On Deleting Aggregate Objects
Draft

David Clarke
email: clad@cs.uu.nl*

January 8, 2001

Abstract

We describe a typed object calculus which makes explicit the nesting between objects. The calculus is based on Abadi and Cardelli's object calculus [1] extended with *regions*.[1] Regions have properties describing their nesting and the bounds on their access. They are used not only in a stack-based manner, but also to store an object's private implementation. This creates opportunities to improve memory management. In particular, the calculus allows the entire private implementation of an aggregate object to be deleted when the interface to the aggregate becomes garbage. The calculus also allows entire aggregate objects to be allocated on stack-based regions.

This work is the first attempt exploration of the opportunities to improve memory management which were inherent in our earlier work. The formal development presented here seems to be correct, though it's properties have not been formally demonstrated. The reader is refered to an earlier work [5] for a more thorough development which does not venture as far as this does, or to the soon-to-appear thesis [8].

1 Introduction

Before starting, let's fix some terminology. Rather than consider objects to be without structure, we exploit the nesting inherent in the notion of *aggregation* used in object-oriented design [12]. An *aggregate object* consists of one or more *interface objects* and a collection of *implementation objects* which are private to the aggregate object. Both the interface and implementation objects may be aggregate objects. Not all objects an object accesses are considered part of its implementation; some objects are not protected. For concision, we use the words *aggregate*, *interface*, and *implementation*.

In previous work we presented a type system for protecting the implementation of an aggregate [6], where the aggregate consisted of only a single interface object and multiple implementation objects. The idea was to enable a more private `private` which prevented direct access to the implementation from outside the aggregate. This type system enforces on that the interface of an aggregate is a *dominator*[2] for access paths from the root of the object graph to the aggregate's implementation. A consequence is that when the interface becomes garbage, its entire implementation can also be deleted.

A typical example the calculus allowed was a linked list where the handle of the list was the interface and the links the implementation. The data in the list is not considered a part of the implementation. The only access to the links is through the handle. Each of the links is afforded the same amount of protection. When the handle is deleted, then so may all the links.

*This work was conducted at the School of Computer Science and Engineering, University of New South Wales, Sydney. From April 2001 the author will be at the Institute of Information and Computer Science, Utrecht University, Utrecht, The Netherlands.

[1] We used the word *context* instead of *region* in previous work.

[2] A node a is a *dominator* for paths from node b to node c if all paths from b to c include a.

This calculus is, however, limited in practice. Although the original proposal lacked inheritance, this can be easily regained [7]. The problem is that the dominance property is too strong to allow the kinds of idioms used in practice. For example, to use an iterator on a linked list meant that the links could not be given the same amount of protection previously stated, because both the handle and the iterator need direct access to the links, which is prevented by the dominator property.

Recent work extended the underlying containment model, recasting the entire type system in terms of Abadi and Cardelli's object calculus [5]. Doing so allowed aggregates with multiple interfaces, but the memory management properties were lost. This was because the types system did not contain enough information to guarantee the desired invariants. Certain syntactic restrictions are possible to avoid certain behaviour in the core calculus and regain previous properties (and limitations). We would, however, rather that the type system stated whenever such properties were satisfied, without the limitations. For example, we want to know whether a certain object is the dominator for access to its implementation.

In this work we strengthen the properties specified by types to allow varying different levels of protection to be specified in the same type system.

Whole aggregate objects can be deleted; the implementation objects, not just the interface. The key to achieving this is to store objects in regions, to have regions associated with both objects and stacks frames, and to attach properties to the regions. The properties control access from objects in one region to objects in other regions, and give bounds on this access, which are used to guarantee certain memory management invariants. In the framework we can also have stack-based memory regions which cannot be accessed outside of the stack frame. These allow complete objects graphs to be allocated and deleted in a stack-based manner (using a stack of heaps).

2 The Calculus

The calculus is a minor extension of our object calculus with ownership and containment [5], which in turn is an extension of Abadi and Cardelli's object calculus [1]. We now briefly describe its syntax, given in Figure 1.

Regions consists of the root region ϵ, which is the only region present when a program begins, and variables, denoting regions introduced using **new**. Only objects are stored in regions. The region ϵ is accessible to all objects. Regions are nested; this is captured by the relation $\prec:$, called *inside*. $p \prec: q$ means that p is inside q; all regions are inside ϵ, that is, $p \prec: \epsilon$ for all p. The nesting of the regions is the basis for controlling access between objects. Specifically playing that role are permissions.

Permissions are used to control access to objects, from which we obtain our desired invariants. The *point* permission $\langle p \rangle$ allows access to objects in region p. The *upset* permission $\langle p\uparrow \rangle$ allows access to objects in any region q where $p \prec: q$, that is any region enclosing p.

Types consist of object types and existential types which hide region names. This departs significantly from traditional region calculi [14], because the region use is no longer lexically scoped. In the object type $[l_i : \Theta_i{}^{i \in 1..n}]^p_q$, p is the region in which the object resides, and q is the region in which it stores its implementation. Existential quantification over regions allows regions to be abstracted away, though their properties are retained, and thus allows new regions to be a part of types, while keeping the type system statically defined. The type $\exists(\alpha \lhd p/P)A$ means that there is some region denoted α which is inside p and satisfies region bound P, described in a moment.

Method types Θ are distinct from types to distinguish between evaluation inside and outside an object. They specify which arguments are required; method evaluation with those arguments is considered to be inside the object — the result is considered to be returned from the inside to the outside of the object.

Values are terms which are the results of computation. Values are either variables x, locations ι, as the object-oriented language is imperative, or the term hiding a region (with some properties)

Regions

$$p \quad ::= \quad \alpha \mid \epsilon$$

Permissions

$$K \quad ::= \quad \langle p \rangle \mid \langle p\uparrow \rangle$$

Types

$$A, B \quad ::= \quad [l_i : \Theta_i{}^{i \in 1..n}]_q^p \mid \exists(\alpha \lhd p/P)A$$

Method Types

$$\Theta \quad ::= \quad A \mid A \to \Theta$$

Values

$$\begin{array}{rcl} u, v & ::= & x \\ & \mid & \iota \\ & \mid & \textbf{hide } p \textbf{ as } \alpha \lhd q/P \textbf{ in } v{:}A \end{array}$$

Terms

$$\begin{array}{rcl} a, b & ::= & v \\ & \mid & v.l\langle\Delta\rangle \\ & \mid & v.l \Leftarrow \varsigma(s : A, \Gamma)b \\ & \mid & \textbf{let } x : A = a \textbf{ in } b \\ & \mid & \textbf{expose } v \textbf{ as } \alpha \lhd p/P, x{:}A \textbf{ in } b{:}B \\ & \mid & \textbf{new } \alpha \lhd p/\bot \textbf{ in } a \\ & \mid & \textbf{new } \alpha \lhd p/P \textbf{ in let } \vec{x} = \vec{a} \textbf{ in } wfo_\alpha \end{array}$$

Object (with implementation region α)

$$\begin{array}{rcl} wfo_\alpha & ::= & [l_i = \varsigma(s_i : A_i, \Gamma_i)b_i{}^{i \in 1..n}]_\alpha^q \\ & \mid & \textbf{hide } \alpha \textbf{ as } \alpha \lhd p/P \textbf{ in } wfo_\alpha{:}A \end{array}$$

Region Bounds

$$\begin{array}{rcl} P & ::= & p \\ & \mid & \bot \end{array}$$

Actual Parameters

$$\Delta \quad ::= \quad \emptyset \mid v, \Delta$$

Formal Parameters

$$\Gamma \quad ::= \quad \emptyset \mid x : A, \Gamma$$

Figure 1: The Syntax

within a value, **hide** p **as** $\alpha \lhd q/P$ **in** $v{:}A$. The hidden region p is generally the implementation region of some object.

Terms are presented in the named form [13] which helps with proving properties.[3] A term is either a value, method selection $v.l\langle\Delta\rangle$ with arguments Δ, method update $v.l \Leftarrow \varsigma(s : A, \Gamma)b$, a **let** expression for local declarations, or an existential unpacking **expose** v **as** $\alpha \lhd p/P, x{:}A$ **in** $b{:}B$. The semantics of these terms are as usual, except that methods take arguments. The remaining terms are more interesting.

[3] Future work may see a change to Gordon and Hankin's concurrent object calculus [9], possibly using Cardelli, Ghelli, and Gordon's Groups [3, 4].

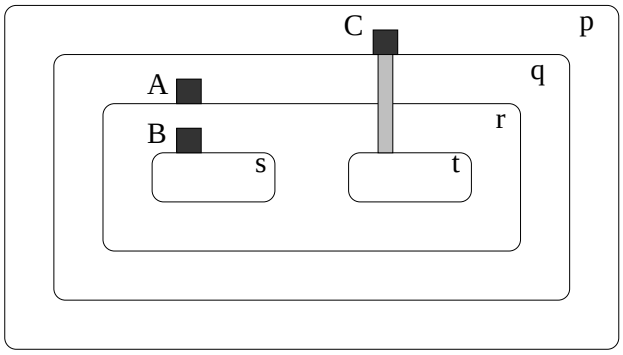

Figure 2: Bounded Region Access

2.1 Objects and Regions

The term **new** $\alpha \lhd p/\bot$ **in** a corresponds approximately to the **letregion** term from the regions calculus [14]. This term creates a new region α which is directly inside the region p, but is bounded so that it cannot be used outside of itself. This means that there cannot be a reference from an object residing outside of α to an object which is in α or a region inside α. In addition this region can be deallocated when term a has finished evaluating.[4] We call this sort of region *exclusive*.

The remaining syntax **new** $\alpha \lhd p/P$ **in let** $\vec{x} = \vec{a}$ **in** wfo_α specified a new object with implementation region α (bound in the scope of this expression). **let** $\vec{x} = \vec{a}$ **in** ... denotes a series of **let** expressions which are used to initialise the object, including its private implementation. The syntax is described in this manner, that is, parameterised by the region α, so that every object has a unique implementation region.

Objects have the following form $[l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]^p_q$. The region the object resides in, it's interface region, is p, and q is the region where it stores its private implementation. These, combined with the nesting of regions, govern which objects can access which other objects. The constraint is stated as follows:

$$\iota \to \iota' \Longrightarrow \mathsf{impl}(\iota) \prec: \mathsf{int}(\iota')$$

where $\mathsf{impl}(\iota)$ is the implementation region and $\mathsf{int}(\iota')$ is the interface region. This states: an object can only access other objects whose interface region is outside its implementation region. Thus the regions play a double role of representing where the objects reside, for memory management purposes, and for access control, from which the invariants on memory management are derived.

An object is capable of *surreptitious* access whenever the implementation region is not directly inside the interface region. The purpose of the bound is to place a cap on the extent of surreptitious access for a particular region. (C in Figure 2.)

The implementation region can be either *bounded* or *exclusive*.

A region created with **new** where the bound P is some region p is called *bounded*. This represents the outermost region which can directly access the given region. Figure 2 illustrates.

Here we have regions which are nested as follows $s \prec: r \prec: q \prec: p$ and $t \prec: r$, and three objects A, B, and C, where

$$\begin{array}{ll} \mathsf{int}(A) = q & \mathsf{impl}(A) = r \\ \mathsf{int}(B) = r & \mathsf{impl}(B) = s \\ \mathsf{int}(C) = p & \mathsf{impl}(C) = t. \end{array}$$

Each object can access the other, but B is less accessible that A, which is less accessible than C. B could be the implementation of aggregate A, and C could be an iterator which is used by someone using the A as a part of its implementation.

Consider when object A was created. Its implementation region r was created to be inside q, with a bound p. The bound represents the maximal region where the interface of C can reside

[4] This property remains to be demonstrated formally.

— C's interface could not be placed in any region further out. Furthermore, any object which resembles C with a protruding interface, but is used inside C, is also bounded by p. (We use transitive bounds, described in a moment, to enforce this.)

If P is $\bot$ then the region again is called *exclusive*. In this case there can exist no object with surreptitious access like C. If the bound on region r is $\bot$, then the maximum interface region an object with implementation region inside A can have is r, for example, B. (When r is a stack-allocated region then there is no object A either, just the region r.) When an objects implementation region is exclusive, then the region does not exhibit the stack-based deallocation described before. It does however ensure that the interface object is the single access point to the aggregate it constructs. This means that when the object becomes garbage, every region inside r and the objects they contain can be safely deleted.

As a part of the type system we have the following functions from region variables to regions: UB and TB. The first gives the upper bound on a region variable. This gives the maximal region where the interface can go. The second gives the upper bound on regions created within the region. This property is inherited by the other regions.

These functions have the following values, as per the kind of behaviour we wish to exhibit:

stack allocated Regions created using **new** $\alpha \lhd p/\bot$ **in** a, where a is not an object, have the following properties:

- $\text{UB}(\alpha) = p$. Note that it is effectively undefined because this is only used when typing objects which store their implementation in α;
- $\text{TB}(\alpha) = \alpha$.

Region is not attached to an object. Prevents any objects like C above.

single entry Regions created using **new** $\alpha \lhd p/\bot$ **in** wfo_α are used to store the implementation of an aggregate. In this case, the object is the single access point to the implementation. Region α has the following properties:

- $\text{UB}(\alpha) = p$;
- $\text{TB}(\alpha) = \alpha$

Again prevents objects like C.

bounded Regions created using **new** $\alpha \lhd p/p'$ **in** wfo_α are used to store the implementation of an aggregate.

- $\text{UB}(\alpha) = p'$; and
- $\text{TB}(\alpha) = p'$.

This allows object's like C, where the bound on the interface region is p', and this bound is inherited by any objects whose implementation region is inside α.

If the bound $P = \epsilon$ for every region, then we have effectively the system described in [5]. If $P = \bot$ for every region, then we effectively have the system described in [6]. In a sense, our two previous major type systems are two extremes of the protection landscape provided by the type system presented here.

3 The Type Rules

Typing environments have the following form:

$$E \quad ::= \quad \emptyset \mid E, x : A \mid E, \iota : A \mid E, \alpha \lhd p/P$$

$E \vdash \diamond$	*good environment*
$E \vdash p$	*good region* p
$E \vdash p \prec: q$	p *is inside* q
$E \vdash K$	*good permission* K
$E \vdash K \subseteq K'$	K *is a subpermission of* K'
$E; K \vdash A$	*good type* A
$E; K \vdash A <: B$	A *is a subtype of* B
$E; K \vdash a : A$	*good expression* a *of type* A
$E; K \vdash \Theta\ meth$	*good method type* Θ
$E; K \vdash (\Theta)(\Delta) \Rightarrow C$	*actuals arguments* Δ *match method type* Θ*; return type is* C

Figure 3: Judgements

The type system consists of the judgements described in Figure 3. (Others dealing with stores and configurations are required to provide a complete account.)

Note that permissions K control both type and term formation. The idea is to limit the objects and locations accessed in a given term to those residing in one of the regions indicated by the permission [5].

The first three type rules follow the standard pattern. Locations have object type.

$$\text{(Env } \emptyset\text{)}\quad \frac{}{\emptyset \vdash \diamond} \qquad \text{(Env } x\text{)}\quad \frac{E; K \vdash A \quad x \notin \mathsf{dom}(E)}{E, x : A \vdash \diamond} \qquad \text{(Env Location)}\quad \frac{E; \langle p \rangle \vdash [l_i : \Theta_i{}^{i \in 1..n}]^p_q \quad \iota \notin \mathsf{dom}(E)}{E, \iota : [l_i : \Theta_i{}^{i \in 1..n}]^p_q \vdash \diamond}$$

There are two rules for adding new region variables to an environment, depending whether the region is bounded or exclusive. A bounded region can be created inside any other region p, so long as we ensure that the bound q is at most the transitive bound of p (Env $\lhd$1). This prevents an object in region α from being accessed directly by an object outside of $\textsc{tb}(p)$, as this would violate the intended meaning of $\textsc{tb}(\ )$. An exclusive region can be creating inside any valid region (Env $\lhd$2).

$$\text{(Env } \lhd 1\text{)}\quad \frac{E \vdash q \prec: \textsc{tb}(p) \quad \alpha \notin \mathsf{dom}(E)}{E, \alpha \lhd p/q \vdash \diamond} \qquad \text{(Env } \lhd 2\text{)}\quad \frac{E \vdash p \quad \alpha \notin \mathsf{dom}(E)}{E, \alpha \lhd p/\bot \vdash \diamond}$$

Given some environment, a valid region is either the top level region ϵ or any declared in the environment.

The nesting of regions is derived from the variable declarations. Additional rules deal with $\bot$ to make subtyping existential types more flexible and uniform. The rules for reflexivity and transitivity of $\prec:$ have been omitted.

$$\text{(Region } \epsilon\text{)}\quad \frac{E \vdash \diamond}{E \vdash \epsilon} \qquad \text{(Region } \alpha\text{)}\quad \frac{\alpha \in \mathsf{dom}(E)}{E \vdash \alpha} \qquad \text{(In } \prec:\text{)}\quad \frac{\alpha \lhd p/P \in E}{E \vdash \alpha \prec: p} \qquad \text{(In } \bot\bot\text{)}\quad \frac{E \vdash \diamond}{E \vdash \bot \prec: \bot} \qquad \text{(In } \bot\text{)}\quad \frac{E \vdash p}{E \vdash \bot \prec: p}$$

The upper and transitive bounds too are derived from the region declarations in the environment, as described in Section 2.1.

$$\frac{\alpha \lhd p/q \in E}{E \vdash q \prec: \mathrm{UB}(\alpha)}\ (\text{In UB1}) \qquad \frac{\alpha \lhd p/\bot \in E}{E \vdash p \prec: \mathrm{UB}(\alpha)}\ (\text{In UB2}) \qquad \frac{\alpha \lhd p/q \in E}{E \vdash q \prec: \mathrm{TB}(\alpha)}\ (\text{In TB1}) \qquad \frac{\alpha \lhd p/\bot \in E}{E \vdash \alpha \prec: \mathrm{TB}(\alpha)}\ (\text{In TB2})$$

These rules are really saying a number of things which have been compressed into a single set of rules. For example, (In UB1) is really saying that

$$\frac{E \vdash q \prec: q \quad \dfrac{\alpha \lhd p/q \in E}{E \vdash \mathrm{UB}(\alpha) = q}}{E \vdash q \prec: \mathrm{UB}(\alpha)} \text{ using } \frac{\alpha \lhd p/q \in E}{E \vdash \mathrm{UB}(\alpha) = q}\ (\text{UB1}) \text{ and } \frac{E \vdash p \prec: q \quad E \vdash q' = q}{E \vdash p \prec: q'}\ (\text{In} =)$$

The rules as presented provide enough information for our present purpose.

Any valid region can be used to create either a point (Perm p) or upset permission (Perm $p\uparrow$). The subpermission relation is derived directly from the nesting of regions. It is based on the idea that if you have permission to access a region, you can access all the regions enclosing it — allied with standard scoping rules. Rules for reflexivity and transitivity of $\subseteq$ have been omitted.

$$\frac{E \vdash p}{E \vdash \langle p \rangle}\ (\text{Perm } p) \qquad \frac{E \vdash p}{E \vdash \langle p\uparrow \rangle}\ (\text{Perm } p\uparrow) \qquad \frac{E \vdash p}{E \vdash \langle p \rangle \subseteq \langle p\uparrow \rangle}\ (\text{SubPerm } p) \qquad \frac{E \vdash q \prec: p}{E \vdash \langle p\uparrow \rangle \subseteq \langle q\uparrow \rangle}\ (\text{SubPerm } \prec:)$$

It is in the rules for types, particularly for objects, where all the machinery starts to work.

The first clause of (Type Object), $E; \langle q\uparrow \rangle \vdash \Theta_i\ meth$, ensures that all parts of the method type which concern object types reside in regions enclosing q. This helps maintain a particular structural invariant on object graphs which allows private implementation to be protected, but also to be accessed by multiple interfaces [5]. The clause $E \vdash q \prec: p$ ensures that the objects can access themselves. The final clause $E \vdash p \prec: \mathrm{UB}(q)$ ensures that the object is not placed in a region which should not have access to region q.

The subtyping rule for objects allows only width but not depth subtyping, though this could easily be extended using variance annotations [1]. Neither region component can change.

$$\frac{E; \langle q\uparrow \rangle \vdash \Theta_i\ meth \qquad \forall i \in 1..n \quad E \vdash q \prec: p \quad E \vdash p \prec: \mathrm{UB}(q)}{E; \langle p \rangle \vdash [l_i : \Theta_i{}^{i \in 1..n}]^p_q}\ (\text{Type Object}) \ (l_i \text{ distinct})$$

$$\frac{E; \langle q\uparrow \rangle \vdash \Theta_i\ meth \quad \forall i \in 1..n+m \quad E \vdash q \prec: p \quad E \vdash p \prec: \mathrm{UB}(q)}{E; \langle p \rangle \vdash [l_i : \Theta_i{}^{i \in 1..n+m}]^p_q <: [l_i : \Theta_i{}^{i \in 1..n}]^p_q}\ (\text{Sub Object}) \ (l_i \text{ distinct})$$

Existential types quantify only over regions (though the usual existential could easily be added). This is to hide the details of the new implementation region part of an object's type. Since the region in which an object resides is used to determine which other objects can access it, from which the subsequent structural invariants are attained, a key aspect of this type rule is to prevent hiding that region. This is done with an additional side condition: from $E \vdash K$ and $\alpha \notin \mathsf{dom}(E)$, it follows that α is not in the permissions K. Since this governs the top level regions which the can occur in object types appearing in type B, we conclude that none of these object types can reside in α.

The subtyping rule follows the standard pattern, except that the p and P components vary in opposite directions. The p component is the region which α is inside. Logically, because $\prec:$

is transitive, α is also inside any region which p is inside. The type system, on the other hand, prevents direct access from any region outside P. By tightening the bound no harm is done. Indeed, when $P' = \bot$ the type prevents any surreptitious references from being created.

$$\frac{E, \alpha \lhd p/P; K \vdash B \quad E \vdash K}{E; K \vdash \exists(\alpha \lhd p/P)B}\ (\text{Type Exists}) \qquad \frac{E \vdash p \prec: p' \quad E \vdash P' \prec: P \quad E, \alpha \lhd p/P; K \vdash A <: A' \quad E \vdash K}{E; K \vdash \exists(\alpha \lhd p/P)A <: \exists(\alpha \lhd p'/P')A'}\ (\text{Sub Exists})$$

Lifting allows types and expressions which can be formed given some permission K to be valid with a larger permission K'.

$$\frac{E; K \vdash A \quad E \vdash K \subseteq K'}{E; K' \vdash A}\ (\text{Type Lift}) \qquad \frac{E; K \vdash A <: B \quad E \vdash K \subseteq K'}{E; K' \vdash A <: B}\ (\text{Sub Lift})$$

The subtype relation $<:$ is also reflexive and transitive, but again these rules have been omitted.

The method type rules allow methods to have any number of arguments and a return type which are all types. Methods cannot take functions as parameters, but functions can be encoded or added with little effort. Presenting method types as a separate colleciton of type rules makes sense with more complicated versions of this type system.

$$\frac{E; K \vdash A}{E; K \vdash A\ meth}\ (\text{Type Return}) \qquad \frac{E; K \vdash A \quad E; K \vdash \Theta\ meth}{E; K \vdash A \to \Theta\ meth}\ (\text{Type Arrow})$$

A number of the term typing rules rely on the function $\lfloor \Gamma \rfloor_C$ which converts the method arguments Γ and the return type C into a method type Θ:

Definition 1 ($\lfloor \Gamma \rfloor_C$)

$$\begin{aligned} \lfloor \emptyset \rfloor_C &\ \widehat{=}\ C \\ \lfloor x : A, \Gamma \rfloor_C &\ \widehat{=}\ A \to \lfloor \Gamma \rfloor_C \end{aligned}$$

Locations and variables are typed as per assumption, so long as the permission is large enough to construct the appropriate type.

$$\frac{x : A \in E \quad E; K \vdash A}{E; K \vdash x : A}\ (\text{Val } x) \qquad \frac{\iota : [l_i : \Theta_i{}^{i \in 1..n}]^p_q \in E}{E; \langle p \rangle \vdash \iota : [l_i : \Theta_i{}^{i \in 1..n}]^p_q}\ (\text{Val Location})$$

The rule (Val Object) includes the usual rule for well-formed objects, taking account of the multiple method arguments. In addition, it restricts the objects and locations which can be accessed in the method bodies to those included in permission $\langle q\uparrow \rangle$, where q is the implementation region. This means an object can access any enclosing region. Because the object's type A is well-formed, we also have that $E \vdash q \prec: p$, thus an object can access itself. We also have the appropriate bound restriction which limits access to the implementation region. The conclusion of the rule states that the object can be accessed in any expression having permssion to access the interface region p.

(Val Object) (where $A \equiv [l_i : \Theta_i{}^{i \in 1..n}]^p_q$ and $\Theta_i \equiv \lfloor \Gamma_i \rfloor_{C_i}$)

$$\frac{E, s_i : A, \Gamma_i; \langle q\uparrow \rangle \vdash b_i : C_i \quad \forall i \in 1..n}{E; \langle p \rangle \vdash [l_i = \varsigma(s_i : A, \Gamma_i) b_i{}^{i \in 1..n}]^p_q : A}$$

Method selection depends on the following additional type rules for checking that the method arguments are correct in type and number. These are (Arg Empty) and (Arg Val).

(Arg Empty)

$$\frac{E; K \vdash C}{E; K \vdash (C)(\emptyset) \Rightarrow C}$$

(Arg Val)

$$\frac{E; K \vdash v : A \quad E; K \vdash (\Theta)(\Delta) \Rightarrow C}{E; K \vdash (A \to \Theta)(v, \Delta) \Rightarrow C}$$

Method selection simply requires that the arguments match the declared type, and the return type is the return type of the method type.

(Val Select)

$$\frac{E; K \vdash v : [l_i : \Theta_i{}^{i \in 1..n}]^p_q \quad E; K \vdash (\Theta_j)(\Delta) \Rightarrow C_j \quad j \in 1..n}{E; K \vdash v.l_j \langle \Delta \rangle : C_j}$$

Method update ensures that the new method has the same type as the old, and that any objects and locations which it accesses are accessible both to the object and in the context which installs the method. (That is, $E \vdash K' \subseteq \langle q\uparrow \rangle$ and $E \vdash K' \subseteq K$ in effect specify that K' is in the intersection of $\langle q\uparrow \rangle$ and K.)

(Val Update) (where $A \equiv [l_i : \Theta_i{}^{i \in 1..n}]^p_q$ and $\Theta_j \equiv \lfloor \Gamma_j \rfloor_{C_j}$)

$$\frac{\begin{array}{c} E; K \vdash v : A \quad E, s : A, \Gamma_j; K' \vdash b : C_j \\ E \vdash K' \subseteq \langle q\uparrow \rangle \quad E \vdash K' \subseteq K \quad j \in 1..n \end{array}}{E; K \vdash v.l_j \Leftarrow \varsigma(s : A, \Gamma_j) b : A}$$

The type rule for **new** introduces a new region with particular properties into an expression. The permission $\langle \alpha \rangle$ in the first premiss allows objects to be creating in the new region and perhaps included as part of an aggregate's implementation, but the second premiss ensures that α does not occur in the type, preventing the new region name from appearing in the return type. This type rule resembles the one for new names in the $\lambda\nu$ calculus [11, 10].

(Val New)

$$\frac{E, \alpha \lhd p/P; K \cup \langle \alpha \rangle \vdash a : A \quad E; K \vdash A}{E; K \vdash \mathbf{new}\ \alpha \lhd p/P\ \mathbf{in}\ a : A}$$

Packing and repacking existential types follow the usual pattern (except for the name change). The additional clause to (Val Hide) ensures that the type $\exists(\alpha \lhd q/P)A$ is well-formed, and thus that **hide** does not abstract away the region in which any objects reside, as this region is essential for controlling object graph structure.

(Val Hide)

$$\frac{E \vdash p \prec: q \quad E; K \vdash v\{\!\{{}^p/_\alpha\}\!\} : A\{\!\{{}^p/_\alpha\}\!\} \quad E; K \vdash \exists(\alpha \lhd q/P)A}{E; K \vdash \mathbf{hide}\ p\ \mathbf{as}\ \alpha \lhd q/P\ \mathbf{in}\ v{:}A : \exists(\alpha \lhd q/P)A}$$

(Val Expose)

$$\frac{E; K \vdash v : \exists(\alpha \lhd p/P)A \quad E; K \vdash B \quad E, \alpha \lhd p/P, x : A; K \cup \langle\alpha\rangle \vdash b : B}{E; K \vdash \textbf{expose}\ v\ \textbf{as}\ \alpha \lhd p/P, x{:}A\ \textbf{in}\ b{:}B : B}$$

Subsumption additionally allows the permission required to type an expression to increase.

(Val Subsumption)

$$\frac{E; K \vdash a : A \quad E; K' \vdash A <: B \quad E \vdash K \subseteq K'}{E; K' \vdash a : B}$$

The operational semantics of the calculus are straightforward extensions of those for the previous version of this system. The regions have no effect on computation, but the semantics tracks new regions and their properties in terms of existing regions. In previous work this was used to prove the appropriate invariants, and we anticipate that this will be the case here.

4 Applications

The calculus has a number of principle applications:

- objects which can be have their entire implementation deleted when they become garbage;
- bounds on access to aggregates used to derive other constraints on object lifetimes; and
- stack-based allocation of objects — which, using the above two features, allows entire aggregate objects to be allocated in a stack-based manner, using a stack of heaps.

But there are other applications in the area for which this calculus was orignially devised. There is a problem with so-called *vampiric* behaviour [5] — which amounts lacking control over region creation — is dealt with adequately by this type system, using the $\bot$ bound. Using the same kind of type in a different manner (if the type system is extended with region parameterisation in methods following [5]) allows something resembling *principals* to be regained, in principle [15]. This is called *borrowing* elsewhere. This occurs when an object passes another object, possibly part of its implementation, to another object's method, and is guaranteed that the other object will not retain a reference in any shape or form [2]. As a simple example, consider a `sort` function which sorts the elements of a list. One would hope that this did not retain any references to the list being sorted or any of its elements, but one would wish that it could use its own intermediate data structures to do the sorting. The form of borrowing available in minor extensions to this type system can accommodate such idioms.

5 Discussion

The notions of bounded containment presented here may seem a little complicated, but they are based on notions which already exist in programming languages. They are in part inspired by the variable scoping of inner classes (and many other things), combined with genuine privacy (which protects objects), and the bounds on the general way aggregates are used in practice.

These notions in classes are generally static, whereas our nesting and bounds are entirely dynamic, and the kinds of protection on offer (such as via privacy annotations) work only at a class or package not object level. Here we deal with the actual objects.

6 Future

Much remains to be done. As we complete the formal development of the type system, we hope also to simplify it by independently studying the underlying containment model. The aim is to discover invariants on object graph structure which can be stated simply. This should also assist with the statement and proof of the memory management properties.

As such, the type system is too complicated to use directly in a programming language. With minimal annotations, perhaps indicating which objects are part of the implementation, and by leveraging off the existing nesting structures already present in programming languages (for example, inner classes), a static program analysis may use the type system as a target as a target, aiming to get a more private `private`.

Once the constraints between object and region life-times have been established, hybrid algorithms which combine regions with garbage collection can then be designed, implemented, and evaluated. This of course will not happen overnight.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] John Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 2001. to appear.

[3] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. In *Theoretical Computer Science; Exploring New Frontiers in Theoretical Informatics. International Conference IFIP TCS 2000*, volume 1872 of *LNCS*, pages 333–347, 2000.

[4] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In *CONCUR 2000 – Concurrency Theory. 11th International Conference*, volume 1877 of *LNCS*, pages 365–379, August 2000.

[5] David Clarke. An object calculus with ownership and containment. In *Foundations of Object-oriented Programming (FOOL8)*, London, January 2001. Available from `http://www.cs.williams.edu/~kim/FOOL/FOOL8.html`.

[6] David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA Proceedings*, 1998.

[7] David Clarke, Ryan Shelswell, John Potter, and James Noble. Object ownership to order. Unpublished manuscript.

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

[9] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *HLCL'98*, Elsevier ENTCS, 1998.

[10] Martin Odersky. A syntactic theory of local names. Technical Report YALEU/DC/RR-965, Yale University, May 1993.

[11] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.

[12] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[13] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *1992 ACM Conference on LISP and Functional Programming*, pages 288–298, San Francisco, CA, June 1992. ACM.

[14] Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

[15] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: A syntactic proof technique. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, Paris, France, September 1999.