Roles for Owners

– Work in Progress –

Dave Clarke
Katholieke Universiteit Leuven
Belgium
dave.clarke@cs.kuleuven.be

Sophia Drossopoulou
Imperial College London
United Kingdom
s.drossopoulou@imperial.ac.uk

James Noble
Victoria Univ. of Wellington
New Zealand
kjx@ecs.vuw.ac.nz

ABSTRACT

Ownership types were proposed to characterize the topology of objects on the heap. They notionally organize objects into boxes, and each box belongs to an object – its owner. In most ownership-based systems, the box *protects the object from its environment* in some way. Thus, the owner may play the owners-as-dominators role, whereby the owner restricts access to the objects in a box, (i.e. the "outside" may not access the "inside"), or, the owner may play the owners-as-modifiers role, whereby the owner restricts modification of the objects in a box (i.e. the "outside" may not modify the "inside").

We propose the dual protection, whereby the box *protects the environment from the object.* We suggest two further roles: in *owners-as-restrictors*, the owner restricts access from the object (i.e. the "inside" may not access the "outside"), and in *owners-as-filters*, the owner restricts the range of modifications from an object (i.e. the "inside" may not modify the "outside").

We explore the design space for possible exact meanings for the four roles. We define the meanings of these roles in terms of the guarantees they make about the heap and about executions.

We sketch parts of a language which supports all four roles, and which allows any of the owner parameters to play any of these four roles. These roles may be enforced statically, dynamically, or via a combination of both.

1. INTRODUCTION

Ownership types were first suggested in 1998 to characterize aliases, and thus control the topology of objects on the heap [15]. Several brands and variations have been proposed since, and the model has been put to different uses, e.g. memory management [19, 16], encapsulation [5], effect systems [7, 6], avoidance of race conditions and deadlock [4, 3], locations [18], parallel programming [2], and program verification [14].

In most ownership systems, objects belong to boxes, and each box belongs to an object; but note that boxes may belong to several objects [17] or objects may belong to several boxes [6].

Furthermore, in most ownership systems, owners *protect* the object from its environment. Thus, the owners were assigned roles such as *owners-as-dominators* [8] whereby the owner controls all accesses to an object (i.e. the "outside" may not access the "inside"), or *owners-as-modifiers* [14] whereby the owner controls all modifications of an object (i.e. the "outside" may not modify the "inside"), or *owners-as-articulation-points* [17] whereby several objects sit at the boundary of the box.

Thus, so far, ownership types have used boxes to provide *protection* to the objects *inside the box* from the objects outside the box. The dual approach uses boxes to protect the objects *outside the box* from the object inside the box.

We propose two new roles for owners: *owners-as-restrictors*, whereby the owner restricts access to the environment from the object (i.e. the "inside" may not access the "outside"), and *owners-as-filters*, whereby the owner restricts the range of modifications to the environment from an object (i.e. the "inside" may not modify the "outside").

Furthermore, we propose that *one* single language can use owners to support all four roles. Namely, ownership determines the boxes (or, topology), and each owner may be assigned one or more of these roles. To our knowledge, this is the first proposal for *one* language to support more than one roles.

The roles of filter and restrictor have practical counterparts in systems where capabilities prevent untrusted applications from interfering with each other [11]. For example, capabilities may be used in Javascript to protect the integrity of the `dom` from the actions of mashups.

The contributions of our current work are: the identification of the new roles, the incorporation of several roles in one language, and an exploration of possible precise meanings for the roles. The design space for the new roles is wide; we need to undertake case studies in order to settle that question. Therefore, we leave a concrete proposal for the new roles, evaluation, full model, and static enforcement to future work.

This paper is organized as follows: In Section 2 we sketch a motivating example. In Section 3 we explore the design space for guarantees given by the four roles. In Section 4 we argue that even though we do not yet know the exact meaning of the roles, it is possible to design a static type system to enforce their guarantees. In Section 5 we discuss how we overcame the challenges in supporting several roles in one language, and the possible relations between roles and topology. Section 6 concludes.

2. MOTIVATING EXAMPLE

As shown in Figure 1, we assume the existence of the following objects: poker and biotronic are Games, object gamebox is a GameBox, object notifier is a Notifier used to send notifications to friends, arbitrator is an Arbitrator used to arbitrate in disputes among friends, activities is an Activities object, friendTable holds a lookup table for friends, and theFaceBook is the object representing Facebook. The "inside" relation is described through dotted rounded boxes; thus, poker and biotronic are inside the gamebox, while gamebox, notifier and arbitrator are within activities, and finally, activities and friendTable are within theFaceBook.

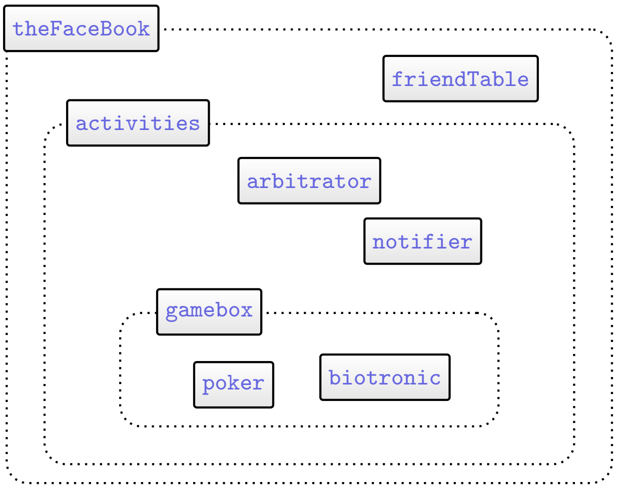

Figure 1: Illustration of the Facebook Example.

We now define the roles as follows:

- activities, notifier, and friendTable should not read or modify the contents of the games, therefore gamebox is the dominator as well as the modifier of poker and biotronic.
- The games should not read nor modify the contents of the friends look-up table; therefore activities is the restrictor and also the filter of poker and biotronic.
- notifier may send notifications to friends; therefore, its filter is theBook. On the other hand, notifier should not modify the friendship relation; therefore its restrictor is activities.
- arbitrator is meant to arbitrate between friends, and therefore affect their friendship status; therefore its restrictor as well as its filter is theFaceBook.

With this approach, the "inner objects" may not directly read or modify the "outer objects", but may do so indirectly, by asking the objects which have the corresponding capabilities. For example, when Sophia reaches a high score on Biotronic, the object biotronic sends a message to notifier, which in its turn reads friendTable, and sends the appropriate notifications. On the other hand, when James cheats Dave in poker, the object poker sends a message to arbitrator which will determine how this affects their friendship, and cause any necessary updates in the contents of friendTable.

3. SKETCH OF THE LANGUAGE $\mathcal{J}ro$

We give an overview of our approach by sketching a little language called $\mathcal{J}ro$, which stands for a **J**ava-like language with different **r**oles for the **o**wners.

$$
\begin{array}{lcl}
\mathit{ClassDecl} & ::= & \texttt{class}\ \mathit{ClassId}\langle\overline{o}\rangle\ \texttt{where}\ \overline{\mathit{Constr}}\ \overline{\mathit{Role}} \\
 & & \quad \{\ \overline{\mathit{FieldDecl}}\ \overline{\mathit{MethDecl}}\ \} \\
\mathit{Constr} & ::= & o \lesssim o' \\
\mathit{Type} & ::= & \mathit{ClassId}\langle\overline{oa}\rangle \\
oa & ::= & o \mid \texttt{this} \mid \texttt{top} \mid \texttt{bot} \\
\mathit{Role} & ::= & o : \mathit{qual} \\
\mathit{qual} & ::= & \texttt{dominator} \mid \texttt{modifier} \\
 & \mid & \texttt{restrictor} \mid \texttt{filter}
\end{array}
$$

Figure 2: $\mathcal{J}ro$ syntax – extracts

3.1 Syntax, roles and the inside relation

As in traditional ownership type systems, a class has one or more ownership parameters, each of which stands for an object at run-time. The first owner parameter is special, in that the object is *directly* inside it—in other words, this parameter denotes *the owner*. The remaining owner parameters are not in any particular topological relation. In particular, in contrast to early ownership systems, we do not expect the owner to be inside the other owner parameters. We do, however, allow the expression of constraints on the owner parameters as to their relative positions, where the "is inside" relation is expressed through the $\lesssim$ symbol.

Furthermore, each class declaration must indicate which roles each owner parameter plays. Here we differ from earlier ownership systems, where only one role was available throughout the system, and this role was played implicitly by the owner. $\mathcal{J}ro$ allows a role to be played by any of the owner parameters of the class, and one owner parameter to play several different roles for the same object.

Part of the syntax for class declarations is given in Figure 2. An example appears in Figure 3, where class A has five owner parameters, and where o2 is expected to be inside o3 and o4. In A the owner parameter o2 is the dominator, o3 is the modifier as well as the restrictor, and o5 is the filter.

```
class A<o1, o2, o3, o4, 05 >
    where o2 < o3, o2 < o4
    o2:dominator, o3:modifier,
    o3:restrictor, o5:filter
{     ...    }
```

Figure 3: Declaration of class A

The syntax of types follows previous work in the standard way: a type consists of a class name followed by the ownership arguments. These are either owner parameters which are in scope (o), or the current object (this), or the most enclosing object (top), or the most enclosed object (bot). Types are legal only if the actual owner parameters satisfy all constraints mentioned in the class declaration. For example, the type `A<o6,o7,o8,o9,this>` is legal only if `o7` is inside `o8` and `o9`. This follows previous work in the standard way. We do not consider subclasses; these would be handled in the standard way [7].

We will also define functions $\mathcal{O}$, $\mathcal{D}$, $\mathcal{M}$, $\mathcal{R}$ and $\mathcal{F}$ which, when applied to a type return the immediate owner, dominator, modifier, restrictor, or filter of that type. For our example we will have that $\mathcal{F}(\texttt{A} < \texttt{o6}, \texttt{o7}, \texttt{o8}, \texttt{o9}, \texttt{this} >) = \texttt{this}$.

The "is inside" relation is defined as the reflexive, transitive closure of the relation "has owner". The "is inside" relation depends on the heap, so for a heap H, and object addresses ι and ι' we will have a judgement $H \vdash \iota' \lesssim \iota$. Similarly, the dominators, modifiers, filters and restrictors of addresses depend on their runtime types; thus we will have lookup functions $\mathcal{D}_H(\iota)$, $\mathcal{M}_H(\iota)$, $\mathcal{R}_H(\iota)$ and $\mathcal{F}_H(\iota)$.

An example can be seen in Figure 4, where the squares represent objects, i.e. we have objects `1`, `2`, `3` etc. Also, the dotted rounded boxes indicate the direct owner relation, e.g. `1` owns `2` and `8`, `2` owns `3` and `7`, etc. Therefore, `5` is inside `4`, `3`, `2`, and `1`.

3.2 Formal prerequisites

In order to define the runtime guarantees in a precise manner, we will need to talk about the most current receiver on a stack of method frames. To do that, we need to be able to distill out of a runtime configuration the sequence of stack frames involved.

We adapt techniques from Cunningham et al. [9]. We define source expressions, e_s, which allow for method calls, but not for nested method activations, and runtime expressions, e_r, which allow for nested method activations:

$$\begin{array}{rcl} e_s & ::= & x \mid \texttt{this} \mid e_s.\texttt{f} \mid e_s.\texttt{f} = e_s \mid e_s.\texttt{m}(e_s) \\ e_r & ::= & e_s \mid \iota \mid e_r.\texttt{f} \mid e_r.\texttt{f} = e_s \mid \iota.\texttt{f} = e_r \\ & \mid & e_r.\texttt{m}(e_s) \mid \iota.\texttt{m}(e_r) \mid \textbf{frame}\,\sigma\, e_r \end{array}$$

We also define frame-free contexts, $E[\,]$, whereby:

$$E[\,] \quad ::= \quad E[\,].\texttt{f} \mid E[\,].\texttt{f} = e \mid \iota.\texttt{f} = [\,] \mid E[\,].\texttt{m}(e) \mid \iota.\texttt{m}(E[\,])$$

and frames $F[\,]$, which allow for nested method activations:

$$F[\,] \ ::= \ E[\,\textbf{frame}\,\sigma\, F[\,]\,] \mid E[\,].$$

A runtime expression of the form $e_r = F[\textbf{frame}\,\sigma\, e_r{}']$ expresses that one of the currently active method calls is σ. A runtime expression of the form $e_r = F[\textbf{frame}\,\sigma\, E[e_s]]$ expresses that the innermost receiver in the stack of method calls is $\sigma(\texttt{this})$. Finally, a runtime expression of the form $e_r = F[\textbf{frame}\,\sigma\, F'[\textbf{frame}\,\sigma'\, e_r{}']]$ expresses that some time during execution, $\sigma(\texttt{this})$ indirectly called a method on $\sigma'(\texttt{this})$.

3.3 Heap guarantees

Dominators and restrictors make guarantees about the topology of the heap.

Dominators.

The dominators guarantee promises that an object o may point to an object o', only if o is inside the dominator of o'. Note that in classical ownership types [8], the immediate dominator of an object is its owner. However, in $\mathcal{J}ro$, this need not be so. In fact, in Figure 4 the immediate dominator of `6` is `2`, while its owner is `3`. The references from `7` and `5` to `6` are legal, but the reference from `8` to `6` is illegal. This guarantee is described formally in Definition 1.

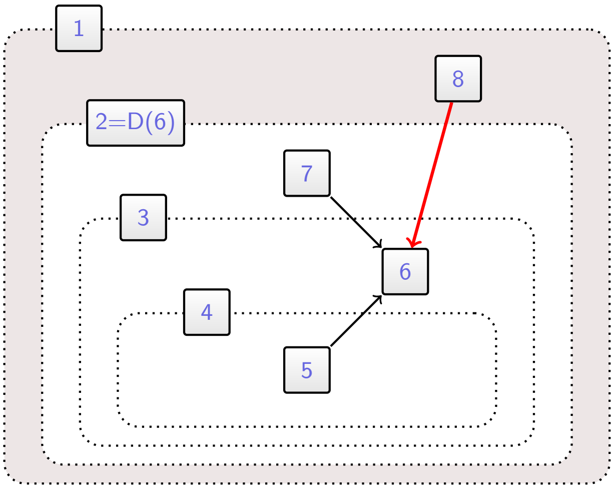

Figure 4: Illustration of the dominators guarantee: the reference from `8` to `6` is forbidden, because `8` is not inside the dominator of `6`.

Definition 1. *A heap H* respects dominators *iff $H(\iota, f) = \iota'$ implies $H \vdash \iota \lesssim \mathcal{D}_H(\iota')$.*

The dominators' guarantee from Definition 1 is local, in that it does not say anything about which paths are possible, and it does not enforce the domination relation in the graph-theoretic sense. For example, if $\mathcal{D}_H(5) = \texttt{1}$, then a path from `8` to `5` to `6` would be legal, even though a direct path from `8` to `6` would be forbidden. A "deeper version" of the guarantee would forbid such paths.

Definition 2. *A heap H* respects dominators deeply *iff $H(\iota, f_1 \ldots f_n) = \iota'$ implies $H \vdash \iota \lesssim \mathcal{D}_H(\iota')$.*

Often, in order to obtain the deep domination guarantee, one requires that the heap respects dominators, and that domination is monotonic with respect to ownership:

Definition 3. *Dominators are monotonic with owners, if $H \vdash \iota \lesssim \iota'$ implies $H \vdash \mathcal{D}_H(\iota) \lesssim \mathcal{D}_H(\iota')$ for all heaps H.*

Furthermore, there are guarantees for method activations: the receiver and arguments of enclosing calls are not inside the receiver of inner calls (such guarantees are crucial for garbage collection):

Definition 4. The calls *of a configuration H, e_r* respect dominators *iff $e_r = F_1[\textbf{frame}\,\sigma_1\, F_2[\textbf{frame}\,\sigma_2\, e_r{}']]$ implies that $H \vdash H(\sigma_1(\texttt{x}_1)) \not\lesssim H(\sigma_2(\texttt{x}_2))$, for any $\texttt{x}_1 \in \mathrm{dom}(\sigma_1)$, $\texttt{x}_2 \in \mathrm{dom}(\sigma_2)$.*

Restrictors.

The restrictors' guarantee promises that an object does not contain references which point outside its restrictor. For example, assume, as shown in Figure 5, that `3` is the restrictor of `5`. Then the reference from `5` to `6` is legal, but the reference from `5` to `7` is illegal. This guarantee is described formally in Definition 5.

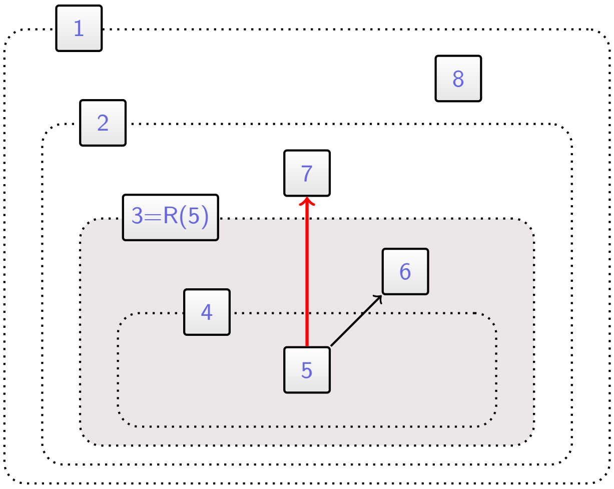

Figure 5: Illustration of the restrictors guarantee: the reference from 5 to 7 is forbidden, because 7 is not inside the restrictor of 5.

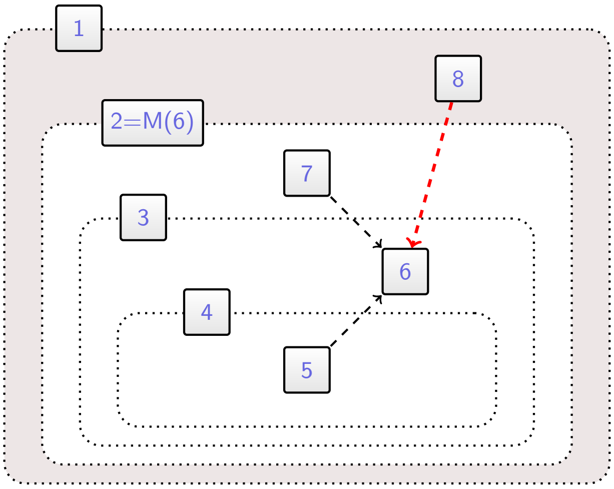

Figure 6: Illustration of the modifiers guarantee: the object 8 may not directly modify the object 6, because 8 is not inside the modifier of 6.

Definition 5. *A heap* respects restrictors *iff $H(\iota, f) = \iota'$ implies $H \vdash \iota' \lesssim \mathcal{R}_H(\iota)$.*

Strengthening the restrictors' guarantee to paths, so that $H(\iota, f_1 ... f_n) = \iota'$ implies $H \vdash \iota' \lesssim \mathcal{R}_H(\iota)$ does not seem useful. Nor do we have a motivation for requiring restrictors to be monotonic with ownership.

3.4 Update guarantees

Modifiers and Filters make guarantees as to who causes updates to an object. We first discuss the meaning of "causes modification of an object": Is it the receiver in *the innermost* execution frame at the point of modification, or just one of the receivers in the execution frames? Both interpretations make sense, and we will allow for both.

Modifiers.

The modifiers guarantee promises that an object o may modify another object o' only if o is inside the modifier of o'. Note that in the classical universes system [14], the modifier of an object is implicitly its owner. However, in $\mathcal{J}ro$, this need not be so. In fact, in Figure 6 the modifier of 6 is 2, while its owner is 3. Thus, 5 and 7 may modify 6, but 8 may not modify 6.

Formal descriptions of this guarantee are given in Definition 6 and in Definition 7, which differ in the exact interpretation of the term "causes the modification of", as discussed above.

Definition 6. *An execution $H, e_r \leadsto H', e_r{}'$* respects modifiers *iff $H(\iota, f) \neq H'(\iota, f)$ and $e_r = F[\mathbf{frame}\, \sigma\, E[e_s]]$ imply $H \vdash \sigma(\mathbf{this}) \lesssim \mathcal{M}_H(\iota)$.*

Definition 7. *An execution $H, e_r \leadsto H', e_r{}'$* weakly respects modifiers *iff $H(\iota, f) \neq H'(\iota, f)$ implies that $\exists$ F, σ, $e_r{}''$, so that $e_r = F[\mathbf{frame}\, \sigma\, e_r{}'']$ and $H \vdash \sigma(\mathbf{this}) \lesssim \mathcal{M}_H(\iota)$.*

The universes system [14] imposes a stronger guarantee than the one from Definition 6, i.e. that only modifiers may cause modifications, and that, except for calls to pure methods, the direct caller of a method on an object is its own modifier.[1]

Definition 8. *An execution $H, e_r \leadsto H', e_r{}'$* respects modifiers in the universes sense *iff*

- *$H(\iota, f) \neq H'(\iota, f)$ and $e_r = F[\mathbf{frame}\, \sigma\, E[e_s]]$ imply $H \vdash \sigma(\mathbf{this}) = \mathcal{M}_H(\iota)$.*
- *$e_r = F[\mathbf{frame}\, \sigma\, E[\mathbf{frame}\, \sigma' e_r{}'']]$ implies that* $\sigma(\mathbf{this}) = \mathcal{M}_{\mathbf{H}}(\sigma'(\mathbf{this}))$

Filters.

The filters guarantee promises that an object o may only modify objects which are inside o's filter. Therefore, in Figure 7, where 3 is the filter of 5, the object 5 may modify 6, but may not modify 7. Depending on the interpretation of the term "causes the modification of" we give a formal description in Definition 9 and stricter version in Definition 10.

Definition 9. *An execution $H, e_r \leadsto H', e_r{}'$* respects filters *iff $H(\iota, f) \neq H'(\iota, f)$ and $e_r = F[\mathbf{frame}\, \sigma\, E[e_s]]$ imply that $H \vdash \iota \lesssim \mathcal{F}_H(\sigma(\mathbf{this}))$.*

Definition 10. *An execution $H, e_r \leadsto H', e_r{}'$* weakly respects filters *iff $H(\iota, f) \neq H'(\iota, f)$ implies that $\exists$ F, σ, $e_r{}''$, so that $e_r = F[\mathbf{frame}\, \sigma\, e_r{}'']$ such that $H \vdash \iota \lesssim \mathcal{F}_H(\sigma(\mathbf{this}))$.*

A very strict version of the guarantee, whereby an object may be modified, only when it is within the filter of all the current callers on the stack, i.e. where $H, e_r \leadsto H', e_r{}'$ and $H(\iota, f) \neq H'(\iota, f)$ and $e_r = F[\mathbf{frame}\, \sigma\, e_r{}'']$ imply that $H \vdash \iota \lesssim \mathcal{F}_H(\sigma(\mathbf{this}))$, would be far too strong. For example, it would prevent the object `arbitrator` from modifying `friendTable` when being called by `poker`.

[1] Note however, that these requirements have been weakened in subsequent work [10].

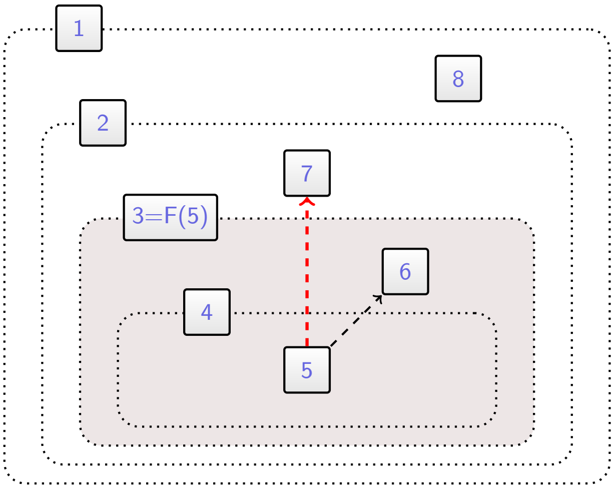

Figure 7: Illustration of the filters guarantee: the object 5 may not directly modify the object 7, because 7 is not inside the filter of 6.

4. TYPE CHECKING

The exact guarantees of the roles are not yet clear, and nor is it clear whether the guarantees should be checked statically or dynamically. Therefore, at the current stage, we do not know enough to design a type system.

Nevertheless, we do know enough to be in a position to argue that such a static type system is possible. In the remainder of this section we will sketch such a type system.

We assume the existence of the usual typing judgement $\Gamma \vdash \mathtt{e} : \mathtt{T}$ giving to e the type T, and a topological judgement $\Gamma \vdash \mathtt{o} \lesssim \mathtt{o}'$ which guarantees that at runtime the object standing for o will be inside the object standing for o′ — such judgements are commonplace in systems which support constraints on ownership parameters, e.g. [7].

The most interesting question is what checks need to be performed upon field assignment and upon method call. Assuming that we wanted to guarantee the properties defined in Definitions 1, 5, 6 and 9, then properties to be checked are outlined in the following rule:

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{x} : \mathtt{T} \qquad \Gamma \vdash \mathtt{y} : \mathtt{T}' \qquad \vdash \mathtt{T}' \leq \mathit{fType}(\mathtt{T},\mathtt{f}) \\ \Gamma \vdash \mathcal{O}(\mathtt{T}) \lesssim \mathcal{D}(\mathtt{T}') \qquad \Gamma \vdash \mathcal{O}(\mathtt{T}') \lesssim \mathcal{R}(\mathtt{T}) \\ \Gamma \vdash \mathtt{this} \lesssim \mathcal{M}(\mathtt{T}) \qquad \Gamma \vdash \mathcal{O}(\mathtt{T}) \lesssim \mathcal{F}(\mathtt{this})\end{array}}{\Gamma \vdash \mathtt{x.f} = \mathtt{y} : \mathtt{T}}$$

In the rule from above, the first three premises are standard. We now explain the remaining four premises:

$\mathcal{O}(\mathtt{T}) \lesssim \mathcal{D}(\mathtt{T}')$ enforces the dominators guarantee (Definition 1): The the object x, i.e. the one which will be holding a reference to y, must be inside the dominator of y.

$\mathcal{O}(\mathtt{T}') \lesssim \mathcal{R}(\mathtt{T})$ enforces the restrictors guarantee (Definition 5): the object to which x will be pointing, i.e. y, must be inside the restrictor of x.

$\mathcal{O}(\mathtt{T}') \lesssim \mathcal{R}(\mathtt{T})$ enforces the modifiers guarantee (Definition 6): the current receiver must be inside the modifier of x.

$\mathcal{O}(\mathtt{T}) \lesssim \mathcal{F}(\mathtt{this})$ enforces the filters guarantee (Definition 9): the object being modified, i.e. x, must be inside the filter of the current receiver.

Interestingly and somewhat surprisingly, the only checks necessary at the point of method call are that the types of the actual parameters match those of the formal parameters.

5. DISCUSSION

Challenges and our Approach.

In the past, we tried to develop a system supporting the modifier as well as the dominator role, but were unable to make progress. Namely, we thought that owners would implicitly pay one of the two roles, and thus, we thought that for some objects the owner would be their dominator, and for others it would be their modifier. We thought that a pluggable type system would help, but had difficulty in determining which protocol (the modifiers or the dominators protocol) to enforce on which objects.

We had also considered the possibility that an object would have two owners: the dominator owner, and the modifier owner; in this case, it is clear which protocol needed to be enforced (namely both), however, we did not know what the topology of the heap should be. Should the modifier owner always be within the dominator owner? If so, why?

We realized that the reason for our difficulties was our expectation that owners would implicitly play certain roles. In this paper, there are no implicit roles for owners. We separate the issue of the topology (ie the box structure), from the role of owners in providing guarantees. We remove the expectation that the direct owner plays any specific role, other than directly containing the object. The class declaration then assigns each of the four roles to one of the owner parameters (not necessarily different ones).

In this sense, our approach is similar to that by Aldrich and Chambers [1], which separates methodology from mechanism; it differs in that their approach supports one role (restriction of links), whereas ours can support any number of roles.

Relations between Roles and Default Roles.

An open question is the relation between the objects playing different roles. For example, it seems sensible to require that the filter should be inside the restrictor and that the modifier should be inside the dominator.

However, it is less clear what, if any inside relation, there should be between the restrictor an the dominator of an object, or even maybe of different objects. Also, should there be some relation between the objects playing a certain role for objects from the same box? Should, for instance, the objects of one box share the same dominator?

Finally, does an object need to be inside the objects playing certain roles for it? If we adopt the ideas from ownership domains [1], then the answer seems to be no. For example, we might want to allow the `arbitrator` to modify the contents of `friendTable`, without giving it the capability to modify other objects within `theFaceBook`, and thus, we might want to make `friendTable` the filter of `arbitrator`, even though `arbitrator` is not inside `friendTable`.

Another pertinent question concerns default values: our system comes with the heavy annotation burden of requiring the provision of four roles per class. It is crucial in such a system to alleviate the annotation burden by supporting sensible default rules. For example, if no dominator has been specified, then it should be the owner, while an unspecified filter defaults to `top`.

We leave all these questions open for further work.

6. CONCLUSIONS AND FUTURE WORK

The main contributions of our work are

- The realization that the remit of owners in describing the heap topology can be separated from the remit in describing protection or capabilities schemes.
- The identification of two new roles for owners, namely restrictors and filters.
- The exploration of the design space for the guarnatees given by the four roles.

Restrictors and filters are of importance when one wants to sandbox some piece of code and protect its environment from its effects. We believe that our ideas are applicable to object-capability systems [13, 12] giving assurance that the code of a particular system implements a given policy.

By separating the remit of owner parameters as a way of describing the heap topology, from its remit in proving capabilities/protection, we have shown how *one* type system can be devised so as to supports four different roles for owners. We expect that more roles could be supported in that way.

We expect that practical systems may adopt only some or a variation of the roles suggested in our paper, however, our work shows how such a combination of the roles can achieved.

More work needs to be done to investigate which guarantees should be enforced, how these can be enforced, what are sensible relations between roles, and how to alleviate the annotation burden.

Acknowledgements We thank Sergio Maffeis for useful discussions about capabilities and Javascript, and the anonymous referees for constructive feedback and comments humorous.

7. REFERENCES

[1] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 1–25. Springer-Verlag, 2004.

[2] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[3] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming.* PhD thesis, MIT, 2004. Available from `pmg.lcs.mit.edu/ chandra/publications/`.

[4] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, Nov. 2002.

[5] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–223, New York, NY, USA, 2003. ACM Press.

[6] N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple Ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 441–460. ACM, 2007.

[7] D. Clarke and S. Drossopoulou. Ownership, Encapsulation and the Disjointness of Types and Effects. In *OOPSLA*, pages 292–310, Seattle, Washington, USA, November 2002. ACM Press.

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, volume 33(10), pages 48–64. ACM Press, 1998.

[9] D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, pages 20–51, 2007.

[10] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.

[11] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc of IEEE Security and Privacy'10*. IEEE, 2010.

[12] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, May 2006.

[13] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Systems Research Laboratory, Johns Hopkins University, 2003.

[14] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[15] J. Noble, J. Potter, and J. Vitek. Flexible alias protection. In *ECOOP*, Brussels, Belgium, July 1998.

[16] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP*, 2007.

[17] J. Vitek and J. Bokowski. Confined types for java. *Software Partice and Experience*, 2001.

[18] Y. Welsch and J. Schäfer. Location types for safe distributed object-oriented programming. In *TOOLS*, 2011.

[19] T. Zhao, J. Baker, J. Hunt, J. Noble, and J. Vitek. Implicit ownership types for memory management. *Science of Computer Programming*, 71(3):213–241, 2008.