

A Fundamental Permission Interpretation for Ownership Types

Yang Zhao

Nanjing University of Science and Technology
200 Xiao Ling Wei, Nanjing, China
yangzhao@mail.njust.edu.cn

John Boyland

University of Wisconsin Milwaukee
2200 E. Kenwood Blvd. Milwaukee, WI USA
boyland@cs.uwm.edu

Abstract

This paper builds a bridge between permissions and ownership types. Ownership is a recognized alias control technique. With ownership, each object is assigned an owner and any access to that object is required to follow some rules based on its owner. Permission is a low-level linear value associated with some piece of state in a program and it is often used to permit certain operations. A permission nesting indicates that some permission is nested in another which intuitively reveals a protection relation between a nested permission and its nester one, with building some restriction among operations furthermore. Permission nesting and ownership behave some common characteristic. In this paper, two ownership models (owners-as-dominators and owners-as-locks) are investigated, and we show they are able to be unified by permission interpretation. Whereafter, we discuss the possibilities of representing multiple ownership by fractional permissions.

1. Introduction

Inter-object aliases are normally implemented in object-oriented programs, since they are powerful and flexible. However, any change happening on an object may potentially affect other objects that refer to it even though they are unaware of it. The usage of aliasing often complicates the program reasoning and verification.

Researchers try to control aliases by restricting visibility of objects. For example, Clarke et al. propose to use ownership types: static types annotated with context declarations that represent object ownership [11]. With ownership, each object may own several objects, but is only owned by just one object. Any object is not allowed to own itself either directly or indirectly, but can be owned by a special global recognized object called “world” which cannot be owned any more. Therefore, the ownership hierarchy is normally acyclic and exhibits as a tree rooting in the “world.” Ownership actually provides a statically enforceable way of speci-

fying object encapsulation: all access paths from the root of the system to an object pass through the owner of that object [10]. This encapsulation property prevents any outside access to an object other than those from its owner. That is, owners act as dominators.

In order to verify race free in a multithreaded program, Boyapati et al. [3, 1] modify the previous owners-as-dominators model into a new one: *owners-as-locks*, such that any reference to an object must guarantee that its root-owner is held by the context thread. Furthermore, they break the single-tree ownership hierarchy into a forest. Any regular object as well as the per thread object `thisThread` is able to become a root that leads an ownership tree independently. Both owners-as-dominators and owners-as-locks models essentially build an ownership order among objects and then implicitly apply certain protection enforcement by type checking.

A permission is a token associated with some piece of state in a program that permits corresponding operations [4, 5, 8]. For a well permission typed program, it is required that no operation is allowed unless it is granted some permission. A permission nesting indicates a relation between the nested permission and its nester one, such that the former is adopted by the latter. It intuitively demonstrates a protection relation which has something in common with ownership. With nesting, we are able to simulate the ownership relation from the permission’s point of view.

In this paper, we expect to connect permissions with ownership types by interpreting different ownership types with low-level permission representation. We first briefly review the permission system including its syntax and semantics in Section 2, then induct the permission representation for two kinds of ownership types in Section 3. Some attempts to interpret multiple ownership by fractional permission are discussed in Section 4. We conclude in Section 5.

$\rho ::= o \mid r$	<i>literal reference, variable</i>
$k ::= \rho.f$	<i>key (field instance)</i>
$\Pi ::=$	<i>permission:</i>
Γ	<i>fact(formula)</i>
$k \rightarrow \rho$	<i>unit permission</i>
v	<i>permission variable</i>
\emptyset	<i>empty permissions</i>
$\Pi + \Pi$	<i>combination</i>
$(\Gamma)?(\Pi) : (\Pi)$	<i>conditional</i>
$\exists r. (\rho.f \rightarrow r + \Pi)$	<i>existential</i>
$\Pi \multimap \Pi$	<i>implication</i>

Figure 1. Permission syntax.

2. Permission Overview

Translated from program annotations, a *permission* is a token associated with some piece of state in a program, where different permissions are granted to permit different operations. We omit the translation process as well as permission inference and transformation since they are out of the scope of this paper. More details about permission could be found in our previous work [4, 5, 8, 6, 7].

Figure 1 shows the permission syntax. An *empty permission* \emptyset permits nothing, while an access to a field instance $\rho.f$ (assuming it is a reference to ρ' currently) is allowed provided a *unit permission* $\rho.f \rightarrow \rho'$ is granted.

A read operation is not able to change the program's state, thus the associated unit permission is preserved safely. A write operation, however, may potentially update a field's value. Within permission, a write operation is not allowed unless a corresponding unit permission is granted in advance, and once the write happens, a possible different unit permission occurs afterwards. The different permission effects for different operations are demonstrated in below:

$$k \rightarrow \rho' \Longrightarrow \boxed{\text{an operation reading } k} \Longrightarrow k \rightarrow \rho'$$

$$k \rightarrow \rho' \Longrightarrow \boxed{\text{an operation writing } k} \Longrightarrow k \rightarrow \rho''$$

where k represents a field instance. $k \rightarrow \rho'$ and $k \rightarrow \rho''$ could be different assuming ρ' and ρ'' are known to be different locations.

A *compound permission* $\Pi + \Pi'$ is constituted from two sub-permissions Π and Π' , and gives one all the rights associated with either of them being compounded. This combination is commutative, associative and using the empty permission \emptyset as the identity.

$$\Pi + \emptyset \equiv \emptyset + \Pi \equiv \Pi$$

Permission is a static notation and it is often unable to decide actual object locations before the runtime. That is

the reason we have to utilize existential variables to represent unknown objects. Assuming there is a unit permission for $\rho.f$ as well as a unit permission for the f' field of its pointed-to object (assuming non-null), then an existentially quantified variable is introduced to represent that unknown pointed-to object:

$$\exists r. (\rho.f \rightarrow r + r.f' \rightarrow \$0)$$

where r is bound for the compound permission and the $r.f'$ is a null pointer¹ currently. This is an *existential permission* and it can be unpacked in the standard way (using skolemization).

A *conditional permission* $(\Gamma)?(\Pi) : (\Pi')$ presents either Π or Π' depending on the truth value of Γ , where Γ is a boolean formulae (a *fact*) using these forms:

- standard boolean logic: true, $\neg(\Gamma)$ or $\Gamma \wedge \Gamma$;
- type assertion ($\rho \in C$): ρ has the type C or its subtype;
- reference equality ($\rho = \rho'$): two references are referring to the same location;
- nesting fact ($\Pi \prec k$): permission Π is nested in a unit permission for the field instance k ;
- named predicate $p(\bar{\rho})$: a conjunction of several facts that usually represents a class invariant.

Facts are treated as non-linear permissions which are allowed to be duplicated arbitrarily. Given a truth value of the condition part, a conditional permission could be “reduced” to its true or false clause, such as

$$\begin{aligned} \Gamma + (\Gamma)?(\Pi) : (\Pi') &\equiv \Gamma + \Pi \\ \neg(\Gamma) + (\Gamma)?(\Pi) : (\Pi') &\equiv \neg(\Gamma) + \Pi' \end{aligned}$$

A *linear implication* $\Pi_1 \multimap \Pi_2$ indicates that one has the rights of the consequent Π_2 , except for the ones of the antecedent Π_1 . There are two related concepts: *permission nesting* and *permission carving*.

Definition 3.1 (Permission Nesting). Any permission Π can be nested in a unit permission for a field instance k and this nesting relation is denoted as a *fact (formula)*:

$$\Pi \prec k$$

Permission nesting indicates that anyone holding the nester permission *owns* the nested permission as well.

A permission nesting primarily has two functionality:

- It intuitively indicates a protection relation. Since the nested permission is not available unless its nester permission is granted, we may consider the latter as a protector for the former.

¹The $\$0$ is used to represent null.

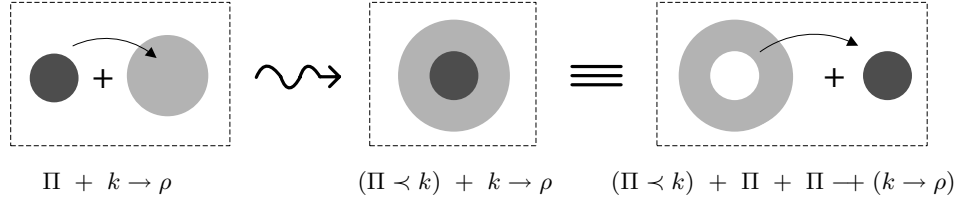


Figure 2. Permission nesting and carving.

- It enables state encapsulation. The nester may be associated to a data group [13], such that it hides some nested permissions that are not supposed to be exposed to clients directly.

Assuming Π and $k \rightarrow \rho'$ are the nested and nester permissions respectively, then after performing a nesting

$$\Pi + (k \rightarrow \rho) \rightsquigarrow (\Pi < k) + (k \rightarrow \rho)$$

the original Π is “gone,” but a nesting fact $\Pi < k$ occurs. Actually, that unit permission (acting as a nester) before and after the nesting action are different. It is “enlarged” to contain the newly nested one. The first transfer step in Figure 2 shows this phenomenon.

Once a nesting happens, the nested permission is no longer directly present any more. It has to be *carved out* if needed. The carving process is performed as

$$\Gamma + (k \rightarrow \rho) \equiv \Gamma + \Pi + \Pi \dashv (k \rightarrow \rho)$$

where Γ should include the nesting fact $\Pi < k$, otherwise it is not sound.

Definition 3.2 (Permission Carving). *A nested permission Π is able to be carved out from its nester permission $k \rightarrow \rho$ for some ρ assuming the nesting fact $\Pi < k$ exists and Π has not been carved out yet.*

It is worthwhile to emphasize that the carving is not just an opposite nesting process. As mentioned before, the nester permission is “enlarged” once a nesting happens, but it cannot shrink after the nested permission is removed. Instead, there will be a “hole” expressed by a linear implication. Thus, the nesting is not undoable and the carving cannot be considered as an *anti*-nesting action. This is the reason we use an one-way arrow “ \rightsquigarrow ” to express a nesting in Figure 2. Nevertheless, there does exist an *anti*-carving action: the carved out nested permission can be filled back into the implication form of its nester. The “ \equiv ” in Figure 2 demonstrates that round-way property.

Nesting (with carving) is one of the hallmarks of the permission system. Permission is used to permit operations in programs and it may be considered as some kind of resources, therefore a permission nesting is to say that one resource is logically protected by another. This is analogous to the protection introduced by ownership types.

3. Ownership Types

Ownership is first proposed to control aliasing in object-oriented programs [11, 10, 2] and it is then extended to multithreaded programs focusing on data-race detection [3, 1].

3.1. Owners-as-dominators

In order to manage and control inter-object aliases, researchers have developed several techniques including uniqueness and object encapsulation. Ownership is originally proposed to implement the core encapsulation mechanism of flexible alias protection [11]. With ownership, each regular object must have another object or a special global “world” as its owner and may own zero or more objects as well. The ownership relation is required to be acyclic, and hence forms an ownership hierarchy tree, where the root is the “world.” The owner of an object is often fixed once the object is instantiated completely.

Ownership provides a statically enforceable way of specifying object encapsulation: any reference to an object is demanded to pass through that object’s owner [10]. This property prevents any direct access to an object other than through its owner. In this model, any object acting as an owner is often considered to be a dominator for all the owned objects, so it is called *owners-as-dominators*.

```
class LinkedList<thisowner> {
    ListNode<this,world> head;
    void insert(Object<world> d) {
        head=new ListNode<this,world>(d,head);
    }
    .....
class ListNode<thisowner,d_owner> {
    ListNode<thisowner,d_owner> next;
    Object<d_owner> datum;
    ListNode( Object<d_owner> d,
              ListNode<thisowner,d_owner> n) {
        next = n;
        datum = d;
    }
    .....
}
```

Figure 3. Owners-as-dominators.

Figure 3 gives a code segment of this ownership

model using a Java-like object-oriented language. Every class definition has a sequence of ownership parameters $\langle \text{thisowner}, \text{otherowner}^* \rangle$, such that thisowner will act as the direct owner of this class instance, and otherowner^* are passed-in ownership parameters that may occur in field or method definitions. For example, class `ListNode` includes two ownership parameters, `thisowner` and `d_owner`, where they act as the owners for this class instance and the pointed-to object by the datum field respectively. When such an instance is created inside of `insert`, two actual parameters `this` and `world` are passed in to substitute the corresponding formal parameters `thisowner` and `d_owner` respectively. Then the newly created `ListNode` object is owned by the receiver of the current `insert` method, while the object referred to by its `datum` field is owned by the global “world.” It matches the ownership (as well as primitive type) requirement at left hand side of this assignment.

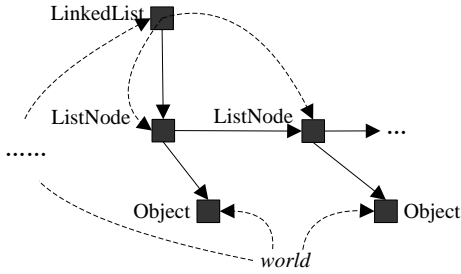


Figure 4. A feasible ownership hierarchy

A possible ownership relation (tree) for this code segment is demonstrated in Figure 4 assuming the list contains at least two nodes. Each dark rectangle represents an object and a solid arrow shows an explicit reference between two objects. A dashed arrow indicates the source owns the sink object. Briefly, we use a “.....” in the ownership arrow from the “world” to that `LinkedList` object to show that the former either directly or indirectly owns the latter.

The ownership property prevents any reference to an object that does not pass through its owner, which basically indicates an encapsulation relation. This, however, overlaps with the nesting concept for permission. A permission to access the state of an object is “owned by” or “nested into” a permission to access the state of its owner, thereby the former is not achievable unless the latter is granted. Formally, it is written as a nesting fact in permission:

$$r_{\text{this}}.\text{All} \prec r_{\text{thisowner}}.\text{Owned} \quad (1)$$

assuming r_{this} and $r_{\text{thisowner}}$ are two objects where the latter owns the former at the same time. “All” and “Owned” are used as special data groups, such that the former is the whole state of an object while the latter holds the state of objects owned by the current object. Following the requirement of *representation containment* for ownership [11], the

state of an owned object is considered as part of its owner’s state, thus another nesting fact is born naturally:

$$r_{\text{this}}.\text{Owned} \prec r_{\text{this}}.\text{All} \quad (2)$$

Once a class is instantiated into an object, these two nesting facts are built innately and are often included in a *class invariant* with regard to a sequence of ownership parameters, where the class invariant is expressed as a named predicate whose body is a conjunction of nesting facts and a primitive type assertion. Assuming r_{this} , $r_{\text{thisowner}}$ and $r_{\text{d_owner}}$ are three object variables representing the `this` object, two actual ownership parameters for `thisowner` and `d_owner` respectively, then the class invariant for `ListNode` is defined as:

$$\begin{aligned} \text{ListNode}(r_{\text{this}}, r_{\text{thisowner}}, r_{\text{d_owner}}) = & \\ & r_{\text{this}}.\text{All} \prec r_{\text{thisowner}}.\text{Owned} \\ \wedge & r_{\text{this}}.\text{Owned} \prec r_{\text{this}}.\text{All} \\ \wedge & \Gamma_{\text{next}} \\ \wedge & \Gamma_{\text{datum}} \\ \wedge & r_{\text{this}} \in \text{ListNode} \end{aligned}$$

where Γ_{next} and Γ_{datum} are *unary field invariants* interpreting ownership relations for fields `next` and `datum` respectively. They are object instance invariants involving one field at a time.

$$\begin{aligned} \Gamma_{\text{next}} &= \exists r_n . (r_{\text{this}}.\text{next} \rightarrow r_n + \Pi_n) \prec r_{\text{this}}.\text{All} \\ \Gamma_{\text{datum}} &= \exists r_d . (r_{\text{this}}.\text{datum} \rightarrow r_d + \Pi_d) \prec r_{\text{this}}.\text{All} \end{aligned}$$

and

$$\begin{aligned} \Pi_n &= (r_n = \$0) ? (\emptyset) : (\text{ListNode}(r_n, r_{\text{thisowner}}, r_{\text{d_owner}})) \\ \Pi_d &= (r_d = \$0) ? (\emptyset) : (\text{Object}(r_d, r_{\text{d_owner}})) \end{aligned}$$

Expressed by two nesting facts, unit permissions to access `next` and `datum` are both nested inside of the “All” group of the `this` object. Actually, all fields within an object are considered to be included as the representation of that object. Π_n is a conditional permission showing that if the pointed-to object referred to by `next` is not null, then it also maintains a corresponding class invariant of itself. Similarly, Π_d demonstrates a requirement for the pointed-to object referred to by `datum`.

According to the same principle, a class invariant for `LinkedList` is:

$$\begin{aligned} \text{LinkedList}(r_{\text{this}}, r_{\text{thisowner}}) = & \\ & r_{\text{this}}.\text{All} \prec r_{\text{thisowner}}.\text{Owned} \\ \wedge & r_{\text{this}}.\text{Owned} \prec r_{\text{this}}.\text{All} \\ \wedge & \Gamma_{\text{head}} \\ \wedge & r_{\text{this}} \in \text{LinkedList} \end{aligned}$$

where

$$\Gamma_{\text{head}} = \exists r . (r_{\text{this}}.\text{head} \rightarrow r + \Pi_h) \prec r_{\text{this}}.\text{All}$$

and

$$\Pi_h = (r = \$0)?(\emptyset) : (ListNode(r, r_{\text{this}}, \$0))$$

There are two $\$0$ used above and they have different meanings. The first represents a null pointer, while the second is borrowed as a variable for the “world” since null pointer is always global visible.

Provided that a $r_{\text{thisowner}}.\text{Owned} \rightarrow \0 ² is granted as well as a class invariant $C(r_{\text{this}}, r_{\text{thisowner}}, r_{\text{otherowner}}^*)$, any access (read or write) to a field of r_{this} object is allowed since the access permission can be carved out by several transformation from the unit permission for its owner. We omit the details in this paper.

3.2. Owners-as-locks

In order to avoid data races in multithreaded programs, people often use mutually exclusive locks to protect critical regions where shared objects are accessed. Borrowing the ownership concept, Boyapati et al. [3, 1] modify the previous owners-as-dominators into a new *owners-as-locks* model requiring a context thread to hold an object’s root-owner when that object is accessed. An object’s root-owner is the root node of an ownership hierarchy tree that contains that object. This property guarantees that shared objects cannot be accessed simultaneously in parallel threads and hence avoids possible data races.

In the owners-as-locks model, there is not such a fixed “world” that directly or indirectly owns all other objects. It also breaks the original single-tree ownership hierarchy into a forest. Either a `thisThread` object coming into scope at the beginning of each thread or an object that owns itself could be a root leading an ownership tree independently. Figure 5 shows two ownership trees rooting in a `threadi` object and a self looped object o_1 respectively.

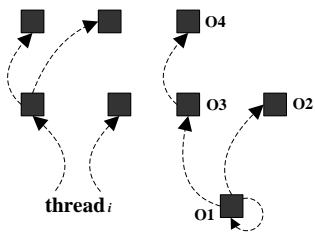


Figure 5. Two feasible ownership hierarchy.

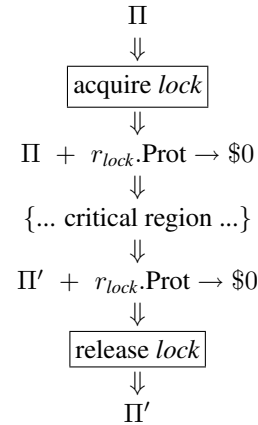
For a multithreaded language without synchronization, the permission reasoning works in a way straightforwardly: the incoming permission is divided into several parts and each part goes into a parallel thread [4]. Combining multithreading with synchronization challenges the traditional permission reasoning. A feasible solution is to think about

²All data groups are given an uninteresting type $\$0$.

permission as some sort of resources, a lock acquisition “borrows” some permissions from an invisible “neutral” thread that holds permissions to access shared objects, while a lock release “returns” them back. Statically, a lock object intends to protect some shared objects. A lock acquisition operation, therefore, “grants” a permission to access a special state of the lock object which is supposed to be “taken back” once a lock release happens later on. That special state of an lock object (represented as an implicit “Prot” data group within permission) basically owns those shared state the lock object is trying to protect.

The owners-as-locks model assures only the root of an ownership tree can be used as a lock which is either a self-owned object or a thread object, whereas the latter is considered to be directly acquired when that thread starts.

Besides the previous “All” and “Owned” groups, we further introduce a “Prot” group for each lock object to contain all state “protected” by it, such that a unit permission to the “Prot” group of an lock object is not available unless that lock is held by the context thread, namely, inside of a synchronized block holding that lock. A simplified permission checking is demonstrated in below, where the r_{lock} is a variable representing the *lock* object.



Assuming a synchronization expression is able to be decomposed into an execution sequence including a lock acquisition, a critical region and a lock release in turn and it is granted an input permission Π originally³, then after executing the lock acquisition, a $r_{\text{lock}}.\text{Prot} \rightarrow \0 is newly granted in the scope. If the critical region is perfectly happy with that, it will give out an output permission compelling that $r_{\text{lock}}.\text{Prot} \rightarrow \0 is still there, since it has to be taken back by the following release operation.

The critical region actually is able to access any shared object that protected by the lock provided those access permissions are “owned by” (or “nested into”) the Prot group of the lock object. Therefore, the permission interpretation

³In fact, an additional requirement is that the Π does not include the $r_{\text{lock}}.\text{Prot} \rightarrow \0 .

for an ownership protection in the owners-as-locks model is not only the classical

$$r_{\text{this}}.\text{Owned} \prec r_{\text{this}}.\text{All} \quad (3)$$

but also

$$(r_{\text{thisowner}} = r_{\text{this}})?(r_{\text{this}}.\text{All} \prec r_{\text{this}}.\text{Prot}) : (r_{\text{this}}.\text{All} \prec r_{\text{thisowner}}.\text{Owned}) \quad (4)$$

This conditional permission distinguishes two possibilities: whether or not the `thisowner` is the `self` object. If an object's owner is itself (the object is a root-owner), then state owned by the object is totally protected by the object itself. Figure 6 demonstrates a permission nesting relation for the ownership relation tree rooting in o_1 at Figure 5 based on (3) and (4). The outmost rectangle represents a unit permission for $o_1.\text{Prot}$, into which a unit permission for $o_1.\text{All}$ is nested. Moreover, $o_1.\text{All}$ includes $o_1.\text{Owned}$ which further includes both $o_2.\text{All}$ and $o_3.\text{All}$ and so on. For example, once o_1 is acquired, a unit permission $o_1.\text{Prot} \rightarrow \0 is granted followed by $o_1.\text{All} \rightarrow \0 , $o_2.\text{All} \rightarrow \0 , $o_3.\text{All} \rightarrow \0 and $o_4.\text{All} \rightarrow \0 , since the latter four can be carved out from the former by some transformation. In other words, o_1 as well as o_2 , o_3 and o_4 are able to be accessed if o_1 is held by the context thread.

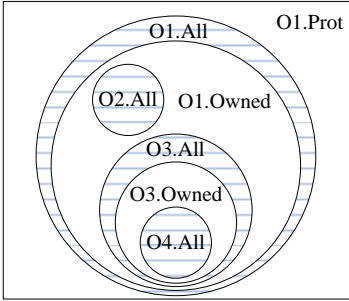


Figure 6. An ownership by permission

For the code segment in Figure 7, a method annotation “requires (e)” indicates the root-owner of that e is required to be held at the method entry. That is to say, the $r_{\text{root}}.\text{Prot} \rightarrow \0 is available, where $\text{isRootOwner}(r_{\text{root}}, r_e)$ is true. Since r_{root} owns r_e either directly or indirectly, $r_e.\text{All} \rightarrow \0 will also be available at the method entry as well. Formally, this is interpreted as a permission carving (transformation):

$$\begin{aligned} & \text{isRootOwner}(r_{\text{root}}, r_e) + r_{\text{root}}.\text{Prot} \rightarrow \$0 \\ & \equiv \text{isRootOwner}(r_{\text{root}}, r_e) + r_e.\text{All} \rightarrow \$0 \\ & \quad + r_e.\text{All} \rightarrow \$0 - r_{\text{root}}.\text{Prot} \rightarrow \$0 \end{aligned}$$

where

- either the $r_e.\text{All} \prec r_{\text{root}}.\text{Prot}$ exists, then $\text{isRootOwner}(r_{\text{root}}, r_e)$; or
- there exists a r , such that $r_e.\text{All} \prec r.\text{Owned}$ and $\text{isRootOwner}(r_{\text{root}}, r)$, then $\text{isRootOwner}(r_{\text{root}}, r_e)$.

```
class Account<thisowner> {
    int balance = 0;
    void deposit(int x) requires (this) {
        balance += x;
    }
    int getbalance() requires some (this) {
        return balance;
    }
    .....
}

Account<thisThread> a1=
    new Account<thisThread>();
Account<self> a2 = new Account<self>();

a1.deposit(100);
fork {synchronized a2 {a2.deposit(100);}}
fork {synchronized a2 {a2.getbalance();}}
```

Figure 7. Owners-as-locks.

If an object is owned by a thread object (directly or transitively), then it is local to that thread and cannot be accessed by any other thread in parallel. In our permission system, we treat this paradigm as: a thread object is always “pre-acquired” when that thread starts. This is to guarantee the special $r_{\text{thread}_i}.\text{Prot} \rightarrow \0 is “pre-granted” for each thread i . Let’s take the code in Figure 7 for example, the `a1` is instantiated to be an `Account` instance owned by the main thread (numbered 0) object. Thereafter, the prerequisite “requires (`this`)” for calling `a1.deposit(100)` is satisfied, since a $r_{\text{thread}_0}.\text{Prot} \rightarrow \0 is granted in advance and $\text{isRootOwner}(r_{\text{thread}_0}, r_{a1})$ is true. Note that the “`this`” occurring in that annotation is substituted as its receiver `a1` and its root-owner is the main thread object certainly.

`a2` is a self-owned class instance and it is able to be shared among threads in parallel, but any access to it should make sure itself (as a lock object) is held by its context thread. The two `fork` expressions create two parallel threads, both of which try to access `a2` by calling a method of it. Since both invocations are in a synchronized block holding the `a2`, also the root-owner of itself, they both satisfy the method prerequisite introduced by annotation.

4. Discussion

Both owners-as-dominators and owners-as-locks focus on building a protection hierarchy among objects. They use annotated static types to group objects and construct them as a tree (or forest), such that a parent node (owner object) always takes the responsibility for its children nodes (ownee

objects) in some sense. Ownership type systems are, therefore, more object-centric and abstract.

Permissions are tightly bound with operations in a program. They are more like resources. Provided you carry some permissions, you are allowed to execute some operations correspondingly. Different operations require the current process (or thread) to carry different permissions, otherwise the program is not well permission typed. The permission system is operation-centric and concrete, and hence it is easier to be formalized and used to verify many program properties, such as ownership, uniqueness, nullity and so on.

4.1. Fractional Permissions

A unit permission permits field accesses, but it does not distinguish reads from writes. Obviously, reads and writes play totally different roles, especially in multithreaded programs. For instance, a non-synchronized write operation in one thread together with any access (either read or write) in its parallel siblings may cause a race problem, but multiple reads simultaneously are safe and do not need to be protected by mutually exclusive locks. Although the owners-as-locks fits well for the former case, it has nothing to do with the latter one. Permission, however, is a promising way to solve the problem assuming we are able to distinguish a read permission from its counterpart.

The *fractional permission* is proposed to scale a permission by a modifier ξ which is a positive fraction (rational number) from zero exclusively to one inclusively. Fractions allow one to split a permission and merge permissions that only differ from fractions, for example:

$$k \rightarrow \rho \equiv \frac{1}{2}k \rightarrow \rho + \frac{1}{2}k \rightarrow \rho$$

A unit permission $k \rightarrow \rho$ is split into two fractional unit permissions, both of which are associated with a fraction “ $\frac{1}{2}$ ”. If it is split further, we may get other fractional permissions, but with some smaller fractions:

$$\frac{1}{2}k \rightarrow \rho \equiv \frac{1}{4}k \rightarrow \rho + \frac{1}{4}k \rightarrow \rho \equiv \dots$$

A unit permission permits both writes and reads, but a fractional unit permission only allows reads if the attached fraction is known to be less than one. This technique principally handles multiple reads in parallel threads by splitting a unit permission into several fractions and distributing them to different parallel threads [4]. The capability of distinguishing reads from writes is one of permission’s most important advantages over previous type systems.

4.2. Multiple Ownership

Most existent ownership systems are required to be single ownership. In other words, no object can have more than

one owners and the ownership hierarchy only forms a tree (or forest) structure. This is too restrictive and cannot model many program paradigms. For instance, a student may join more than one associations and he (or she) can then be considered to be “owned” by those associations simultaneously. Furthermore, an employee may be assigned a series of tasks which may belong to different projects, then each task object may have two direct owners: an employee object and a project object at the same time. The phenomenon that one belongs to (or depends on) more is totally natural, so we cannot blindly avoid multiple ownership in a program.

Cameron et al. propose to extend single ownership to multiple ownership and they provide a solution based on a Java-like object-oriented language [9]. As mentioned in their paper, constructing the ownership relation is likely to put objects into boxes: each owner object is associated a box containing its ownee objects. For any two boxes, single ownership requires that either they are totally disjoint or one is completely nested in another (directly or indirectly), while multiple ownership allows boxes to overlap with each other and the objects occurring in the overlapped area have multiple owners since they belong to multiple boxes coinstantaneously.

Syntactically, every class definition may then have a new form of ownership parameters: $\langle \text{thisowner}^+; \text{otherowner}^* \rangle$, such that thisowner^+ is a sequence of formal ownership parameters that will own the current class instance at the same time. See the code segment in Figure 8 for an example. The `Task` class is defined as having two formal ownership parameters for any instance of itself. When object `t` is created, it then has two direct owners: an `Employee` object `e` and a `Project` object `p`.

```
class Task<thisowner1, thisowner2; t_owner> {
    Time<t_owner> time;
    void delay() { ... }
    .....
}
class Employee<thisowner> {.....}
class Project<thisowner> {.....}

final Employee<world> e = new Employee<world>;
final Project<world> p = new Project<world>;
.....
Task<e, p, world> t = new Task<e, p, world>;
e.addTask(t);
p.addTask(t);
```

Figure 8. Multiple ownership.

Instead of putting the whole state of an object $r_{\text{this}}.All \rightarrow \0 into its owner $r_{\text{thisowner}}.Owned$ in the single ownership model (as shown in (1) in page 4), we split

an object’s state and nest them into multiple owners with the help of fractional permission. For example, the class invariant for `Task` could be:

$$\begin{aligned} \text{Task}(r_{\text{this}}, r_{\text{thisowner1}}, r_{\text{thisowner2}}, r_{\text{towner}}) = & \\ & \frac{1}{2}r_{\text{this}}.\text{All} \prec r_{\text{thisowner1}}.\text{Owned} \\ & \wedge \frac{1}{2}r_{\text{this}}.\text{All} \prec r_{\text{thisowner2}}.\text{Owned} \\ & \wedge r_{\text{this}}.\text{Owned} \prec r_{\text{this}}.\text{All} \\ & \wedge \Gamma_{\text{time}} \\ & \wedge r_{\text{this}} \in \text{Task} \end{aligned}$$

In this case, any `Task` object has two owners and each owner takes *half* responsibility for it. From the permission’s point of view, either of the two owner holds half access permission for that object and only permits read accesses independently, but a combination of those two certainly permits write accesses.

$$\begin{aligned} r_{\text{to1}}.\text{Owned} \rightarrow \$0 + r_{\text{to2}}.\text{Owned} \rightarrow \$0 + \\ \text{Task}(r, r_{\text{to1}}, r_{\text{to2}}, r_{\text{to}}) \\ \equiv r.\text{All} \rightarrow \$0 + \\ \frac{1}{2}r.\text{All} \rightarrow \$0 + r_{\text{to1}}.\text{Owned} \rightarrow \$0 + \\ \frac{1}{2}r.\text{All} \rightarrow \$0 + r_{\text{to2}}.\text{Owned} \rightarrow \$0 + \\ \text{Task}(r, r_{\text{to1}}, r_{\text{to2}}, r_{\text{to}}) \end{aligned}$$

Different $\frac{1}{2}r.\text{All} \rightarrow \0 nested in different owner locations can be carved out and combined into a unit permission $r.\text{All} \rightarrow \0 which permits write accesses to the object r .

There is another form of multiple ownership requirement that may happen in multithreaded programs with synchronization. In the owners-as-locks model, the whole state of an object has to be protected by a single lock (its root-owner). This is a little bit restrictive. The permission system, however, can do better with more field annotations. A programmer may specify different guards that protect different parts of an object’s state by attaching “`guarded_by(...)`” to field (or data group) definitions, such as:

```
class C<thisowner; g> {
  int f1 guarded_by (g);
  int f2;
  . . . . .
```

In this case, any access to field `f1` is guarded by a formal class parameter `g` (often called *ghost variable* [12]). Therefore, the whole state of the object is split into two parts owned by two different objects.

$$\begin{aligned} C(r_{\text{this}}, r_{\text{thisowner}}, r_g) = & \\ & r_{\text{this}}.\text{All} \prec r_{\text{thisowner}}.\text{Owned} \\ & \wedge \Gamma_{f1} \\ & \wedge \Gamma_{f2} \\ & \wedge r_{\text{this}} \in C \end{aligned}$$

where

$$\begin{aligned} \Gamma_{f1} &= r_{\text{this}}.f1 \rightarrow \text{int} \prec r_g.\text{Prot} \\ \Gamma_{f2} &= r_{\text{this}}.f2 \rightarrow \text{int} \prec r_{\text{this}}.\text{All} \end{aligned}$$

This permission representation shows that different fields in one object may be assigned different nester locations, and hence protected by different owners.

5. Conclusions

In this paper, we show how to use permissions to interpret two kinds of ownership types: owners-as-dominators and owners-as-locks. Permission nesting is the core technique that simulates the ownership relation. We also discuss the possibilities of using the fractional permission to interpret multiple ownership.

References

- [1] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02*, pages 211–230. ACM Press, Nov. 2002.
- [2] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL '03*, pages 213–223, New York, NY, USA, 2003. ACM Press.
- [3] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. *SIGPLAN Not.*, 36(11):56–69, 2001.
- [4] J. Boyland. Checking interference with fractional permissions. In *SAS '03*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
- [5] J. Boyland. Why we should not add `readonly` to Java, yet. In *7th ECOOP Workshop on Formal Techniques for Java-like Programs*, 2005.
- [6] J. Boyland. Semantics of fractional permissions with nesting. Technical report, University of Wisconsin–Milwaukee, Department of EE & CS, 2007.
- [7] J. Boyland, W. Retert, and Y. Zhao. Iterators can be independent “from” their collections. In *ECOOP 2007 Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, 2007.
- [8] J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *POPL '05*, pages 283–295, New York, NY, USA, 2005. ACM Press.
- [9] N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple ownership. In *OOPSLA '07*, Oct. 2007.
- [10] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Australia, 2001.
- [11] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98*, pages 48–64, New York, NY, USA, 1998. ACM Press.
- [12] C. Flanagan and S. N. Freund. Types-based race detection for Java. In *PLDI '00*, pages 219–232. ACM Press, 2000.
- [13] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98*, pages 144–153. ACM Press, Oct. 1998.