JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 4, May–June 2007

Existential Owners for Ownership Types

Tobias Wrigstad, Stockholm University/Royal Institute of Technology, Sweden
Dave Clarke, CWI, The Netherlands

This paper describes a lightweight approach to adding run-time checked downcasts to a language in the presence of ownership types without the need for a run-time representation of owners. Previous systems [6] have required owners of objects to be tracked and matched at run-time which is costly in terms of memory and performance. Our proposal avoids run-time overhead to deal with owners and also extends the expressiveness of ownership-based systems enough to handle the Java equals idiom for structural equality comparison. The price is that it is sometimes impossible to downcast a type into a type that can be statically aliased. Our proposal is completely orthogonal and combinable with previous work.

1 INTRODUCTION

Recent years have seen an intense research on various ways of strengthening encapsulation in object-oriented systems. One notable such system is Clarke et al.'s Ownership Types [10, 8]. In ownership types, the objects' nesting relations are captured in the program code by means of ownership annotations on types—every object is owned by another object in the system, and the nesting relations control how an object may be referenced from other objects. Somewhat syntactically burdensome, ownership types has a strong containment invariant that facilitates alias management [13, 8] and makes the job of developing lightweight specification languages easier [19]. Various forms of ownership types have also been used to enforce architectural restrictions [2], to avoid deadlocks and data-races [5], and for reasoning about disjointness of effects and absence of aliasing [9]. Ownership types have also been combined with uniqueness to enable uniquely referenced aggregates [4, 3, 12] and to overcome a problem with abstraction introduced by unique pointers in an object-oriented setting [12, 21].

Most statically typed class-based object-oriented languages include a downcasting operation. The downcasting operation is checked at run-time. To this end, the run-time system keeps track of an object's class and matches it against the class cast to modulo subtyping. An invalid cast generally raises some kind of error or exception, preventing the use of the object as the wrong type, which preserves type-safety.

As ownership types extend types and class declarations with ownership and nesting information it complicates the matter of downcasting. When casting from one type to another, not only must classes and the inheritance tree be considered, but also the owners associated with the types. Boyapati, Lee and Rinard [6] describe a way of allowing downcasts in the presence of ownership in Mini Safe Concurrent Java. Mini Safe Concurrent Java handles downcasting in the obvious fashion by keeping track of owners at run-time. To enable the dynamic check, each object holds references to its owners, which is a pretty severe overhead.

In this paper, we present an alternate way of implementing downcasts in the presence of ownership influenced by the concept of existential types [20]. Our proposal voids the need for keeping owners around at run-time and adds additional flexibility to an ownership types system that allows the implementation of "standard methods" that generally rely on downcasting, such as the Java [15] *equals* method, an idiom that was not previously possible to express in deep ownership-types systems. The downside of our proposal is the impossibility in most cases to downcast into an existing type that can be statically aliased.

Outline This paper continues as follows: in Section 2, we present deep ownership types in additional detail to set the scene for our discussion. In Section 3, we discuss the downcasts of Mini Safe Concurrent Java and show why they are not powerful enough to express Java equals methods. Section 4 introduces our own proposal, named Existential Downcasts, and discusses its strengths and weaknesses in relation to Mini Safe Concurrent Java. Section 5 sketches a formalisation of Existential Downcasts in the context of the Joline programming language, and Section 6 concludes.

2 OWNERSHIP TYPES

Deep ownership types [10] enforces the conceptual structural property that an object's representation (the subobjects conceptually belonging to it) is *inside* its *enclosing* object and cannot be exported outside it. This is called the *owners-as-dominators property* and gives strong encapsulation.

Ownership types introduces the notion of objects as *owners* and representation objects are *owned* by their enclosing objects. Classes are parameterised by ownership information and types are formed by instantiating these parameters with actual owners.

Deep ownership enables constraining of the object graph by capturing the nesting of objects in the types in a simple and elegant manner. Representation objects are ordered *inside* their enclosing objects, and references to representation are not allowed to flow to the outside world. As the nesting is captured in the class declarations, the nesting information is propagated through the program, giving control over the global structure of the object graph. By prohibiting references owned by some owner x to flow to objects outside x, a strong, but flexible, containment invariant is achieved that cannot be circumvented as in shallow ownership, causing indirect representation exposure [8].

An owner can be seen as the permission to reference a group of objects. Types are formed from classes and *owner parameters*, which serve as placeholders to give permissions to reference external objects. Thus, a type does not denote a set of possible instances of a class, but a set of possible instances of a class *with a particular set of permissions to reference other objects.* Types with different owner parameters are not compatible and references of types with different owners cannot be aliases [9].

Annotating Classes with Ownership Information

To be able to statically control ownership nesting, class declarations are extended with annotations which describe the relations between owner parameters to thread nesting information through the program. As an example, the class `StringList` below takes two owner parameters where the first parameter is nested inside the second and both are outside the owner `owner`.

```
class StringList< owner1 outside owner, owner2 outside owner1 >
```

Ownership parameters of a class must always be outside the implicit owner `owner`, the owner of the instance. This is key to avoiding the problem of indirect representation exposure in shallow ownership [21].

The omnipresent owner `world` is *outside* all owners, is visible in all scopes and denotes global objects, accessible everywhere in the object graph. In addition to `owner` and `world`, a class body has access to the owners declared in its class header, and the owner `this`, which denotes itself and is *inside* `owner`.

For subclassing, the extends clause is extended with a mapping relation from the owners of the subclass to those of the superclass. The number of owner parameters in a subclass may grow or shrink depending on the relations between the owners in the superclass.

```
class List< some outside owner > extends Object { ... }
class StringList< owner1 outside owner, owner2 outside owner1 >
                                   extends List< owner2 > { ... }
```

The owner must be preserved through subtyping as it acts as the permission governing access to the object. Preserving it by subsumption is a key to achieving a sound system [8]. In the example above, `StringList`'s second owner parameter will be mapped to `some` when viewed as its superclass. This is valid if `owner2` is outside `owner`, a requirement derived from `List`'s class header. That the requirement is fulfilled can be derived from the class header of `StringList` as nesting is a transitive relation (`owner2` is outside `owner1` and `owner1` is outside `owner` implies `owner2` is outside `owner`).

Forming Types

Types have the following syntax:

```
owner:ClassName<owner_1, ..., owner_n>
```

where `owner` is the owner of the type, `ClassName` is a class name and `owner`$_{1..n}$ are visible permissions (in the current context) to reference external objects.

When forming types from a class, the nesting requirements of the owners in the class' header must be satisfied by the owners in scope to which they are bound. The object graph is well-constructed with respect to the nesting requirements specified in the classes.

Below are a few examples of types with ownership using the recent class declaration examples.

```
class StringList<owner1 outside owner, owner2 outside owner1>
                                extends List<owner2>
{
  this:StringList<owner, owner1>  representation;
  owner:StringList<owner1, owner2>  outgoing;
  // owner1:StringList<this, owner2>  illegal;
  // world:StringList<this, owner2>  alsoIllegal;

  owner:List<owner2>  super = outgoing;
}
```

In the code example above, the third and fourth variable declarations are illegal as the owners in scope do not satisfy the requirements of the class header of `StringList` as `this` is inside both `world` and `owner1`.

The variable `representation` holds a representation object with permission to reference back to the object itself as it is parameterised with `owner`.

The variable `outgoing` has the same type as the current instance. As the type of `outgoing` does not have `this` as its owner, it cannot point to a representation object as all representation objects are owned by `this` and types with different owners are not assignment compatible. Furthermore, the type is not given explicit permission to reference `this` (`this` is not an owner in the type). This means that references to representation cannot be stored in an object referenced by the variable. Such violations are statically checkable and will not compile. Actually, having this in the type of `outgoing` would not be valid as that would give an external object permission to reference the current representation. This is prevented by the restriction that the owner must be inside all other owner parameters.

Last, `super` demonstrates subsumption—`owner:List<owner2>` is a super type of `owner: StringList<owner1, owner2>` and we can therefore assign from `outgoing` to `super`. Note the remapping and hiding of owner parameters as discussed on the previous page.

Owner-Polymorphic Methods

Owner-polymorphic methods take owners as parameters, similar to type parameters, and allow code to be reused with different owners [7, 8, 12, 18]. The scope of owner parameters is limited to the current method, and an argument with owner parameters in its type cannot be statically aliased by the receiver (see Wrigstad's dissertation [21] for an extended discussion of this topic, in particular the "hide owner" pattern that lifts this restriction in certain situations). As an illustration, an owner-polymorphic method in the Joline system [12, 21] looks like this:

```
<temp inside world> void method(temp:Blah arg) { ... }
```

Inside the method body the owner `temp` is statically known to be inside the owner `world`. When invoked, the owner argument is supplied at the call-site and can be thought of as giving a temporary permission to the receiver to reference the object (albeit only for dynamic aliases).

Ownership Types in Trouble: The Java Equals Method

In Java, downcasting is frequently used to overcome the shortcomings of the static type system. Prior to Java 5.0, and the introduction of parametrically polymorphic classes, container classes stored data objects as instances of `Object`. As a consequence, type information of an object stored in a container was lost when the object was later retrieved and dynamically checked downcasting was essential to regain the type information.

Even in Java 5.0, downcasts are frequently used. A good example can be seen in the `equals` method declared in `Object`, the superclass of all objects. This method is supposed to be overridden in all classes for which structural equality is sensible. In the Java API, all equals methods have the same signature: `boolean equals(Object other)`. The common implementation of such a method is to check that the argument is of the correct type; if so, cast the argument to the desired type, and then perform equality tests of the contents of the objects.

In a system without the possibility of downcasting implementation of `equals` will break overriding as the signatures of equals methods cannot stay the same:

```
boolean equals(x:List<y> other)
{
  ...// code omitted, no need for casting
}
```

Apart from breaking overriding, the method above has another serious problem: `x` and `y` can only be selected from the owner parameters known to the receiver and the selection of `x`, most likely `owner`, is fixed for life. Comparing different lists in this way is not possible without making the equals method owner-polymorphic, which adds additional syntactic baggage to the implementation:

```
<x inside world, y outside x> boolean equals(x:List<y> other)
{
  ...// code omitted, no need for casting
}
```

but the necessary signature would (still) prevent overriding, which limits flexibility.

In conclusion, in an ownership types system, the implementation of equals methods is complicated by the presence of owner parameters. The methods will most certainly be owner-polymorphic, as they will otherwise not allow structural equality tests of objects with different owners, which is clearly too restrictive.

Even in a system with regular downcasting, implementing equals methods is problematic, which the upcoming section will show.

Note that the problems pointed out there apply to other methods, such as clone, as well. Later sections will revisit this discussion in the presence of downcasting.

3 RELATED WORK

In this section, we discuss previous approaches to downcasting in the presence of ownership. As our focus is on deep ownership, we focus most of our attention on Mini Safe Concurrent Java, as it is a deep ownership system but also briefly cover ArchJava. The systems are very similar, and both fail to deal with implementation of Java equals methods, our driving example.

Downcasting in Mini Safe Concurrent Java

In Mini Safe Concurrent Java, Boyapati, Lee and Rinard [6] propose a way of doing *dynamically checked* downcasts with deep ownership where ownership information is stored at run-time. The implementation is straight-forward: after successful type-checking, a Mini Safe Concurrent Java program can be translated into a normal Java program where each class is extended by a field for each owner used by the class. The Java translation of the class `StringList` on Page 143 would for example have become the fields

```
Object owner1, owner2;
```

to be able to hold references at run-time to the owners so they can be used for checking the validity of downcasts.

The obvious downside of this implementation is the overhead of keeping track of owners at run-time. If a class has four owner parameters, this means four additional pointers to the corresponding actual owners in each instance in a naïve implementation. As the mandatory owner parameter may sometimes not be used internally in a class, but only be used externally to track nesting and manage aliasing, Boyapati et al. allow the programmer to explicitly declare the the owner parameter as *anonymous*. This removes the owner field from the translation, avoiding the need for one owner field in the translation. (Inferring anonymity would require that all classes in an inheritance hierarchy were analysed before compilation.) Anonymous owners somewhat lowers the memory overhead required to do downcasting in Mini Safe Concurrent Java, but the overhead is still high.

How Mini Safe Concurrent Java Handles Equals

Interestingly, the downcasting enabled Mini Safe Concurrent Java does not allow the implementation of Java equals methods. Consider the code for the equals method in a List class below:

```
boolean equals(x:Object other)
{
  x:List<y> list = (x:List<y>) other;
  ...// rest of the code omitted
}
```

Just like in our initial example of equals, the implementation above has two problems—`x` and `y`. Unless we make the method owner-polymorphic, we can only compare ourselves with lists owned by the owner `x`, probably `owner` in a real implementation, which is severly limiting. Making all equals methods polymorphic in the owner parameter of their parameter lifts this restriction, but we must still consider the remaining parameters, in this case `y`. If we must include the remaining owners in the methods' owner parameters to enable downcasting, equals methods in classes with different owner parameters will have different signatures, again breaking overriding. Thus, the possible choices of `y` will be restricted to the set of owners statically accessible to the receiver, and `x`, the single owner parameter. Clearly, this is inflexible and not satisfactory for real-world applications.

Downcasting in ArchJava

The ArchJava language [2, 1] uses ownership annotations as part of a system to enforce architectural constraints in programs. The ArchJava ownership is shallow, meaning that nesting relations between objects are not enforced, which results in a weaker encapsulation than that of deep ownership.

The ArchJava language implements downcast support using a technique similar to that of Mini Safe Concurrent Java. Owners are being stored at run-time, and downcasting can only be made to types using owners in the current scope. The system thus suffer from the same problems as Mini Safe Concurrent Java in terms of being able to encode Java equals methods. However, an upside of the weaker ownership system is that the owner representation overhead is lower than for Mini Safe Concurrent Java.

4 OUR PROPOSAL: EXISTENTIAL DOWNCASTS

The idea behind our proposal is simple: if a Java-style downcast (disregarding ownership) of an object to some class c succeeds, then we should be able to infer the owner parameters necessary to form the new type from c's class header. We call the inferred owners existential owners, and types that use them existential types.

For example, if we can check that the *class* of the object stored in a variable typed `a:Object` is List, we know that its actual type is `a:List<b>` for some owner `b` outside `a`. In our proposal, downcasts will simply add `b` to the current scope, nested outside `a`. Adding owners in this fashion can never enable breaking owners-as-dominators as we must already have permission to reference the object (its owner), and owner parameters of well-formed classes are always outside the owner parameter.

Existential owners completely avoid a run-time owner representation to the price of now being able to downcast a type into a previously existing type. The shortcomings of this approach are explored below.

The syntax of existential downcasting could follow standard:

```
(a:List<b>) x; // x has type a:Object
```

If `x` actually holds a list object, the cast expression would introduce `b` as a new owner (possibly aptly named by the programmer) that is outside `a`, inferred from the header of the `List` class. (If `x` does not contain a list, we would raise a class cast exception as in Java.) The only information necessary at run-time to check this is the class of the object in `x`, which is present in most object-oriented languages, including Smalltalk [14], Java and C# [17].

We define existential owners and existential types thus:

Definition 4.1 (Existential Owner)**.** An existential owner is an owner that is introduced by a type cast.

Existential owners have no statically known relations to other owners, except for the owners in the type cast where they were introduced and the owner parameter of the cast object, which is always known. Existential types are bound to their enclosing scope.

Definition 4.2 (Existential Type)**.** In our setting, an existential type is a type that uses an existential owner for one or more of its owner parameters.

As the owners of an existential type are bound to the enclosing scope, so is the type itself. As existential owners are introduced in the scope of a method, a class definition can never use existential types for fields and thus, references with existential type can never be stored in the fields of the current receiver as the types will never be compatible. For the same reasons, existential types can never appear as parameter types, or return types.

Again, if `x` holds a `List` object, the object must have (at least) two owners, the already known owner `a` and an owner of the data objects in the list, here `b`. As we clearly have permission to reference `a` which is inside all the "non-visible owner parameters" of the object in `x`, we can safely access them without breaking encapsulation.

Owner-Polymorphic Methods + Existentials = Equals

Apart from avoiding run-time owner representations, the existential owners present a solution to the problem of the Java equals method. As we saw in Section 3, combining standard subtype polymorphism and downcasting to allow structural equivalence tests of objects belonging to different representations does not suffice, as the owners in the type cast to must be in the receiver's type or be `this`. This prevents structural equality tests of objects with different owners, which is severly limiting. Using owner-polymorphic methods as discussed in Section 2 to pass owners in to the equals method to overcome this limitation will break overriding as the equals method's signature will vary with the implementing class' owner parameters. As existential owners can be introduced "out of the blue" without having to be explicitly passed in, these problems are now overcome, and combining existential downcasting with owner-polymorphic methods allows us to form a generic signature for the equals method that will work for all cases:

```
<temp inside world> boolean equals(temp:Object other)
```

Inside the equals method, existential downcasting can be used in a straightforward way to allow access to `other`'s complete protocol. An example of this is shown in Figure 1.

As we see it, allowing the downcasting operation to introduce existential owners is a simple way to overcome the (technical) difficulty of deriving what owner parameters should have been passed to the method would the desired type of the argument be known outside, something we expect to be generally impossible. Our solution preserves pure polymorphism for methods without introducing any additional complexities in the system. It also preserves abstraction as it is not visible external to the method how the method will downcast its argument object.

```
// In List class
<temp inside world> boolean equals(temp:Object other)
{
  if (other == this)
    {
      return true;
    }
  if (other != null && other instanceof List)
    {
      // the existential owner 'ex' is introduced here
      let list = (temp:List<ex>) other in
        {
          if (list.length() == this.length())
            {
              for (int i=0; i<this.length(); ++i)
                {
                  if (this.get(i).equals<ex>(list.get(i)) == false)
                    return false;
                }
              return true;
            }
          else
            {
              return false;
            }
        }
    }
  return false;
}
```

Figure 1: Existential downcasting used in implementation of an `equals` method.

Limitations of Existential Downcasting

The price of existential downcasting is being forced to conservatively treat existential owners derived from a cast as different from all other owners in scope, possibly even when the same variable is downcast twice (unless we can clearly see that it has not been modified since the last downcast using program analysis). This is much weaker than Mini Safe Concurrent Java, but fully functional in many situations and does not require burdensome ownership information to be kept at run-time. Interestingly, the existential owners are regular owners that can be used to instantiate objects etc. Thus, we can use existential owners introduced through a downcast to create objects and update the downcast object, for example populate a list. We just need to do it without losing track of what owner belongs to what object, such as within the body of a single method.

As existential types are tied to the scope of the method that creates them, references of existential types cannot be captured in fields of the receiver. To be stored in a field, these types need to be subsumed into a non-existential type, that is, a type where all the existential owners are forgotten. This is possible for all types whose owner is not existential or an owner parameter to a method as all classes are derived from `Object`. Thus, the reference can be saved as being of a more general type, which will require a downcasting operation before each use of the contents of the field as the more specific type.

Downcasting into a Non-Existential Type

An interesting observation of downcasts is that the introduction of existential owners is only necessary when the subclass extends the set of owner parameters of the superclass. Casting from a supertype to a subtype using the same set of owner parameters, no existential owners need be introduced. Consider the following pieces of code:

```
class List<data outside owner> extends Object { ... }
class StringList<other outside owner> extends List<other> { ... }
```

If casting from `x:Object` to `x:List<y>`, an existential owner `y` must be introduced for the data owner parameter. However, when casting from `x:List<z>` to `x:StringList<z>`, the set of owners is unchanged. As `z` is already in the environment, as `x:List<z>` is a well-formed type, we can safely allow the downcast into a *non-existential type*. The only necessary dynamic check is to check that the actual type of the value is a subtype of `StringList`. A similar remark can be found in Boyapati's dissertation [4], but only for types with single owners.

5 ADDING EXISTENTIAL DOWNCASTING TO JOLINE

In this section, we show how to extend the Joline programming language [12, 21], with an existential downcast statement. For brevity, we present only the parts of the Joline system that are absolutely necessary to grasp the new extension. One intension is to show the straightforwardness of our idea and how easy it can be incorporated with the existing Joline language. Joline has been described elsewhere [11, 12] (see Wrigstad's dissertation [21] for an extended description including the dynamic semantics and proofs of the owners-as-dominators property and subject reduction).

To simplify the formal account, existential downcasting is formulated in standard let-expression style to provide a scope for the existential owners, rather than to allow expressions to extend the static type environment. How to map the let-style expressions to standard casts should be obvious.

Table 1: Syntax of Joline

c	$\in$	**ClassName**	f	$\in$	**FieldName**	md	$\in$	**MethodName**
x, y	$\in$	**TermVar**	α, β	$\in$	**OwnerVar**	R	$\in$	$\{\prec^*, \succ^*\}$

P	$::=$	$class_{i\in 1..n}\ s\ e$	*Program*
$class$	$::=$	$\texttt{class}\ c\langle \alpha_i\, \mathsf{R}_i\, p_{i\in 1..m}\rangle\ \texttt{extends}\ c'\langle p'_{i'\in 1..n}\rangle\ \{\ fd_{j\in 1..r}\ meth_{k\in 1..s}\ \}$	*Class*
fd	$::=$	$t\ f = e;$	*Field*
$meth$	$::=$	$\langle \alpha_i\, \mathsf{R}_i\, p_{i\in 1..m}\rangle\ t\ md(t_i\ x_{i\in 1..n})\ \{\ s\ \texttt{return}\ \ e\ \}$	*Method*
$lval$	$::=$		*l-value*
		x	*variable*
		$e.f$	*field*
e	$::=$		*Expression*
		$\texttt{this}$	*this*
		$lval$	*l-value*
		$\texttt{new}\ t$	*new*
		$\texttt{null}$	*null*
		$e.md\langle p_{j\in 1..m}\rangle(e_{i\in 1..n})$	*method call*
s	$::=$		*Statement*
		$\texttt{skip;}$	*skip*
		$t\ x = e;$	*variable declaration*
		$e;$	*expression*
		$lval = e;$	*update of lvalue*
		$s_1;\ s_2$	*sequence*
		$\texttt{let}\ x = (t)\ e\ \texttt{in}\ \{\ s\ \}$	*existential downcast*
		$\texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}$	*if-statement*
p, q	$::=$		*Owners*
		$\texttt{this}$	*this*
		α	*owner parameter*
		$\texttt{world}$	*world*
		$\texttt{owner}$	*owner*
t	$::=$		*Type*
		$p{:}c\langle p_{i\in 1..n}\rangle$	

Joline's Syntax

A subset of the syntax of Joline is displayed in Table 1. It is basically a subset of Java extended with ownership types and should be familiar to anyone with some experience of Java.

A program is a collection of classes followed by a statement and a resulting expression that are the equivalent of Java's main method. We could have followed Java's example and use a static main method etc., but chose this way out for simplicity.

Classes are parameterised with owner parameters. Each owner parameter (except the implicit, first, parameter `owner`) must be related to either `owner` or some previously declared parameter of the same class. Classes contain fields and methods. Fields must be initialised. Object creation requires the owner parameters specified in the class header to be bound to actual owners.

Static Semantics

In this section, we present the static type environment for Joline along with the definition of well-formed type environment and well-formed owners.

Static Type Environment

The type environment E records the types of free term variables and the nesting relation on owner parameters:

$$E \quad ::= \quad \epsilon \quad | \quad E, x :: t \quad | \quad E, \alpha \succ^* p \quad | \quad E, \alpha \prec^* p$$

Above, ϵ is the empty environment, $x :: t$ is a variable to type binding and $\alpha \succ^* p$ means that owner parameter α is outside owner p. Conversely, $\alpha \prec^* p$ means that owner parameter α is inside owner p.

Good environment

$$\overset{(\textsc{env-}\epsilon)}{\frac{}{\epsilon \vdash \diamond}} \qquad \overset{(\textsc{env-}x)}{\frac{E \vdash t \qquad x \notin dom(E)}{E, x :: t \vdash \diamond}}$$

$$\overset{(\textsc{env-}\alpha \succ^*)}{\frac{E \vdash p \qquad \alpha \notin dom(E)}{E, \alpha \succ^* p \vdash \diamond}} \qquad \overset{(\textsc{env-}\alpha \prec^*)}{\frac{E \vdash p \qquad \alpha \notin dom(E)}{E, \alpha \prec^* p \vdash \diamond}}$$

The rules for good environment are straightforward. (ENV-ϵ) states that the empty environment, ϵ, is well-formed. (ENV-x) states that adding a variable name to type binding, $x :: t$ to a good environment E produces another good environment provided x is not already bound to a type in E and t is a well-formed under E. The rules (ENV-$\succ^*$) and (ENV-$\prec^*$) deal with inside and outside orderings of owners—(ENV-$\succ^*$) states that adding a $\alpha \succ^* p$ ordering of two owners to a good environment E produces a good environment if p is a good owner under E and α is not in E. The (ENV-$\prec^*$) rule states the same, but for the $\prec^*$ relation.

Good owner

$$\text{(OWNER-VAR)}\ \frac{\alpha \,\mathsf{R}\, \_ \in E}{E \vdash \alpha} \qquad \text{(OWNER-THIS)}\ \frac{\texttt{this} : t \in E}{E \vdash \texttt{this}} \qquad \text{(OWNER-WORLD)}\ \frac{E \vdash \diamond}{E \vdash \texttt{world}}$$

The rules for good owners state that an owner is well-formed if it is defined in the static environment. Also, if present in the environment, the special variable `this` is also a good owner. The owner `world` is globally defined, and thus always valid.

Owner Orderings

$$\text{(IN-ENV1)}\ \frac{\alpha \prec^* p \in E}{E \vdash \alpha \prec^* p} \qquad \text{(IN-ENV2)}\ \frac{\alpha \succ^* p \in E}{E \vdash p \prec^* \alpha} \qquad \text{(IN-WORLD)}\ \frac{E \vdash p}{E \vdash p \prec^* \texttt{world}}$$

$$\text{(IN-THIS)}\ \frac{\texttt{this} :: t \in E}{E \vdash \texttt{this} \prec^* \texttt{owner}} \qquad \text{(IN-REFL)}\ \frac{E \vdash p}{E \vdash p \prec^* p} \qquad \text{(IN-TRANS)}\ \frac{E \vdash p \prec^* q \quad E \vdash q \prec^* q'}{E \vdash p \prec^* q'}$$

The inside and outside relations are derived from the owner orderings in E. The relations are transitive and reflexive and each others' inverses. Owners and their ordering form a tree (since an owner can only be ordered inside one owner by (IN-ENV1)). From (IN-WORLD), we see that all owners are inside `world`. Importantly, if `this` is a valid owner, it is always ordered inside `owner`, which is the owner of the object denoted by `this`.

Owner Substitutions

Substitution is denoted σ, where σ is a map from owner variables to owners.

As an illustration, if type $p\!:\!List\langle q\rangle$ is formed from the class definition

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

we sometimes write $p\!:\!List\langle\sigma\rangle$ for the same type where $\sigma = \{\texttt{data} \mapsto q\}$.

We write σ^p to mean $\sigma \cup \{\texttt{owner} \mapsto p\}$ and σ_n to mean $\sigma \cup \{\texttt{this} \mapsto n\}$ and σ^p_n for the combination. Applying a substitution to an owner is written $\sigma(p)$. For brevity, we write $\sigma(\alpha \,\mathsf{R}\, p)$ for applying a substitution to a pair of owners related with R. The application is defined thus:

$$\begin{array}{rcl} \sigma(p) & = & q, \text{ if } p \mapsto q \in \sigma \\ \sigma(p) & = & p, \text{ if } p \mapsto q \notin \sigma \\ \sigma(p \,\mathsf{R}\, q) & = & \sigma(p) \,\mathsf{R}\, \sigma(q) \text{ if } p \in dom(\sigma) \\ \sigma(p \,\mathsf{R}\, q) & = & p \,\mathsf{R}\, q \text{ if } p \notin dom(\sigma) \end{array}$$

Types

A type is well-formed whenever the substituted owner arguments satisfy the ordering on parameters specified in the class header.

$$\text{(TYPE)}\quad \frac{\texttt{class}\ c\langle \alpha_i\, \mathsf{R}_i\, p_{i\in 1..n}\rangle \cdots \in P \qquad \sigma = \{\texttt{owner} \mapsto q, \alpha_i \mapsto q_{i\in 1..n}\} \quad E \vdash \sigma(\alpha_i\, \mathsf{R}_i\, p_i)_{i\in 1..n}}{E \vdash q : c\langle q_{i\in 1..n}\rangle}$$

Subtyping

Subtyping in Joline must care to preserve the owner to preserve the encapsulation. A supertype may have fewer owners than a subtype. Mapping between owners of a subclass and its superclass is specified explicitly in the extends clause. The subtyping rule states that the owner must remain the same.

$$\text{(SUB-CLASS)}\quad \frac{E \vdash p : c\langle \sigma^p\rangle \quad \texttt{class}\ c\langle \ldots\rangle\ \texttt{extends}\ c'\langle p'_{i\in 1..n}\rangle \cdots \in P}{E \vdash p : c\langle\sigma\rangle \le p : c'\langle \sigma(p'_{i\in 1..n})\rangle}$$

Subtyping is derived from subclassing, modulo names of the owner parameters. As this corresponds to the composition of two order-preserving functions, it is order-preserving. This is required to preserve deep ownership, see Clarke's dissertation [8]. In particular, subtyping preserves the owner that is fixed for life. Letting the owner vary, as in Cyclone [16], would be unsound in a system with deep ownership, as observed by Clarke and Drossopoulou [9].

$$\text{(SUB-REFL)}\quad \frac{E \vdash t}{E \vdash t \le t} \qquad\qquad \text{(SUB-TRANS)}\quad \frac{E \vdash t \le t' \quad E \vdash t' \le t''}{E \vdash t \le t''}$$

As expected, the subtype relation is reflexive and transitive.

Parameters

We write $\mathcal{P}_c$ to denote the owner parameter declaration for a class c. For example, given the class definition

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

we have $\mathcal{P}_{\texttt{List}} = \{\texttt{data} \succ^* \texttt{owner}, \texttt{other} \succ^* \texttt{data}\}$ in straightforward fashion.

Existential Downcast

This section shows the single type rule necessary to extend Joline with existential downcasts.

$$\frac{\begin{array}{c}E \vdash e :: t \qquad E' \subseteq \sigma^p(\mathcal{P}_c) \\ E, E' \vdash p\!:\!c\langle\sigma\rangle \leq t \qquad E, E', x :: p\!:\!c\langle\sigma\rangle \vdash s; E''\end{array}}{E \vdash \texttt{let}\ x = (p\!:\!c\langle\sigma\rangle)\ e\ \texttt{in \{}\ s\ \texttt{\}}; E}\ \text{(EXISTENTIAL-DOWNCAST)}$$

The (EXISTENTIAL-DOWNCAST) rule states that the result of expression e of type t can be downcast to some type $p\!:\!c\langle\sigma\rangle$ if $p\!:\!c\langle\sigma\rangle$ is a subtype of t. This downcast will possibly introduce existential owners, E' for the scope of the block `{` s `}`. Note that E' is a subset of the nesting relations, $\mathcal{P}_c$, from the class header of c applied to σ, the mapping from parameter names from the class header of c to the actual names used by the programmer. The subset relation is necessary as a downcast need not introduce additional owners. To illustrate the rule, we provide a few examples based on the class declarations on Page 143.

Examples

Consider the following statement, type checked in a well-formed environment E in which `obj` is a variable typed `this:Object` (*i.e.*, `obj`::`this:Object` is an element in E):

```
let super = (this:List<ex>) obj; in { ... }
```

By (SUB-CLASS), if `this:List<ex>` is a well-formed type, it is a subtype of the type `this:Object`. $\mathcal{P}_{\texttt{List}} = \{\texttt{some} \succ^* \texttt{owner}\}$ and $\sigma^p = \{\texttt{owner} \mapsto \texttt{this}, \texttt{some} \mapsto \texttt{ex}\}$.

Thus, $\sigma^p(\mathcal{P}_{\texttt{List}}) = \{\texttt{ex} \succ^* \texttt{this}\}$. If `ex` is free in E and `this` is a well-formed owner (which it must be, as `this:Object` is a well-formed type), E can be extended to $E' = E, \texttt{ex} \succ^* \texttt{this}$, an environment under which `this:List<ex>` is a well-formed type. Thus, $E' \vdash$ `this:List<ex>` holds and by (SUB-CLASS), $E' \vdash$ `this:List<ex>` $\leq$ `this : Object`.

Interestingly, if we were to downcast `super` to `StringList` *in the nested scope of the let expression*, we would only have to introduce one new existential owner:

```
let example = (this:StringList<ex, other>) super; in { ... }
```

In this case, $\mathcal{P}_{\texttt{List}} = \{\texttt{owner1} \mapsto \texttt{ex}, \texttt{owner2} \mapsto \texttt{other}\}$ and $\sigma^p(\mathcal{P}_{\texttt{List}}) = \{\texttt{ex} \succ^* \texttt{this}, \texttt{other} \succ^* \texttt{ex}\}$. As `ex` is already in the environment, we only need to introduce `other`. Note that if `List` did not introduce additional parameters other than what was in the superclass, downcasting would not need to introduce any (new) existential parameter. This is dealt with automatically by the subset relation between E' and $\sigma^p(\mathcal{P}_c)$. Thus, when all the owners are statically known, and the subclass does not introduce new owners not in the superclass, E' in the rule above will be empty and our downcasting proposal have equal strength as Mini Safe Concurrent Java, but without the run-time overhead.

Again, note that the let-expression is not strictly necessary but simplifies the formalism. We could have allowed the cast expression to introduce new owners for the remainder of the enclosing scope. This would have retained the standard syntax for casts, but would have broken away from our existing formalisation, as expressions do not update the static type environment.

6 CONCLUSION

We have presented a novel way of implementing downcasting in the presence of ownership types. The benefits of our system are twofold: it does not need a run-time representation of ownership and it can safely introduce owners into a scope without breaking any containment invariants or require these owners to be passed around. This increases the expressiveness and flexibility of the system and enables encoding of idioms such as the Java equals method without breaking overriding due to changing signatures. Our proposal also deals naturally with downcasting where no additional introduction of owners is necessary. In such cases, downcasting into a previously known, non-existential type is possible, with the same strengths as in Mini Safe Concurrent Java but without the run-time overhead.

In contrast to our proposal, Mini Safe Concurrent Java [6] allows downcasting into a type that extends the owner parameters with parameters already in scope. This is more powerful, but comes with the cost of keeping track of a potentially large mass of ownership information at run-time.

The existential system is simple, both statically, as we have shown, and dynamically (just rely on standard Java-style downcast checking) and completely orthogonal to and therefore combinable with previous downcast proposals. Finally, nothing would prevent using our system in a shallow ownership setting such as in ArchJava [2].

References

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

[2] Aldrich, J., Chambers, C., and Notkin, D. ArchJava: Connecting software architecture to implementation. In *ICSE* (May 2002).

[3] Aldrich, J., Kostadinov, V., and Chambers, C. Alias annotations for program understanding. In *Proceedings of the OOPSLA Conference on*

Object-Oriented Programming, Systems, Languages and Applications (November 2002).

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

[5] Boyapati, C., Lee, R., and Rinard, M. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications* (November 2002).

[6] Boyapati, C., Lee, R., and Rinard, M. Safe runtime downcasts with ownership types. In *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming* (July 2003), D. Clarke, Ed., UU-CS-2003-030, Utrecht University.

[7] Buckley, A. Ownership types restrict aliasing. Master's thesis, Department of Computer Science, Imperial College of Science, Technology, and Medicine, Queen's Gate, London, June 2000.

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

[9] Clarke, D., and Drossopolou, S. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications* (November 2002).

[10] Clarke, D., Potter, J., and Noble, J. Ownership types for flexible alias protection. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications* (1998).

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

[12] Clarke, D., and Wrigstad, T. External uniqueness is unique enough. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (Darmstadt, Germany, July 2003), L. Cardelli, Ed., vol. 2473 of *Lecture Notes In Computer Science*, Springer-Verlag, pp. 176–200.

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

[14] Goldberg, A., and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[15] Gosling, J., Joy, B., Steele, G., and Bracha, G. *Java(TM) Language Specification, The (3rd Edition) (Java Series).* Addison-Wesley Professional, July 2005.

[16] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (June 2002).

[17] Hejlsberg, A., and Wiltamuth, S. *C# Language Specification.* Microsoft Corporation, 2000.

[18] Krishnaswami, N., and Aldrich, J. Permission-based ownership: encapsulating state in higher-order typed languages. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM Press, pp. 96–106.

[19] Leavens, G. T. Why specification languages need ownership types. In *Dagstuhl workshop Types for Tools: Applications of Type Theoretic Techniques* (Dagstuhl, Germany, June 2005). (based on joint work with Peter Müller and Arnd Poetzsch-Heffter).

[20] Pierce, B. C. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

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

ABOUT THE AUTHORS

Tobias Wrigstad is a lecturer at the department for computer and systems sciences, Stockholm University/Royal Institute of Technology, Kista. His main areas of interest are object-oriented programming, formal methods and dynamic programming languages. He can be reached at tobias@dsv.su.se.
Also see http://dsv.su.se/ tobias.

Dave Clarke is a postdoctoral researcher in the SEN3 "Coordination Languages and Models" Group at CWI in The Netherlands. His main areas of interest are coordination languages, object- and aspect-oriented programming, game theory and formal methods. He can be reached at David.Clarke@cwi.nl.
Also see http://homepages.cwi.nl/ dave/