquad \frac{{}^{s}\Gamma \vdash x : \_}{{}^{s}\Gamma \vdash x \text{ enc}}\ \text{E\_VAR}$$

$$\frac{{}^{s}\Gamma \vdash \texttt{new}\ {}^{s}T\,\texttt{()} : \_}{{}^{s}\Gamma \vdash \texttt{new}\ {}^{s}T\,\texttt{()} \text{ enc}}\ \text{E\_NEW} \qquad \frac{{}^{s}\Gamma \vdash (\,{}^{s}T\,)\ e : \_ \quad {}^{s}\Gamma \vdash e \text{ enc}}{{}^{s}\Gamma \vdash (\,{}^{s}T\,)\ e \text{ enc}}\ \text{E\_CAST}$$

$$\frac{{}^{s}\Gamma \vdash e_0.f : \_ \quad {}^{s}\Gamma \vdash e_0 \text{ enc}}{{}^{s}\Gamma \vdash e_0.f \text{ enc}}\ \text{E\_READ} \qquad \frac{{}^{s}\Gamma \vdash e_0.f = e_1 : \_ \quad {}^{s}\Gamma \vdash e_0 : {}^{s}N_0 \quad {}^{s}\Gamma \vdash e_0 \text{ enc} \quad {}^{s}\Gamma \vdash e_1 \text{ enc} \quad \text{om}({}^{s}N_0) \in \{\texttt{self}, \texttt{peer}, \texttt{rep}\}}{{}^{s}\Gamma \vdash e_0.f = e_1 \text{ enc}}\ \text{E\_WRITE}$$

$$\frac{{}^{s}\Gamma \vdash e_0.m\texttt{<}\overline{{}^{s}T}\texttt{>}(\overline{e}) : \_ \quad {}^{s}\Gamma \vdash e_0 : {}^{s}N_0 \quad {}^{s}\Gamma \vdash e_0 \text{ enc} \quad {}^{s}\Gamma \vdash \overline{e} \text{ enc} \quad \text{om}({}^{s}N_0) \in \{\texttt{self}, \texttt{peer}, \texttt{rep}\} \ \vee\ \text{MSig}({}^{s}N_0, m, \overline{{}^{s}T}) = \texttt{pure <\_> \_}\ m\,\texttt{(\_)}}{{}^{s}\Gamma \vdash e_0.m\texttt{<}\overline{{}^{s}T}\texttt{>}(\overline{e}) \text{ enc}}\ \text{E\_CALL}$$

To enforce the owner-as-modifier discipline, the update of fields of objects in arbitrary contexts must be forbidden. Therefore, field updates are only allowed if the main modifier of the receiver type is `self`, `peer`, or `rep` (E_WRITE). For a method call, either the main modifier of the receiver type is `self`, `peer`, or `rep`, or the called method is pure (E_CALL). Pure methods can be called on arbitrary receivers because they do not modify existing objects.

The encapsulation judgment prevents method `main` (Figure 4) from updating the field `key` of the object referenced by `n`, since the main modifier of `n` is `any`. The update would preserve the topology, but violate the owner-as-modifier discipline because the object referenced by `n` is in a statically unknown context.

5.2. Pure Expression

To focus on the essentials of the type system, we under-specify what we mean with a pure expression. All we need for the proof of the owner-as-modifier property (Theorem 5.7, given later) is expressed in the following assumption: pure expressions do not modify objects that exist in the prestate of the expression evaluation; formally, we have the following.

ASSUMPTION 5.2 (PURE EXPRESSION).

$$\left.\begin{array}{l} h, {}^{r}\Gamma : {}^{s}\Gamma \ \mathsf{OK} \\ {}^{s}\Gamma \vdash e : \_ \\ {}^{s}\Gamma \vdash e \ \mathsf{pure} \\ {}^{r}\Gamma \vdash h, e \rightsquigarrow h', \_ \end{array}\right\} \Longrightarrow \forall \iota \in \mathrm{dom}(h), f \in h(\iota)\downarrow_2 \,.\, h(\iota.f) = h'(\iota.f)$$

As an example that satisfies this assumption, we give a strict definition of pure expressions [Dietl and Müller 2005]. This definition forbids all field updates, and calls to nonpure methods.

Definition 5.3 (*Strictly Pure Expression*).

$\boxed{{}^{s}\Gamma \vdash e \ \mathsf{strictly\ pure}}$ strictly pure expression

$$\frac{{}^{s}\Gamma \vdash \texttt{null} : \_}{{}^{s}\Gamma \vdash \texttt{null} \ \mathsf{strictly\ pure}}\ \text{SP\_NULL} \qquad \frac{{}^{s}\Gamma \vdash x : \_}{{}^{s}\Gamma \vdash x \ \mathsf{strictly\ pure}}\ \text{SP\_VAR}$$

$$\frac{{}^{s}\Gamma \vdash \texttt{new}\ {}^{s}T\texttt{()} : \_}{{}^{s}\Gamma \vdash \texttt{new}\ {}^{s}T\texttt{()} \ \mathsf{strictly\ pure}}\ \text{SP\_NEW} \qquad \frac{{}^{s}\Gamma \vdash \texttt{(}{}^{s}T\texttt{)}\ e : \_ \quad {}^{s}\Gamma \vdash e \ \mathsf{strictly\ pure}}{{}^{s}\Gamma \vdash \texttt{(}{}^{s}T\texttt{)}\ e \ \mathsf{strictly\ pure}}\ \text{SP\_CAST}$$

$$\frac{{}^{s}\Gamma \vdash e_0.f : \_ \quad {}^{s}\Gamma \vdash e_0 \ \mathsf{strictly\ pure}}{{}^{s}\Gamma \vdash e_0.f \ \mathsf{strictly\ pure}}\ \text{SP\_READ}$$

$$\frac{\begin{array}{l} {}^{s}\Gamma \vdash e_0.m\texttt{<}\overline{{}^{s}T}\texttt{>(}\overline{e}\texttt{)} : \_ \\ {}^{s}\Gamma \vdash e_0 : {}^{s}N_0 \quad {}^{s}\Gamma \vdash e_0 \ \mathsf{strictly\ pure} \quad {}^{s}\Gamma \vdash \overline{e} \ \mathsf{strictly\ pure} \\ \mathrm{MSig}({}^{s}N_0, m, \overline{{}^{s}T}) = \texttt{pure <>}\ \_\ m\texttt{(}\_\texttt{)} \end{array}}{{}^{s}\Gamma \vdash e_0.m\texttt{<}\overline{{}^{s}T}\texttt{>(}\overline{e}\texttt{)} \ \mathsf{strictly\ pure}}\ \text{SP\_CALL}$$

Approaches that allow the modification of newly created objects [Salcianu and Rinard 2005] also fulfill this assumption.

5.3. Encapsulated Method Declaration

For an encapsulated method, we require that for a pure method, the method body is a pure expression and that for a nonpure method, the method body is an encapsulated expression.

Definition 5.4 (*Encapsulated Method Declaration*).

$\boxed{{}^s\Gamma, C \vdash md \ \mathsf{enc}}$ encapsulated method declaration

$$\frac{\begin{array}{l}{}^s\Gamma, C \vdash p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^sN_l}\texttt{>}\ {}^sT\ m(\overline{{}^sT_q\ pid})\ \{\ e\ \}\ \ \mathsf{OK}\\ {}^s\Gamma = \{\overline{X'_k \mapsto {}^sN'_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \_\}\\ {}^s\Gamma' = \{\overline{X'_k \mapsto {}^sN'_k},\ \overline{X_l \mapsto {}^sN_l}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \overline{pid \mapsto {}^sT_q}\}\\ (p{=}\texttt{pure} \Longrightarrow {}^s\Gamma' \vdash e\ \ \mathsf{pure}) \qquad (p{=}\texttt{impure} \Longrightarrow {}^s\Gamma' \vdash e\ \ \mathsf{enc})\end{array}}{{}^s\Gamma, C \vdash p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^sN_l}\texttt{>}\ {}^sT\ m(\overline{{}^sT_q\ pid})\ \{\ e\ \}\ \ \mathsf{enc}}\ \text{EMD\_DEF}$$

The first requirement checks that the method is a topologically well-formed method. We then determine the static environments ${}^s\Gamma$ and ${}^s\Gamma'$ that we need to check the method body expression. The construction of these environments corresponds to the topological well-formed method judgment (Definition 4.16). Finally, a pure method is required to have an expression as method body that fulfills our assumption of a pure expression (Assumption 5.2) and a nonpure method needs a method body that fulfills the encapsulated expression judgment (Definition 5.1).

5.4. Encapsulated Class and Program Declaration

The final two judgments define encapsulated class, *Cls* enc, and encapsulated program, $\vdash P$ enc. They simply propagate the checks to the lower levels.

Definition 5.5 (*Encapsulated Class Declaration*).

$\boxed{Cls\ \mathsf{enc}}$ encapsulated class declaration

$$\frac{\begin{array}{l}\texttt{class}\ Cid\texttt{<}\overline{X_k\ \texttt{extends}\ {}^sN_k}\texttt{>}\ \texttt{extends}\ C\texttt{<}\overline{{}^sT}\texttt{>}\ \{\ \overline{fd}\ \overline{md}\ \}\ \ \mathsf{OK}\\ {}^s\Gamma = \{\overline{X_k \mapsto {}^sN_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ Cid\texttt{<}\overline{X_k}\texttt{>}, \_\} \qquad {}^s\Gamma, Cid \vdash \overline{md}\ \ \mathsf{enc}\end{array}}{\texttt{class}\ Cid\texttt{<}\overline{X_k\ \texttt{extends}\ {}^sN_k}\texttt{>}\ \texttt{extends}\ C\texttt{<}\overline{{}^sT}\texttt{>}\ \{\ \overline{fd}\ \overline{md}\ \}\ \ \mathsf{enc}}\ \text{EC\_DEF}$$

$$\frac{}{\texttt{class Object}\ \{\}\ \ \mathsf{enc}}\ \text{EC\_OBJECT}$$

A class declaration is correctly encapsulated if it fulfills the topological rules of well formedness and all methods are encapsulated in the corresponding environment ${}^s\Gamma$ (the construction of ${}^s\Gamma$ again corresponds to the construction of the topological judgment, this time from Definition 4.14). Class `Object` is always correctly encapsulated.

Definition 5.6 (*Encapsulated Program Declaration*).

$\boxed{\vdash P\ \mathsf{enc}}$ encapsulated program

$$\frac{\begin{array}{l}\vdash \overline{Cls},\ C,\ e\ \ \mathsf{OK}\\ \overline{Cls_k\ \ \mathsf{enc}}\\ \{\emptyset\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<>}\} \vdash e\ \ \mathsf{enc}\end{array}}{\vdash \overline{Cls},\ C,\ e\ \ \mathsf{enc}}\ \text{EP\_DEF}$$

A program is encapsulated if the program is topologically well formed, all classes are correctly encapsulated, and the main expression is correctly encapsulated.

5.5. Examples

5.5.1. Purity Examples. Consider the following examples on method purity. Method `getData` below is pure in both the weak (Assumption 5.2) and strict definitions (Definition 5.3):

```
class C {
  rep Data f;
```

```
  pure rep Data getData() {
    return f;
  }
}
```

The method simply returns a reference to the internal state and does not modify any state.

On the other hand, consider

```
class D {
  pure rep Data computeData() {
    rep Data x = new rep Data();
    x.addInfo(...);
    return x;
  }
}
```

assuming some nonpure method `addInfo` in class `Data`. Method `computeData` is pure according to the weak definition of purity from Assumption 5.2, since it does not modify objects that exist in the prestate of the method call. It creates a new object, modifies only this new object, and finally returns it. However, the strict definition from Definition 5.3 forbids method `computeData` because all nonpure method calls are forbidden.

Finally, the following method is nonpure by both definitions:

```
class E {
  rep Data f;
  rep Data cachedData() {
    if( f==null ) {
      f = new rep Data();
      f.addInfo(...);
    }
    return f;
  }
}
```

The modification of field `f` violates both purity definitions. However, caching and lazy initialization are common in query methods. The current research on observational purity [Cok and Leavens 2008; Naumann 2007; Barnett et al. 2004] tries to remedy this problem, but the relation to the owner-as-modifier discipline has not yet been investigated.

5.5.2. Encapsulation Examples. The following examples illustrate the encapsulation judgments. The code fragment below is well encapsulated:

```
peer Data dp = ...;
dp.addInfo(...);
dp.count = 4;

rep Data dr = ...;
dr.addInfo(...);

any Data da = ...;
any Info ia = da.getInfo();
int count = da.count;
```

Calls of the nonpure method `addInfo` are on receivers with a `peer` or `rep` type; also, the field update is on a receiver with a `peer` type. We have no static knowledge of the ownership of `da`; therefore, only the `pure` method `getInfo` can be called on it, and the field can only be read.

Note that the encapsulation judgment is concerned only with the main modifier of the receiver type. For example, consider

```
peer List<any Data> pla = new peer List<any Data>();
pla.add(new peer Data());
```

This code is well encapsulated. We know that the list referenced by `pla` has the same owner as the current object, and the current object can therefore modify the object referenced by `pla`. We have no static knowledge of the ownership of the type arguments, but this is not needed to enforce the owner-as-modifier discipline.

Static types always represent static approximations of the runtime behavior. Consider

```
any Data d;
if (...) {
  d = new rep Data();
} else {
  d = new peer Data();
}
d.addInfo(...);
```

This code is not well encapsulated. At runtime, variable `d` will always reference an object that is either owned by the current object or by the owner of the current object. However, the static type loses this ownership information and the encapsulation system rejects the call to the nonpure method `addInfo`. The code must be rewritten to retain the static knowledge.

5.6. Properties of the Encapsulation System

The owner-as-modifier discipline is expressed by the following theorem. The evaluation of an encapsulated expression e in an encapsulated program P and a well-formed environment can only modify those objects that are (transitively) owned by the owner of `this`.

THEOREM 5.7 (OWNER-AS-MODIFIER).

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{enc} \\ {}^{s}\Gamma \vdash e \ \ \mathsf{enc} \\ h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ {}^{r}\Gamma \vdash h, e \rightsquigarrow h', \_ \end{array}\right\} \Longrightarrow \begin{array}{l} \forall \iota \in \mathrm{dom}(h)\,.\, \forall f \in \mathrm{dom}(h(\iota)\!\downarrow_2)\,. \\ \quad h(\iota.f) = h'(\iota.f) \ \vee \\ \quad \mathrm{owner}(h, {}^{r}\Gamma(\mathtt{this})) \in \mathrm{owners}(h, \iota) \end{array}$$

The proof of Theorem 5.7 runs by rule induction on the operational semantics. The interesting cases are field updates and calls of nonpure methods. In both cases, the encapsulation rules enforce that the receiver expression does not have the main modifiers `lost` or `any`. That is, the receiver object is owned by `this` or the owner of `this`. The case for pure method calls relies only on Assumption 5.2 and not on a more restrictive definition of purity.

6. RELATED WORK

Early work on object-oriented programming already discussed the problems of object aliasing; for example, see the descriptions of Meyer [1988, 1997]. Guides to secure

programming, for example for Java [Sun Developer Network 2010], also highlight the problems of aliasing.

The "Geneva convention on the treatment of object aliasing" [Hogg et al. 1992] illustrates the problems, and outlines four possible treatments: detection, advertisement, prevention, and control.

The Islands system [Hogg 1991] was the first approach to combat aliasing; however, it has a high annotation overhead. Balloon types [Almeida 1997; 1998] use a type system and static analysis to give strong encapsulation guarantees. Both severely restrict the expressiveness and forbid many useful programs.

We structure the rest of this section as follows. In Section 6.1 we discuss the relation to other ownership type systems, in Section 6.2 we give an overview of the work on the Universe type system; and in Section 6.3 we discuss type systems that support read-only references and immutable objects. Finally, in Section 6.4 we briefly discuss object-oriented verification.

6.1. Ownership Type Systems

6.1.1. Ownership Types. Flexible alias protection [Noble et al. 1998] presents a mode system and discusses its use to protect against the negative effects of aliasing. Clarke et al. [1998] and Clarke [2001] developed an ownership type system for flexible alias protection. It enforces the owner-as-dominator property for references stored in fields as well as in local variables. To compare the owner-as-dominator and owner-as-modifier properties, we want to distinguish three kinds of references: (1) references to peers or representation objects; (2) references to objects in enclosing contexts; and (3) arbitrary other objects. Both encapsulation systems support references of the first kind. In owner-as-dominator systems, context parameters express role separation and allow an object, o, to reference objects in ancestor contexts of the context that contains o. These references of the second kind violate neither the owner-as-dominator nor the owner-as-modifier property. Still, we require references to ancestor contexts to be `any` to prevent methods from modifying objects in ancestor contexts because such modifications complicate verification [Müller et al. 2003]. References of the third kind are permitted by the owner-as-modifier property (typed as `any`), but forbidden by the owner-as-dominator property. Even though our encapsulation system is more permissive than owner-as-dominator for references of kind 3, the invariant enforced by our encapsulation system is not strictly weaker than the containment invariant of ownership types because of the restricted handling of references to enclosing contexts,

Context parameters allow a fine-grained specification of the ownership relationship. In contrast, the combination of type parameters and the `any` modifier allow GUT to choose between parameterizing the class and using an abstraction of the ownership. The abstract `any` types can replace context parameters in many situations, impose less annotation overhead, and lead to programs that are easier to reason about. Using type parameters allows us to parameterize a class by both ownership and class information. For many examples, we believe that the type parameters found in GUT will be expressive enough to model the desired ownership structures. Furthermore, type parameters and `any` references allow multiple objects to reference one representation, which is not supported by the owner-as-dominator model used in ownership types. However, such nonowning references to a representation are used in common implementations such as iterators or shared data structures. Both encapsulation models have important applications and drawbacks that need to be considered [Boyland 2005].

In ownership types, the static visibility function SV is used to protect `rep` references from exposure. However, it forbids all access to representation from non-owners. In contrast, the viewpoint adaptation function used in GUT (see Section 4.1) will introduce `lost` ownership information if the exact ownership relation cannot be expressed.

This still allows limited access to the representation of other objects. The substitution of context parameters also roughly corresponds to viewpoint adaptation in Universe types, which adapts ownership information and replaces type arguments for type parameters. Clarke's Ph.D. dissertation [Clarke 2001] gives a detailed development of an object calculus with ownership types and proves a containment invariant.

For a detailed comparison of parametric ownership type systems to nongeneric Universe types, see Section 6.2.3 and Cameron and Dietl [2009]. Extending this work to GUT is future work.

Clarke and Drossopoulou [2002] extended the original ownership type system to support inheritance. Their type system is ownership-parametric, and enforces a slightly weaker form of the owner-as-dominator property by not restricting the references that are computed as intermediate results during expression evaluation. However, this relaxation is not sufficient to handle common patterns such as lists with external iterators. Based on their type system, Clarke and Drossopoulou present an effects system and use it to reason about aliasing and noninterference.

Multiple Ownership for Java-like Objects (MOJO) [Cameron et al. 2007] is an ownership type system that enforces a more flexible topology, supporting more than one owner per object and path types. The type system does not enforce an encapsulation topology. The wildcard owner "?" provides ownership abstraction similar to `any` references in Universe types. This system is only parametric in ownership contexts, not in types.

6.1.2. SafeJava. SafeJava [Boyapati 2004; Boyapati et al. 2003] is very similar to ownership types, but enforces an encapsulation discipline that is slightly less restrictive than owner-as-dominator: An object and all associated instances of inner classes can access a common representation. For instance, iterators can be implemented as inner class of the collection, and therefore directly reference the collection's representation. However, more general forms of sharing are not supported. SafeJava is more flexible than ownership types, but the Universe type system is both syntactically simpler and more permissive. SafeJava has been applied to prevent data races and deadlocks [Boyapati et al. 2002; Boyapati et al. 2003].

Boyapati et al. [2003] present a space-efficient implementation of downcasts in SafeJava. Their implementation inspects each class, C, to determine whether downcasts for C objects potentially require dynamic ownership information. If not, ownership information is not stored for C objects. "Anonymous owners," marked as "-" in class declarations, are used to mark owner parameters that are not used in the class body and do not need runtime representation. This optimization does not work for the Universe type system, where `any` references to objects of any class can be cast to `peer` or `rep` references, and therefore, objects of every class potentially need runtime ownership information.

SafeJava [Boyapati 2004] supports type parameters and ownership parameters independently, but does not integrate both forms of parametricity. This leads to significant annotation overhead. Also, the combination with type parameterization is not formalized. No implementation is available.

6.1.3. Ownership Domains. Ownership domains [Aldrich and Chambers 2004] support a further relaxation of the owner-as-dominator discipline. Contexts can be structured into several domains. Domains can be declared public, which permits reference chains to objects in the public domain that do not pass through their owner. Programmers can control whether objects in different domains can reference each other. For instance, iterators in a public domain of a collection are accessible for clients of the collection. They can be allowed to reference the representation of the collection stored in another

domain. However, the use of public domains and linking of domains can lead to complex systems [Nägeli 2006].

Ownership domains have been used to visualize software architectures [Abi-Antoun and Aldrich 2007a, 2009]. They have also been encoded in Java 5 annotations [Abi-Antoun and Aldrich 2007b]; however, the limited capabilities of Java 5 annotations required a complex encoding.

The concept of ownership domains is powerful and allows many forms of sharing. The link declarations allow programmers to declare the intended sharing in the system, whereas our system enforces a uniform encapsulation discipline for all objects. However, the link declaration further increases the annotation overhead of ownership systems with context parameters.

Ownership Domains [Aldrich and Chambers 2004] combine type parameters and domain parameters into a single parameter space, and thereby reduce the annotation overhead of earlier type parametric ownership systems. However, type parameters are not covered by their formalization. Ownership Domains are integrated in the ArchJava compiler [Aldrich 2003].

6.1.4. Systems by Lu and Potter. The Acyclic Region Type System (ARTS) [Lu and Potter 2005] separates the heap into regions and ensures that reference cycles can only occur within a region. The core language does not use ownership; it provides a strong encapsulation discipline that prohibits common patterns.

Effective ownership types [Lu and Potter 2006b] enforce the owner-as-modifier discipline. They allow a method to modify objects in ancestor contexts of its receiver, and are thus slightly less restrictive than our encapsulation system. This extra flexibility requires context parameters and effective owner declarations for methods, which leads to a significant annotation overhead. Effective ownership types do not separate the topology from the encapsulation system.

Variant ownership types [Lu and Potter 2006a] support both ownership and accessibility modifiers, allowing a fine-grained access scheme. Context variance allows us to abstract over ownership information.

None of the three systems is type parametric and no implementation is available.

6.1.5. Ownership Generic Java. Ownership Generic Java (OGJ) [Potanin et al. 2006; Potanin 2007] allows programmers to attach ownership information through type parameters. OGJ enforces the same owner-as-dominator discipline as the original ownership types. It piggybacks ownership information on type parameters. In particular, each class C has a type parameter to encode the owner of a C object. This encoding allows OGJ to use a slight adaptation of the normal Java type rules to also check ownership, which makes the formalization very elegant. Similarly to OGJ, the Generic Confinement system [Potanin et al. 2004; Potanin 2007] encodes package-level ownership on top of a generic type system. WOGJ [Cameron and Noble 2009] presents an encoding of ownership on top of a generic type system that supports wildcards and requires less change to the underlying type system than OGJ.

We believe that adapting OGJ to separate the topological system from the encapsulation system or to support the owner-as-modifier discipline is not easily accomplished. We need to loosen up the static ownership information by allowing certain references to point to objects in any context. In OGJ, the subtype relation between any types and other types would require covariant subtyping—for instance, that `Node<This>` is a subtype of `Node<Any>`. In OGJ, there is no covariant subtyping, because the underlying Java (or C♯) type system is nonvariant. Therefore, piggybacking ownership on the standard Java type system is not possible in the presence of `any` ownership.

6.1.6. Existential Types. Higher-order functional ownership by Krishnaswami and Aldrich [2005] allows the abstraction of ownership information similar to `any` references.

Existential owners for ownership types [Wrigstad and Clarke 2007] provides a mechanism that allows ownership types to support some downcasts without requiring a runtime representation of ownership. This system does not provide the full flexibility of `any` references.

Jo∃ [Cameron and Drossopoulou 2009; Cameron 2009] combines the theory on existential types with a parametric ownership type system. Ownership information is passed as additional type parameters, and existential types can be used to allow subtype variance. $\text{Jo}\exists_{deep}$ provides optional enforcement of the owner-as-dominator discipline. The Jo∃ system provides theoretical underpinnings and builds a theoretically sound basis. It does not provide a practical language design and no language implementation.

We discussed the relationship between a subset of Jo∃, called $\text{Jo}\exists^{-}$, and the nongeneric Universe type system [Cameron and Dietl 2009]; see our discussion in Section 6.2.3. Analyzing the correspondence between GUT and Jo∃ will provide interesting future work.

A formalization of wildcards [Cameron et al. 2008] uses existential quantification to model Java wildcards. It could also provide insights into how to model `lost` and `any` in future systems.

6.1.7. Other Ownership Type Systems. Confined types [Bokowski and Vitek 1999] have been designed for the development of secure systems and guarantee that objects of a confined type cannot be referenced in code declared outside the confining package. Thus, they support encapsulation on the package level, rather than the object level. A confinement system has also been used to ensure correct behavior of Enterprise Java Beans [Clarke et al. 2003].

Banerjee and Naumann use ownership to prove a representation independence result for object-oriented programs [Banerjee and Naumann 2002; 2004]. They show their main result for a simplistic ownership system that enforces the owner-as-dominator discipline for references from stack and heap locations. Their system requires that, for a given pair of classes C, D, all instances of D are owned by some instance of C. This is clearly too restrictive for many implementations.

The work on Simple Loose Ownership Domains and Boxes [Schäfer and Poetzsch-Heffter 2007, 2008] provide a model of encapsulation that is based on Ownership Domains [Aldrich and Chambers 2004], but allows "loose" references to representation domains, abstracting multiple domains with a single type. It was also adapted to active objects in CoBoxes [Schäfer et al. 2008]; a compiler for JCoBox, the realization of CoBoxes for Java, is also available [Schäfer 2008].

Pedigree types [Liu and Smith 2008] provide additional ownership modifiers that allow a finer description of ownership relations, similar to path types supported in other systems. They also present an interesting inference system.

Ownership has received considerable attention for real-time and concurrent applications—for example, the work on SafeJava [Boyapati et al. 2002, 2003; Boyapati 2004] mentioned above; scoped types for real-time applications [Andrea et al. 2006]; the use for components and process calculi [Hirschkoff et al. 2005]; the use of an ownership topology for concurrency [Cunningham et al. 2007]; the use of a dynamic ownership model for concurrency in Spec♯ [Jacobs et al. 2005]; and an ownership system for object race detection [von Praun and Gross 2001].

The work on gradual encapsulation and decapsulation [Herrmann 2008] in the context of ObjectTeams [Herrmann et al. 2009] provides interesting discussions.

6.2. Universe Type System

The original design goals of the Universe type system, according to Müller and Poetzsch-Heffter [1999], were

(1) to have simple semantics;
(2) be easy to apply;
(3) be statically checkable;
(4) guarantee an invariant that is strong enough for modular reasoning; and
(5) be flexible enough for many useful programming patterns.

These goals were also our guiding principles for the development of Generic Universe Types.

In the following we first discuss different formalizations of the Universe type system, we then discuss the relation to dependent type systems, and finally compare Universe types to parametric ownership systems that support existential quantification.

6.2.1. Formalizations. The Universe type system (UTS) was first introduced by Müller and Poetzsch-Heffter [1999, 2000]. The early syntax was different to the current syntax, and Type Universes were later removed. These early formalizations already use a "type combinator" to adapt a declared type to a changed viewpoint. The syntax using the three ownership modifiers `peer`, `rep`, and `readonly` was first used by Müller and Poetzsch-Heffter [2001]. The UTS was used by Müller to develop a modular verification methodology for object-oriented programs [Müller 2001, 2002].

In Dietl and Müller [2005] we present the integration of the Universe type system into the Java Modeling Language (JML). We implemented a type checker, runtime checks, and bytecode information for the UTS in the JML tools.

Universe Types with Transfer [Müller and Rudich 2007] realize ownership transfer for nongeneric Universe types.

All formalizations mentioned before do not distinguish between enforcing the ownership topology and the owner-as-modifier discipline. However, for some applications of ownership, for example, for concurrency [Cunningham et al. 2007], enforcing the heap topology only is enough; enforcing the owner-as-modifier discipline is an unneeded restriction.

The separation of the topological structure and the encapsulation discipline was developed previously for UT [Cunningham et al. 2008]. That work renamed the modifier `readonly` to `any`, because this ownership modifier now signifies only that the object is in an arbitrary ownership context, and not necessarily that it is used for reading only. The `any` modifier is a "don't care" modifier, which expresses that, for the annotated reference, the ownership of the referenced object is of no concern. That work also introduced the new ownership modifier `lost`, which signifies that static ownership information for a reference is not available. In contrast to `any`, `lost` is a "don't know" modifier—from the current viewpoint, we cannot express the ownership relation, but the declared type might have a constraint. Updates of `any` variables are always possible, since the owner of their value is not of interest. Updates of `lost` variables must be forbidden, since the ownership information required for type-safe updates is not statically known. Using `lost` in the nongeneric system allowed the clean separation of the topology from the encapsulation system and simplified the rules and their presentation. The encapsulation system in Cunningham et al. [2008] made the distinction between an encapsulation property and an owner-as-modifier property. In this terminology, our owner-as-modifier, Theorem 5.7, corresponds to the encapsulation property; a corresponding owner-as-modifier property could be proved using a small-step semantics or using execution traces. Klebermaß's master's thesis [Klebermaß 2007] used Isabelle [Nipkow et al. 2002] to formalize UT and prove type soundness for a Java subset based

on Featherweight Java [Igarashi et al. 2001; Foster and Vytiniotis 2006] and Jinja [Klein and Nipkow 2004, 2006]. To our knowledge, this is the first soundness proof of an ownership type system in a theorem prover.

A previous version of Generic Universe Types [Dietl et al. 2007, 2006] always enforces the owner-as-modifier discipline. A particularly interesting aspect of that version is how generics and ownership can be combined in the presence of an `any` modifier, in particular, how a restricted form of ownership covariance can be permitted without runtime checks. For this ownership covariance to be safe, it recursively changes the enclosing main modifiers of a type to `any` and relies on the enforcement of the owner-as-modifier discipline, which always forbids modifications through `any` references. This resulted in the rules for subtyping and viewpoint adaptation to be tightly coupled.

In our current work, we allow ownership covariance using the `lost` modifier, and thereby cleanly separate the topological system, including subtyping and viewpoint adaptation, from the enforcement of an encapsulation system. In type arguments, we use the `lost` modifier to express that ownership information was lost locally, and do not need to change enclosing ownership modifiers. We also retain more static knowledge about the ownership structure because we do not need to change enclosing ownership modifiers to `any`. For a detailed comparison of the two systems, and for a discussion of how arrays, exceptions, and static fields and methods are handled in GUT, see the thesis of the first author [Dietl 2009], in particular Sections 2.5 and 2.6; here we outline an example:

```
class Client {
  peer Map<rep ID, any Data> m;
}
...
peer Client c;
```

In our current system, the type of `c.m` is `peer Map<lost ID, any Data>`. However, in our earlier work [Dietl et al. 2007], the type of `c.m` is less precise, namely it is `any Map<any ID, any Data>`. This is so because in Dietl et al. [2007], we had no way of making the distinction between a "don't care" (`any`) and a "don't know" (`lost`) owner. If `c.m` had the type `peer Map<any ID, any Data>`, then the update `c.m.put(k, v)` would be legal for any ownership of `k` and `v`, even though the update is not guaranteed to preserve the required heap topology for `c.m`. For this reason, and to avoid such harmful updates, in Dietl et al. [2007] we changed the enclosing modifiers to `any`, and in the process lost topological information, which through the introduction of `lost`, we could retain in our current work.

Erasure and Expansion of Type Arguments. It is possible to erase a GUT program into a Universe Types program without generics [Cunningham et al. 2008], using casts. The interpretation of casts and type arguments is the same: both are from the viewpoint of the current object. Therefore, the casts inserted into the erased program use the same types that are used as type arguments. The simple example from Figure 14 can be erased to the UT program in Figure 15.

However, we cannot translate a GUT program into a UT program by "expansion," that is, generating a new class for each type where the type arguments are substituted for the type variables. In our example, this expansion would produce the code in Figure 16.

This does not work because the declared types will be adapted from the type of the receiver to the current viewpoint. In the example above, this viewpoint adaptation will result in lost ownership information, which will forbid the call of method `push` and return less precise information for the result of `pop`. For some combinations of receiver

```
class Stack<X extends any Object> {
    void push(X p) {...}
    X pop() {...}
}

class C {
    void m() {
        peer Stack<rep Data> x = new peer Stack<rep Data>();
        x.push( new rep Data() );
        rep Data y = x.pop();
    }
}
```

Fig. 14. A simple stack and client.

```
class Stack {
  void push(any Object p) {...}
  any Object pop() {...}
}

class C {
  void m() {
    peer Stack x = new peer Stack();
    x.push( new rep Data() );
    rep Data y = (rep Data) x.pop();
  }
}
```

Fig. 15. Erasure of the stack from Figure 14 to a nongeneric Universe Types program.

```
class StackrepData {
  void push(rep Data p) {...}
  rep Data pop() {...}
}

class C {
  void m() {
    peer StackrepData x = new peer StackrepData();
    x.push( new rep Data() ); // error
    rep Data y = x.pop();  // error
  }
}
```

Fig. 16. The stack from Figure 14 after "expansion" of the type argument.

and parameter types this expansion is possible, but in general, as illustrated by the example above, such an expansion is not possible.

We could adapt the runtime model to be closer to Java: keeping the runtime type variables in the environment and in the heap is not done in Java. If we forbid creation of type variables and casts with type arguments, as is done in Java, they are not needed. We could still store the main owner in the heap in order for downcasts from `any` to `rep` or `peer` to be checked at runtime.

6.2.2. Dependent Types. Ownership type systems structure the heap and enforce restrictions on the behavior of a program. Virtual classes [Madsen and Møller-Pedersen 1989; Ernst 1999, 2001] express dependencies between objects. Similar to virtual methods, a class A can declare a dependent class B by nesting class B within the definition

of *A*. Virtual classes can be overridden in subclasses, and the runtime type of an object determines the concrete definition of a virtual class. Recent work [Odersky et al. 2003; Ernst et al. 2006; Clarke et al. 2007; Igarashi and Viroli 2007; Saito et al. 2007] formalized and extended virtual classes. All of these systems have in common that dependency is expressed by nesting of classes.

Dependent classes [Gasiunas et al. 2007a, 2007b] are a generalization of virtual class systems that allows one class to depend on multiple objects. Dependency is expressed by explicit declaration of the depended-upon objects as class parameters. This allows us to declare dependencies independently of the nesting of classes, which increases the expressive power and reduces the coupling between classes. Even more generally, constrained types [Nystrom et al. 2008; Charles et al. 2005] can express multiple constraints and is parametric in the constraint system.

Ownership type systems, in particular the Universe type system and Ownership Domains [Aldrich and Chambers 2004], can be expressed using dependent classes [Dietl and Müller 2008]. The ownership structure is made explicit by adding a dependency on an immutable `owner` field, similarly to the dynamic encoding in Dietl and Müller [2005]. The ownership modifiers of the UTS can be directly expressed as constraints on this `owner` field. Even more fine-grained relationships can be expressed, for example, that an object is in an unknown context, but in the same context as some other object.

In the dependent classes [Gasiunas et al. 2007a] syntax, we can declare the following class:

```
class OwnedObject( owner: Object ) {}
```

We can then express the topology of the following simple program that uses Universe types:

```
class C {...}
class D {
  rep C f = new rep C();
}
```

The field `f` may reference only `C` objects that have the current `D` object as owner. We can express this in the dependent classes encoding as follows:

```
class C( Object owner ) extends OwnedObject {...}
class D( Object owner ) extends OwnedObject {
  C(owner: this) f = new C(owner=this);
}
```

In the above program, we make explicit that the owner of the referenced `C` object is the current `this` object. Similarly, a `peer` reference is translated into the constraint that the referenced object has the same owner as the current object.

6.2.3. Parametric Ownership with Existential Types. Parametric ownership types [Noble et al. 1998; Clarke et al. 1998; Clarke 2001; Clarke and Drossopoulou 2002; Potanin et al. 2006] discussed above and the nongeneric Universe type system [Dietl and Müller 2005] are two ownership type systems that describe an ownership hierarchy and statically check that this hierarchy is maintained. They both provide (different) encapsulation disciplines.

Ownership types can describe fine-grained heap topologies, whereas Universe types are more flexible and easier to use. No direct encoding of one type system in the other has been possible: the abstraction provided by `any` references in the Universe type system could not be modeled with parametric ownership types.

Recently, parametric ownership has been extended with existential quantification of contexts [Cameron and Drossopoulou 2009; Cameron 2009]. This extension, called Jo∃,

provides the possibility to abstract from concrete ownership information—similarly to any references in Universe types.

We show in Cameron and Dietl [2009] that the descriptive parts of the Universe type system [Cunningham et al. 2008] and a variant of Jo∃, which we call Jo∃⁻, are equivalent. In Jo∃⁻ we use a single owner parameter that corresponds to the single ownership modifier of the UTS. Note that full Jo∃ allows multiple owner parameters, and is thus more expressive than nongeneric Universe types.

We formalize this correspondence as encodings between the two systems. We have proved that the encodings from Universe types to Jo∃⁻ and from Jo∃⁻ to Universe types are sound; thus, we have shown that the two systems are equivalent with respect to type checking. As an intermediate step in the encoding we give an alternative formalization of the UTS, which is closer to the underlying existential types model.

Consider the program P_1 using Universe types:

```
class C {
  peer Object f1;
  any Object f2;

  void m(any C x) {
    this.f1 = new peer Object(); // 1: OK
    x.f1 = new peer Object();    // 2: error
    x.f2 = new peer Object();    // 3: OK
  }
}
```

and the program P_2 using Jo∃⁻ types

```
class C<owner> {
  Object<owner> f1;
  ∃o. Object<o> f2;

  void m(∃o. C<o> x) {
    this.f1 = new Object<owner>(); // 1: OK
    x.f1 = new Object<owner>();    // 2: error
    x.f2 = new Object<owner>();    // 3: OK
  }
}
```

These two programs are equivalent, that is, both describe the same topology and type checking in both systems rejects expression 2.

In P_1, the field update `x.f1` in expression 2 is forbidden, as the viewpoint adaptation `peer Object` from `any` results in `lost Object` and `lost` is forbidden in the adapted field type. On the other hand, the field update `x.f2` in expression 3 is allowed, as `any Object` from `any` results in `any Object` and the right-hand side is a correct subtype.

In expression 2 of P_2, the type of `x` must be unpacked before it can be used. Therefore, the field type lookup for field `f1` in type `C<o1>` is performed, where `o1` is a fresh context variable. This lookup gives the type `Object<o1>`. There is no subtype relationship between `Object<owner>` and `Object<o1>` because their parameters do not match and subtyping of unquantified types is invariant.

In expression 3, the lookup for field `f2` in type `C<o1>` results in `∃o. Object<o>`, which is a supertype of `Object<owner>`, due to the variance of existential types, and the assignment is allowed.

The investigation of encoding ownership type systems in dependent types and the relationship between Universe types and a parametric ownership system with

existential quantification gave us valuable insights into these type systems and promising ideas for future research for combining these systems with GUT.

6.3. Read-only References and Immutability

Skoglund [2002, 2003] as well as Birka and Ernst [2004] present type systems for readonly types that are similar to readonly references when the owner-as-modifier discipline is enforced. Birka and Ernst's type system is more flexible than ours, as it allows us to exclude certain fields or objects from the immutable state. Neither Skoglund nor Birka and Ernst combined readonly types with ownership. The combination with ownership gives more context to decide when a downcast to a read-write reference is valid.

Our readonly types leave the owner of an object unspecified. Whenever precise information about the owner is needed, a downcast with a dynamic type check is used. This approach is similar to soft typing [Cartwright and Fagan 1991; Cartwright and Felleisen 1996], where a compiler does not reject programs that contain potential type errors, but introduces runtime checks around suspect statements. In soft typing, these runtime checks are added automatically by the compiler, whereas we require programmers to introduce casts manually.

Immutability Generic Java (IGJ) [Zibin et al. 2007] allows the covariant change of type arguments of readonly and immutable types. The unsoundness of the covariant change is prevented by forbidding modifications through readonly and immutable types. However, the erased-signature rule is needed to ensure that overriding methods cannot introduce an unsoundness; this rule also requires that type variables that appear in the parameter types of pure methods are marked as nonvariant. In contrast, in GUT the loss of ownership information in a covariant argument change is detected for the method call, and we can therefore safely allow more methods.

Unique references and linear types [Boyland 2001; Boyland et al. 2001; Fähndrich and DeLine 2002; Wadler 1990] can be used for a very restrictive form of alias control. For ownership type systems, a weaker form of uniqueness [Clarke and Wrigstad 2003; Wrigstad 2006] is sufficient to enable ownership transfer. Universe Types with Transfer [Müller and Rudich 2007] realize ownership transfer for nongeneric Universe types.

Ensuring that an object is immutable cannot be checked by the GUT type system. Immutability is present in other type systems (e.g., Javari [Tschantz and Ernst 2005]; Jimuva [Haack et al. 2007]; IGJ [Zibin et al. 2007]; and Joe_3 [Östlund et al. 2008]) and in a dynamic encoding of ownership called frozen objects [Leino et al. 2008]. Investigating the combination with Universe types is interesting future work.

6.4. Object-Oriented Verification

Work on the formal verification of object-oriented programs is of particular interest to this thesis. Verification tools for the Java Modeling Language (JML) [Burdy et al. 2003; Burdy et al. 2003; Cheon 2003; Cheon and Leavens 2002; Cok and Kiniry 2004; Flanagan et al. 2002; Jacobs 2004; Jacobs and Poll 2001] and Spec♯ [Barnett et al. 2004] all have to deal with the problems of aliasing for program verification. Preventing representation exposure [Detlefs et al. 1998] and cross-type aliasing [Dhara and Leavens 2001] also provide a discussion of aliasing problems and possible solutions.

A recent technical report provides an overview and comparison of different behavioral interface specification languages [Hatcliff et al. 2009]. Specification and verification challenges were also recently summarized [Leavens et al. 2007]. A framework for verification techniques for object invariants [Drossopoulou et al. 2008] describes different techniques using seven parameters.

Müller's thesis [Müller 2002] provides the basis for the owner-as-modifier discipline that we enforce in GUT. Ownership was also used to reason about frame properties in

JML [Müller et al. 2003]. Spec♯'s [Barnett et al. 2004] dynamic ownership model [Leino and Müller 2004] (also called the Boogie methodology) for reasoning about invariants is based on a dynamic ownership encoding similar to the one described in Dietl and Müller [2005]. In this methodology, any reference can be used to modify an object, provided that all transitive owners of this object are made mutable by applying a special unpack operation. In practice, this requirement is typically met by following the owner-as-modifier discipline: the owner unpacks itself before initiating the modification of an owned object. The Boogie methodology supports ownership transfer. It is future work to investigate how the topological system of GUT can be used together with the Boogie methodology. Reasoning using ownership [Müller 2007] is a very promising approach, but some remaining obstacles need to be overcome.

Lu et al. [2007] divide their system into validity invariants and the validity effect to describe which objects need to be revalidated. The system is based on effective and variant ownership types [Lu and Potter 2006b, 2006a].

Poetsch-Heffter and Schäfer [2006, 2007] describe the modular specification of components based on their boxes type system [Schäfer and Poetzsch-Heffter 2007, 2008; Schäfer et al. 2008].

Alternative approaches to the formal verification of systems are, for example, separation logic [Reynolds 2002], regional logic [Banerjee et al. 2008], and (implicit) dynamic frames [Kassios 2006; Smans et al. 2009]. It will be interesting future work to investigate the relationships between these different approaches and ownership type systems.

7. CONCLUSION

Encapsulation of object structures is an important challenge in programming language research with many practical applications. In this article, we described the design and formalization of Generic Universe Types (GUT), a lightweight ownership type system that combines type genericity and ownership. Our system distinguishes itself from earlier work in three ways. First, we separate the topological structure from the enforcement of the encapsulation discipline. This separation allows the separate development and reuse of these parts. Second, we formally integrate ownership and type genericity, which allowed us to study subtle interactions such as the difference between the existential modifiers `any` and `lost`. Third, our encapsulation system enforces the owner-as-modifier discipline, which is more permissive than the discipline used in other ownership systems, but provides guarantees that are strong enough for many applications of ownership, in particular, verification of object invariants. The GUT system is rigorously formalized, proved sound, and implemented in the JML compiler.

As future work, we plan to further improve the expressiveness and tool support for GUT. The expressiveness would greatly benefit from combining GUT with existing solutions for ownership transfer [Müller and Rudich 2007] and object immutability [Leino et al. 2008]. Both extensions seem possible, but we need to investigate how we can preserve the modularity of the formalization. Universe Types with Transfer rely on both the topological and the encapsulation system, which we want to keep separate, and immutability should ideally be formalized as an extra system complementing the existing two. The tool support could be improved by extending our static inference tool for nongeneric Universe Types [Dietl et al. 2009] to GUT. This inference tool uses a SAT solver to resolve the constraints on ownership modifiers coming from the type system. The main challenge there is how to encode the constraints of the GUT, which are considerably more complex than those of the nongeneric system.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We are grateful to Alexander J. Summers, David Cunningham, and the anonymous ECOOP '07 and FOOL/-WOOD '07 reviewers for their helpful comments.

REFERENCES

ABI-ANTOUN, M. AND ALDRICH, J. 2007a. Compile-time views of execution structure based on ownership. In *Proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*.

ABI-ANTOUN, M. AND ALDRICH, J. 2007b. Ownership domains in the real world. In *Proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*.

ABI-ANTOUN, M. AND ALDRICH, J. 2009. Static extraction of sound hierarchical runtime object graphs. In *Proceedings of the International Workshop on Types in Language Design and Implementation (TLDI)*.

ÁDÁM DARVAS AND LEINO, K. R. M. 2007. Practical reasoning about invocations and implementations of pure methods. In *Fundamental Approaches to Software Engineering* (FASE), M.B. Dwyer and A. Lopes Eds., Lecture Notes in Computer Science, vol. 4422. Springer, Berlin, 336–351.

ALDRICH, J. 2003. Using types to enforce architectural structure. Ph.D. dissertation, University of Washington.

ALDRICH, J. AND CHAMBERS, C. 2004. Ownership domains: Separating aliasing policy from mechanism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 3086, Springer, Berlin, 1–25.

ALDRICH, J., KOSTADINOV, V., AND CHAMBERS, C. 2002. Alias annotations for program understanding. In *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 311–330.

ALMEIDA, P. S. 1997. Balloon types: Controlling sharing of state in data types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 1241, Springer, Berlin, 32–59.

ALMEIDA, P. S. 1998. Control of object sharing in programming languages. Ph.D. dissertation, Imperial College London.

ANDREA, C., COADY, Y., GIBBS, C., NOBLE, J., VITEK, J., AND ZHAO, T. 2006. Scoped types and aspects for real-time systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 4067, Springer, Berlin, 124–147.

BANERJEE, A. AND NAUMANN, D. A. 2002. Representation independence, confinement, and access control. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 166–177.

BANERJEE, A. AND NAUMANN, D. A. 2004. Ownership confinement ensures representation independence for object-oriented programs. Tech. rep. 2004-14, Stevens Institute of Technology.

BANERJEE, A., NAUMANN, D. A., AND ROSENBERG, S. 2008. Regional logic for local reasoning about global invariants. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 5142. Springer, Berlin, 387–411.

BARNETT, M., DELINE, R., FÄHNDRICH, M., LEINO, K. R. M., AND SCHULTE, W. 2004. Verification of object-oriented programs with invariants. *J. Object Technol. 3*, 6, 27–56.

BARNETT, M., NAUMANN, D. A., SCHULTE, W., AND SUN, Q. 2004. 99.44% pure: Useful abstractions in specification. In *Proceedings of the Formal Techniques for Java-like Programs Workshop (FTfJP)*. 51–60.

BIERMAN, G. M., PARKINSON, M. J., AND PITTS, A. M. 2003. An imperative core calculus for Java and Java with effects. Tech. rep. 563, Computer Laboratory, University of Cambridge.

BIRKA, A. AND ERNST, M. D. 2004. A practical type system and language for reference immutability. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, New York,

BOKOWSKI, B. AND VITEK, J. 1999. Confined types. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM New York, 82–96.

BOYAPATI, C. 2004. SafeJava: A unified type system for safe programming. Ph.D. dissertation, MIT, Cambridge, MA.

BOYAPATI, C., LEE, R., AND RINARD, M. 2002. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM New York, 211–230.

BOYAPATI, C., LEE, R., AND RINARD, M. 2003. Safe runtime downcasts with ownership types. In *Proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming* (IWACO).

BOYAPATI, C., LISKOV, B., AND SHRIRA, L. 2003. Ownership types for object encapsulation. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 213–223.

BOYAPATI, C., SALCIANU, A., JR., AND RINARD, M. 2003. Ownership types for safe region-based memory management in real-time Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, 324–337.

BOYLAND, J. 2001. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exper. 31*, 6, 533–553.

BOYLAND, J. 2005. Why we should not add read-only to Java (yet). In *Proceedings of Formal Techniques for Java-like Programs Workshop* (FTfJP).

BOYLAND, J., NOBLE, J., AND RETERT, W. 2001. Capabilities for aliasing: A generalization of uniqueness and read-only. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 2072. Springer, Berlin, 2–27.

BURDY, L., CHEON, Y., COK, D., ERNST, M., KINIRY, J., LEAVENS, G. T., LEINO, K. R. M., AND POLL, E. 2003. An overview of JML tools and applications. In *Proceedings of the Formal Methods for Industrial Critical Systems (FMICS)*. Elsevier, Amsterdam, 73–89.

BURDY, L., REQUET, A., AND LANET, J.-L. 2003. Java applet correctness: A developer-oriented approach. In *Proceedings of the International Symposium on Formal Methods Europe (FME)*. Lecture Notes in Computer Science, vol. 2805, Springer, Berlin, 422–439.

CAMERON, N. 2009. Existential types for variance | Java wildcards and ownership types. Ph.D. dissertation, Imperial College London.

CAMERON, N. AND DIETL, W. 2009. Comparing universes and existential ownership types. In *Proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*.

CAMERON, N. AND DROSSOPOULOU, S. 2009. Existential quantification for variant ownership. In *Proceedings of the European Symposium on Programming Languages and Systems (ESOP)*.

CAMERON, N., DROSSOPOULOU, S., AND ERNST, E. 2008. A model for Java with wildcards. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 5142, Springer, Berlin, 2–26.

CAMERON, N., DROSSOPOULOU, S., NOBLE, J., AND SMITH, M. 2007. Multiple ownership. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 441–460.

CAMERON, N. AND NOBLE, J. 2009. OGJ gone wild. In *Proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*.

Cartwright, R. and Fagan, M. 1991. Soft typing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, 278–292.

CARTWRIGHT, R. AND FELLEISEN, M. 1996. Program verification through soft typing. *ACM Comput. Surv. 28*, 2, 349–351.

CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 519–538.

CHEON, Y. 2003. A runtime assertion checker for the Java modeling language. Ph.D. dissertation, Iowa State University.

CHEON, Y. AND LEAVENS, G. T. 2002. A runtime assertion checker for the Java Modeling Language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. CSREA Press, 322–328.

CLARKE, D. G. 2001. Object ownership and containment. Ph.D. dissertation, University of New South Wales.

CLARKE, D. G. AND DROSSOPOULOU, S. 2002. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 292–310.

CLARKE, D. G., DROSSOPOULOU, S., NOBLE, J., AND WRIGSTAD, T. 2007. Tribe: A simple virtual class calculus. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*.

CLARKE, D. G., POTTER, J. M., AND NOBLE, J. 1998. Ownership types for flexible alias protection. In *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, New York.

CLARKE, D. G., RICHMOND, M., AND NOBLE, J. 2003. Saving the world from bad beans: deployment time confinement checking. In *Proceedings of the Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*. ACM New York, 374–387.

Clarke, D. G. and Wrigstad, T. 2003. External uniqueness is unique enough. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Lecture Notes in Computer Science, vol. 2743. Springer, Berlin, 176–200.

Cok, D. R. and Kiniry, J. 2004. ESC/Java2: Uniting ESC/Java and JML. In *Proceedings of the Conference on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS).* Lecture Notes in Computer Science, vol. 3362, Springer, Berlin, 108–128.

Cok, D. R. and Leavens, G. T. 2008. Extensions of the theory of observational purity and a practical design for JML. In *Proceedings of the Conference on Specification and Verification of Component-Based Systems (SAVCBS).* 43–50.

Cunningham, D., Dietl, W., Drossopoulou, S., Francalanza, A., Müller, P., and Summers, A. J. 2008. Universe types for topology and encapsulation. In *Proceedings of the Conference on Formal Methods for Components and Objects (FMCO).* Lecture Notes in Computer Science, vol. 5382, Springer, Berlin, 72–112.

Cunningham, D., Drossopoulou, S., and Eisenbach, S. 2007. Universe types for race safety. In *Proceedings of the Conference on Verification and Analysis of Multi-threaded Java-like Programs (VAMP).* 20–51.

Detlefs, D. L., Leino, K. R. M., and Nelson, G. 1998. Wrestling with rep exposure. SRC Res. rep. 156, Digital Systems Research Center.

Dhara, K. K. and Leavens, G. T. 2001. Preventing cross-type aliasing for more practical reasoning. Tech. rep. 01-02a, Department of Computer Science, Iowa State University.

Dietl, W. 2009. Universe Types: Topology, encapsulation, genericity, and tools. Ph.D. dissertation, ETH 18522, Department of Computer Science, ETH Zurich.

Dietl, W., Drossopoulou, S., and Müller, P. 2006. Formalization of generic Universe Types. Tech. rep. 532, Department of Computer Science, ETH Zurich.

Dietl, W., Drossopoulou, S., and Müller, P. 2007. Generic Universe Types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Lecture Notes in Computer Science, vol. 4609, Springer, Berlin, 28–53.

Dietl, W., Ernst, M. D., and Müller, P. 2009. Tunable Universe Type inference. Tech. rep. 659, Department of Computer Science, ETH Zurich. Dec.

Dietl, W. and Müller, P. 2004. Exceptions in ownership type systems. In *Proceedings of the Formal Techniques for Java-like Programs Workshop (FTfJP).* 49–54.

Dietl, W. and Müller, P. 2005. Universes: Lightweight ownership for JML. *J. Object Technol. 4,* 8, 5–32.

Dietl, W. and Müller, P. 2008. Ownership type systems and dependent classes. In *Proceedings of the Conference on Foundations of Object-Oriented Languages (FOOL).*

Drossopoulou, S., Francalanza, A., Müller, P., and Summers, A. J. 2008. A unified framework for verification techniques for object invariants. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Lecture Notes in Computer Science, vol. 5142, Springer, Berlin, 412–437.

Emir, B., Kennedy, A. J., Russo, C., and Yu, D. 2006. Variance and generalized constraints for C] generics. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Lecture Notes in Computer Science, vol. 4067, Springer, Berlin, 279–303.

Ernst, E. 1999. gbeta—A language with virtual attributes, block structure, and propagating, dynamic inheritance. Ph.D. dissertation, Department of Computer Science, University of Aarhus, Arhus, Denmark.

Ernst, E. 2001. Family polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Lecture Notes in Computer Science, vol. 2072, Springer, Berlin, 303–326.

Ernst, E., Ostermann, K., and Cook, W. R. 2006. A virtual class calculus. In *Proceedings of the Symposium on Principles of Programming Languages (POPL).* ACM, New York, 270–282.

Ernst, M. D. 2008. Type annotations specification (JSR 308). http://types.cs.washington.edu/jsr308/.

Fähndrich, M. and DeLine, R. 2002. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI).* ACM, New York, 13–24.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. 2002. Extended static checking for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI).* ACM, New York, 234–245.

Flatt, M., Krishnamurthi, S., and Felleisen, M. 1999. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java,* Lecture Notes in Computer Science, vol. 1523. Springer, Berlin, 241–269.

FOSTER, J. N. AND VYTINIOTIS, D. 2006. A theory of Featherweight Java in Isabelle/HOL. In *Archive of Formal Proofs*, G. Klein et al. Eds. http://afp.sf.net.

GASIUNAS, V., MEZINI, M., AND OSTERMANN, K. 2007a. Dependent classes. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 133–152.

GASIUNAS, V., MEZINI, M., AND OSTERMANN, K. 2007b. vcn—A calculus for multidimensional virtual classes. www.st.informatik.tu-darmstadt.de/static/pages/projects/mvc/index.html.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2005. *The Java Language Specification* 3rd Ed., Addison-Wesley, Reading, MA.

HAACK, C., POLL, E., SCHÄFER, J., AND SCHUBERT, A. 2007. Immutable objects for a Java-like language. In *Programming Languages and Systems*, Lecture Notes in Computer Science, vol. 4421, Springer, Berlin, 347–362.

HATCLIFF, J., LEAVENS, G. T., LEINO, K. R. M., MÜLLER, P., AND PARKINSON, M. 2009. Behavioral interface specification languages. Tech. rep. CS-TR-09-01, School of EECS, University of Central Florida.

HERRMANN, S. 2008. Gradual encapsulation. *J. Object Technol.7*, 9, 47–68.

HERRMANN, S., HUNDT, C., AND MOSCONI, M. 2009. ObjectTeams/Java language definition version 1.2 (OTJLD). http://www.objectteams.org/def/1.2/.

HIRSCHKOFF, D., HIRSCHOWITZ, T., POUS, D., SCHMITT, A., AND STEFANI, J.-B. 2005. Component oriented programming with sharing: Containment is not ownership. In *Generative Programming and Component Engineering*, Lecture Notes in Computer Science, vol. 3676, Springer, Berlin, 389–404.

HOGG, J. 1991. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 271–285.

HOGG, J., LEA, D., WILLS, A., CHAMPEAUX, D. D., AND HOLT, R. 1992. The Geneva convention on the treatment of object aliasing. *OOPS Messenger, Report on ECOOP'91 Workshop W3 3,* 2, 11–16.

IGARASHI, A., PIERCE, B. C., AND WADLER, P. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst. 23*, 3, 396–450.

IGARASHI, A. AND VIROLI, M. 2007. Variant path types for scalable extensibility. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 113–132.

JACOBS, B. 2004. Weakest precondition reasoning for Java programs with JML annotations. *J. Logic Algebraic Program. 58*, 61–88.

JACOBS, B., PIESSENS, F., LEINO, K. R. M., AND SCHULTE, W. 2005. Safe concurrency for aggregate objects with invariants. In *Proceedings of the Conference on Software Engineering and Formal Methods (SEFM)*. IEEE, Los Alamitos, CA, 137–147.

JACOBS, B. AND POLL, E. 2001. A logic for the Java modeling language JML. In *Proceedings of the Conference on Fundamental Approaches to Software Engineering (FASE)*. Lecture Notes in Computer Science, vol. 2029, Springer, Berlin, 284–299.

KASSIOS, I. T. 2006. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Proceedings of the Symposium on Formal Methods (FM)*.

KENNEDY, A. AND SYME, D. 2001. Design and implementation of generics for the .NET common language Runtime. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, 1–12.

KLEBERMA, M. 2007. An Isabelle formalization of the Universe type system. M.S. thesis, Technical University Munich and ETH Zurich.

KLEIN, G. AND NIPKOW, T. 2004. A machine-checked model for a Java-like language, virtual machine and compiler. Tech. rep. 0400001T.1, National ICT Australia. http://www4.informatik.tumuenchen.de/~nipkow/pubs/Jinja/.

KLEIN, G. AND NIPKOW, T. 2006. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Program. Lang. Syst. 28*, 4, 619–695.

KRISHNASWAMI, N. AND ALDRICH, J. 2005. Permission-based ownership: encapsulating state in higher-order typed languages. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, 96–106.

LEAVENS, G. T., LEINO, K. R. M., AND MÜLLER, P. 2007. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects Comput.* 19, 2, 159–189.

LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D., MÜLLER, P., KINIRY, J., CHALIN, P., ZIMMERMAN, D. M., AND DIETL, W. 2008. *JML Reference Manual.* http://www.jm/specs.org.

LEINO, K. R. M. AND MÜLLER, P. 2004. Object invariants in dynamic contexts. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 3086, Springer, Berlin, 491–516.

LEINO, K. R. M., MÜLLER, P., AND WALLENBURG, A. 2008. Flexible immutability with frozen objects. In *Proceedings of the Verified Software: Theories, Tools, and Experiments (VSTTE)*. Lecture Notes in Computer Science, vol. 5295, Springer, Berlin, 192–208.

LIU, Y. D. AND SMITH, S. 2008. Pedigree types. In *Proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*.

LU, Y. AND POTTER, J. 2005. A type system for reachability and acyclicity. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 3586, Springer, Berlin, 479–503.

LU, Y. AND POTTER, J. 2006a. On ownership and accessibility. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 4067, Springer, Berlin, 99–123.

LU, Y. AND POTTER, J. 2006b. Protecting representation with effect encapsulation. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 359–371.

LU, Y., POTTER, J., AND XUE, J. 2007. Validity invariants and effects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 4609, Springer, Berlin, 202–226.

MADSEN, O. L. AND MOLLER-PEDERSEN, B. 1989. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York.

MEYER, B. 1988. *Object-Oriented Software Construction* 1st Ed., Prentice Hall, Englewood Cliffs, NJ.

MEYER, B. 1997. *Object-Oriented Software Construction* 2nd Ed., Prentice Hall, Englewood Cliffs, NJ.

MÜLLER, P. 2002. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science, vol. 2262, Springer, Berlin.

MÜLLER, P. 2007. Reasoning about object structures using ownership. In *Proceedings of the Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*. Lecture Notes in Computer Science, vol. 4171, Springer, Berlin, 93–104.

MÜLLER, P. AND POETZSCH-HEFFTER, A. 1999. Universes: A type system for controlling representation exposure. In *Programming Languages and Fundamentals of Programming*, 131–140.

MÜLLER, P. AND POETZSCH-HEFFTER, A. 2000. A type system for controlling representation exposure in Java. Tech. rep. 269, Fernuniversitat Hagen.

MÜLLER, P. AND POETZSCH-HEFFTER, A. 2001. Universes: A type system for alias and dependency control. Tech. rep. 279, Fernuniversitat Hagen.

MÜLLER, P., POETZSCH-HEFFTER, A., AND LEAVENS, G. T. 2003. Modular specification of frame properties in JML. *Concurrency Comput.: Pract. Exper. 15*, 117–154.

MÜLLER, P., POETZSCH-HEFFTER, A., AND LEAVENS, G. T. 2006. Modular invariants for layered object structures. *Sci. Comput. Program. 62*, 253–286.

MÜLLER, P. AND RUDICH, A. 2007. Ownership transfer in Universe Types. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA). ACM, New York, 461–478.

NÄGELI, S. 2006. Ownership in design patterns. M.S. thesis, Department of Computer Science, ETH Zurich.

NAUMANN, D. A. 2007. Observational purity and encapsulation. *Theor. Comput. Sci*. 376, 205–224.

NIPKOW, T., PAULSON, L., AND WENZEL, M. 2002. *Isabelle/HOL– A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, vol. 2283, Springer, Berlin.

NOBLE, J., VITEK, J., AND POTTER, J. M. 1998. Flexible alias protection. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 1445, Springer, Berlin.

NYSTROM, N., SARASWAT, V., PALSBERG, J., AND GROTHOFF, C. 2008. Constrained types for objectoriented languages. In *Proceedings of the Conference on Object-oriented Programming Systems Languages, and Applications (OOPSLA)*. ACM, New York, 457–474.

ODERSKY, M. 2008. The Scala Language specification, version 2.7. Programming Methods Laboratory, EPFL, Switzerland.

ODERSKY, M., CREMET, V., ROCKL, C., AND ZENGER, M. 2003. A nominal theory of objects with dependent types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 2743, Springer, Berlin, 201–224.

Östlund, J., Wrigstad, T., Clarke, D. G., and Akerblom, B. 2008. Ownership, uniqueness, and immutability. In *Objects, Components, Models and Patterns*, Lecture Notes in Business Information Processing, vol. 11, Springer, Berlin, 178–197.

Papi, M. M., Ali, M., Jr., Perkins, J. H., and Ernst, M. D. 2008. Practical pluggable types for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 201–212.

Poetzsch-Heffter, A. and Schäfer, J. 2006. Modular specification of encapsulated object oriented components. In *Formal Methods for Components and Objects*, Lecture Notes in Computer Science, vol. 4111, Springer, Berlin, 313–341.

Poetzsch-Heffter, A. and Schäfer, J. 2007. A representation-independent behavioral semantics for object-oriented components. In *Formal Methods for Open Object-Based Distributed Systems*, Lecture Notes in Computer Science, vol. 4468. Springer, Berlin, 157–173.

Potanin, A. 2007. Generic ownership: A practical approach to ownership and confinement in object-oriented programming languages. Ph.D. dissertation, Victoria University of Wellington.

Potanin, A., Noble, J., Clarke, D. G., and Biddle, R. 2006. Generic ownership for generic Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 311–324.

Potanin, A., Noble, J., Clarke, D. G., and Biddle, R. 2004. Featherweight generic confinement. In *Proceedings of the Conference on Foundations of Object-Oriented Languages (FOOL)*.

Reynolds, J. C. 2002. Separation logic: A logic for shared mutable data structures. *Logic Comput. Sci. 55*.

Saito, C., Igarashi, A., and Viroli, M. 2007. Lightweight family polymorphism. *J. Funct. Program. 18*, 285–331.

Salcianu, A. and Rinard, M. C. 2005. Purity and side-effect analysis for Java programs. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Lecture Notes in Computer Science, vol. 3385, Springer, Berlin, 199–215.

Schäfer, J. 2008. JCoBox compiler. http://softech.informatik.uni-kl.de/Homepage/JCoBox.

Schäfer, J. and Poetzsch-Heffter, A. 2007. A parameterized type system for simple loose ownership domains. *J. Object Technol. 6*, 5, 71–100.

Schäfer, J. and Poetzsch-Heffter, A. 2008. CoBoxes: Unifying active objects and structured heaps. In *Formal Methods for Open Object-Based Distributed Systems*, Lecture Notes in Computer Science, vol. 5051, Springer, Berlin, 201–219.

Schäfer, J., Reitz, M., Gaillourdet, J.-M., and Poetzsch-Heffter, A. 2008. Linking programs to architectures: An object-oriented hierarchical software model based on boxes. In *The Common Component Modeling Example*, Lecture Notes in Computer Science, vol. 5153. Springer, Berlin, 238–266.

Sewell, P., Nardelli, F. Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., and Strnisa, R. 2007. Ott: Effective tool support for the working semanticist. In *Proceedings of the International Conference on Functional Programming (ICFP)*. ACM, New York, 1–12.

Skoglund, M. 2002. Sharing objects by read-only references. In *Proceedings of the Conference on Algebraic Methodology and Software Technology (AMAST)*. Lecture Notes in Computer Science, vol. 2422, Springer, Berlin, 457–472.

Skoglund, M. 2003. Investigating object-oriented encapsulation in theory and practice. Ph.D. dissertation, Stockholm University/Royal Institute of Technology.

Smans, J., Jacobs, B., and Piessens, F. 2009. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, Springer, Berlin, 148–172.

Sun Developer Network. 2010. Secure coding guidelines for the Java programming language, version 3.0. http://java.sun.com/security/seccodeguide.html.

Tschantz, M. S. and Ernst, M. D. 2005. Javari: Adding reference immutability to Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM Press, 211–230.

von Praun, C. and Gross, T. R. 2001. Object race detection. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 70–82.

Wadler, P. 1990. Linear types can change the world! In *Proceedings of the Conference on Programming Concepts and Methods (PROCOMET)*. M. Broy and C. B. Jones Eds.

Wrigstad, T. 2006. Ownership-based alias management. Ph.D. dissertation, Royal Institute of Technology, Sweden.

Wrigstad, T. and Clarke, D. G. 2007. Existential owners for ownership types. *J. Object Technol. 6*, 4, 141–159.

ZIBIN, Y., POTANIN, A., ALI, M., ARTZI, S., KIEZUN, A., AND ERNST, M. D. 2007. Object and reference immutability using Java generics. In *Proceedings of the European Software Engineering Conference/- Foundations of Software Engineering (ESEC/FSE)*.

ZIBIN, Y., POTANIN, A., LI, P., ALI, M., AND ERNST, M. D. 2010. Ownership and immutability in generic Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, New York.

Received June 2010; revised July 2011; accepted September 2011