A Confinement Framework for OO Programs

Shu Qin and Qiu Zongyan*
LMAM and Department of Informatics
School of Math., Peking University
{shuqin,zyqiu}@pku.edu.cn

Wang Shuling†
Institute of Software
Chinese Academy of Sciences
wangsl@ios.ac.cn

ABSTRACT

We present a framework for specifying and verifying about object confinement in OO programs. Instead of expressing the expected confinement requirements within a class, the requirements are specified by the clients of the class. We add an optional conf clause in class declarations for specifying the confined attribute-paths. A "same type and confinement" notation is introduced for expressing type dependencies among variables, parameters and return values of methods in classes. Based on the extension to a Java-like language and existing techniques of alias analysis, we define a sound type system to check well-confinedness of objects in programs.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming Languages, Language Constructs

General Terms

Language, Theory, Verification

Keywords

Object-Orientation, Confinement, Type System

1. INTRODUCTION

The sharing of object references (aliasing) is essential in object-oriented (OO) programming. Aliasing brings much benefit to OO practice and is utilized widely, however, it is also the source of program vulnerabilities [1, 2]. Current OO languages do not provide linguistic support for confinement to exclude undesired aliasing. For achieving confinement, early attempts include Hogg's Islands [3] and Almeida's Balloon types [4] to enforce *full encapsulation*. J. Vitek *et al.* propose *confined classes* [5] that confine object references at a package level. D. Clarke *et al.* propose *ownership types* [6, 7] to confine objects inside some object instances in more fine-grained schemes. *Universes* [8] and *Ownership Domains* [9] support more precise and flexible aliasing protection as variants to ownership types.

Ownership types clarify many issues related to confinement, but its specification form might not be quite satisfactory. The essential idea of ownership types is that owner parameters for objects are introduced to the syntax of an OO language, and the owner parameters are used to declare types of attributes, local variables, parameters and return values of methods which will be instantiated by their owners later. This *preparation for future confinement* is good and benefits a clear and straightforward type system [6], but sometimes may not be easy to apply properly for all possible clients.

```
class Node {                 class Node<own> {
  Node next;                   Node<own> next;
  T data;                      T data;
  ...                          ...
}                            }
```

Figure 1: Class Node without and with Ownership Types

Fig. 1 shows a class $Node$ in common syntax (left) and its extension with ownership types (right), where T is type of attribute $data$. In ownership typed code, $Node$ is defined with an owner parameter for the later possibility to confine $Node$ objects and their attribute $next$ in some higher-level classes. Given a simple class as $Node$, we may introduce owner parameters to support all the possible ownership relations for later use of the class. But for a little more complex class with a dozen of attributes and methods, introducing owner parameters for all possibilities will disturb the class declaration and make the class unacceptable, because in each use of the type we will have to instantiate all the ownership parameters correctly. A specification for an entity in a program such as a class should let us concentrate on the facilities of the entity itself instead of some unclear future requirements. Thus a class such as $Node$ has no duty to offer facilities for plausible confinement in its future use. If another class $List$ needs to confine a sequence of $Node$ objects and to use them as its representation, it should have the whole duty to specify this requirement. Of course, the language should provide mechanisms for programmers to express the intention.

We introduce a clause conf in class declarations to express that objects referred by the paths are confined inside this object. To avoid tedious writing, we propose a "same type and confinement" notation by using a type form ctype$[p]$. Objects of type ctype$[p]$ have the same type and confinement requirement as the object referred by p. We define a type system to check programs in a core language CμJava statically. The type system is sound according to our proof. Due to the page limitation, we leave many technical details in our report [10].

The main contributions of this paper are:

- A novel idea for specifying confinement policies in class declarations. It is a more flexible and up-down way to express confinement specification. Classes can be used with different forms with respect to confinement.

* Supported by NNSFC Grand No.90718002.
† Supported by the projects NSFC-60970031, NSFC-60736017, and NSFC-91018012.

IWACO'2011 Lancaster, UK

- A type system for checking programs with confinement specification. Soundness of the system is proved based on our previously published work [11].

Outline. Section 2 illustrates our idea intuitively; Section 3 extends a Java-like language with confinement specification; Section 4 defines typing environments and related notations, based on which a type system is defined in Section 5; Section 6 concludes.

2. CONFINEMENT SPECIFICATION

This section illustrates our approach by some examples, with some comparison to ownership types. We allow also programmers to write confinement requirements in programs, and define also a type system to check whether the requirements are satisfied during execution. The essential difference between our approach and ownership types is that, instead of assigning objects ownership contexts with ownerships parameters in lower-level classes, we specify these requirements in higher-level client classes, and meanwhile, we do not needs to say anything in the classes of these internal objects.

2.1 Internal Representation

The first example is taken from [6], where a class $Pair$ has two attributes fst of type X and snd of type Y. Class $Intermediate$ has two attributes p_1 and p_2 of type $Pair$, where p_1, $p_1.fst$ and $p_2.fst$ are its internal representation, but the rest are public. The $Main$ class has a public attribute $safe$ of type $Intermediate$, and method a cannot be called by $safe$ in $Main$ class, so $a := safe.a()$ is wrong. There are some other wrong statements in $Main$.

The example can be realized by our approach as shown in Fig. 2. First of all, we do no need to annotate any confinement requirement in class $Pair$, because it does not want to confine anything itself. While for class $Intermediate$, we need to confine the paths rooting from the attributes declared with rep in the ownership example. In consequence, we define $p_1, p_1.fst$ and $p_2.fst$ to be confined paths in the conf clause, and moreover, in order to preserve the prefix closure of conf set, we need to define p_2 to be confined. Notice that in the original example p_2 is public, but this does not mean that we have more confinement restrictions for class $Intermediate$. In fact, although p_2 is public, its attribute fst is declared with rep, so according to ownership types, p_2 can only be accessed by the current class. In later work of ownership types, Boyapati *et al* [12] proposed the *constraints on owners* principle, which corresponds to the "prefix closure" requirement of confined paths here.

Thanks to the type notation ctype$[p]$, we can specify type dependencies between variables within a class. For all types with rep context in the ownership example, we use ctype$[p]$ instead, where p is some confined path, to denote that the declared variable (or parameter, return value) has the same type and confinement requirement as p. For example, method a of class $Intermediate$ in Fig. 2 returns a value of the same type as p_1, therefore we define ctype$[p_1]$ as its return type. Other methods can be defined similarly.

Finally, the class $Main$ in the ownership example declares two local variables a and b with rep context. It seems to be problematic in our approach to declare local variables with rep context when there is no conf set in the class. But in fact, local variables with rep context will only be managed in current class without affecting the objects of the class at all. Therefore, we declare these variables as public instead, as seen in $Main$ class of Fig. 2.

Our type system produce the same result as ownership types while checking this example. Compared to ownership types, our type system allows more valid programs. For instance, if we assume that in class $Pair\ Y$ is a subtype of X, we can add one more method declared as: **void** $swap()\{\ fst\ :=\ snd;\}$. According to ownership types, the assignment inside the body is invalid due to the different ownership contexts of fst and snd. However, in our opinion, it is too early to exclude such a behavior, since m and n may be instantiated with the same context in the future. Our approach leaves the type checking to the later position(s) when method $swap$ of class $Pair$ is used.

2.2 Recursive Internal Representation

Fig. 3 implements a typical linked list class with a head node. Class $Node$ declares a recursive attribute $next$, and also an attribute $data$ of type T. Method $setNext$ sets $next$ of this object to be parameter n, and $getNext$ returns the $next$ attribute. Class $List$ declares a $head$ attribute of type $Node$, and confines paths $head$, $head.data$, and $head.\{next\}^+$. Here $head.\{next\}^+$ represents all the nodes of the linked list except for the first one denoted directly by $head$. Notice that only the $data$ of head node is confined, while the data of all other nodes are public. This is meaningful in practice because the first node is often used to store some important information of the list such as its length. The confinement structure of such a linked list is illustrated in Fig. 4.

Although $head$ and $head.\{next\}^+$ are of the same type $Node$ and both confined in $List$, they have different confinement schemes: $head$ confines one more attribute ($data$) than $head.\{next\}^+$. In order to distinguish them, we attach to the confinement scheme of each object the confinement information of its components, which will be discussed in detail in Section 4.

Method $addNode$ in $List$ first creates a new node, denoted by local variable $temp$, and then adds the node after the head node via calling $getNext$ and $setNext$ of $Node$. The newly added node has the same confinement type with other nodes referred by $head.next$. When a method is declared without confinement types, it is allowed to be called by either public or confined objects. But for the later case, we need to check whether the actual invocation will break confinement, e.g. $head.getNext(temp)$ in the method body. Our framework can conclude that $addNode$ is valid.

Similar to ownership types, if a method is declared with confinement requirement in types, it can only be called by objects with the same type and confinement status as this such as $getHead$ here. Method $violate$ in $List$ attempts to assign the head of this with the head of another node from the parameter ls. The assignment is not valid because the nodes are confined inside two different lists. In addition, $ls.head$ is even not allowed to occur here because it is invisible for current object. However, if we change the type of ls to be ctype[this] instead, then the assignment would be valid. The notation ctype[this] provides a way to express that multiple objects share the same confinement domain.

Going back to ownership types, we find that the list class in Fig. 3 can not be implemented by a single $Node$ class with ownership types as defined in Fig. 1. Another class needs to be introduced specially for implementing the $head$ node:

```
class HNode<Nowner, Towner>{
    Node<Nowner> next;
    T<Towner> data;
    ...
}
```

and $List$ needs to be modified accordingly.

2.3 Subtyping

We use the example in Fig. 5 to illustrate how confinement is specified in the presence of subtyping. Here class A has two attributes a_1 and a_2. a_1 is confined. Class B extends A with two new attributes b_1 and b_2. b_1 is confined. Class $Main$ defines two confined attributes, a of type A and b of type B respectively, and

```
class Pair {
  X fst, Y snd;
}

class Intermediate {
  Pair p1, p2;
  conf {p1, p1.fst, p2, p2.fst};
  ctype[p1] a() { return p1; }
  ctype[p2] b() { return p2; }
  ctype[p1.fst] x() { return p1.fst; }
  Y y() { return p1.snd; }
  void update() { p1.fst := new X(); }
}

class Main {
  Intermediate safe;
  void main(){
    Pair a, b;
    X x, Y y;
    safe := new Intermediate();
    a := safe.a();    // wrong
    b := safe.b();    // wrong
    x := safe.x();    // wrong
    y := safe.y();    // valid
    safe.update();    // valid
  }
}
```

Figure 2: Example from [6]

```
class Node {
  Node next; T data;
  void setNext(Node n){
    this.next := n; }
  Node getNext(){
    return this.next; }
}

class List {
  Node head;
  conf {head, head.data, head.{next}+};
  void addNode(T val){
    ctype[head.next] temp;
      temp := new Node();
      temp.data := val;
      temp.next := head.getNext();
      head.setNext(temp);
      head.data := ...;
  }
  ctype[head] getHead(){
    return head; //valid iff invoked from this
  }
  void violate(List ls){
    head := ls.head;      // wrong
  }
}
```

Figure 3: Linked List

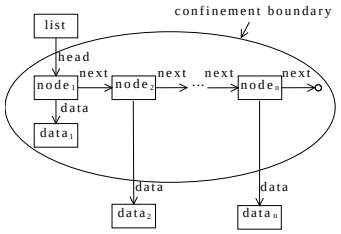

Figure 4: Confinement Structure of Linked List

```
class A : Object {
   T1 a1; T2 a2; conf {a1}; ...
}
class B : A {
   S1 b1; S2 b2; conf {b1}; ...
}
class Main {
   A a; B b; conf {a, a.a2, b, b.b2};
   void main (){
      ctype[a] x; ctype[b] y;
      y := new B();
      x := y; // wrong
      ...
   }
}
```

Figure 5: Example With Subtyping

furthermore, confines attributes a_2 of a and b_2 of b. In method $main$, local variables x and y are declared to have the same type and confinement requirement as a and b respectively. A new object of class B is created and assigned to y.

Our key idea for confinement in relation to subtyping is that what is confined in a superclass should also be confined in its subclasses. By our typing rule, the confinement scheme of y is not a subtype of the one of x, since attribute a_2 of x is confined while a_2 of y is not. Therefore, assignment $x := y$ is invalid.

The examples in this section give some intuitive ideas of our approach for specifying and checking confinement. In the following sections, we turn to our formal framework.

3. CμJava

To investigate the confinement problem, we define a simple OO language CμJava for the work that is a Java-like language extended with confinement specifications. Here is its syntax:

$$
\begin{array}{lll}
e & ::= & \mathsf{null} \mid \mathsf{this} \mid x \\
c & ::= & \mathsf{skip} \mid x := e \mid e.a := x \mid x := e.a \mid x := (T)e \\
 & \mid & x := e.m(\overline{e}) \mid x := \mathsf{new}\, T() \mid c; c \\
T & ::= & \mathsf{Object} \mid C \\
p & ::= & a \mid p.a \mid p.s^+ \\
TC & ::= & T \mid \mathsf{ctype}[\mathsf{this}] \mid \mathsf{ctype}[p] \\
md & ::= & TC\ m(\overline{TC\ x})\{\overline{T\ a};\ c; \mathsf{return}\, e\} \\
cd & ::= & \mathsf{class}\, C : C\{\ \overline{T\ a};\ [\mathsf{conf}\ \{\overline{p}\};]\ \overline{md}\ \} \\
prog & ::= & \overline{cd}
\end{array}
$$

To keep the language simple for a formal investigation to confinement problems, we restrict the common aspects of CμJava to some extent which are not essential.

We use x to denote a variable name, C a class name, a an attribute name, and m a method name. We use $\overline{e}$ to denote a sequence of expressions and correspondingly e_i the i-th element. We assume that there is always a statement return e as the last statement in method declarations. If there is no return statement in a method, a statement as "return null" is added by default. The return keyword **void** in previous examples is only a shorthand for brevity.

We introduce the *path expressions* p for specifying the representations of objects, where s is a finite set of attribute names, and form s^+ is used to define paths in recursive data structures. All paths start from the current object this by default. For example, if $head$ refers to the head node of a linked list, then $head.\{next\}^+$ denotes all nodes of the list except the head node. We say p is a *simple path* (*expression*) when it does not contain the form s^+. A valid simple path p denotes an attribute of some class.

In a class declaration, a new conf clause is introduced for specifying the confinement requirements for attributes of the class with path expressions, $\{\overline{p}\}$, to mean that this.$\{\overline{p}\}$ forms the internal representation of this object and is confined in it. We introduce a type notation ctype$[p]$ to stand for the same type and confinement requirements as the object denoted by path p, here p must be a confined simple path of the class. In particular, ctype[this] represents the same type and confinement requirement as the variable this.

We call the syntactic category TC *confinement type*. In CμJava, all parameters, local variables and return values are typed with TC. We will use cpath(C) to denote the set of confined paths of class C (without the ones declared in its superclasses).

For making confinement specification meaningful, it needs to conform to the following restrictions:

DEFINITION 1 (VALID CONFINEMENT SPECIFICATION). *A confinement specification of a class C is* valid*, iff*

- *All the confinement types used in C are well-formed: either TC is an ordinary type T, or of the form* ctype[this] *or* ctype$[p]$ *for $p \in$ cpath(C);*
- cpath(C) *satisfies the property of* prefix closure*: if a path is in* cpath(C)*, then all its non-empty prefixes are also in* cpath(C).

- $\mathsf{cpath}(C)$ *satisfies the property of* confinement inheritance*: a subclass inherits all confined paths of its superclasses (similar to attribute inheritance), but cannot override them. That is, only paths starting from newly declared attributes can appear in the* conf *clause of the subclass.*
- $\mathsf{cpath}(C)$ *satisfies the property of* path visibility*: only non-confined attributes of other classes can appear in* $\mathsf{cpath}(C)$. □

It is easy to formalize and check these restrictions statically. In the following, we only consider valid confinement specifications.

4. TYPING: ENVIRONMENTS AND NOTATIONS

Given a program P, its typing environment consists of two components, where Γ_P records basic type and inheritance information of classes in P and Θ_P records information relevant to confinement. We will define confinement schemes and the subtyping relation between the schemes according to the typing environment.

4.1 Standard Typing Environment Γ_P

The standard typing environment of a program P is a tuple:

$$\Gamma_P \;\widehat{=}\; \langle \mathsf{cname}_P, \mathsf{super}_P, \mathsf{attr}_P, \mathsf{method}_P, \mathsf{locvar}_P \rangle$$

where cname_P is the set of class names declared in P plus predefined Object and Null; super_P is a map associating each class to its direct superclass; attr_P maps each class name to its set of attribute-type pairs including the inherited attributes; method_P maps each pair of a class name and a method name to the signature of the corresponding method; and locvar_P maps a tuple of a class name, a method name and a variable name to the type of the variable. Types recorded in method_P and locvar_P are confinement types TC. The construction of Γ_P is routine [13] and omitted here.

4.2 Confinement Typing Environment Θ_P

Our type system use information stored in environment Θ_P to check whether an access to a variable or a method in a given context is permitted. Θ_P maps each class to its confinement tree, which will be defined below.

4.2.1 Confinement scheme

A *confinement scheme* CS records the type of an object plus the confinement information, defined as:

$$\begin{array}{rcl} \omega & ::= & T\langle C, \overline{a \mapsto \omega}, \overline{b \mapsto +}\rangle \\ CS & ::= & \omega \mid T\langle \mathsf{this}\rangle \mid T \end{array}$$

The confinement scheme of general form ω defines a type with a confinement context. An object of scheme $T\langle C, \overline{a \mapsto \omega}, \overline{b \mapsto +}\rangle$ means that the object is of type T and is confined in class C and the rest following C are the attribute-paths which are also confined inside class C, plus their confinement schemes recursively. The notation $b \mapsto +$ means that b has the same confinement scheme as the current object. It is used in defining schemes of objects with confined recursive attributes. The special form $T\langle \mathsf{this}\rangle$ represents that current object has the same type and confinement context as the variable this of class T. The final form T without confinement context means that current object is of type T and not confined.

We use $\mathsf{dtype}(CS)$ to extract the pure type from a scheme CS, i.e., $\mathsf{dtype}(T\langle C, \ldots\rangle) = T$.

4.2.2 Confinement tree

Each confinement type in a class corresponds to a confinement scheme. To calculate confinement schemes for confinement types in programs, we construct a confinement tree for each class.

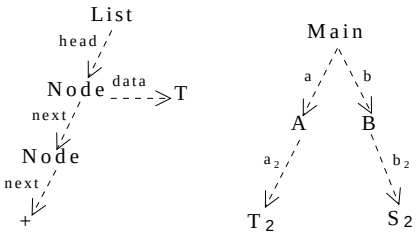

Figure 6: Confinement Trees for Examples in Section 2

DEFINITION 2 (CONFINEMENT TREE). *A* confinement tree *is a rooted, labeled and directed tree, defined as a quadruple:*

$$\mathcal{T} = (r, \mathcal{N}, \mathcal{A}, \mathcal{E})$$

where r is the root representing a class type, $\mathcal{N}$ is the node set representing class types or $+$, $\mathcal{A}$ is the label set representing attributes, and $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{A} \times \mathcal{N}$ is the edge set. □

Suppose we are considering the confinement tree of class G. An edge $(C, a, D) \in \mathcal{E}$ (resp. $(C, a, +) \in \mathcal{E}$) means that class C has an attribute a of type D (resp. C) and a is confined in G. The confinement tree of a class C, denoted by $\mathcal{T}_C$, can be constructed based on its confined set $\mathsf{cpath}(C)$. It roots at class node C, from which all the confined paths are drawn, with intermediate nodes representing the types of corresponding prefixes of these paths. As an example, the confinement trees for classes $List$ in Fig. 3 and $Main$ in Fig. 2 are present in Fig. 6.

We then define the confinement typing environment Θ_P for program P as follows:

$$\Theta_P \;\widehat{=}\; \{C \mapsto \mathcal{T}_C\}_{C \in \Gamma_P.\mathsf{cname}}$$

Given a confinement tree $\mathcal{T}$ of class C and a path p, we use $\mathsf{subtree}(\mathcal{T}, p)$ to denote the subtree that roots from the target node referred by p in $\mathcal{T}$. We call such subtree a confinement tree in the context of C. Given a confinement tree $\mathcal{T}$ in the context of class C, we get the corresponding confinement scheme by traversing the tree recursively, denoted by $\mathsf{trans}(\mathcal{T}, C)$, as follows:

- First, starting from the root r, for all edges $\overline{(r, a, D) \in \mathcal{T}}$, we construct $\overline{a \mapsto \mathsf{trans}(\mathcal{T}', C)}$, where $\overline{\mathcal{T}'}$ are the subtrees originating from the nodes $\overline{D}$ respectively;
- Second, for all edges $\overline{(R, b, +) \in \mathcal{T}}$, we construct $\overline{b \mapsto +}$;
- Finally, we define the confinement scheme $\mathsf{trans}(\mathcal{T}, C)$ as

$$r\langle C, \overline{a \mapsto \mathsf{trans}(\mathcal{T}', C)}, \overline{b \mapsto +}\rangle$$

Based on the confinement trees in Fig. 6, we can calculate confinement schemes for all confined paths. For example in class $List$, the confinement scheme for $\mathsf{ctype}[head]$ is $Node\langle List,\ next \mapsto Node\langle List, next \mapsto +\rangle, data \mapsto T\langle List\rangle\rangle$, while the one for $\mathsf{ctype}[head.next]$ is $Node\langle List, next \mapsto +\rangle$. For class $Main$, the confinement scheme for $\mathsf{ctype}[a]$ is $A\langle Main,\ a_2 \mapsto T_2\langle Main\rangle\rangle$.

4.3 Confinement scheme for confinement type

Based on Γ and Θ, we introduce a function $\sigma(TC, C)$ to return the confinement scheme for TC declared in class C, defined as follows:

$$\sigma(TC, C) \;\widehat{=}\; \begin{cases} T & \text{if } TC = T \in \Gamma.\mathsf{cname}, \\ C\langle \mathsf{this}\rangle, & \text{if } TC = \mathsf{ctype}[\mathsf{this}] \\ \mathsf{trans}(\mathsf{subtree}(\Theta(C), p), C), & \text{if } TC = \mathsf{ctype}[p] \wedge \\ & \quad p \in \mathsf{cpath}(C) \end{cases}$$

The confinement scheme of T is itself. The confinement scheme of $\mathsf{ctype}[\mathsf{this}]$ is $C\langle\mathsf{this}\rangle$. When TC is of form $\mathsf{ctype}[p]$ in class C, we first look up the confinement tree $\mathcal{T}_C$, then get the subtree corresponding to p by using $\mathsf{subtree}$, and finally transform the subtree into corresponding confinement scheme in the context of C.

4.4 Confinement schemes for attributes

As we defined above, the confinement scheme for an object not only records the type and confinement information of itself, but also those of its confined attribute-paths. Given a confinement scheme CS and an attribute a, we can calculate the confinement scheme of a under Γ and Θ as follows:

$$\mathsf{t}_{\Gamma,\Theta}(CS, a) \;\widehat{=}\; \begin{cases} CS & \text{if } CS = T\langle C, \overline{x \mapsto CS_x}\rangle \wedge \{\overline{x \mapsto CS_x}\}(a) = + \\ CS' & \text{if } CS = T\langle C, \overline{x \mapsto CS_x}\rangle \wedge \{\overline{x \mapsto CS_x}\}(a) = CS' \\ \sigma(\mathsf{ctype}[a], C) & \text{if } CS = C\langle\mathsf{this}\rangle \wedge a \in \mathsf{cpath}(C) \\ \Gamma.\mathsf{attr}(T)(a) & \text{if } \mathsf{dtype}(CS) = T \wedge a \in \mathsf{va}(T) \end{cases}$$

where $\mathsf{va}(T)$ in the last line of the definition is a set including all the non-confined attributes of class T.

The first two cases are for the situation when a is confined in the confinement context of CS. For this situation, when there is $a \mapsto +$ in CS, it means that a is a recursive attribute and its scheme is CS itself, otherwise, it is directly the projection of a in CS. The third case is when a is a confined attribute of the declaring class, so it can only be accessed by objects declared with $\mathsf{ctype}[\mathsf{this}]$. The last case is when a is not confined, then the scheme of a is its declared type.

$\mathsf{t}_{\Gamma,\Theta}(CS, a)$ only calculates the scheme of attribute a that is visible in the confinement context of CS. Therefore, the visibility of attributes can be enforced by the function implicitly.

4.5 Subtyping

Based on the super relation in Γ, we define an extended subtyping relation $\preceq_e$ between confinement schemes as follows:

$$\frac{\Gamma.\mathsf{super}(T_2) = T_1}{\Gamma \vdash T_2 \preceq_e T_1} \quad \Gamma \vdash \mathsf{Null} \preceq_e CS \quad \Gamma \vdash T\langle\mathsf{this}\rangle \preceq_e T$$

$$\frac{\Gamma \vdash T_2 \preceq_e T_1 \quad \{\overline{a \mapsto \omega_a}\} \subseteq \{\overline{b \mapsto \omega_b}\} \quad \{\overline{b}\} \setminus \{\overline{a}\} \nsubseteq \Gamma.\mathsf{attr}(T_1)}{\Gamma \vdash T_2\langle C, \overline{b \mapsto \omega_b}\rangle \preceq_e T_1\langle C, \overline{a \mapsto \omega_a}\rangle}$$

Null is subtype to all confinement schemes. $T\langle\mathsf{this}\rangle$ is always subtype to T. Given two confinement schemes ω_1 and ω_2 with non-empty confinement contexts, $\omega_2 \preceq_e \omega_1$ iff the declared type of ω_2 is subtype to that of ω_1; they have the same confinement context; ω_2 inherits all the confined attributes of ω_1 and can confine more new declared attributes which do not belong to ω_1. Finally, the subtyping relation is reflexive and transitive.

4.6 Static Visibility

Variables or methods declared with confinement types are invisible from outside. Especially such methods can only be invoked by objects of type $\mathsf{ctype}[\mathsf{this}]$ (including this) in current class. We use static visibility to represent this access constraint.

DEFINITION 3 (STATIC VISIBILITY). *For expression e and confinement type TC, we say TC is visible to e if $\mathsf{sv}(e, TC)$ holds:*

$$\mathsf{sv}(e, TC) \;\widehat{=}\; \neg(e : \mathsf{ctype}[\mathsf{this}] \vee e = \mathsf{this}) \Rightarrow TC \neq \mathsf{ctype}[p] \quad \textit{for any confined path } p$$

where $e : \mathsf{ctype}[\mathsf{this}]$ means that the declared type of e is $\mathsf{ctype}[\mathsf{this}]$.

5. A TYPE SYSTEM FOR CONFINEMENT

Under typing environments, we define a type system for checking CμJava programs with confinement specifications.

First of all, the typing judgments for expressions take the form $\Gamma, \Theta, C, m \vdash e : CS$, stating that expression e in the method m of class C has confinement scheme CS under the typing environment Γ and Θ. The typing rules for variables x and this are given as follows:

$$\frac{\Gamma.\mathsf{locvar}(C, m, x) = TC}{\Gamma, \Theta, C, m \vdash x : \sigma(TC, C)} \qquad \Gamma, \Theta, C, m \vdash \mathsf{this} : C\langle\mathsf{this}\rangle$$

The confinement scheme of variable x can be calculated from its confinement type TC by using $\sigma(TC, C)$. For the special variable this in C, its confinement scheme is $C\langle\mathsf{this}\rangle$.

The typing judgement for statements c takes the form "$\Gamma, \Theta, C, m \vdash c : \mathsf{com}$", stating that c is well-confined in the context of m in C under Γ and Θ. Except for method invocation, the typing rules for statements can be defined in the traditional way. For example, an assignment is well-confined iff the confinement scheme of the assignee is subtype to the one of the assigner. The typing rules for methods, classes and programs can also be defined as traditional way. In the following we will mainly build the typing rule for method invocation, while the others can be found in detail in our report [10].

Ownership types specify object dependencies when a class is declared but our approach delays the process to the using stage of the class. As a consequence, new dependencies may be introduced by clients of classes, and for this case, we have to retreat to check confinement of methods when they are called.

For checking method invocation $x := e_1.m_1(\overline{e})$, we have two typing rules. The first one is for the cases when method m is declared with confinement types, or both caller e_1 and actual arguments $\overline{e}$ are not confined:

[tp-methinv]

$$\frac{\begin{array}{c}\Gamma, \Theta, C, m \vdash x : CS_x \quad \Gamma, \Theta, C, m \vdash \overline{e : CS_e} \\ \Gamma, \Theta, C, m \vdash e_1 : CS_{e_1} \quad CS_{e_1} \preceq_e \mathsf{dtype}(CS_{e_1}) \\ (\mathsf{dtype}(CS_{e_1}), m_1(\overline{TC_y : y}) : TC) \in \Gamma.\mathsf{method} \\ \mathsf{sv}(e_1, TC) \quad \mathsf{sv}(e_1, \overline{TC_y}) \quad \Gamma, \Theta, C, m_1 \vdash \overline{y : CS_y} \\ \overline{CS_e \preceq_e CS_y} \quad \sigma(TC, C) \preceq_e CS_x\end{array}}{\Gamma, \Theta, C, m \vdash x := e_1.m_1(\overline{e}) : \mathsf{com}}$$

where CS_{e_1} is $C\langle\mathsf{this}\rangle$ or some ordinary type T if CS_{e_1} is subtype of $\mathsf{dtype}(CS_{e_1})$. We first require that method m_1 is visible to the caller e_1, represented by $\mathsf{sv}(e_1, TC)$ and $\mathsf{sv}(e_1, \overline{TC_y})$. Then as usual, the confinement schemes of actual arguments $\overline{e}$ are subtype to the ones of formal parameters $\overline{y}$ and the confinement scheme for return type TC is subtype to the one of assigner x.

From this rule, when m_1 is declared with confinement types, it can only be called by objects of type $C\langle\mathsf{this}\rangle$. For this case, the caller e_1 and method m_1 are in the same context corresponding to this, therefore, the method invocation is well-confined. On the other hand, if m_1 is declared without confinement types, this rule restricts that the caller e_1 and actual arguments $\overline{e}$ be not confined in current class C, i.e., they are of ordinary types like T. Under this condition, no more object dependence of the method body being called will be introduced during the actual invocation, and therefore, the method invocation is well-confined.

By applying this rule, the statement $a := safe.a()$ in method $main$ is not well-confined, because method a of class $Intermediate$ is not visible to $safe$, i.e., $\mathsf{sv}(safe, \mathsf{ctype}[p_1])$ is not satisfied; but the statement $y := safe.y()$ is well-confined, since method y of class $Intermediate$ is declared without confinement type, and the caller $safe$ is not confined.

Another rule [tp-methinv'] is defined for the case when method m_1 is declared without confinement types, and the caller e_1 or actual arguments $\overline{e}$ are confined in current class C. For this case, besides the usual type checking, we need to further check whether the new object dependencies induced during actual invocation will break confinement or not. This is achieved by checking the alias set of the method body being called after its instantiation over e_1 and $\overline{e}$.

Given a program P, we introduce its alias summary AS, which maps a pair of class and method to a set of aliasing sets:

$$AS(C, m) \widehat{=} \{A_1, A_2, \cdots, A_n\}$$
$$\text{for all } C \in \Gamma_P.\mathsf{cname}, m \in \Gamma_P.\mathsf{method}$$

Each set A_i is composed of attribute-paths that are aliasing to each other in m. For any $i \neq j$, we have $A_i \cap A_j = \emptyset$. The aliasing summary AS records the aliasing information of all methods of the program. It can be built by statical analysis of body commands of methods. There have been a range of work on this, one among which is [14]. We assume the existence of AS here and will not detail the construction process, because it is not the main concern of the paper. The rule is then defined as follows:

[tp-methinv']

$$\frac{\begin{array}{c}\Gamma, \Theta, C, m \vdash e_1 : CS_{e_1} \quad \Gamma, \Theta, C, m \vdash x : CS_x \\ \Gamma, \Theta, C, m \vdash \overline{e : CS_e} \\ (\mathsf{dtype}(CS_{e_1}), m_1(\overline{T_y : y}) : T) \in \Gamma.\mathsf{method} \\ \overline{\mathsf{dtype}(CS_e) \preceq_e T_y} \quad T \preceq_e \mathsf{dtype}(CS_x) \\ \forall D \preceq_e \mathsf{dtype}(CS_{e_1}), A_k \in AS(D, m_1), p_i, p_j \in A_k. \\ (CS_i \preceq_e CS_j \vee CS_j \preceq_e CS_i)\end{array}}{\Gamma, \Theta, C, m \vdash x := e_1.m_1(\overline{e}) : \mathsf{com}}$$

where CS_i denotes the confinement scheme for the path p_i after instantiation, i.e., $\sigma(\mathsf{ctype}[p_i[e_1/\mathsf{this}, \overline{e}/\overline{y}, x/\mathsf{res}]], C)$, and CS_j the same.

Except for the usual type checking in the first two lines, we need to further check: first, the declared types of actual arguments are subtype to the ones of formal parameters, and the return type is subtype to the declared type of x being assigned. In this step, we don't check confinement; second, the method body of m_1 being executed actually should not break confinement of related objects. Because of dynamic binding, we consider all the definitions of m_1 in the subclasses of declared type of e_1, i.e., D in the rule. For method m_1 of class D, we require that any two aliasing paths should have compatible confinement schemes during the method invocation, defined by the last line of the hypothesis. For CS_i and CS_j, which one is subtype to the other is actually determined by the subtyping relation between their corresponding declared types, which is guaranteed by the type checking of the method body.

By applying this rule, the statement $temp.next := head.getNext()$ in Fig. 3 is well-confined. Formally, the alias set for method $getNext$ is $\{\{\mathsf{this}.next, res\}\}$, and furthermore, the confinement schemes of $\mathsf{this}.next[\mathsf{this}.head/\mathsf{this}]$ and $\mathsf{res}[temp.next/\mathsf{res}]$ are the same.

6. CONCLUSION

In this short paper, we present a new framework for specifying and reasoning about confinement of OO programs. Our basic idea is inspired by ownership types. In our approach, a class is not responsible for specifying future confinement requirements of other client classes. If a class wants to confine its representation, it has to only express this requirement entirely in its declaration by itself.

The confinement specification becomes simpler and more direct, by delaying confinement requirements of classes to the later employing phase. But on the other hand, this to some extent burdens the design of the type system. As shown above, the typing rule for method invocation needs to check whether confinement of the method body is broken because of the new object dependence introduced during the invocation. We use alias summary to solve this problem, though a little complicated, our approach aims at facilitating programmers to express their confinement requirements.

Currently, we restrict that an object cannot be the representations of more than one object at the same time, thus suffers the limitation to implement common programming idioms such as external iterators. One considerable solution to relax this restriction in our approach is to define confinement types visible for multiple classes by connecting internal domain of these classes.

7. REFERENCES

[1] Hogg, J., Lea, D., Wills, A., de Champeaus, D., Holt, R.: The geneva convention on the treatment of object aliasing. ACM SIGPLAN OOPS Messenger **3**(2) (1992) 11–16

[2] Clarke, D., Drossopoulou, S., Müller, P., Noble, J., Wrigstad, T.: Aliasing, confinement, and ownership in object-oriented programming (IWACO). In: Proc.of ECOOP'08, Springer (2008)

[3] Hogg, J.: Islands: Aliasing protection in object-oriented languages. In: Proc. of OOPSLA'91, ACM Press (1991)

[4] Almeida, P.S.: Balloon types: Controlling sharing of state in data types. In: Proc. of ECOOP'97, Springer (1997)

[5] Bokowski, B., Vitek, J.: Confined types. In: Proc.of OOPSLA'99, ACM Press (1999)

[6] Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: Proc.of OOPSLA'98, ACM Press (1998)

[7] Clarke, D.: Ownership types and containment. PhD thesis, University of New South Wales, Australia (2001)

[8] Müller, P.: Modular specification and verification of object-oriented programs. PhD thesis, FernUniversit at Hagen, *LNCS 2262*, Springer (2002)

[9] Aldrich, J., Chambers, C.: Ownership domains: Separating alias policy from mechanism. In: Proc.of ECOOP'04, Springer (2004)

[10] Shu, Q., Qiu, Z., Wang, S.: A confinement framework for OO programs. Technical Report 2011-011, School of Math., Peking University (2011) Avaliable at: http://www.mathinst.pku.edu.cn/download.php.

[11] Wang, S., Shu, Q., Liu, Y., Qiu, Z.: A semantic model of confinement and locality theorem. Frontiers of Computer Science **4**(1) (2010) 28–46

[12] Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: Proc.of POPL'03, ACM Press (2003)

[13] Qiu, Z., Wang, S., Quan, L.: Sequential μJava: Formal foundations. In *AWSF'07*. Technical Report 2007-035, School of Math., Peking University (2007) Avaliable at: http://www.mathinst.pku.edu.cn/download.php.

[14] Meyer, B.: The theory and calculus of aliasing. CoRR **abs/1001.1610** (2010)