Ownership Types

Fabian Muehlboeck

April 11, 2012

Objects encapsule data and are a nice way to structure a stateful, imperative program, enabling us to have what we call separation of concerns. We would like to reason about objects or classes individually rather than about the whole program. Therefore, we often specify some invariants on objects, and then see to it that all method calls to an object preserve these invariants. However, those invariants might depend on the status of some other objects our object is related to: a stack might be implemented using a linked list. As long as that linked list is solely controlled by the stack, everything is fine. But as we often need to share objects, we need some form of aliasing, that is having references to the same objects at multiple points in the program. An uncontrolled form of aliasing would make a scenario in which some other object has access to the linked list that is vital ot the representation of our stack possible - that other object may not know about the stack's invariants and internal state and might hence violate the stack's invariants when it manipulates the list that it has access to. With uncontrolled aliasing, we never know which objects hold references to which other objects, which forces us to take the whole program into account when reasoning about the correctness of a class.

1 An Introduction to Ownership Types

The basic idea of ownership types is to eliminate a scenario like above by assigning an owner to each object. From this, we can build an ownership hierarchy as a tree, where owned objects are the child nodes of their owners. Such a child node may access anything their parent can access (if the parent shares it with the child), i.e. all their ancestors plus their immediate children, as well as their siblings. In general, every access to an owned object must go through the owner if it comes from/via higher up in the ownership tree.

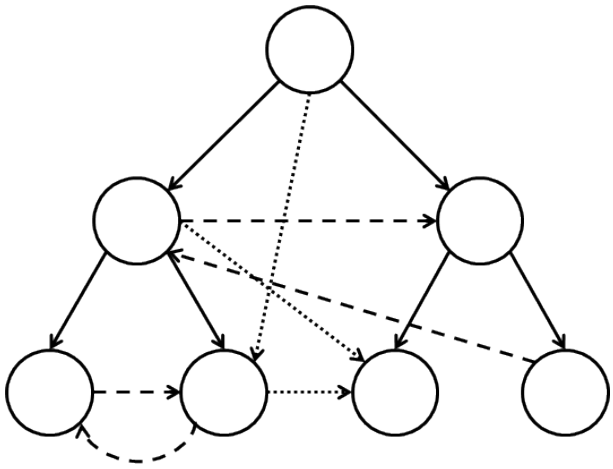

Figure 1: An example ownership tree. Each circle is an object (except for the root, which is a dummy). Solid arrows represent ownership relations, dashed arrows are valid references, dotted arrows represent invalid references

Our interest is to include the concept of ownership into static typechecking, which means we will rule out some perfectly fine programs and possibly lose some programming constructs we want to have. In the following, we will examine some attempts for statically checked ownership type systems and see another application of ownership types than just simply controlling aliasing, namely in the context of concurrency.

Our running example will be a that of a stack that is implemented as a linked list. The stack will offer the methods *push* and *pop* to add and remove elements from the stack. We will view the nodes as relevant to the stack's representation and thus its invariant, in particular $s.push(x); x = s.pop();$ should always work for a stack s and some object of a right type x. The

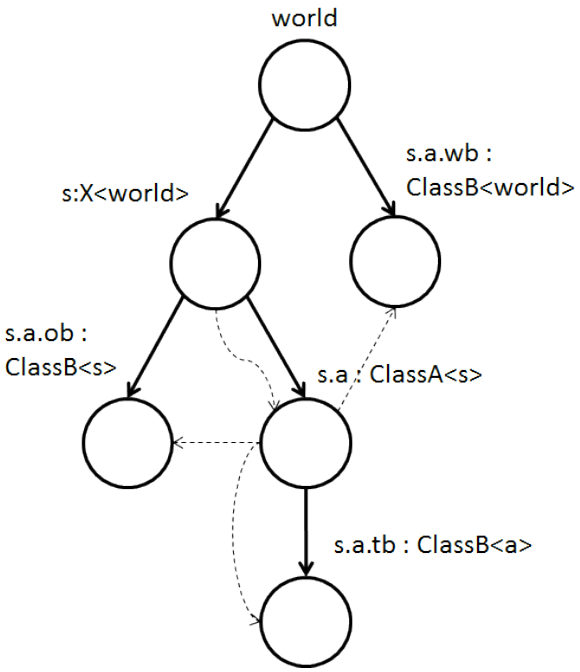

Figure 2: A more detailed example of an ownership tree. Assume X is a type that has a field of type ClassA, where the object in that field is owned by the instance of X. Dashed lines again represent references, solid lines ownership relations.

objects the stack stores however are not relevant to its representation and should be accessible from the outside.

2 Basic Ownership Types

2.1 Our Language

We will use a Java-like language, first presented by Clarke et. al, but slightly adapted by Boyapati et al. to demonstrate how we would use ownership types to program a safe stack. We will introduce the most important constructs here, the full syntax and explanation can be found in Appendix B. The important notation that we are going to introduce are ownership annotations, which look just like generic type parameters:

```
class ClassA<o>
{
    ...
}
```

This is the simplest form of a class declaration. The parameter *o* will be instantiated with the owner of an object, which, depending on where in the code that is, might be either *this*, *world* or another existing parameter name. Here, *world* is a dummy owner representing the root of the ownership tree. A program in our language consists of a bunch of class definitions followed by an expression to be evaluated. That might just be $(newClassA\langle world\rangle).main()$. That main-method could then create some global objects owned by the world or objects that are owned by that object of ClassA.

```
class ClassA<o>
{
        ClassB<world> wb;
        ClassB<o> ob;
```

```
        ClassB<this> tb;
        void main() {...}
}
class ClassB<o>
{
        ...
}
```

Here we see that an object of ClassA contains three fields of ClassB, all with (possibly) different owners. Figure 2 shows an example setup. The first field, *wb*, is a global object, independent of the actual owner of the ClassA object. The second field, *ob*, will have the same owner as the ClassA object. That means, if our ClassA object is a global object, *ob* will be globally accessible, too. But we do not know this at compile-time, hence an assignment like $this.ob = this.wb$; is illegal, as is $this.wb = this.ob$; . Lastly, *tb* is owned by our ClassA object. This means among other things that neither *wb* nor *ob* can ever access *tb* directly.

2.2 Programming our Stack

Given this concept of ownership, we create our stack. As we do not have generics or subtyping, we will have to create a stack for a specific type. Let us say it is a stack of papers of some class $Paper\langle o\rangle$. For now, we will assume that all the papers are global objects. We will also leave out checks one would normally write for *pop*() to ensure that the stack is not empty and use some notational shorthands not mentioned in the grammar in the Appendix but closer to Java.

```
class Node<no>
{
        Paper<world> paper;
        Node<no> next;
}
class Stack<o>
{
        Node<this> first;
        void push(Paper<world> paper)
        {
                first = new Node(paper, first);
        }
        Paper<World> pop()
        {
                Paper<world> p = first.paper;
                first = first.next;
                return p;
        }
}
```

Now this stack has a totally safe internal representation. The ownership type system will statically check that Nodes are never accessible from outside of the stack. Note that all nodes are owned by the stack rather than by their predecessor node. This is important because there is no notion of a transfer of ownership in the presented type system, hence the first created node will always be owned by the stack, and any subsequent node that is put on top of the stack thus must point to a node that has this ownership type.

2.3 Ownership Polymorphism

We might not always know what owners the objects that the stack should store will have. They might be at any point in the ownership tree, provided that the owner of the stack and thus the stack itself can access them. At this point, we are forcing them to be at the global ownership level, accessible to everyone, but we may not want that. Instead, we can introduce additional ownership parameters:

```
class ClassA<o,p,q>
{
    ...
}
```

The first parameter, *o*, still has a special status: it refers to the actual owner of an instance of the type. The other ownership parameters enable us to specify owners of contents of our object, and we can decide on instantiation wheter to provide *this* or *world* or any other available ownership parameter for every of the parameters of the new object.

This way, we can replace *world* as the owner of our paper objects by a parameter that we add to the stack and node classes:

```
class Node<no,np>
{
        Paper<np> paper;
        Node<no,np> next;
}
class Stack<o,p>
{
        Node<this,p> first;
        void push(Paper<p> paper) { ... }
        Paper<p> pop() { ... }
}
```

If the stack is instantiated with *world* for p, we effectively have the same setup as before. However, this time we may also have any other owner for our papers, so long as that owner makes the papers still accessible to the stack and its nodes. We therefore require all context parameters of an object to be owners of the owner of that object.

3 Type-checking basic Ownership

First, look at the last statement in the previous section: all context parameters of an object must be owners of the owner of that object (i.e. the first context parameter). Suppose this was not the case, i.e. we had at least two context parameters where the first one is not owned by the second one. Then we can just have a field that's first context parameter (and therefore its owner) is the second context parameter that we have. The content of that field is now either owned by something that our object owns (if the second context parameter is owned by the first context parameter) or even by something somewhere else in the ownership tree - in both cases, we would have access to an object without going through its owner, thus violating our ownership structure.

Now recall our ownership trees from above. The key idea to having ownership prevent access to certain objects is encoded in the so-called **Static Visibility Constraint**:

$$SV(e,t) := (e \neq \textbf{this}) \Rightarrow \textbf{this} \notin contexts(t)$$

where

$$contexts(cn\langle m_1 ... m_n \rangle) := \{m_1, ..., m_n\}$$

We use this constraint in the typing rules for field accesses and method calls, in the way that we use some $e.x$, where x is either a field name or a method name plus arguments, and t is the type of the whole expression $e.x$. Now if $e \neq$ **this**, this means that we are potentially accessing another object's fields or methods. If the (return) type of that field or method includes a **this**, then by our rule above that the first context parameter must be owned by all the others (and the observation that **this** is the lowest possible value for an owner - because you can't have access to something that is owned by someone lower in the ownership tree) the owner of that object must be e. And because e is not **this**, we may not access it from our current

class. Note that **this** is used in two different meanings here: the ownership parameter **this** is relative to e, whereas the object reference **this** is relative to our current object.

In short, static visibility says that if we access a field or method in some other object, the value we get back from that must not be owned by that other object. This limitation is a little stronger than what the ownership structure permits, because if e were the owner of our current object, we actually may have references to other objects owned by e. The result of this that we cannot just retrieve those objects from e, but rather e has to give us references to the objects it wants to share with us.

The last things we need for type-checking are some small helpers. $Node\langle no, np\rangle$ in the class declaration header is called an ownership scheme. We may insert anyhthing valid by the rules above for no and np. In the typing rules, σ stands for a function that maps an ownership scheme to a type declaration. If we have an object of type $Node\langle$**this,p**$\rangle$ (that is, it is owned by our current object, and the second parameter is bound to the current object's binding for p, that is, an object somehwere above in the ownership tree), σ for that type would map no to **this** and np to p. Hence when we ask for the field $next$, we know that that Node in there is also owned by our current object since we replaced the first context parameter with $this$. In addition, we have a function ϕ that extracts such a σ when given a type already instantiated with ownership parameters. Lastly, we assume that we have some dictionaries that tell us the types of fields and methods by writing $t.fn : t'$, meaning that the field with name fn in the class of type t is of value t', and $t.mn : t_1...t_k \to t'$ gives us the type of the method with name mn in the class of type t. An example would be: $Node\langle no, np\rangle.next : Node\langle no, np\rangle$.

Let us check (parts of) the following method:

```
Paper<p> pop()
{
        Paper<p> pp = first.paper;
        first = first.next;
        return pp;
}
```

First, we check the field access $first.paper$:

$$\frac{... \; \Gamma \vdash e : t \qquad \sigma = \phi(t) \qquad t.fn : t' \qquad SV(e, t')}{... \; \Gamma \vdash e.fn : \sigma(t')} \text{ (Field Access)}$$

This means that we want to check that $e(= first).fn(= paper) : \sigma(t')(= Paper\langle p\rangle)$ under some environment that we will not explain further. In order to check that this is correct, we first have to check that e has some type t. In our case, $first$ has type $Node\langle this, p\rangle$. We retrieve our σ from that as $\{no \Rightarrow this, np \Rightarrow p\}$. The field named *paper* in our the class of our type t has type $t' = Paper\langle np\rangle$. Luckily, our σ therefore gives us the right type of $e.fn$, namely $Paper\langle p\rangle$. The last thing we have to check is static visibility, i.e. since e is not **this**, the object in the field *paper* may not be owned by e. Since $Paper\langle np\rangle$ does not contain **this**, we are fine.

Checking field update works very similar to field access - we only have to also check the expression that we are evaluating.

$$\frac{... \; \Gamma \vdash e : t \qquad \sigma = \phi(t) \qquad t.fn : t' \qquad ... \; \Gamma \vdash e' : \sigma(t') \qquad SV(e, t')}{... \; \Gamma \vdash e.fn = e' : \sigma(t')} \text{ (Field Update)}$$

We can check $first = first.next$ (because technically, it would have to be $this.first = this.first.next$). The type of e is **this**, so our sigma will be the identity function. The type of the field is $Node\langle this, p\rangle$, and e' has to be checked with field access against that type, which we will assume worked out. Since $e =$ **this**, static visibility is trivial, and our type-check succeeds.

Method calls work very similar:

$$\frac{\begin{array}{c} ... \; \Gamma \vdash e : t \qquad \sigma = \phi(t) \qquad t.mn : t_1...t_n \to t' \\ ... \; \Gamma \vdash e_1 : \sigma(t_1) \; , \; ... \; , \; ... \; \Gamma \vdash e_n : \sigma(t_n) \qquad SV(e, t'), SV(e, t_1)...SV(e, t_n) \end{array}}{... \; \Gamma \vdash e.mn(e_1, ..., e_n) : \sigma(t')} \text{ (Method Call)}$$

The e must have some type t that has some context parameter substitution σ and a method with name mn that takes arguments of types $t_1 ... t_n$ and returns an object of type t'. We have to check that the expressions we use for our arguments are well-typed and evaluate to something of a type with the right context parameter substitution, and all the arguments as well as the return type must be visible to our current context (i.e. we cannot give objects as arguments to something if we do not have access to them).

Say that we have some object that has a field s of type $Stack\langle this, this\rangle$. Let us check a method call to $s.pop()$. e is a field access that is well-typed and has type $Stack\langle this, this\rangle$. The σ therefore is $\{o \Rightarrow \mathbf{this}, p \Rightarrow \mathbf{this}\}$, the method *pop* is of type $\rightarrow Paper\langle p\rangle$. There are no arguments to type-check, and only one static visibility constraint. $e \neq \mathbf{this}$, but $Paper\langle p\rangle$ does not contain **this** either, so we are fine. Applying the σ, we get the type $Paper\langle \mathbf{this}\rangle$ as the type of the result of the method call.

4 Iterators

Our stack works: we can put objects onto it and we can retrieve them again. However, what if we want to examine the whole content of the stack without removing all the elements? We can add iterator-functionality to the stack by adding the methods *current*(), *next*() and *reset*() and it will work just fine. However, often we would like to have several iterators at the same time. In our current model, this is not possible: the iterator would have to be an object of another class, and it had to be visible outside of the stack, hence it cannot be owned by the stack. But if it is not owned by the stack, it cannot have access to the nodes. This means that there is no way to implement an efficient iterator with our current set of possibilities.

Obviously, we have to violate our ownership structure at some point. The problem with that is that if we allow this arbitrarily, we might lose our ability to reason locally about parts of our program. Boyapati et. al therefore suggested to use inner classes as a natural way to describe such breaches in the ownership structure. The idea is that even if inner classes can access inner parts of the outer object while not being owned by the outer object, we still have a bound on the extent of the violation - i.e. we just reason locally about a class and an inner class together. There is only one interesting thing that inner classes add to our syntax, which is the reference to the context of the outer class. Our stack with iterator would look like this:

```
class Node<no,np>
{
        Paper<np> paper;
        Node<no,np> next;
}
class Stack<o,p>
{
        Node<this,p> first;
        void push(Paper<p> paper) { ... }
        Paper<p> pop() { ... }
        class Iter<io>
        {
                Node<Stack.this,p> current;
                Paper<p> next()
                {
                        Paper<p> pp = current.paper;
                        current=current.next;
                        return pp;
                }
                bool hasNext() { return current != null; }
        }
        Iter<i,p> getIter<i>{ return new Iter<i>(first); }
}
```

One novelty here is the context parameter for the method getIter, which we can use to specify the owner for the new iterator. The other, already mentioned interesting thing is the type of the field *current* in *Iter*.

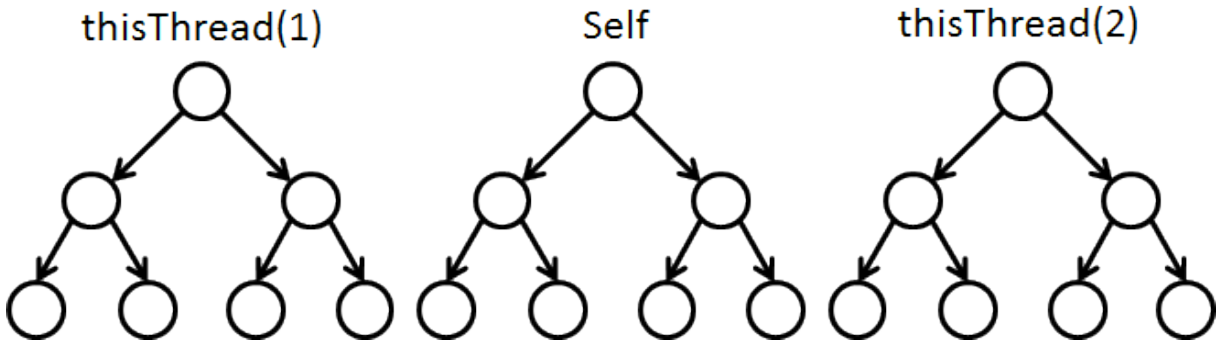

Figure 3: An example ownership forest in a concurrent program. The left tree is only accessible in the code executed by thread 1, the right tree is only accessible in the code executed by thread 2, and the middle tree is accessible by both, but has to be locked

Its ownership context is defined as $Stack.this$. This is just the way to express that the node is actually owned by the outer stack object, a notation that we can only use with inner classes.

5 Application: Concurrent OO-Programming

Boyapati et al. present a very intuitive way to use ownership types to prevent data races and are also able to give a simple scheme to prevent deadlocks.

5.1 Data Races

The idea is that instead of just one tree, we have a forest of ownership trees. The root nodes have one of two special ownership contexts, which is either *thisThread* or *self*. If the ownership context is *thisThread*, then the object and all the objects it owns are only accessible from within the thread where they were created, thus we know that we can safely manipulate them without synchronization.

On the other hand, if the ownership context of an object is *self*, then the object is and all the objects it owns are shared between all the threads and thus have to be locked before they can be accessed. Whereas it is usually hard to figure out which part of a composite object to lock best, we have to lock the root of the *self*-owned ownership tree here.

Now we can statically check that there is a synchronized-statement around every object that is in a *self*-owned ownership tree and therefore guarantee freedom from data races (provided that the programmer set the synchronized blocks in such a way that they actually guard the actions he considers atomic).

5.2 Deadlocks

When we introduce locking, we usually automatically introduce the possiblity of deadlocks, which occur when one thread has already locked resource A and now waits for resource B while another thread has locked resource B and waits for resource A. The simple idea to eliminate deadlocks in this scheme is to assign a lock level to every root of *self*-owned ownership trees. Then we can statically check if locks are aquired in the order of the lock levels and hence guarantee the absence of deadlocks.

6 Conclusion

Ownership types are a field of ongoing research, and the papers presented are from a quite early era of that research. While the presented systems can express ownership quite nicely, they also have some limitations in particular do not enable change of ownership, or shared ownership, which might be a - desirable thing in more complex scenarios (a very limited system of changing ownership dynamically in certain situations

has been proposed in the concurrency paper, though, and some kind of shared ownership is presented in the formalization paper by Clarke et al., but this system on the other hand has the disadvantage that the number and structure of ownership contexts must be statically know). As always with type systems, the question is wheter the safety benefit we gain for our programs outweighs the loss of perfectly fine or good enough programs that are outruled by the type system. So far, ownership types do not seem to have won that case.

A The papers

A.1 Ownership Types for Flexible Alias Protection

D. G. Clarke, J. M. Potter and J. Noble OOPSLA 1998: This paper lays the groundwork for ownership types and introduces the basic ownership type system presented in the beginning. They are able to proof some important properties of their type system, in particular on what they call Representation Containment, i.e. the property we showed in figure 1. Types consist of class names and ownership parameters, annotated like type parameters in generic Java - hence it is not possible to change ownership dynamically. They do not specify inheritance or subtyping.

A.2 Simple Ownership Types for Object Containment

D. G. Clarke, J. Noble and J. M. Potter - ECOOP 2001: This paper formalizes the concept of ownership types. Here, they are presented as an extension to the object calculus of Abadi and Cardelli. They limit themselves to a scenario where the number and structure of ownership contexts must be known beforehand and provided as a partial order, though. This still allows many applications, and especially allows more flexible sharing since the relations now do not necessarily form a tree anymore, but a DAG.

A.3 Ownership Types for Object Encapsulation

C. Boyapati, B. Liskov and L. Shrira - POPL 2003: Building on the type system presented by Clarke et. al in 1998, the authors present an extended version of it which also includes inheritance, subtyping an basic effect clauses. The most important contribution of this paper is the idea to use inner classes as a natural way of providing access to objects in a way that would violate important properties of ownership relations, which in this case does not matter because these violations are contained within the outer class and therefore local reasoning is still possible so long as a class and its inner classes are looked at together. This enables us to implement iterators.

A.4 Ownership Types for Safe Programming: Preventing Data Races and Deadlocks

C. Boyapati, R. Lee and M. Rinard - OOPSLA 2002: Here, the authors incrementally present some extensions to the language in the above paper to handle concurrent programming as presented in Section 5. They also give an overview of possible extensions like basic ownership transfer, using DAGs instead of trees for the ownership hierarchy, possibilities of type inference and dynamic assignment of lock levels. In addition, there is an extensive survey on the state of the art in ownership types.

B Grammars

B.1 Basic language

$$
\begin{array}{lcl}
p & ::= & defn* \ e \\
defn & ::= & \textbf{class } cn\langle m+\rangle \ body \\
body & ::= & \{ \ field* \ meth* \ \} \\
meth & ::= & type \ mn(arg*) \ \{ \ e \ \} \\
field & ::= & type \ fn \\
arg & ::= & type \ vn \\
type & ::= & cn\langle ow+\rangle \\
ow & ::= & m \mid \textbf{this} \mid \textbf{world} \\
e & ::= & \textbf{new } type \mid x \mid \textbf{let } (arg = e) \textbf{ in } \{ \ e \ \} \mid x.fn \mid x.fn = x \mid x.mn(x*) \mid e; e \\
 & & \\
m & \in & \textbf{context parameter names} \\
cn & \in & \textbf{class names} \\
mn & \in & \textbf{method names} \\
fn & \in & \textbf{field names} \\
vn & \in & \textbf{variable names}
\end{array}
$$

The syntax is rather self-explanatory, the main feature are the context parameters in angle brackets. A class declaration contains at least one variable context parameter, possibly more, where the first one is the owner of an instance of that class. The actual parameters are given on instantiation, and field/method type declarations, where they can either be **this**, **world**, or one of the parameters in the class declaration.

B.2 Extended language

$$
\begin{array}{lcl}
p & ::= & defn* \ e \\
defn & ::= & \textbf{class } cn\langle m+\rangle \textbf{ extends } c \ body \\
c & ::= & cn\langle ow+\rangle \mid \textbf{Object}\langle ow+\rangle \mid c.cn\langle ow+\rangle \\
body & ::= & \{ \ defn* \ field* \ meth* \ \} \\
meth & ::= & type \ mn\langle m*\rangle(arg*) \ \{ \ e \ \} \\
field & ::= & type \ fn \\
arg & ::= & type \ vn \\
type & ::= & c \\
ow & ::= & m \mid \textbf{this} \mid \textbf{world} \mid cn.\textbf{this} \\
e & ::= & \textbf{new } type \mid x \mid \textbf{let } (arg = e) \textbf{ in } \{ \ e \ \} \mid x.fn \mid x.fn = x \mid x.mn\langle ow*\rangle(x*) \mid e; e \\
 & & \\
m & \in & \textbf{context parameter names} \\
cn & \in & \textbf{class names} \\
mn & \in & \textbf{method names} \\
fn & \in & \textbf{field names} \\
vn & \in & \textbf{variable names}
\end{array}
$$

The main difference to the syntax presented before is that we allow inner class definitions in the body and gave context parameters to methods, too (an example of their use can be seen in the Iterator example). We also see subclassing introduces, therefore we also need to be able to refer to Object as a common superclass. An important limitation for subclassing is that the first context parameter of the superclass in the class declaration must also be the first context parameter of the subclass, else one object could have different owners depending on the declared type of the object.