Formalizing Composite State Encapsulation

Adrian Fiech
Memorial University, St. John's, NL, A1B 3X5, Canada
afiech@cs.mun.ca

Ulf Schünemann
3S-Smart Software Solutions GmbH, Kempten, Germany
u.schuenemann@3s-software.com

Abstract

Representation exposure is a well documented and studied problem in object-oriented systems. We introduce the *Potential Access Path* methodology as a tool to reason about composite objects and protection of their representation. Our system enforces the owner-as-modifier disciplin, which does not restrict aliasing but requires that all modifications to an encapsulated aggregate are initiated by the aggregate's owner. A novel design choice in our system is the `free` mode that allows read-only aliases. This new weak uniqueness property provides us with additional flexibility to transfer subcomponents from one aggregate to another.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.1.5 [*Programming Techniques*]: Object-oriented Programming

General Terms Languages, Security, Theory

Keywords Alias protection, Representation exposure, Ownership types, State encapsulation, Java

1. Introduction

The recursive combination of smaller objects to one composite object (object composition) is a central technique in the construction of object-oriented software. The encapsulation of such composite objects is an important criterion for the quality of object-oriented designs. A lack of encapsulation makes the composite object's correct functioning depend on its context – its implementation cannot be verified in a modular way and cannot be safely reused in new contexts.

When discussing object-oriented software systems one often considers three related notions of the object concept. At the base-level, the system is a flat "sea" of elementary ***implementation objects*** – instances of concrete classes. At the top level we have ***abstract objects***, which are defined solely by their operations' externally visible behaviour (e.g described with the concept of interfaces in Java along with some behavioural specification). The implementation of this behaviour is delegated to the in-between level – structures of collaborating implementation objects rooted in a ***representative***. That representative, aided by its collaborators, provides the desired functionality specified in the interface. This cluster of cooperating objects will be referred henceforth as the ***composite object*** (or the ***aggregate***).

An abstract object's invariant – specified in the behavioural component of the interface – may depend on the internal structure of the composite object. In general, object-oriented languages do not prevent "outsiders" from obtaining references to the internal structure. Such exposure of the internal representation can lead to mutation of the structure while the representative object remains completely oblivious to the changes. The invariant may be violated and the implementation of the abstract object might behave inconsistent with its specification.

Among the first attempts to address the perils of representation exposure are [11] and [3]. Here the composite object is fully encapsulated and neither incoming nor outgoing references are allowed. The absence of incoming references guarantees that any modifications to the internal structure of the composite object must be triggered through the representative object's interface. Unfortunately the full encapsulation is well too restrictive and many common object-oriented idioms are impossible to implement in such systems.

The ownership type (***OT***) system introduced in [8] relaxes the restriction on the outgoing references. Each object is treated as a representative of a certain composite object, which is owned by the representative. Which objects constitute the owned composite object is specified by program annotations (ownership contexts). A tree-like ownership structure is established among the run-time objects and the system satisfies the owner-as-dominator property: All reference chains from the root object to any other object *o* (thus any reference chain leading inside the composite object to which o belongs) must pass through *o* object's owner (the representative). The owner-as-dominator property guarantees again that any modifications to the internal structure of the composite object must be triggered through the representative object's interface. Outgoing references are permitted, but only upwards in the ownership tree structure. Incoming references are still prevented by the type system. Although more flexible than full encapsulation, many popular design patterns cannot be implemented using ownership types. An often cited example is that of the Iterator pattern. A composite object – collection – often provides clients with an external iterator that allows the client to travers the elements stored in the collection. To move from one element to another, the iterator must be able to access the internal structure of the collection composite object. But because the ownership type system does not permit incoming references, the iterator must itself be part of the encapsulated composite object. This again prevents the external client to acces the iterator.

Systems that enforce the owner-as-modifier discipline [14], [17], [13] and [10] constitute a natural evolution of the ownership type system. We still have a tree-like hierarchy of object ownership and the mode annotations determine membership in the composite object. But unlike in the OT systems, the ownership information determines the legality of method calls. Arbitrary incoming and outgoing references are permitted. At the same time the owner-as-modifier property is satisfied: If an object *o* is modified (the composite object changes) then the change has been triggered through a sequence of method calls originating in the *o*'s owner (the representative).

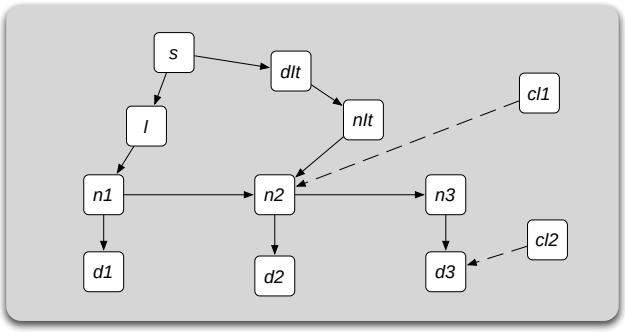
In this workshop paper we describe the ***Potential Access Pass*** methodology introduced in [17] as a tool to reason about composite objects and representation protection. We also put forward a novel weak uniqueness property for reference paths that generalizes the standard notion of free or unique references by allowing read-only

aliases. This property provides us with an additional flexibility to transfer sub-components from one composite object to another.

Outline In section 2 we present an example – the *Set* composite object. Section 3 introduces the ***Potential Access Path*** methodology. In section 4 we present our language ***JaM*** and develop the formal mode-system. Section 5 provides the operational semantics for our language. Next we formally verify the CSE property (section 6). In section 7 we revisit the *Set* example – guiding the reader through the design and implementation process and and providing detailed *JaM* code. Related Work follows in section 8. We briefly conclude the paper with section 9.

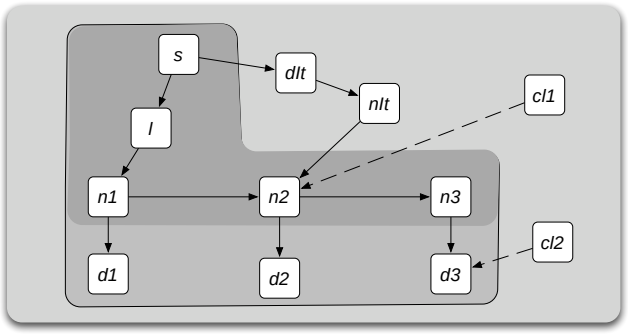
2. The Set Example

We will consider now the implementation of an abstract object *Set*. Our *Set* shall provide the standard methods *add*, *remove* and *contains*. We will also supply an external iterator that provides clients with the possibility to traverse all the *Set* elements. How would we go about implementing the *Set*? Let's assume that we have available a pre-existing *List* component (implemented as a composite object with a single linked list of nodes *ni* with data objects *di*, node iterator *nIt* and the representative *l*). We can use the *List* component to implement our *Set*. The node iterator can be used to implement an iterator over the data elements. Assuming three elements in our set, we end up with the following (run-time) object structure – *s* is the representative of the composite *Set* object and *dIt* an iterator over the set elements:



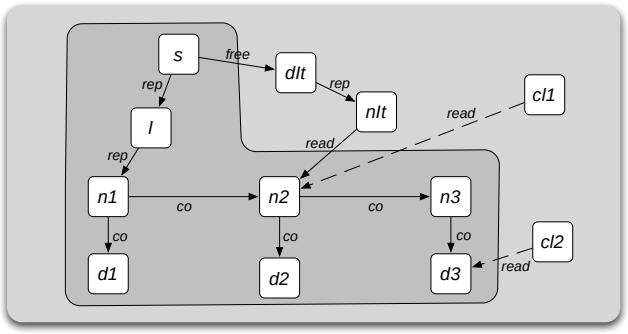
In general, it is possible that other objects obtain references into the above structure. Through these references the *Set* composite object might be modified. For example, an object *cl1* with a reference to *n2*, could send the message *setNext* to *n2* with `nil` as parameter, destroying the integrity of the *Set* composite object. What's worse, the representative object *s* will not even be aware about the changes. Such situation certainly must be prevented. In a different setting, an outsider, *cl2*, might obtain a reference to *d3*. Here it is not so clear, if *cl2* should be allowed to modify *d3*. It all depends on what elements are stored in the set. If the set is used by an online lottery to maintain a viewable list of winning numbers, it would be undesirable to allow some dishonest players to make modifications to the element *d3* (replacing legitimate number with one selected by the devious player). Only the owners of the set (e.g. lottery providers) should be allowed to make modifications to *d3*. On the other hand, if the set is used to keep track of players registrations, the participants should be permitted to make modifications to their registration data (update e-mail address, telephone, etc.). From the set (or lottery organizers) perspective, what matters is that the created registrations are preserved, not their contents. In both scenarios we want the representative object *s* to protect the integrity of the composite object *Set*. What differs is the extent of the protection.

What does it mean that *s* protects certain objects? We take the viewpoint that any changes to the state of these objects should be initiated by *s* itself. We will be referring to the set of objects protected by s as the ***composite*** of *s*. In our *Set* example, depending on the context, we have two different composites. (The dark grey area represents composite that corresponds to the set of player registrations; the larger boundary corresponds to the set of winning numbers.):



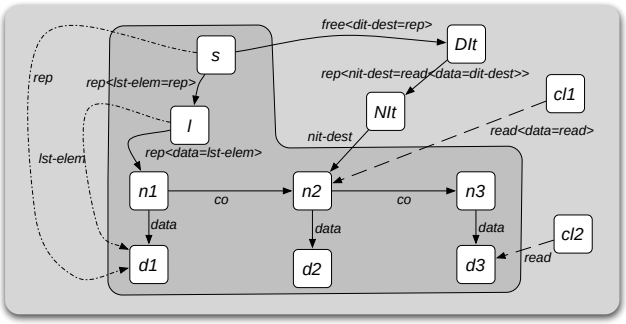
We still need to specify what it means that changes to these objects are initiated by *s*. To this effect we separate the methods of *s* into two kinds: ***observers*** and ***mutators***. When executing an observer on *s* we are guaranteed of no changes to its instance variables, whereas mutators have the right to modify them. But this is not enough. Observers shall never affect changes to instance variables of any object in the composite of *s*. As a consequence, for an object in *s*' composite to change its state, *s* must be executing a mutator – *s* is aware of the fact that it's composite state might change.

The next issue we have to address is: How can programmers specify membership inside a composite object? We borrow here from work of others [11], [15], [8] and use ***mode annotations*** on object's references. To begin with, we will use three kinds of modes: `rep`, `co` and `read`. The most important mode is `rep` - if a reference from an object o to an object ω is designated by programmer as `rep`, we put ω into o's composite. The `co` mode states that the two objects linked by a `co`-reference belong to the same composite object. The third mode, `read`, tells us that based on that particular reference no statement about composite membership of both objects can be determined. In our original example we could assign the following modes (if we want to model the winning numbers set – in the other case replace `co`-references from n to d with `read`-references):



Unfortunately, to address both *Set* scenarios, we would need to define the class *Node* twice - once with a `co`-reference to its data and in the second case with a `read`-reference. If we want to reuse components, the modes `rep`, `co` and `read` are not enough. We need

more flexibility. To this effect we introduce an additional class of modes $\alpha \in \mathbb{A}$ and parameterize the base modes with correlations. Now our complete modes are of the form $\mu < \alpha = \mu' >$. The intuition is as follows: if *o* has a reference of mode $\mu < \alpha = \mu' >$ to ω and ω has a reference of mode α to another object ω', then *o* can potentially obtain a μ' reference to ω (via a series of method calls). An α-reference $o \xrightarrow{\alpha} \omega$ does not tell us anything about *o'* and ω's membership in a composite object. We need some external reference to *o*, to possibly determine ω's membership. The described ***association modes***[1] and ***correlations*** are crucial for the structural flexibility of the mode technique. They allow a class to fix the modes of references in its instances without fixing the reference targets' assignment to a composite object. This decision is postponed to each instance's clients. Hence the same class can be reused in many different structural contexts. In our example we obtain the following situation:



The above diagram tells us, that *l* could potentially obtain an `lst-elem`-reference to *d1*, and therefore *s* could obtain a `rep`-reference to *d1*. *s* could also obtain a `rep`-reference to *d2* (via *l*, *n1* and *n2*). Additionally we can infer that *dIt* (the data iterator) can obtain a `read`-reference to *n2* and `dit-dest`-reference to *d2*. Hence with the help of *dIt*, *s* can avail of a `rep`-reference to *d2* – this time via a different path.

To allow a safe transfer of sub-components, we introduce one additional mode: `free`. By assigning a `free`-reference from *o* to ω we state that ω belongs to a special part of *o*'s composite – its ***movables***. *o* has the right to transfer the sub-component represented by ω to another aggregate. Note, that ω can be aliased by other `read` references.

In section 7 we revisit this example and elaborate more on transfer of sub-components.

3. Potential Access Paths and Composite Objects

During the execution of object-oriented programs, new objects are created, old ones are destroyed and links between objects (through which messages can be exchanged) are established. The run-time system constitutes a graph with objects as nodes and references as edges. If the programmer has the option of annotating references with the previously introduced modes, these annotations will be reflected in the graph.

Our tool in reasoning about composite object protection are paths of references between two (not necessarily directly) connected objects.

Paths in a graph are non-empty sequences $\pi = h_1, \ldots, h_n$ of object references $h_i = o_i \xrightarrow{\mu_i} \omega_i$ with $o_{i+1} = \omega_i$ (also written $\pi = o_1 \xrightarrow{\mu_1} o_2 \ldots o_n \xrightarrow{\mu_n} o_{n+1}$). Among all the possible paths in a given graph, we are only interested in certain kinds, namely those that allow us to make judgements about membership in composite objects. We already discussed in section 2 how to arrive at the judgment for immediate paths of length one. Now let's turn our attention to paths that emerge from combination of two adjacent edges. We first look at the following path: $o \xrightarrow{rep} \omega \xrightarrow{co} \varphi$. ω belongs to *o*'s composite and therefore φ must also belong to *o*'s composite (remember the intuition behind *co*). We could as well imagine an inferred edge $o \overset{rep}{\dashrightarrow} \varphi$ in our graph. This inferred edge tells us directly, that φ belongs to *o*'s composite. In contrast to $o \xrightarrow{rep} \omega$, $o \overset{rep}{\dashrightarrow} \varphi$ does not tell us if *o* has direct access to φ. But potentially, *o* could obtain a direct access to φ if there is a method of φ that returns `this` and if φ is propagated then to *o* as a result of some ω method. Therefore we will refer to the inferred edge $o \overset{rep}{\dashrightarrow} \varphi$ as a ***potential access path***. As mentioned before, many of the paths in the graph will convey no meaningful information about composite object membership. From our perspective interesting access paths are defined as follows:[2]

$$\frac{< o, \mu, \omega > \in \mathfrak{g}}{\mathfrak{g} \vdash < o, \mu, \omega > \in PAP(o, \mu, \omega)}$$

$$\frac{\mathfrak{g} \vdash \pi_1 \in PAP(o, \mu, q) \quad \mathfrak{g} \vdash \pi_2 \in PAP(q, co, \omega)}{\mathfrak{g} \vdash \pi_1 \bullet \pi_2 \in PAP(o, \mu, \omega)}$$

For association modes with correlations we have the additional rule:

$$\frac{\mathfrak{g} \vdash \pi_1 \in PAP(o, \mu\langle \ldots, \alpha = \mu', \ldots \rangle, q) \quad \mathfrak{g} \vdash \pi_2 \in PAP(q, \alpha <>, \omega)}{\mathfrak{g} \vdash \pi_1 \bullet \pi_2 \in PAP(o, \mu, \omega)}$$

The modes `rep` (and `free`) not only decide about the composite object membership, but also allow us to make statements about the yet to be defined object ownership. If there is a reference $o \xrightarrow{rep} \omega$, then *o* is considered the owner of object ω (having complete control about the changes to ω's state). But the ownership property can extend beyond objects reachable directly via *rep*-references (consider the extensions via *co*- or α-references). We formalize these concepts with the following definition.

DEFINITION 1. *In a graph* $\mathfrak{g}$ *we call an object o the* ***owner*** *of an object* ω *iff there exists an ownership path from* o *to* ω. *The set of ownership paths from* o *to* ω *is defined as follows:*

$$Osh_{\mathfrak{g}}(o, \omega) = PAP_{\mathfrak{g}}(o, rep, \omega) \cup PAP_{\mathfrak{g}}(o, free, \omega)$$

For each object o in $\mathfrak{g}$, *the corresponding* ***composite object*** *is defined as:*

$$composite_{\mathfrak{g}}(o) = \{o\} \cup \bigcup_{Osh_{\mathfrak{g}}(o,\omega) \neq \varnothing} composite_{\mathfrak{g}}(\omega).$$

o is called the **representative** *of the composite object. If* $\omega \in composite_{\mathfrak{g}}(o)$ *then o* ***dominates*** ω.

In general it is possible that an object has more than one owner in a given object graph. This is counterintuitive to our understanding that the owner controls the changes to the state of objects in its composite. We lose this exclusive control right, if an object has more than one owner. It is desirable (and for our composite state encapsulation property essential) that all objects in a given object graph have a unique owner.

DEFINITION 2. *We say that an object graph* $\mathfrak{g}$ *has the* ***Unique Owner (UO)*** *property,* $\mathfrak{g} \vDash UO$, *iff* $\forall o, \tilde{o}, \omega \; . \; (Osh_{\mathfrak{g}}(o, \omega) \neq \varnothing \wedge Osh_{\mathfrak{g}}(\tilde{o}, \omega) \neq \varnothing) \Rightarrow (\tilde{o} = o)$.

[1] They are related to, but nevertheless different from ownership parameters.

[2] For technical reasons, the *PAP*'s are determined in the object graph to which inverses of all `co`-labeled edges are added

There might be more than one ownership path from o to ω, and in case of `rep`-paths, two of these paths may start with different references outgoing from o, e.g $o \xrightarrow{\texttt{rep}} \psi \xrightarrow{\texttt{co}} \omega$ and $o \xrightarrow{\texttt{rep}} \chi \xrightarrow{\texttt{co}} \varphi \xrightarrow{\texttt{co}} \omega$. This is fine, as long as the owner is unique. The situation changes in case of `free` paths. `free`-references were introduced to allow safe transfer of sub-components (after destructive read). In the previous situation when replacing `rep` with `free` we arrive at following paths: $o \xrightarrow{\texttt{free}} \psi \xrightarrow{\texttt{co}} \omega$ and $o \xrightarrow{\texttt{free}} \chi \xrightarrow{\texttt{co}} \varphi \xrightarrow{\texttt{co}} \omega$. We still might have a unique owner of ω, but this is not enough. Even if we read destructively the `free`-reference from o to χ, o will retain another `free` access path to ω (via ψ) and it would not be safe to pass this sub-component to another composite object. Here we need a stronger property.

DEFINITION 3. *We say that an object graph $\mathfrak{g}$ has the **Unique Head (UH)** property, $\mathfrak{g} \vDash UH$, iff $\forall o, \tilde{o}, \omega, h, \pi, \tilde{h}, \tilde{\pi}$.($h \bullet \pi \in PAP_{\mathfrak{g}}(o, \text{free}, \omega) \ \wedge \ \tilde{h} \bullet \tilde{\pi} \in Osh_{\mathfrak{g}}(\tilde{o}, \tilde{\omega})) \Rightarrow (\tilde{h} = h \ \wedge \ mult(h, \mathfrak{g}) = 1)$.*

UH tells us that if we have multiple `free` paths to an object ω, the initial `free`-reference must be unique (thereafter we can have multiple `co`-paths leading to ω). Under such circumstances, after destructive read of the initial reference we are guaranteed that o does not own ω anymore.

Both UO and UH are properties of the object graph at a frozen point in time. When the graph changes, so might its compliance with UO and UH.

When an object ω executes a method f, we can find in the object graph a path of references through which a sequence of method calls leading to the call of f took place. When the method f is a mutator, the state of ω (and therefore the composite state of any object o to whose composite ω belongs) might change. We expect that o actually initiates the change. The next two properties help us with it.

DEFINITION 4. *The **Representative Control (RC)** property ensures that if ω (belonging to the composite object o) executes a mutator, then this mutator execution is nested inside a mutator execution on o. The **Mutator Control Path (MCP)** property ensures that a mutator on an object ω is always invoked via a sequence of calls along the edges of an ownership path to ω.* [3]

Notice, that o does not necessarily control the membership in its composite object – through temporary `rep`, `co` or `free` references in the execution of observers new paths can be established that add an object to $composite_{\mathfrak{g}}(o)$. Even though this addition is only temporary, it is a change of $composite_{\mathfrak{g}}(o)$ not necessarily controlled by o. The desired state encapsulation property does not require us to impose control on *temporary* additions since temporary members cannot be used to represent the composite's state. To represent state, only a core of composite object's members can be used that remains in the composite between method invocations.

DEFINITION 5. *For an object o, its **state representation** is defined as:*

$$strep_{\mathfrak{g}}(o) = composite_{\widehat{\mathfrak{g}}}(o),$$

where $\widehat{\mathfrak{g}} \subseteq \mathfrak{g}$ is a subgraph containing references stored only in the instance variables of objects. The composite state of an object o is defined as

$$compState_{\mathfrak{g}}(o) = \bigcup_{\omega \in strep_{\mathfrak{g}}(o)} state(\omega)$$

$strep_{\mathfrak{g}}(o)$ is a set of implementation objects which collectively represent the composite object's state by virtue of their shallow states.

[3] A formal definition of these properties can be found in [17]

At this point we state the main property of our system:

DEFINITION 6. *The **Composite State Encapsulation (CSE)** property ensures that if an execution step of a* JaM *program transforms an object graph $\mathfrak{g}$ into $\mathfrak{g}'$ such that $compState_{\mathfrak{g}}(o) \neq compState'_{\mathfrak{g}}(o)$ then o is executing a mutator.*[4]

One of the features of our system is the ability to move sub-components from one aggregate to another. The set of all objects belonging to a composite object can be divided into two parts, depending on the ability of the composite object to transfer these objects.

DEFINITION 7. ***Fixtures** is the subset of $composite_{\mathfrak{g}}(o)$ that is reachable via `rep` path sequences only:*

$$fixtures_{\mathfrak{g}}(o) = \bigcup_{PAP_{\mathfrak{g}}(o, rep, \omega) \neq \emptyset} \Big(\{\omega\} \ \cup \ fixtures_{\mathfrak{g}}(\omega) \Big)$$

***Movables**, another subset of $composite_{\mathfrak{g}}(o)$ is defined as:*

$$movables_{\mathfrak{g}}(o) = composite_{\mathfrak{g}}(o) \setminus (fixtures_{\mathfrak{g}}(o) \cup \{o\})$$

Objects in $fixtures_{\mathfrak{g}}(o)$ can never be removed from the composite object. $movables_{\mathfrak{g}}(o)$ on the other hand contains objects, which can be safely transferred between different agregates (via destructive read of `free` references). However, the transfer can happen only as sub-components and not as single objects.

DEFINITION 8. *Let $\omega \in composite_{\mathfrak{g}}(o)$ and let φ be an object such that $\mathfrak{g} \vdash \pi \in PAP(\varphi, \texttt{free}, \omega)$ for some path π. Then $composite_{\mathfrak{g}}(\omega)$ is called a (movable) **sub-component** of o.*

4. Mode Checking in JaM

So far we did not introduce the syntax of our language ***JaM*** (Java with Modes). It is a fully orthogonal extension of a Java subset that classifies object references by mode annotations. To reduce the complexity of the formal treatment, we omit several nonessential features (e.g. static methods, subclassing). The grammar of *JaM* is defined below:

$$\begin{array}{rcl}
p \in P & ::= & D^* \\
d \in D & ::= & \texttt{class}\ \mathbb{C}\ \{(\mathcal{T}\ Id;)^*\ Meth^*\} \\
f \in Meth & ::= & \mathcal{K}\ \mathcal{T}\ Id((\mathcal{T}\ Id)^*)\ \{(\mathcal{T}\ Id;)^*\ S;\ \texttt{return}\ E\} \\
\kappa \in \mathcal{K} & ::= & \texttt{mut} \mid \texttt{obs} \\
t \in \mathcal{T} & ::= & \mathcal{M}\ \mathbb{C} \\
\mu \in M & ::= & (\texttt{rep} \mid \texttt{co} \mid \texttt{read} \mid \texttt{free} \mid \mathbb{A}) < \Delta > \\
\delta \in \Delta & ::= & (\mathbb{A} = M)^* \\
s \in S & ::= & S; S \mid V = E \mid \texttt{if}(E \odot E)\{S\} \\
 & & \mid \texttt{while}(E \odot E)\{S\} \\
o \in \odot & ::= & \texttt{==} \mid \texttt{!=} \\
e \in E & ::= & \texttt{val}(V) \mid \texttt{destval}(V) \mid \texttt{null} < \Delta > \\
 & & \mid \texttt{new} < \Delta > \ C() \mid E \Leftarrow Id(E^*) \\
v \in V & ::= & v \mid \texttt{this}.v
\end{array}$$

Notable difference from Java is the introduction of modes, marking of methods as mutator or observer and the explicit read operations (`val` and `destval`). All *JaM* programs stripped of their annotations are legal Java programs. Java programs can be translated into legal JaM programs by annotating all variable/parameter declarations with `co`, declaring all methods as `mut` and introducing the explicit non-destructive read operator `val`.

[4] A more formal definition of CSE can be found in section 6

In general, when executing *JaM* programs, the properties listed in section 3 will not hold. We rule out illegal *JaM* programs with the help of a mode-system, which is orthogonal to the standard Java type-system.[5] For space reasons we concentrate here on deriving the correct modes for expressions. When checking method definitions in the class c, we must verify that the method body is well typed (moded) and the result is of the same type as declared in the method signature. The verification of the method body happens with respect to a type/mode assignment Γ. It is constructed by assigning to `this` the mode `ref co`, and by assigning to each local variable/parameter with the type τ in the signature, the type `ref` τ.[6]

The typing judgment $\Gamma, \kappa \vdash e : \tau$ expresses that term e is legal inside a method of kind κ (`mut` or `obs`) and has static type τ in the context of type assumptions Γ for local variables and parameters. Selected rules for deriving types/modes in *base-JaM* are given below:

$$\frac{\Gamma, \kappa \vdash \nu : \texttt{ref}\ \tau \quad \tau^* = \tau[\texttt{free} \mapsto \texttt{read}]}{\Gamma, \kappa \vdash \texttt{val}(\nu) : \tau^*}$$

$$\frac{\Gamma, \kappa \vdash \nu : \texttt{ref free}\ c \quad \nu = \texttt{this}.y \Rightarrow \kappa = \texttt{mut}}{\Gamma, \kappa \vdash \texttt{destval}(\nu) : \texttt{free}\ c}$$

$$\frac{\vdash c\ \texttt{ok}}{\Gamma, \kappa \vdash \texttt{new}{<}\delta{>}\ \ c() : \texttt{free}{<}\delta{>}\ \ c}$$

$$\frac{\Gamma, \kappa \vdash \nu : \texttt{ref}\ \tau \quad \Gamma, \kappa \vdash e : \tau' \quad \vdash \tau' \leq_m \tau \quad \nu \neq \texttt{this} \quad \nu = \texttt{this}.y \Rightarrow \kappa = \texttt{mut}}{\Gamma, \kappa \vdash \nu = e : \texttt{Cmd}}$$

$$\frac{\Gamma, \kappa \vdash e : \mu\ c \quad \vdash (f : \overline{\tau_i} \xrightarrow{\kappa^*} \tau) \in \Sigma(\mu\ c) \quad \kappa^* = \texttt{mut} \Rightarrow \kappa = \texttt{mut} \wedge \mu \neq \texttt{read} \quad \overline{\Gamma, \kappa \vdash e_i : \tau_i'} \quad \overline{\vdash \tau_i' \leq_m \tau_i}}{\Gamma, \kappa \vdash e \Leftarrow f(\overline{e_i}) : \tau}$$

Non-destructive read of a variable ν is assigned the type τ of the reference stored in that variable. But this works only if the mode of the reference is `rep`, `co` or `read`. If the mode is `free`, we cannot do it. The copy of that reference could be then stored in another variable with the mode `rep` (as `free` can be converted to any other mode) and the UH property would be violated. We do not want to disallow a non-destructive read of `free` variables and therefore change in such cases the resulting mode to `read` (which is always safe).

We can destructively read only `free` variables (we would not gain anything by allowing it for `rep`, `co` or `read`). There is no restriction on reading local variables and parameters, but if the read variable is an instance variable, we can do it only inside mutators (we are setting the instance variable to `nil`, therefore modifying the state of the object executing the method).

In the creation expression we decided to specify the correlation set δ to be added to the `free` mode of its value. Although not necessary, it simplifies the formal treatment.

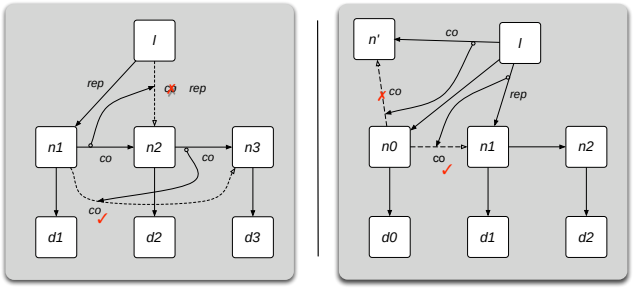
Assignment to instance variables is legal only inside mutators. Also, the mode of the value must be compatible with the mode of the variable.[7]

[5] We also disallow certain class of modes – see section 6

[6] In the full formal system the set of types is extended to include `ref` τ, so we can distinguish between values of type τ and variables that hold values of that type. As `ref` types are not visible to the programmer, we excluded them from the JaM syntax.

[7] We have $\mu{<}\delta{>} \leq_m$ `read`$<\delta>$, `free`$<\delta> \leq_{\texttt{m}}$ `rep`$<\delta>$, `read`$<\delta, \alpha = \mu, \delta'> \leq_{\texttt{m}}$ `read`$<\delta, \delta'>$ and `read`$<\alpha = \mu> \leq_{\texttt{m}}$ `read`$<\alpha = \mu'>$ if $\mu \leq_m \mu'$. We elaborate more in section 6.

Method invocations $e \Leftarrow f(e_1, \ldots, e_n)$ are rather tricky. Assuming that f has the (mode) signature $\mu \rightarrow \tilde{\mu}$ it is tempting to return to the sender of a message f as a result reference with the mode specified in the signature of f, namely $\tilde{\mu}$. But what the signature of f tells us is merely the mode of the temporary reference to the result that the receiving objects has. When this reference is passed from the receiver to the sender, that mode might have to be adjusted. Let's consider the following situation:



Object *l* invokes `next()` on `Node` *n1* which returns the `co` reference *n1* $\rightarrow$ *n2*. The reference *l* $\rightarrow$ *n2* which *l* obtains must not be a `co` reference, since *n2* must be owned by *l*. The return of the `co` reference can be better understood as the mode-preserving shortening of two-reference path *l* $\xrightarrow{\texttt{rep}}$ *n1* $\xrightarrow{\texttt{co}}$ *n2* to a one-reference path *l* $\xrightarrow{\texttt{rep}}$ *n2*. Should, on the other hand, the node *n1* call `next()` on its `co`-object *n2*, then the returned reference's mode is not adapted, since the return simply shortens `co` path *n1* $\xrightarrow{\texttt{co}}$ *n2* $\xrightarrow{\texttt{co}}$ *n3* to *n1* $\xrightarrow{\texttt{co}}$ *n3*. Analogously, the mode of references passed as parameters has to be adapted: If *l* has created a new `Node` object *n0* in its composite (to be included in the list structure), then it should supply to *n0*'s `setNext` operation (expecting a `co` reference) one of its `rep` references, namely *l* $\xrightarrow{\texttt{rep}}$ *n1*, and not a reference *l* $\xrightarrow{\texttt{co}}$ *n'* to a node that is a `co`-object in the same composite as *l*.

Consequently, two notions of signatures have to be distinguished:

Exported signatures The interfaces which all instances of a class c export have a signature $\Sigma(c)$ defined by the class. Its entries $f : \overline{\mu_i\ d_i} \rightarrow \mu\ d$ specify the types of the parameter values which implementations of operation f expect to receive, and the type of the result values which they ensure to produce. Against this signature, the operations' implementations in class c are type-checked.

Imported signatures The interfaces which senders import through μ_r-references to c-objects have the signature $\Sigma(\mu_r c)$ with modes from c-objects' signature $\Sigma(c)$ adapted relative to call-link mode μ_r. Its entries $f : \overline{\mu_r \circ \mu_i\ d_i} \rightarrow \mu_r \circ \mu\ d$ specify the types of the parameter values which the sender must ensure to supply, and the type of the result values which the sender can expect to obtain.

The method invokation rule type checks against the imported signature. The following rule provides us with the imported signature of a class c relative to the call link μ_r (here we show the simplified rule for modes without correlations):

$$\frac{\vdash (f : \overline{\mu_i\ d_i} \xrightarrow{\kappa} \mu\ d) \in \Sigma(c) \quad \mu_i \neq \texttt{rep} \quad \mu_i = \texttt{co} \Rightarrow \mu_r \neq \texttt{read}}{\vdash (f : \overline{\mu_r \circ \mu_i\ d_i} \xrightarrow{\kappa} \mu_r \circ \mu\ d) \in \Sigma(\mu_r\ c)}$$

How do we read this rule? Our system disallows `rep` as parameter mode and permits calling of methods with `co` parameters only via references that are not `read`. If the call over the μ_r link is legal,

the sender must supply an argument of mode $\mu_r \circ \mu_i$ (not just μ_i). The returned reference is viewed by the sender as $\mu_r \circ \mu$ (and not μ). The adaptation, called the ***import*** of μ through μ_r and written $\mu_r \circ \mu$, is defined as follows:

$$\mu_r \circ \texttt{read}<\alpha = \mu> = \texttt{read}<\alpha = \mu_r \circ \mu>$$
$$\mu_r \circ \texttt{free}<\alpha = \mu> = \texttt{free}<\alpha = \mu_r \circ \mu>$$
$$\mu_r \circ \texttt{rep}<\alpha = \mu> = \texttt{read}<\alpha = \mu_r \circ \mu>$$
$$\mu_r \circ \alpha<> = \mu' \quad \text{if} \quad \mu_r = \mu<\alpha = \mu'> .$$

Let's verify the plausibility of these definitions with respect to the result $\hat{\mu}$-reference returned via a μ_r-reference (we leave out correlations for now):

- $\hat{\mu} = \texttt{read}$: The receiver returns a `read` reference and therefore does not know anything about the targets owner. Without this information, the sender can accept it only as a `read` reference – any other choice would be unsafe. This is what $\mu_r \circ \texttt{read} = \texttt{read}$ gives us.
- $\hat{\mu} = \texttt{free}$: The sender can safely accept a `free` reference from the receiver as `free`, since it was the unique initial segment of ownership paths to all `co`-objects reachable through it, and all these old ownership paths are destroyed by the removal of the receiver's `free` handle from the graph (via destructive read). $\mu_r \circ \texttt{free} = \texttt{free}$ does it.
- $\hat{\mu} = \texttt{rep}$: If the receiver returns a `rep` reference, the receiver may still possess further `rep` handles with the same target, and thus remain the target's owner. Hence the sender cannot accept the handle as `free` or `rep` without risking a violation of unique ownership. Accepting it as `co` would make the sender a `co`-object of the target, and thus also owned by the receiver. This might raise a uniqueness conflict with any old owner of the sender. Only `read` is safe and $\mu_r \circ \texttt{rep} = \texttt{read}$ gives us the right mode.
- $\hat{\mu} = \texttt{co}$: If the returned reference is `co`, i.e., points to the receiver's `co`-object, the sender best accepts it with the mode μ_r of the call-link: If μ_r is `rep` or `free`, then the sender already had an ownership path to the target by concatenation of the call-link and the receiver's `co` handle. Hence it is reasonable to shorten it to a direct μ_r handle. In case of `free`, the accepted reference will replace the unstored `free` call-link as the unique initial edge of `free` ownership paths to the receiver and all its `co`-objects. If μ_r is `co` then sender and target were already `co`-objects through the call-link and the handle of the receiver, so that a direct `co`-handle is safe. And if μ_r is `read` then the accepted handle can only be `read`, since a `read` call-link gives the sender no information about the receiver's owner. Again $\mu_r \circ \texttt{co} = \mu_r$ is the right choice.

Similarly we consider the parameter passing mechanism. The receiver of a method expects a parameter of mode $\hat{\mu}$. The sender passes $\tilde{\mu}$ argument over μ_r-reference. The call rule tells us that $\tilde{\mu}$ must be compatible with $\mu_r \circ \hat{\mu}$. Is it sensible?

- $\hat{\mu} = \texttt{read}$: A parameter of mode `read` means that the receiver makes no assumptions about the target's place in the object graph. Hence the sender can supply references of any mode and any mode is compatible with $\texttt{read} = \mu_r \circ \texttt{read}$.
- $\hat{\mu} = \texttt{free}$: If the receiver expects a `free` parameter then only `free` references of the sender (which are destroyed in the call step) can guarantee the necessary uniqueness of the initial ownership path segments. $\mu_r \circ \texttt{free} = \texttt{free}$ does the trick.
- $\hat{\mu} = \texttt{rep}$: If the receiver expects a `rep`-reference then a reference to an object in the receiver's composite object must be passed. However, no mode on sender's reference can guarantee that the target is in the receiver's composite. Hence methods with `rep` parameters are disallowed.
- $\hat{\mu} = \texttt{co}$: A parameter of mode `co` means that the receiver expects a handle to an object with the same owner as itself. If the call-link is of mode $\mu_r = \texttt{read}$ then the sender has no information about the receiver's owner and thus cannot know which handle's target would have the same status. This situation is disallowed by the typing rule. The other call-links provide us with enough information about the owner of the receiver. If the call-link is of mode $\mu_r = \texttt{co}$, then sender and receiver have both the same owner. Therefore it is safe to pass a co-reference (the object at its end has again the same owner). And if the call-link is of mode $\mu_r = \texttt{rep}$ or `free` then respectively, `rep` or `free` handle of the sender guarantees that receiver and target have the same owner, namely the sender. In all three cases $\mu_r \circ \texttt{co} = \mu_r$ does the trick.

Notice that we are also required to recursively import the modes "hidden" in the correlations. A related discussion can be found in [17].

5. Operational Semantics

The formalization of the execution of *JaM* programs is provided in the style of *small-step semantics*. We take the standard approach, where transformation of program terms $(e, \vec{\eta}, s, om) \rightarrow (e', \vec{\eta'}, s', om')$ is defined in the following three contexts: a dynamic stack $\vec{\eta}$ of ***environments*** $\eta_i \in Env$ that maps local identifiers to locations, a changing ***store*** $s \in \mathfrak{Store}$ that maps locations $\ell \in \mathcal{L}oc$ to values currently at these locations and a growing ***object map*** $om \in Omap$ that maps identifiers $o \in \mathbb{O}$ of created objects to object "values": tuples of field environments ρ_o (mapping field names to locations), and method suites F_o (mapping operation names to methods).

The reduction steps are the expected ones, but we include three non-standard features specifically for accommodating reasoning about composite objects:

- We formalize reference values as so-called ***handles***: A handle is not just the object-identifier ω of the target object, but a triple $h =< o, \mu, \omega >$ which also includes the identifier o of the source object and the mode μ of o's reference to ω.
- We record *call-links* (references through which method invocations are made) in the computational state (as annotations to the stack environments)
- Object graph is included (and manipulated) as an explicit fourth context $\mathfrak{g}$ of the term reduction rules.

The meaning of program p as a computational process is formalized now as a sequence of reduction steps $(e, \vec{\eta}, s, om, \mathfrak{g}) \Longrightarrow (e', \vec{\eta'}, s', om', \mathfrak{g})$. The transformations starts with the expression $\texttt{new}<> \ c_n() \Leftarrow main()$ in the initial context $(\eta_0, \mathfrak{s_0}, om_0, \mathfrak{g_0}) = (\emptyset^{obs}_{<nil,read,nil>}, \emptyset, \emptyset, \emptyset)$. Each reduction step replaces in the term e one subterm, the *redex*, by another term. In particular, locations $\ell \in \mathcal{L}oc$ are substituted for identifiers x (using η) and for field names $\texttt{this}.x$ (using $\rho_{\texttt{this}}$). Values of variables are substituted for read access expressions (using s) and method bodies are substituted for operation call expressions (using om). Through these substitutions, the transformed terms are not just the statements and expressions of the program syntax, but belong to the larger category $\mathcal{R}$ of ***runtime terms***.[8]

The sources in all handles in the store and the runtime term should coincide with the object to which the corresponding store location or method nesting level belongs (***source consistency***). At

[8] Detailed syntax for $\mathcal{R}$ can be found in [17]

locations $\ell = \rho_o(x)$ of fields x of object o, we expect to find only handles $s(\ell) = h$ whose source is o. Then the object-map is source consistent, in symbols, $\models_s om$. Analogously, at locations $\ell = \eta_i(x)$ of local variables and parameters x in environments η_i of invocations with receiver $\mathtt{r}$, we expect to find only handles $s(\ell) = h$ whose source is $\mathtt{r}$. Then the environment is source consistent, $\models_s \eta$. And at all method nesting levels in the runtime term e with corresponding receiver $\mathtt{r}$, we expect to find only handles h with source $\mathtt{r}$, and locations ℓ containing handles $s(\ell) = h$ with source $\mathtt{r}$. If this is the case the runtime term is source consistent, in symbols, $\models_{s,\eta} e$.

We split the definition of reduction steps into two complementary aspects. On one side are sub-terms that can be completely substituted in one step to a new term. This substitution will be captured in redex replacement rules $(e, \vec{\eta}, s, om, \mathfrak{g}) \longrightarrow (e', \vec{\eta'}, s', om', \mathfrak{g})$. On the other side we must select a suitable sub-term for the next substitution. This selection can be conveniently specified with the help of a ***reduction context***. A reduction context $\mathcal{E}$ is a runtime term "with a hole" symbolized by '$\Box$'. A complete runtime term $\hat{e} = \mathcal{E}[e]$ is obtained by filling a term e into the hole. Reduction steps are then written $(\mathcal{E}[e], \vec{\eta}, \mathtt{s}, om, \mathfrak{g}) \Longrightarrow (\mathcal{E}[e'], \vec{\eta'}, \mathtt{s}', om', \mathfrak{g}')$ and performed according to the following reduction rule:

$$\frac{\mathcal{E} \in \mathcal{R}^{\Box} \quad (e, \vec{\eta}, s, om, \mathfrak{g}) \longrightarrow (e', \vec{\eta'}, s', om', \mathfrak{g})}{(\mathcal{E}[e], \vec{\eta}, \mathtt{s}, om, \mathfrak{g}) \Longrightarrow (\mathcal{E}[e'], \vec{\eta'}, \mathtt{s}', om', \mathfrak{g}')}$$

A selection of redex replacement rules is given below.

$$\frac{s(l) =< o, \mu, \omega > \quad \mu' = \mu[\mathtt{free} \mapsto \mathtt{read}]}{(\mathtt{val}(l), \vec{\eta}, s, om, \mathfrak{g}) \longrightarrow (< o, \mu', \omega >, \vec{\eta}, s, om, \mathfrak{g} \oplus o \xrightarrow{\mu'} \omega)}$$

$$\frac{s(l) =< o, \mu, \omega >}{(\mathtt{destval}(l), \vec{\eta}, s, om, \mathfrak{g}) \longrightarrow (< o, \mu, \omega >, \vec{\eta}, s[l \mapsto < o, \mu, \mathtt{nil} >], om, \mathfrak{g})}$$

$$\frac{\begin{array}{l} h =< s, \mu, r > \quad \vdash VarMths(c) = \langle \{\overline{x_i : \mathtt{ref}\ t_i}\}, F \rangle \\ \mathtt{fresh}\ o \in \mathbb{O}_c \quad \mathtt{fresh}\ l_i \in [\![\mathtt{ref}\ \mu_i\ c_i]\!] \\ \rho = \{\overline{x_i \mapsto l_i}\} \quad h_i =< o, \mu_i, \mathtt{nil} > \end{array}}{\begin{array}{l} (\mathtt{new}\!<\!\delta\!>\ c(), \vec{\eta} \bullet \eta_h^{\kappa}, s, om, \mathfrak{g}) \longrightarrow \\ (< r, \mathtt{free}\!<\!\delta\!>, o >, \vec{\eta} \bullet \eta_h^{\kappa}, s[l_i \mapsto h_i], \\ om[o \mapsto < \rho, F >], \mathfrak{g} \oplus r \xrightarrow{free} o) \end{array}}$$

$$\frac{\begin{array}{l} r \in \mathbb{O}_c \quad om(r) =< \ldots, F > \\ F(f) = \kappa^*\ \tau\ f(\overline{\mu_i\ c_i\ y_i})\{\overline{\mu_j'\ c_j'\ z_j'}; s; \mathtt{return}\ e\} \\ \mathtt{fresh}\ l \in [\![\mathtt{ref}\ \mathtt{co}\ c]\!] \quad \mathtt{fresh}\ l_i^y \in [\![\mathtt{ref}\ \mu_i\ c_i]\!] \\ \mathtt{fresh}\ l_i^z \in [\![\mathtt{ref}\ \mu_i'\ c_i']\!] \\ \hat{\eta} = \{\mathtt{this} \mapsto l, y_i \mapsto l_i^y, z_j \mapsto l_j^z\} \\ s' = s[l \mapsto < r, \mathtt{co}, r >, l_i^y \mapsto < r, \mu_i, o_i >, \\ \quad l_j^z \mapsto < r, \mu_j', \mathtt{nil} >] \\ \mathfrak{g}' = \mathfrak{g} \ominus s \xrightarrow{\mu"} o_i \oplus r \xrightarrow{co} r \oplus r \xrightarrow{\mu_i} o_i \end{array}}{\begin{array}{l} (< s, \mu_r, r > \Leftarrow f(< s, \mu_i", o_i >), \vec{\eta}, s, om, \mathfrak{g}) \longrightarrow \\ (\ll s; \mathtt{return}\ e \gg, \vec{\eta} \bullet \hat{\eta}^{\kappa^*}_{<s,\mu_r,r>}, s', om, \mathfrak{g}') \end{array}}$$

$$\frac{l \in Loc_{\mu\ c}}{\begin{array}{l} (l =< o, \tilde{\mu}, \tilde{\omega} >, \vec{\eta}, s, om, \mathfrak{g}) \longrightarrow \\ (\varepsilon, \vec{\eta}, s[l \mapsto < o, \mu, \tilde{\omega} >], om, \mathfrak{g} \ominus o \xrightarrow{\tilde{\mu}} \tilde{\omega} \ominus s(l) \oplus o \xrightarrow{\mu} \tilde{\omega}) \end{array}}$$

$$\frac{\begin{array}{l} (s' = s[l \mapsto \bot \mid l \in im(\hat{\eta})] \\ \mathfrak{g}' = \mathfrak{g} \oplus s \xrightarrow{\mu_r \circ \mu} \omega \ominus s \xrightarrow{\mu_r} r \ominus r \xrightarrow{\mu} \omega \ominus s(im(\hat{\eta})) \end{array}}{\begin{array}{l} \ll \mathtt{return} < r, \mu, \omega > \gg, \vec{\eta} \bullet \hat{\eta}^{\kappa^*}_{<s,\mu_r,r>}, s, om, \mathfrak{g}) \longrightarrow \\ (< s, \mu_r \circ \mu, \omega >, \vec{\eta}, s', om, \mathfrak{g}') \end{array}}$$

Below we provide the rationale for the given semantics rules (explanation for graph modifications is provided separately):

- Non-destructive read access $\mathtt{val}(l)$ copies the value from the store (at location ℓ) to the runtime term (at the redex position). This value is always a handle $< o, \mu, \omega >$. In case of a `free` handle, an exact copy would immediately violate UH. The copy is safe if its mode is weakened to `read`.
- Destructive read access $\mathtt{destval}(l)$ evaluates to the value at location ℓ, but resets the store at ℓ to a `nil`-handle.
- An object creation expression instantiates the class c to a new object with a fresh object-identifier `o`. It evaluates to a `free` handle from the current (creator) object `r` to the new object `o`. Instantiating c also involves the initialization of fresh locations ℓ_i of respective types `ref` μ_i c_i, to `nil`-handles with source `o` and modes μ_i. Furthermore `o` is mapped to an object value $< \{\overline{x_i \mapsto l_i}\}, F >$.
- A method invocation is executed after all its subexpressions, arguments and the receiver have evalueted. The execution continues with the body $\ll$`s; return e`$\gg$. The newly created environment contains `this`, parameters and local variables. They are bound to fresh locations of corresponding `ref`-types. These locations are initialized to: a handle to the receiver (of mode `co`), argument expression values adapted to the parameters' modes, and `nil`-handles of the local variables' modes.
- An assignment statement is executed after the left-hand side has reduced to a location ℓ and the right-hand side to a value $< o, \mu', \omega' >$. It updates the store at ℓ to the handle with the mode adapted according to the location's store partition.
- A `return` statement is executed after its return expression has evaluated to a result handle. Evaluation continues in the environment $\vec{\eta}$ with the result handle adapted to the calling context, i.e., with the sender as the new source and with a mode adapted to the sender's perspective. The current top-level environment is removed from the stack and the locations of the names in it (parameters, locals, and `this`) are removed from the store.

The object graph is formalized as a *multiset* $\mathfrak{g} \in \mathbb{N}^{\mathbb{O} \times \mathcal{M} \times \mathbb{O}}$ of edges whose multiplicity represents the *number* of the corresponding handlesoccurrences in s, η, or e. The multiplicity of edges is increased and decreased in accordance with the addition and removal of handles to/from e, η and s.

We will examine now some of the graph transformations during different reduction steps.

- Non-destructive read increases the multiplicity of the handle $h =< o, \mu', \omega >$. This models the redex's substitution to h, which increases the number of h's occurences in the term.
- Destructive read of a (free) variable leaves the object graph unchanged: The new occurence of handle $h =< o, \mu, \omega >$ in the term is balanced by removing one occurence from the store.
- New object creation adds creator object `r`'s initial reference to the new object ω to the object graph: $\mathfrak{g}' = \mathfrak{g} \oplus < \mathtt{r}, \mathtt{free}, \omega >$. This models the redex's substitution to $< \mathtt{r}, \mathtt{free}, \omega >$.
- Method invocation equips the receiver with a new `this` reference $< \mathtt{r}, \mathtt{co}, \mathtt{r} >$ and with a parameter handle $< \mathtt{r}, \mu_i, \omega_i >$ for every argument handle $< \mathtt{s}, \mu_i'', \omega_i >$ supplied by the sender. That is, the multiplicity of $< \mathtt{r}, \mathtt{co}, \mathtt{r} >$ and edges $< \mathtt{r}, \mu_i, \omega_i >$ increases, while that of edges $< \mathtt{s}, \mu_i'', \omega_i >$ decreases. This matches the arguments' disappearance from the term and the parameters' and the `this`-reference's appearance at fresh locations in the store. The call-link $< s, \mu_r, r >$ is

not changed. Its disappearance from the term is balanced by its occurence in the new top-level environment.

- Variable update converts a handle $< o, \tilde{\mu}, \tilde{\omega} >$ to $< o, \mu, \tilde{\omega} >$, i.e., decreases the multiplicity of the first handle and increases that of the second one. This matches, respectively, the disappearence of the right-hand side handle $< o, \tilde{\mu}, \tilde{\omega} >$ from the term and the appearance of the handle $< o, \mu, \tilde{\omega} >$ at location ℓ in the store. Additionally, the multiplicity of the old handle $< o, \mu, \omega >$ at location ℓ decreases since the update at location ℓ overwrites it.
- Method return combines the call-link $< \mathtt{s}, \mu_r, \mathtt{r} >$ and the return handle $< \mathtt{r}, \mu, \omega >$ to the edge $< \mathtt{s}, \mu_r \circ \mu, \omega >$ in the sender, i.e., the former two edge's multiplicity decreases while the latter one's multiplicity increases. This matches the appearence of $< s, \mu_r \circ \mu, r >$ in the runtime term and the disappearence of handle $< \mathtt{r}, \mu, \omega >$ from the term and of call-link $< s, \mu_r, \mathtt{r} >$ (together with the finished invocation) from the environment stack. Additionally, since the locations of the finished invocation's variables in the store are reset, the multiplicities of all (non-`nil`) handles lost by this are decreased to keep the object graph in sync.

For the implementations of the *JaM* language, no representation of the object graph at runtime is needed. The graph has no impact on the computation and is invisible from outside of the program. It can actually be calculated from the other run-time contexts. We included it in the rules to make the nature of transformations more obvious.

6. Verifying Run-Time Properties

The reduction rules are a tool that enables us to establish the properties that we expect to hold during execution of legal *JaM* programs. (In the following, $e_0 = \mathtt{new}\!<\!>\ c_n() \Leftarrow main()$, is the initial expression of a legal program p.)

The ownership paths in all object graphs reachable in the execution of legal *JaM* programs satisfy the Unique Owner and Unique Head integrity invariants.

THEOREM 1. *If* $(e_0, \vec{\eta_0}, \mathbf{s}_0, om_0, \mathbf{g}_0) \Longrightarrow^* (e, \vec{\eta}, \mathbf{s}, om, \mathbf{g})$ *then*

$$\mathfrak{g} \models UH, UO$$

The structure of mutator access as recorded in the environment stack during the execution of legal *JaM* programs is always consistent with ownership paths as captured in the integrity invariants Representative Control and Mutator Control Path.

THEOREM 2. *If* $(e_0, \vec{\eta_0}, \mathbf{s}_0, om_0, \mathbf{g}_0) \Longrightarrow^* (e, \vec{\eta}, \mathbf{s}, om, \mathbf{g})$ *then*

$$\mathfrak{g}, \vec{\eta} \models \mathsf{RC} \text{ and } \mathfrak{g}, \vec{\eta} \models \mathsf{MCP}$$

The following theorem is the main result, which establishes the *Composite State Encapsulation* property.

THEOREM 3. *Let* $(e_0, \vec{\eta_0}, \mathbf{s}_0, om_0, \mathbf{g}_0) \Longrightarrow^* (e, \vec{\eta}, \mathbf{s}, om, \mathbf{g}) \Longrightarrow (e', \vec{\eta'}, \mathbf{s}', om', \mathbf{g}')$. *Then for all* $o \in \mathrm{dom}(om)$*:*

$$compState_{\mathfrak{g}}(o) \neq compState_{\mathfrak{g}'}(o)$$
$$\Rightarrow \exists i \leq n_\bullet\ \mathtt{r}_i = o\ \wedge\ \kappa_i = \mathbf{mut},$$

where $\vec{\eta} = \eta_1{}^{\kappa_1}_{h_1}, \ldots, \eta_n{}^{\kappa_n}_{h_n}$ *with* $h_i =< \mathtt{s}_i, \mu_i, \mathtt{r}_i >$.

The theorem simply states that if a state of a composite object (represented by o) changes, then the representative o is executing a mutator.

The proofs for the first two theorems are by induction on the number of reduction steps from e_0 to e. Once we establish these properties (plus several helpful lemmas), the proof for Composite State Encapsulation is straightforward.

Complete set of proofs for *JaM* and *base-JaM* (without association modes and correlations) can be found in [17]. Although often tedious and lengthy, proving these results in *base-JaM* is fairly manageable. Things change, when introducing association modes with correlations. While the formal treatment of many *JaM* properties is a simple forward adaptation from *base-JaM*, the proofs of the unique owner and unique head invariants must be redone completely. Potential access paths in *JaM* have much more complicated structure than in base-*JaM*. The possibility of extending μ-paths to non-μ paths is the culprit. We were forced to restrict permissible association modes and their correlations.

- We don't consider extensions $o \overset{\mathtt{co}}{\dashrightarrow} q \overset{\alpha}{\dashrightarrow} \omega$ and $o \overset{\alpha'}{\dashrightarrow} q \overset{\alpha}{\dashrightarrow} \omega$ of `co`- and association paths by association paths. Also the extension $o \overset{\mu}{\dashrightarrow} q \overset{\alpha}{\dashrightarrow} \omega$ of potential access paths by association paths to `co`- and `free` paths $o \overset{\mathtt{co}}{\dashrightarrow} \omega$ and $o \overset{\mathtt{free}}{\dashrightarrow} \omega$ is disallowed. This simplification is reflected in constraints on the nesting structure of mode-terms. Only modes `free`, `rep` and `read` are parameterized by correlations (`free`$<\delta>$, `rep`$<\delta>$ and `read`$<\delta>$ are legal, but `co`$<\delta>$ and $\alpha<\delta>$ are not). Also, only correlations to `rep`, `read` and association modes are permitted ($\mu<\alpha =$ `rep`$<\delta>>$, $\mu<\alpha =$ `read`$<\delta>>$ and $\mu<\alpha=\gamma<>>$).
- Implicit mode-conversions from `free`$<>$ to `co`$<>$ or $\alpha<>$ caused by assignment or parameter passing is disallowed. (Tedious invariants about all sequences of association paths starting from targets of `free`$<>$ paths are needed in order to show that such conversions preserve the uniquness of ownership.) This simplification is reflected in the definition of the mode compatibility relation $\leq_m$.

At this point we want to comment on the sub-mode rules introduced in section 4. There we constrain the width- (more or fewer correlations) and depth- (correlations with compatible modes) compatibility between modes. Without this restriction it would be possible for an object to convert `read` references to `rep`. We could weaken a `rep<data=rep<>>` reference to ω to a `rep<data=read<>>` reference. Through this reference the source could store a `read` reference in ω as a `data` reference and read it back through the original reference as a `rep`. The same scenario can be set up using width-compatibility. Two distinct `rep<data=rep<>>` and `rep<data=read<>>` references to ω could be converted to the same mode `rep<>` and then linked by a `co`-reference. By reading it back through the original references the source can obtain, as with depth-compatibility, a `rep<data=rep<>>` reference and a `rep<data=read<>>` reference to the same object. Depth- and width- compatibility in *JaM*'s type system exists only between `read` modes. The `read` modes are compatible because through converted `read` references nothing can be stored in the target (since only observers can be called on the target).

7. The Set Example Revisited

We take a look now at concrete implementation of our *Set* composite object. The relevant interfaces are: `Iterator<T>`, `List<T>` and `Set<T>`. The types of variables, parameters and results are prefixed by our mode annotations (e.g. `rep Node<T>`).[9] We don't show `val`

[9] Although generic types are not part of our syntax, we use them in the example. This does not really affect our system, as the mode annotations are completely orthogonal to the standard Java types.

and destval in our code.[10] We also use void methods, which are not declared in our syntax.[11]

```
interface Iterator<T>{
   void step();
   dest T current();
}

interface List<T>{
   void add(lst-elem T e);
   void remove(read T e);
   lst-elem T contains(read T e);
   free<nit-dest=lst-elem> Iterator<Node<T>>
      getNodeIter();
}

interface Set<T>{
   void add(set-elem T e);
   void remove(read T e);
   set-elem T contains(read T e);
   free<dit-dest=set-elem> Iterator<T>
      getDataIter();
}
```

Inspecting the add method of the List<T> interface we see, that to add an element to a List, we must supply an element that has the same composite membership as the other list elements. Notice, that we are not saying anything about what the membership will be (e.g in the composite of the list object itself or in some other composite). This will depend on the context in which List<T> instantiations are used. The contains method tells us, that when we look for an element in the list, we can supply any element without worrying about its composite membership. But if we find this object, we return it with the information that it belongs to the same composite as all other list elements. The getNodeIter method returns an iterator that can be passed to other components (this is the meaning of free). At the same time we specify that this iterator's destination is in the same composite as the list elements (this is what the correlation <nit-dest=lst-elem> tells us). In the case of the Set<T> interface we can extract similar information: we can only add elements that belong to the same composite as all the other set elements. The set iterator can again be passed to other components, and the iterator's destination is in the same composite as all the set elements.

The *List* abstract object will be implemented with objects of the class Node<T>:

```
class Node<T>{
  co Node<T> next;
  data T value;

  void setNext(co Node<T> n){
    this.next = n;
  }

  void setValue(data T p){
    this.value = p;
  }

  co Node<T> getNext(){
    return this.next;
  }

  data T getValue(){
    return this.value;
  }
}
```

Here we notice that the next link points to an object in the same composite as the referring node (co Node<T> next). The values stored in our nodes belong to some yet unspecified composite data. The signature of the setNext method tells us, that we must provide an object belonging to the same composite as the node executing that method.

Next we implement a *node iterator*, that allows us to traverse nodes contained in some structure:

```
class NodeIt<T> implements Iterator<Node<T>>{
   nit-dest Node<T> curnode;

   void startAt(nit-dest Node<T> n){
      this.curnode = n;
   }

   void step(){
      this.curnode = this.curnode<=getNext();
   }

   nit-dest Node<T> current(){
      return this.curnode;
   }
}
```

Here the iterrator points to a current node that belongs to some composite nit-dest. To set up the initial point of the node traversal, we need to supply a node that belongs to that nit-dest composite.

We use the *node iterator* to implement an iterator DataIt<T> that traverses not the nodes themselves, but the values stored in the nodes:

```
class DataIt<T> implements Iterator<T>{
   rep<nit-dest=read<data=dit-dest>>
      Iterator<Node<T>> nIt;

   void wrap(free<nit-dest=read<data=dit-dest>>
                    Iterator<Node<T>> newnIt){
      this.nIt = newnIt;
   }

   void step(){
      this.nIt<=step();
   }

   dit-dest T current(){
      dit-dest T res;
      if (this.nIt<=current() != null){
         res = this.nIt<=current()<=getValue();
      }
      return res;
   }
}
```

This iterator "wraps" the *node iterator*. It puts the internal node iterator nIt into the composite controlled by the *data iterator* (the base mode rep does it). Any changes to the state of nIt can only be initiated on *data iterator*'s instigation. At the same time we specify that we don't expect any information about nit-dest composite

[10] The compiler can deduce, based on the context, which read operation should be used.

[11] We can view them as syntactic sugar for methods that return this as result and assign it right back to the variable trough wich the method was invoked. This is particularly helpful, if we try to send mutators over free references without losing them.

(setting `<nit-dest=read<...>>` says exactly this). The method `current` returns objects in `dit-dest` composite.

Now we are ready to direct our attention to the *List* implementation:

```
class ListImp<T> implements List<T>{
   rep<data=lst-elem> Node<T>  anchor;

   void add(lst-elem T e){
      rep<data=lst-elem> temp;

      temp = anchor;
      this.anchor = new<data=lst-elem> Node<T>();
      this.anchor<=setData( e );
      this.anchor<=setNext( temp );
   }

  void remove(read T e){
     ...
  }

  lst-elem T contains(read T e){
     ...
  }

  free<nit-dest=rep<data=lst-elem>> Iterator<T>
  getNIt(){
     free<nit-dest=rep<data=lst-elem>> NodeIt<T>
        nIt;

     nIt = new<nit-dest=rep<data=lst-elem>>
              NodeIt<T>()<=startAt(this.anchor);
     return nIt;
  }
}
```

The `anchor` points to the initial node that is put into the list's composite (via `rep`). Concurrently we specify that objects in that node's `data` composite belong to list's `lst-elem` composite. `getNIt` provides a *node iterator* over the node structure. The iterator belongs to *List*'s movables (`free` mode) and therefore can be safely passed to other composite objects (e.g the `SetImp<T>`). Also, the iterator's destination objects (the nodes) belong to list's composite (via `rep`).

The remaining part of the puzzle is the `SetImp<T>` class implementation:

```
class SetImp<T> implements Set<T> {
   rep<lst-elem=set-elem> T  entryList;

   void add(set-elem T e){
      if (entryList<=contains(e) == null){
         entryList<=add(e)
      }
   }

   set-elem contains (read T e){
      return entryList<=contains(e)
   }


   void remove(read T e){
      entryList<=remove(e)
   }


   free<dit-dest=set-elem> Iterator<T> getDIt(){
      free<nit-dest=read<data=set-elem>>
         NodeIt<T> nIt;
      free<dit-dest = set-elem>  dIt;

      nIt = entryList<=getNIt();
      dIt = new<dit-dest=set-elem> DataIt();
      return dIt<=wrap(nIt);
   }
}
```

The *Set* is implemented with the help of a *List* component. The list is put into set's composite, so only the set can make changes to list's structure. Even if some other objects have references to the nodes, they cannot send `setNext()` to them. The list elements (in the composite `lst-elem`) end up in the composite `set-elem`.

In the following code we define the class `OnlineLottery` that holds both, the winning numbers and the players' registrations. `winningNumbers` are in `OnlineLottery`'s composite and therefore cannot ever be transferred to another composite object. `playersReg` on the other hand is part of `OnlineLottery`'s movables and can at any point be "sold" to another "lottery enterprise".

```
class OnlineLottery {
   rep<set-elem=rep> SetImp<NUM>  winningNumbers;

   free<set-elem=reg> SetImp<REG>  playersReg;

   free<dit-dest=rep> Iterator<NUM> getWinIt(){
      return winningNumbers<=getDIt();
   }

   free<dit-dest=reg> Iterator<REG> getRegIt(){
      return playersReg<=getDIt();
   }

   void newDraw() {
      rep<dit-dest=rep> Iterator<T> internalIt
         = this<=getWinIt();

      while (internalIt<=current() != null) {
         internalIt<=current()<=setNum(random);
         internalIt<=step()
      }
   }

   free<set-elem=reg> SetImp<REG> sellRegSet() {
      return destval(playersReg);
   }

   read<set-elem=reg> SetImp<REG> exposeRegSet() {
      return val(playersReg);
   }
}
```

`OnlineLottery` class provides two iterators `getWinIt()` and `getRegIt()`, which either iterate other the set of winning numbers or the set of registrations. The `getWinIt()` iterator, when used internally (e.g. in the `newDraw` method), returns `rep` references that can be modified. When requested by an external client, the exported mode is adapted to `free<dit-dest=read>` allowing only read acces to the numbers.[12] Player's registrations belong to some composite `reg` and therefore `getRegIt()` iterator returns immutable references to an `OnlineLottery` object. On the other hand, clients that own `reg`, will receive mutable references from `getRegIt()`.

[12] See the import operation in section 4

Our `OnlineLottery` objects can sell the registration set, `playersReg`, to another gambling provider. Notice the `destval` in `sellRegSet`. The new owner can modify the `sellRegSet` (adding and removing registrations), but cannot change the content of any registrations (only the owner of the reg composite can). We can also expose the `playersReg` set, but the value of such operation is not clear here (the set cannot be modified - we can only obtain an iterator from such set, which is identical to the iterator obtained via `getRegIt()`).

If we want the `OnlineLottery` to own both, the list of registrations and the registrations themselves and later on transfer the ownership of the list and the registrations to another provider, we might be tempted to declare `playersReg` with the mode `free<set-elem=free>`. Unfortunately we cannot do it – our system does not allow it. We simply cannot guarantee the uniqueness of the references to the set elements (e.g. we could repeatedly call observers that return "`free`" references to the same set element and every-time store them in a different `free` variable of `OnlineLottery`). To make the described transfer feasible, we would need to modify the `Node` class. The mode of `value` needs to be change to `free`. The `OnlineLottery` (or `playersReg` set) owns now the registrations indirectly through the nodes, which are the direct owners of the registrations). The `DataIt` must be changed now as well, returning a `read` reference (via `val` – not `destval`).

8. Related Work

Blake and Cook were the first to characterize the problem of composite object encapsulation [4]. They warned that the common handing out of references to part objects enables clients to modify them in a way violating the integrity of the whole.

The Islands approach [11] proposes three techniques for making object interaction more predictable: the observer/mutator distinction, the uniqueness of certain references and the isolation of specific regions in the object graph (Islands). The work also contributes a system of access mode annotations with `read`, `unique` and `free`. `read` references cannot be assigned to variables but they can be bound to parameters. Island's `free` indicates references to whose target no other reference exist. `unique` is a variation on `free` with temporary aliases. Only un-captured references are allowed in or out of Islands. They must be either `read` or aliases of `unique`.

Flexible Alias Protection (FAP) [15] is another approach towards encapsulation of composite objects. FAP addresses the coupling caused through sharing of mutable state by a two-pronged strategy: the absence of all inbound references into composite objects representation and the independence of container objects from their contents' state. It is the first system to introduce the `rep` mode, which describes references from an object to its state-representing components. The ability to specify `rep` references by some kind of annotation is fundamental to nearly all typing disciplines for composite object encapsulation. FAP also introduces association roles α, for a user-defined classification of object references according to different semantic roles.

Ownership Types (OT) [8] was the first system of composite object encapsulation presented with complete formal definitions (typing rules, interpretation of annotations, encapsulation property) and a proof sketch. The authors introduced the graph-theoretical notion of dominator to define the relaxed hiding policy of representation containment. The concept of `co` references was introduced in OT (under the name `owner`). The authors also observed the importance of `co` for the proper typing of `this`. α roles from FAP reappear here as *context parameters* to the class. Like any hiding policy, OT excludes iterators and other common patterns. Some of the OT descendants and variations are [6], [5], [1], [12] and [16].

The Calculus in [9] is an ambitious foundational work on the isolation of regions in the object graph with several technical innovations. The OT system is generalized to cover the missing language features and make it more flexible. The formalization is done with the help of an object calculus. The decisive step towards more flexibility was to loosen the connection between the structure of object composition and the nesting of protection domains, the ownership contexts.

Universes [14] is the first technique that enforces a policy of encapsulation without hiding. Universes simplify OT by replacing OTs problematic context parameters by runtime ownership checks. Universes prevent flexible object creation and composition by fixing new objects owner always to their creator.

AliasJava [2] is characterized as a capability-based system. It combines aliasing annotations with ownership annotations. It makes aliasing patterns explicit and enforces a relaxed hiding policy. The authors are the first to develop a constraint-based algorithm for inferring the new annotations, and the first to report on the usability of their system for real-world software like Java's standard library. A drawback of AliasJava is its need to represent ownership parameters at runtime.

The work most closely related to ours is [13] and [10] (the later evolved from [14]). [13] introduces a novel type system "Effective Ownership Types" (EOT). Each method definition is provided with effective owners. A method owned by o can only update objects with an owner that dominates o. The ownership tree is established in the same manner as in OT. The static type system tracks down the unsafe mutations. As in our system, object references and non-mutating access are unrestricted. OT system is a special case of EOT, where all methods belong to the owner of the defining object. Unlike in our system, mutator calls via inside-out references are permitted. EOT also can express mutating iterators. Such iterator carries a reference to its collection object and can therefore add and delete elements by making calls on the collection's interface. This is not possible in our system. Our system allows the safe transfer of sub-components (inside the *movables*) from one composite object to another. In EOT the owner of an object is fixed for its lifetime and transfer of sub-components is not possible.

Transfer of ownership has been first described in [7]. The authors introduce the concept of "external uniqueness". Here `unique` describes the only reference into an aggregate from outside the aggregate. Internal aliases to a unique reference are permitted. The authors work in the owner-as dominator setting. In *JaM*, `free` references can have arbitrary `read` aliases and the `free` reference as well as its aliases can be captured in variables.

Ownership transfer is not possible in the Universe Types system [14], [10]. As in our system `read-only` references (or `any` in [10]) are allowed to cross the boundary of encapsulation. In both cases modification of objects through such references is disallowed. [14] cannot produce iterators that deliver mutable objects (a dynamic downcast from `read` to `rep` is required). This has been rectified in [10]. The Viewpoint Adaptation in [10] is closely related to our Signature Import. In our system we have to deal additionally with `free` modes. Without them Viewpoint Adaptation and Signature Import appear almost identical.

9. Conclusion

We presented the *Potential Access Path* methodology as a toll to reason about composite objects and their state. The main technical result of this paper is the *Composite State Encapsulation* property – a guarantee that modifications to composite object's state are controlled solely by its representative.

Our system enables the definition of nested composite objects with a complex internal structure, their observation through external iterator objects, their incremental construction, and their trans-

fer across abstraction boundaries. It is a purely static system in which container objects and their iterator objects can each be encapsulated individually (state-protected from one another).

The flexibility of our system results from a novel weak uniqueness property for reference paths. That property generalizes the standard notion of free or unique references (which are not aliased by any references). We believe that our system is one of the first to combine object-as-modifier discipline with transfer of ownership.

In our system all ownership information is removed from objects. This should mitigate the loss of ownership information problem in subclassing. The association modes (or roles) are not placeholders for reference target's owner, but uninterpreted type tags on object's references. The available roles are not limited by a parameter list, nor by the references targeting it. Our system allows the bottom-up construction process, in which sub-objects are created before their owners.

10. Acknowledgments

We would like to thank Gary Leavens for his encouragement and insightful comments on Schünemann's thesis.

Fiech's and Schünemann's research was partially supported by NSERC grant OGP0170497.

References

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP '04: Proceedings of the 18th European Conference on Object-Oriented Programming*, 2004.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 311–330, New York, NY, USA, 2002. ACM Press.

[3] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECCOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 32–59. Springer-Verlag, 1997.

[4] Edwin H. Blake and Steve Cook. On including part hierarchies in object-oriented languages with an implementation in smalltalk. In *ECOOP '87: Proceedings of the European Conference on Object-Oriented Programming*, pages 41–50, London, UK, 1987. Springer-Verlag.

[5] Chandrasekhar Boyapati, Barbara Liskov, and Liuba 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] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 292–310, New York, NY, USA, 2002. ACM Press.

[7] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP '03: Proceedings of the 17th European Conference on Object-Oriented Programming*, 2003.

[8] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64, New York, NY, USA, 1998. ACM Press.

[9] David Gerard Clarke. *Object ownership and containment*. PhD thesis, University of New South Wales, 2002.

[10] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP '07: Proceedings of the 21st European Conference on Object-Oriented Programming*, 2007.

[11] John Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA '91: Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, pages 271–285, New York, NY, USA, 1991. ACM Press.

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

[13] Yi Lu and John Potter. Protecting representation with effect encapsulation. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 359–371, New York, NY, USA, 2006. ACM Press.

[14] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical report, Fernuniversität Hagen, 2001.

[15] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 158–185, London, UK, 1998. Springer-Verlag.

[16] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic java. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 311–324, New York, NY, USA, 2006. ACM Press.

[17] Ulf Schünemann. *Composite objects: dynamic representation and encapsulation by static classification of object references*. PhD thesis, Memorial University of Newfoundland, 2005.