A Confinement Framework for OO Programs

(Revised Version)

Shu Qin[1], Qiu Zongyan[1], and Wang Shuling[2]

[1] LMAM and Department of Informatics, School of Mathematical Sciences, Peking University
[2] Institute of Software, Chinese Academy of Sciences
Email: `shuqin@pku.edu.cn, qzy@math.pku.edu.cn, wangsl@ios.ac.cn`

Abstract. We present a framework for specifying and reasoning about object confinement in object-oriented programs. It supports the static checking of confinement of dynamic object references as required. Different from existing work, the confinement requirement for an object is specified from outside to its constituents inside. Formally, on the language level, we add an optional `conf` clause in class declaration for specifying the confinement requirement of internal attribute-paths, and meanwhile introduce a "same type and confinement" notation for expressing type dependencies among variables, parameters and returns of methods in the class. Based on these linguistic extensions to a Java-like modeling language and the existing techniques on alias analysis, we define a sound type system to statically check the confinement of objects in programs.

1 Introduction

The sharing of object references (aliasing) is essential in object-oriented (OO) programming. It brings much benefits to OO practice and has been utilized widely, however, it is also the source of many program defects and vulnerabilities. The aliasing out of control makes behaviors of OO programs hard to understand or modify, since changes to an object might affect all objects relying on it without awareness. It becomes an even more serious problem when some objects inside critical regions for security get leaked by unintended aliasing. Obviously, techniques for aliasing control, especially object *confinement*, are crucial for safe OO programming.

Unfortunately, current OO languages do not provide linguistic support for confinement. The *access control* in OO languages is just for protecting names, but not for confining objects. For example, a **private** attribute of an object can be easily exposed as a return value of a public method, or by an assignment to variables accessible outside. People have paid much attention to this problem, and proposed interesting ideas and technologies. Some schemes are proposed to confine object references inside a module by using *confined class* [1, 2], or inside some object instances in more fine-grained schemes by using *ownership types* [3–5]. Some others insist to define some objects as *unique* or *linear* [6–8], etc.

However, after about two decades of study, there is still no finally widely accepted technique for specifying and reasoning about object confinement, especially in the practical OO development. Therefore, novel ideas and techniques should continually be explored, tested, and compared, and finally, to find more acceptable solution for software development.

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

Fig. 1. Class Node without and with Ownership Types

For an OO program, objects created during its execution spread over the global heap without any restriction, and potentially they are accessible from everywhere. For achieving necessary confinement, one possibility is to find a way to classify the objects into different domains which are not overlapped but may be nested, and then define access restrictions among them. Inside each domain, objects are accessible between each other. However, the only allowed access from objects in a domain to another different one is via some special interface objects of the accessed domain [5].

As one of the well-known achievements in this field, *ownership types* [3] provide a flexible way to specify object confinement and enable the static check, which was fully developed in [9]. The idea is that, a kind of owner parameters for objects is introduced to the syntax of an OO language, and then used to declare attributes, as well as local variables, parameters and returns of methods for specifying run-time ownership relations between objects. Fig. 1 shows a class $Node$ in common syntax (left) and its extension with ownership types (right), where T is the type of attribute $data$. In the ownership typed code, we intent to give owner parameter to $Node$ for the later possibility to confine objects of $Node$ and its attribute $next$ in some up-level classes, which can instantiate the parameter as the representation so it owns the $Node$ object and the $next$ field, recursively. This *preparation for future confinement* is good and benefits a clear and straightforward type system [3] but may sometimes not be complete enough to apply properly such as for some possible client classes of $Node$.

Of course, given a class as simple as $Node$, we may introduce as many owner parameters as possible, to support all possible ownership relations of the attributes and methods of the class. However, given a little more complex class with a dozen of attributes and methods, introducing owner parameters for all possibilities will disturb the class declaration seriously and make the class unacceptable, because in each use of the type, we may have to instantiate correctly a dozen of ownership parameters.

Ownership types clarify many issues related to confinement, but its specification form might not be quite satisfactory. For a specification form, in our opinion, when used in specifying an entity in a program, e.g., a class, it is desired that we can concentrate our attention on the facilities of the entity itself, but not some unclear future requirements. Thus, a class like $Node$ has no duty to offer facilities for plausible confinement in its future use, because this is not the requirement of itself. If a using class, e.g., $List$, wants to confine a sequence of $Node$ objects to make them as the internal representation, then it should have the whole duty to specify this requirement. Of course, the language should provide mechanisms for programmers to express these intentions.

In this paper, we develop a confinement framework based on the above ideas, which can offer similar expressive power as ownership types. We introduce an optional conf clause in class declaration for specifying a set of paths to represent that objects referred by the paths are confined inside this object, i.e., as the *internal representation*. Except for attributes, we also need to consider confinement requirement of local variables, parameters and returns of methods. To avoid tediously writing these requirements, we propose a "same type and confinement" notation, with which a variable, a parameter, or return of a method can be declared with type $\mathsf{ctype}[p]$ to express that it has the same type and confinement requirement as the object referred by p. Examples in following section will show that this notation is very beneficial.

These two linguistic extensions support to classify objects in the heap into object-based domains with different access restrictions, and reach a language with confinement specification. We will define a type system to check programs with confinement specification in CμJava statically, which is proved to be sound and guarantees that a well-typed program is always well-confined with respect to the confinement specification in the program. Our framework provides a way to manage the classification of objects, and different from ownership types, specifies object confinement from outside to its constituents inside.

The paper is organized as follows. Section 2 illustrates our idea intuitively and compares it with ownership types. Section 3 extends a Java-like language with confinement specification. Section 4 defines the typing environments and some related notations as basis for defining type system, which is presented in Section 5. Section 6 shows how to apply the type system to check confinement of programs via several examples. Section 7 discusses related work and Section 8 concludes. Finally, the proof for the soundness of the type system is put in Appendix A for reference.

2 Confinement Specification

To avoid the unintended modification to objects, we provide an approach to protect sensitive objects inside a specific domain, to support object confinement. Inspired by ownership types, we allow programmers to express confinement requirements for objects in programs and define a type system for checking statically whether the requirements are satisfied during the execution. The essential difference between ownership types and our approach is that, instead of assigning objects *ownership contexts* [3] using ownership parameters in lower-level classes, we specify directly in each class declaration which internal objects it intends to confine from the higher containment level, while at the same time, these confined objects do not need to say anything in their class declarations. Now we illustrate our idea via some examples and give the formal definition in the next section.

2.1 Internal Representation

To compare our approach with ownership types, we take an example from [3], as shown in Fig. 2. Here class $Pair$ has two context parameters m and n, which represent ownership contexts for its attributes fst of type X and snd of type Y, respectively. Class $Intermediate$ has two attributes of type $Pair$, where p_1, $p_1.fst$ and $p_2.fst$ are defined as its internal representation, but $p_1.snd$, p_2 and $p_2.snd$ public. A $Main$ class declares

```
class Pair<m, n>{
  m X fst, n Y snd;
}

class Intermediate {
  rep Pair<rep, norep> p_1;
  norep Pair<rep, norep> p_2;
  rep Pair<rep, norep> a() { return p_1; }
  norep Pair<rep, norep> b() { return p_2; }
  rep X x() { return p_1.fst; }
  norep Y y() { return p_1.snd; }
  void update() { p_1.fst := new rep X(); }
}

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

Fig. 2. Example in Ownership Types

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

class Intermediate {
  Pair p_1, p_2;
  conf {p_1, p_1.fst, p_2, p_2.fst};
  ctype[p_1] a() { return p_1; }
  ctype[p_2] b() { return p_2; }
  ctype[p_1.fst] x() { return p_1.fst; }
  Y y() { return p_1.snd; }
  void update() { p_1.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
  }
}
```

Fig. 3. Example in Our Approach

a public attribute *safe* of type *Intermediate*. According to the type system of ownership types, methods with rep context are only visible in current class, and assignments are valid only when expressions on both sides have the same ownership context. As a result, method a cannot be called by *safe* in *Main* class, and similarly, other statements in *Main* can be checked statically whether they are valid or not.

The example written using our notation is given in Fig. 3 which can achieve the same confinement requirements. First of all, we assign no confinement requirement for objects of *Pair*. We then confine paths originating from attributes (including themselves) of class *Intermediate* with rep context, by using a conf clause, which means that all objects referred by these pathes are confined as the representation of current *Intermediate* object. Here we define $p_1, p_1.fst$ and $p_2.fst$ to be confined paths, and besides, in order to preserve the prefix closure of conf set, we also define p_2 confined. Notice that with ownership types p_2 (in Fig. 2) is public, but it does not imply that we have more confinement restrictions for class *Intermediate*. In fact, according to the type system of ownership types, the public attribute p_2 with rep constituent can only be operated in current class. In later work of ownership types [10], they proposed the *constraints on owners* principle, which actually corresponds to the "prefix closure" requirement of confined paths here.

Thanks to the type notation $\mathsf{ctype}[p]$, we can easily specify type dependencies between variables within a class. For all types with rep context in Fig. 2, we use $\mathsf{ctype}[p]$ instead, where p is a confined path, to denote that the decorated variable (or parameter, return) has the same type and confinement requirement with p. For example, in Fig. 2 method a returns a value of exactly the same type with p_1, therefore we define $\mathsf{ctype}[p_1]$ as its return type. Other methods are similar.

Finally, for class $Main$ in Fig. 2, it has no internal representation but declares two local variables a and b with rep context. It seems to be problematic at first sight that in our approach how to declare local variables with rep context when there is no conf set in a class, like $Main$ here. In fact, such local variables with rep context will only be managed in current class without affecting any representation of the class at all. Therefore, we only need to declare these variables with public types, as shown in $Main$ class in Fig. 3.

Our type system to be defined will gain the same results with the type system of ownership types when checking the example. But compared to ownership types, our type system allows more valid programs. For instance, we assume that in class $Pair$, Y is a subtype of X, and add one more method shown below:

$$\textbf{void}\ \ swap()\{\, fst\ :=\ snd; \}$$

According to the type system of ownership types, the assignment inside the body is invalid because of the context parameters of *fst* and *snd* are different. But in our opinion, it is too early to exclude such behavior, since in the later invocation of this method, m and n may be instantiated with a same context. Our approach leaves the type checking to the later stage when the method $swap$ of class $Pair$ is called.

2.2 Recursive Internal Representation

Fig. 4 presents a linked list example with a head node. Class $Node$ declares a recursive attribute $next$, and another $data$ of type T respectively. Method $setNext$ sets $next$ of this object to be of parameter n, and $getNext$ returns $next$ attribute. Class $List$ declares a $head$ attribute of type $Node$, and confines $head, head.data$, and $head.\{next\}^+$ of current object. Here $head.\{next\}^+$ represents all the nodes of the linked list except for the first one referred by $head$. Notice here we confine the data of the head, while leave all other data publicly accessible. It is meaningful in practice because the first node is used to store some important information of the list, e.g. the length of the list. The confinement structure of such a linked list is illustrated in Fig. 5.

Although $head$ and $head.\{next\}^+$ have the same declared type $Node$, and are both confined in $List$, they have different confinement schemes. This is because $head$ confines one more attribute $data$ than other nodes referred by $head.\{next\}^+$. In order to tackle this problem in static checking, we attach to the confinement scheme of each object the confinement information of its related constituents, which will be discussed in Section 4.

Method $addNode$ in $List$ adds a node after the head node, in which local variable $temp$ records the new created node with the same confinement requirement as other

```
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; }

  void violate(List ls){
    head := ls.head; }        // wrong
}
```

Fig. 4. A Linked List with Recursive Structure

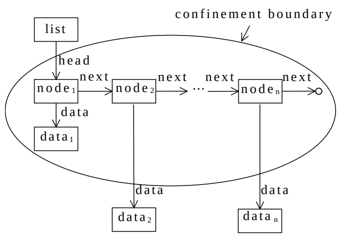

Fig. 5. Confinement Structure of a Linked List

non-head nodes, so it is declared with type ctype$[head.next]$. The node is then set as the next node of $head$ via invocations to $getNext$ and $setNext$ of $Node$. When a method is declared without confinement types in the signature, such method is allowed to be called by either public or confined objects. For the later case, we need to check whether the invocation breaks the confinement or not, by analyzing the aliasing set of the method body being called. Here $addNode$ is valid. Similar to ownership types, if a method is declared with confinement types, it can only be called by objects with the same type and confinement as this, like $getHead$ here. Our typing rules can deal with all the cases.

Method $violate$ in $List$ attempts to assign the head of this by the head of another node from the parameter ls. The assignment is not valid in our typing system because they are confined inside two different lists. $ls.head$ is not allowed to occur here because it is invisible in current object. However, if we change the type of ls to be ctype[this] instead, then the assignment will be valid. The type notation ctype[this] provides a way for expressing that multiple objects share the same confinement domain.

Go back to ownership types, we find that the list with such confinement requirements as in Fig. 4 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 $head$ node :

```
class $A$ : Object {
    $T_1$ $a_1$;
    $T_2$ $a_2$;
    conf $\{a_1\}$;
    ...
}
class $B$ : $A$ {
    $S_1$ $b_1$; $S_2$ $b_2$;
    conf $\{b_1\}$;
    ...
}
```

```
class $Main$ {
    $A$ $a$; $B$ $b$;
    conf $\{a, a.a_2, b, b.b_2\}$;
    void $main$ (){
        ctype$[a]$ $x$;
        ctype$[b]$ $y$;
        $y$ := new $B()$;
        $x$ := $y$; // wrong
        ...
    }
}
```

Fig. 6. Example With Subtyping

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

2.3 Subtyping

We use an example in Fig. 6 to illustrate how confinement is specified with the presence of subtyping. Here class A has two attributes a_1 and a_2, in which a_1 is confined. Class B extends A by two new attributes b_1 and b_2, in which b_1 is confined. Class $Main$ defines two confined attributes, a of type A and b of type B respectively, and furthermore, confines attributes a_2 of a and b_2 of b. In method $main$, local variable 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 related to subtyping is, what confined should not be exposed to the outside. Therefore, what confined in a superclass should also be confined in its subclasses. By our typing rule, the confinement scheme of y is not subtyping to the one of x, since attribute a_2 of $x.a$ is confined while a_2 of $y.a$ is not. Therefore, assignment $x := y$ is invalid.

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

Now we define a small OO language CμJava used in this work. It is a Java-like language extended with confinement specifications.

3.1 Syntax

The syntax of CμJava is as follows:

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

Here we use x to denote a variable name, C a class name, a an attribute name, and m a method name, respectively. We use $\overline{e}$ to denote a sequence of expressions and correspondingly e_i the i-th element. The same overline convention will be used throughout the paper. As in Java, this denotes current object, and null is special to mean that a variable or attribute does not refer to anything.

To keep the language simple for the formal investigation on the confinement problem, we restrict CμJava in some aspects. There are only some special forms of assignments, including plain assignment, attribute lookup $x := e.a$, and update $e.a := x$. Besides, we take object creation $x := \mathsf{new}\, C()$ also as a special form of assignments, that creates a new object of class C, initializes all its attributes with null, and lets variable x refer to it finally. The restrictions on creation will leave all confinement related problems to method call. The sequential composition is included, but other structural statements are omitted here. All of these restrictions are not essential, and all the discussion below can be extended easily to the full language.

As usual, we take Object as the super type of all classes. We assume an internal type Null as the type of null, which is the subtype of all class types, and is used only for defining type system. In method declaration, we assume that there is always a statement return e as the last statement. If there is not any return statement in a method, a statement as "return null" is added by default. The return symbol **void** in previous examples is only a shorthand for brevity.

We introduce a category of *path expressions* p for specifying the representation of objects of the class, where a is an attribute name, and s is a finite set of attribute names. All paths in class declarations start implicitly from current object. Form s^{+} is used to define paths in recursive data structures. For example, if l refers to the head node of a linked list, then $l.\{next\}^{+}$ represents all nodes of the list except for the head. If r refers to the root of a binary tree, we can use $r.\{left, right\}^{+}$ to refer to all tree nodes except the root node. We say p is a *simple path* (*expression*) when it does not contain the form s^{+}. A valid simple path p usually denotes an attribute of some class.

Now we give some explanations to the confinement specifications. In a class declaration, the clause leaded by keyword conf specifies the confinement requirement for attributes of the class. Here we can write a set of path expressions, $\{\overline{p}\}$, to mean that

$\mathsf{this}.\{\overline{p}\}$ forms the internal representation of this object and is confined in it. We will use $\mathsf{cpath}(C)$ to denote the set of confined paths defined by conf clause of class C (without the ones declared in its superclasses). Moreover, we introduce a type form $\mathsf{ctype}[p]$, where ctype is a reserved word, p is a simple path, to stand for the same type and confinement requirement as the object denoted by p. We require that p must be some confined path of the class where the method is declared. In particular, the type form $\mathsf{ctype}[\mathsf{this}]$ represents the same type and confinement requirement as this.

We call the syntactic category TC *confinement type*. In CμJava, all parameters, local variables and returns are typed with TC.

3.2 Valid Confinement Specification

Now we give some definitions and properties, which aims to define the well-formedness and validation of confinement specifications in classes.

We use $\mathsf{cattr}(C)$ to denote the set of attributes directly declared in class C, and $\mathsf{attr}(C)$ to denote all attributes of C, including the ones inherited from its superclasses.

The property *prefix closure* for conf clauses is presented as follows:

Definition 1 (Prefix Closure). *Given a* conf *clause of class C, its confined set* $\mathsf{cpath}(C)$ *satisfies the* prefix closure *property, iff* $\forall p \in \mathsf{cpath}(C) \Rightarrow \mathit{prefix}(p) \subseteq \mathsf{cpath}(C)$. □

For attribute a, we define $\mathit{prefix}(a) = \emptyset$; and for $x \in \{a, \{s\}^+\}$, $\mathit{prefix}(p.x) = \{p\} \cup \mathit{prefix}(p)$. This property implies that if a path p is in $\mathsf{cpath}(C)$, then all its non-empty prefixes must also be in $\mathsf{cpath}(C)$. It is necessary, otherwise the confinement of a path can be easily broken by invoking the methods of its non-confined prefix.

We introduce the concept *confinement behaviorial subtyping* as follows:

Definition 2 (Confinement Behaviorial Subtyping). *Given a* conf *clause of class C, its confined set* $\mathsf{cpath}(C)$ *satisfies the* confinement behaviorial subtyping *property, iff* $\forall p \in \mathsf{cpath}(C) \wedge \mathit{prefix}(p) = \emptyset \Rightarrow p \in \mathsf{cattr}(C)$. □

The confinement behaviorial subtyping property states that any subclass obeys the same confinement requirement as its superclass for the paths starting from the inherited attributes. In fact, trying to confine, in a subclass, a path starting from an inherited non-confined attributes of its superclasses would not keep them confined. Such requirement may be broken by upcasting objects to its superclass.

According to above two properties, only paths starting from new declared attributes can appear in conf clause of a class. A subclass inherits all confined paths of its superclasses (similar to the attribute inheritance), but cannot override them. We will use $\mathsf{cf}(C)$ to denote all the confined paths of class C including inherited ones. Obviously, if B is the direct superclass of C, then $\mathsf{cf}(C) = \mathsf{cf}(B) \cup \mathsf{cpath}(C)$.

Now we define the visibility of paths in conf clauses where only non-confined attributes of other classes can appear here.

Definition 3 (Path Visibility). *Given a* conf *clause of class C, its confined set* $\mathsf{cpath}(C)$ *satisfies the* path visibility *property, iff* $\forall p \in \mathsf{cpath}(C)$*, predicate* $\mathsf{visPath}(p, C)$ *is* true*, where:*

$$\begin{array}{lcl}
\mathsf{visPath}(a, C) & \Leftrightarrow & a \in \mathsf{cattr}(C) \\
\mathsf{visPath}(p.a, C) & \Leftrightarrow & \mathsf{visPath}(p, C) \ \wedge a \in \mathsf{va}(\mathsf{dtype}(p)) \\
\mathsf{visPath}(p.s^+, C) & \Leftrightarrow & \mathsf{visPath}(p, C) \ \wedge s \subseteq \mathsf{va}(\mathsf{dtype}(p))
\end{array}$$

Here $\mathsf{dtype}(p)$ *is the static type of the attribute denoted by (simple path) p, which may be a primitive type or a class type in program, and the confinement augmentation is not taken into account;* $\mathsf{va}(C)$ *denotes non-confined attributes of C defined as:*

$$\mathsf{va}(C) \mathrel{\widehat{=}} \mathsf{attr}(C) \setminus \mathsf{cf}(C) \qquad \square$$

Definition 4 (Confinement Type Well-formedness). *A confinement type TC in a class C is well-formed, iff either TC is an ordinary type T, or of the form* $\mathsf{ctype}[\mathsf{this}]$ *or* $\mathsf{ctype}[p]$ *for* $p \in \mathsf{cpath}(C)$. $\square$

Finally, we define the valid confinement specification as follows:

Definition 5 (Valid Confinement Specification). *A confinement specification of a class C is* valid*, iff*

- *All the confinement types used in C are well-formed, and*
- $\mathsf{cpath}(C)$ *satisfies the properties of prefix closure, confinement behaviorial subtyping, and path visibility.* $\square$

These properties are easy to be checked statically. In the rest of this paper, we will consider only valid confinement specifications.

4 Typing: Environments and Notations

Before presenting the type system, we need to build the typing environment first. Given a program P, our typing environment consists of two components, where Γ_P records type and inheritance information of classes in P; and Θ_P records confinement information represented by confinement schemes of objects, each of which is composed of the confinement context, plus inductively the confinement schemes for the attribute-paths of current object confined in the same context. We also consider the subtyping relation between confinement schemes, used for checking whether each assignment is valid.

4.1 Standard Typing Environment Γ_P

Given a program P, the standard typing environment Γ_P records all the typing information for classes, methods, and variables in P, as well as the inheritance information. It is defined as a tuple:

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

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 in P to its set of attribute-type pairs including the inherited attributes; method_P maps each pair of class name and method name in P 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. Notice that the types occurring in method_P and locvar_P are confinement types TC as defined in the syntax.

The typing environment Γ_P can be extracted directly from program P, and we omit it due to page limit. Readers can refer to our report [11] for details. In the following, we will omit the subscript P when there is no confusion. We will use $\Gamma.\mathsf{cname}$, $\Gamma.\mathsf{super}$, *etc* to denote the cname, super, *etc* components of Γ, respectively.

4.2 Confinement Typing Environment Θ_P

For a program P, each class C has a confined path set $\mathsf{cpath}(C)$, which may be empty. We record these information in environment Θ_P using confinement scheme introduced below. Our type system will use the information to check if an accessing to a variable or a method in a given context is valid or not.

Confinement scheme We use *confinement scheme* to represent the type of objects, denoted by CS, which has the form defined as:

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

Confinement scheme ω defines a type with confinement context. If the scheme of an object is $T\langle C, \overline{a \mapsto \omega}, \overline{b \mapsto +}\rangle$, this means that the object is of type T, and moreover, C here denotes the domain in which the object is confined, and the rest followed by C are the attribute-paths which are confined inside class C, accompanied with their confinement schemes. Then, only the attributes confined in C are recorded in the list. The ω for attribute a must also have the form $T'\langle C, \cdots\rangle$ assuming a is of type T'. The notation $b \mapsto +$ means that b has the same confinement scheme as current object. It is used mainly in schemes of objects with confined recursive attributes.

Scheme $T\langle \mathsf{this}\rangle$ represents that the object under consideration has the same type and confinement context as this object of class T. Finally, a type T without confinement context means that the object is of type T and not confined.

For future use, we overload function $\mathsf{dtype}(CS)$ to extract the pure type from scheme CS, thus we have $\mathsf{dtype}(T\langle C, \ldots\rangle) = T$, $\mathsf{dtype}(T\langle \mathsf{this}\rangle) = T$, and $\mathsf{dtype}(T) = T$.

Confinement tree

Definition 6 (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. □

An edge $(C, a, D) \in \mathcal{E}$ means that class C has an attribute a of type D and a is confined in some class; and similarly, $(C, a, +) \in \mathcal{E}$ means that class C has a recursive attribute a of type C and a is confined in some class. In the following, we will use Tr, Tr' to stand for confinement trees, C, D for nodes, a, b for labels, and p, q for sequences of labels (attribute-paths).

Given a confinement tree Tr and a path p, we use $\mathsf{subtree}(\mathsf{Tr}, p)$ to denote the subtree that roots at the target node referred by p in Tr. From our assumption that attribute names in a class are distinct, the target node is unique when existing, so the subtree is also unique. In the context of class C, given a confinement tree T with root type R, we can construct a confinement scheme by traversing the tree recursively, denoted by $\mathsf{trans}(\mathsf{Tr}, C)$, as follows:

- Starting from the root, first, for all the edges $\overline{(R, a, D) \in \mathsf{Tr}}$, we construct maps $\overline{a \mapsto \mathsf{trans}(\mathsf{Tr}', C)}$, where $\overline{\mathsf{Tr}'}$ are the subtrees originating from the node $\overline{R}$ respectively;
- Second, for all edges $\overline{(R, b, +) \in \mathsf{Tr}}$, we construct $\overline{b \mapsto +}$ in correspondence;
- At last, the confinement scheme $\mathsf{trans}(\mathsf{Tr}, C)$ is defined as

$$R\langle C, \overline{a \mapsto \mathsf{trans}(\mathsf{Tr}', C)}, \overline{b \mapsto +}\rangle$$

We use confinement tree to model the confinement structure of a class. Given a class C, we build its confinement tree Tr_C based on its confined set $\mathsf{cpath}(C)$, as follows:

- If $\mathsf{cpath}(C)$ is empty, then Tr_C is an empty tree.
- Otherwise, choose a path $p \in \mathsf{cpath}(C)$ randomly, all the prefixes of p will be also in $\mathsf{cpath}(C)$ by the prefix closure assumption of $\mathsf{cpath}(C)$. Denote the first element of p by $\mathit{fr}(p)$, and the rest by $rs(p)$. Clearly $\mathit{fr}(p)$ is an attribute of C, take its type as D. We then add the edge $(C, \mathit{fr}(p), D)$ to the tree Tr_C if it does not exist, otherwise do nothing. Now we consider the rest path $rs(p)$ recursively. When the construction process reaches some node N, and the corresponding element in the path is of form s^+, where s is a set of attributes, then we add for each attribute $e \in s$, the edge $(N, e, +)$ to Tr_C. If there is a further element c followed by s^+ in the path, we add the edge (N, c, M) to the tree, where M is the type of c. Continue the process till the rest path is empty, we have the sub-tree corresponding to the path p.
- Repeat the second process for each path in $\mathsf{cpath}(C)$, we finally build the confinement tree Tr_C for class C.

Finally, we define the confinement typing environment Θ_P for program P as follows:

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

Similar to Γ, we often omit the subscript and write directly Θ when there is no confusion. As shown above, given a CμJava program, both typing environments for it can be built statically. In the following, we simply assume their existence, and concentrate on how to define the type system based on them.

4.3 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 7 (Static Visibility). *For expression e and confinement type TC, we say TC is visible to e if* $\mathsf{sv}(e, TC)$ *holds:*

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

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

Having this definition, we can check whether a method is visible or confined in a given context.

4.4 Confinement scheme for confinement type

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

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

For pure type T, the confinement scheme is itself. When TC is of form $\mathsf{ctype}[\mathsf{this}]$, the confinement scheme will be $C\langle\mathsf{this}\rangle$ that is different form the pure C. When it is of form $\mathsf{ctype}[p]$, where p is a confined path, we first look up the confinement tree Tr_C from the $\varTheta$ by using $\varTheta(C)$, then get the subtree that p corresponds to in the tree by using subtree, and finally transform the subtree into corresponding confinement scheme by using the function trans, which is exactly the confinement scheme of $\mathsf{ctype}[p]$.

4.5 Examples

Now we take an example from Section 2 to illustrate our approach to get the confinement scheme for the confinement type. First of all, the confinement trees for them are built via their conf clause and shown in Fig. 7.

Then based on the confinement tree for class $Intermediate$, the confinement scheme of $\mathsf{ctype}[p_1]$ is built: Firstly, we obtain the subtree rooted with the node that p_1 points to; and then apply trans to the subtree to obtain corresponding scheme of $\mathsf{ctype}[p_1]$, i.e., $Pair\langle Intermediate,\ fst \mapsto X\langle Intermediate\rangle\rangle$. Similarly, we get the confinement scheme of $\mathsf{ctype}[p_2]$, which is the same as the one of $\mathsf{ctype}[p_1]$. Moreover, we get the confinement schemes for $p_1.fst$ and $p_2.fst$, both of which are $X\langle Intermediate\rangle$.

We do similar work for class $List$ and $Main$. $List$ is recursive, following the rules, the confinement scheme for $\mathsf{ctype}[head]$ is $Node\langle List,\ next \mapsto Node\langle List, next \mapsto +\rangle, data \mapsto T\langle List\rangle\rangle$, the scheme for $\mathsf{ctype}[head.data]$ is $T\langle List\rangle$, and the scheme for $\mathsf{ctype}[head.next]$ is $Node\langle List, next \mapsto +\rangle$. For class $Main$ (ref. Fig 6 and Fig. 7) with subtyping, the confinement scheme for $\mathsf{ctype}[a]$ is $A\langle Main,\ a_2 \mapsto T_2\langle Main\rangle\rangle$, the one for $\mathsf{ctype}[b]$ is $B\langle Main,\ b_2 \mapsto S_2\langle Main\rangle\rangle$, the one for $\mathsf{ctype}[a.a_2]$ is $T_2\langle Main\rangle$, and the one for $\mathsf{ctype}[b.b_2]$ is $S_2\langle Main\rangle$.

4.6 Calculating types of attributes from confinement scheme

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 attribute-paths which are confined in the same domain. Given a confinement scheme CS and an attribute a of an object, we can calculate the confinement scheme of a under typing environments $\varGamma$ and $\varTheta$ as follows:

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

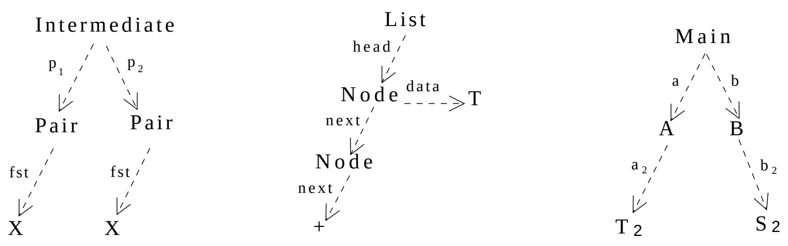

Fig. 7. Confinement Trees for Examples in Section 2

There are three main possibilities: 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, then the scheme of a will be CS itself, otherwise, it is directly the projection of a in CS. The third case is when a is a confined attribute, and it can only be accessed by objects declared with ctype[this]. The last case is when a is not confined, then the scheme of a is its declared type.

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

4.7 Subtyping

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

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

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

$$\Gamma \vdash CS \preceq_e CS \qquad \frac{\Gamma \vdash CS_2 \preceq_e CS_1 \quad \Gamma \vdash CS_3 \preceq_e CS_2}{\Gamma \vdash CS_3 \preceq_e CS_1}$$

The type Null is subtype to all confinement schemes. The confinement scheme $T\langle\mathsf{this}\rangle$ 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, and they have the same confinement context; and furthermore, ω_2 inherits all the confined attributes of ω_1, and in addition, confines more new declared attributes which do not belong to ω_1. The subtyping relation is reflexive and transitive.

5 A Type System for Confinement

Now we are ready to define a type system for CμJava programs with confinement specifications, for checking whether a program is well-confined or not. Given a program P, the type system is defined under the two typing environments Γ_P and Θ_P.

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

Fig. 8. Typing Expressions

5.1 Typing Expressions

Typing judgments for expressions take the form of $\Gamma,\Theta,C,m \vdash e : CS$, which states that expression e in the method m of class C has confinement scheme CS under typing environments Γ and Θ. The typing rules for expressions are present in Fig. 8.

A variable x in method m of class C has the confinement scheme corresponding to its declared type TC using $\sigma(TC,C)$. From the definition of $\sigma(TC,C)$, if TC is a pure type T, then the confinement scheme is still T and thus variable x is publicly accessible; otherwise, if TC is a confinement type $\mathsf{ctype}[p]$, the variable has the same confinement scheme as p, which can be obtained from the confinement typing environment Θ, and in this case, the variable can only be accessed in C. For the special variable this in C, we define its confinement scheme as $C\langle\mathsf{this}\rangle$. As a result, all the confined attributes of this will be visible and can be accessed. As usual, the constant null has confinement scheme Null.

5.2 Typing Statements and Programs

Now we are ready to present the typing rules for statements except for method calls, which is more complicated and will be considered in the following Subsection 5.3. The typing judgement for statements c takes the form "$\Gamma,\Theta,C,m \vdash c : \mathsf{com}$", which means that command c is well-confined in the context of m in C under typing environments Γ and Θ. Furthermore, we use the judgements "$\Gamma,\Theta,C \vdash m : \mathsf{ok}$", "$\Gamma,\Theta \vdash cd : \mathsf{ok}$" and "$\Gamma,\Theta \vdash P : \mathsf{ok}$" to state that method m, class declaration cd, and program P are well-confined under the environments respectively. The typing rules for statements and programs are listed in Fig. 9.

The statement skip is always well-confined, as shown in rule **[tp-skip]**. For assignment $x := e$ (rule **[tp-assign]**), it is required that the confinement scheme of e must be subtype to the one of x. For update $e.a := x$ (rule **[tp-update]**), the confinement scheme of $e.a$ is calculated from the one of e by using $\mathsf{t}_{\Gamma,\Theta}(CS_e, a)$, from whose definition we know that a must be a visible attribute of e in current class C. Besides it is still required that the confinement scheme of assignee x is subtype to the one of assigner $e.a$. Rule **[tp-lookup]** for lookup $x := e.a$ is similarly defined. The type cast $x := (T)e$ requires that the declared type of e is subtype to cast type T, which is subtype to the declared type of x, and moreover, the confinement scheme of e must be subtype to the one of x, as shown in rule **[tp-asncast]**. The object creation $x := \mathsf{new}\, T()$ creates an object of class T and then lets x to refer to it. The rule **[tp-objcreate]** requires that T is subtype to the declared type of x. The sequential composition $c_1; c_2$ is well-confined iff c_1 and c_2 are well-confined, as shown in rule **[tp-sequence]**.

As shown in rule **[tp-method]**, a method declaration in class C is well-confined, iff its method body is well-confined, and moreover, the confinement scheme of the return value is subtype to the one corresponding to return type. A class is well-confined iff

$$\text{[tp-skip]}\quad \frac{}{\Gamma, \Theta, C, m \vdash \mathsf{skip} : \mathsf{com}}$$

$$\text{[tp-assign]}\quad \frac{\Gamma, \Theta, C, m \vdash x : CS_x \quad \Gamma, \Theta, C, m \vdash e : CS_e \quad CS_e \preceq_e CS_x}{\Gamma, \Theta, C, m \vdash x := e : \mathsf{com}}$$

$$\text{[tp-update]}\quad \frac{\begin{array}{c}\Gamma, \Theta, C, m \vdash x : CS_x \quad \Gamma, \Theta, C, m \vdash e : CS_e \\ \mathsf{t}_{\Gamma,\Theta}(CS_e, a) = CS_a \quad CS_x \preceq_e CS_a\end{array}}{\Gamma, \Theta, C, m \vdash e.a := x : \mathsf{com}}$$

$$\text{[tp-lookup]}\quad \frac{\begin{array}{c}\Gamma, \Theta, C, m \vdash x : CS_x \quad \Gamma, \Theta, C, m \vdash e : CS_e \\ \mathsf{t}_{\Gamma,\Theta}(CS_e, a) = CS_a \quad CS_a \preceq_e CS_x\end{array}}{\Gamma, \Theta, C, m \vdash x := e.a : \mathsf{com}}$$

$$\text{[tp-asncast]}\quad \frac{\begin{array}{c}\Gamma, \Theta, C, m \vdash x : CS_x \quad \Gamma, \Theta, C, m \vdash e : CS_e \\ T \preceq_e \mathsf{dtype}(CS_x) \quad \mathsf{dtype}(CS_e) \preceq_e T \quad CS_e \preceq_e CS_x\end{array}}{\Gamma, \Theta, C, m \vdash x := (T)e : \mathsf{com}}$$

$$\text{[tp-objcreate]}\quad \frac{\Gamma, \Theta, C, m \vdash x : CS_x \quad T \preceq_e \mathsf{dtype}(CS_x)}{\Gamma, \Theta, C, m \vdash x := \mathsf{new}\, T() : \mathsf{com}}$$

$$\text{[tp-sequence]}\quad \frac{\Gamma, \Theta, C, m \vdash c_i : \mathsf{com},\ i = 1, 2}{\Gamma, \Theta, C, m \vdash c_1; c_2 : \mathsf{com}}$$

$$\text{[tp-method]}\quad \frac{\Gamma, \Theta, C, m \vdash c : \mathsf{com} \quad \Gamma, \Theta, C, m \vdash e : CS_e \quad CS_e \preceq_e \sigma(TC, C)}{\Gamma, \Theta, C \vdash TC\ m(\overline{TC_1\ x})\{\overline{TC_2\ x};\ c; \mathsf{return}\, e\} : \mathsf{ok}}$$

$$\text{[tp-class]}\quad \frac{\Gamma, \Theta, C \vdash \overline{md} : \mathsf{ok}}{\Gamma, \Theta \vdash \mathsf{class}\, C : D\{\overline{T\ a}; [\mathsf{conf}\ \{\overline{p}\};]\ \overline{md}\} : \mathsf{ok}}$$

$$\text{[tp-program]}\quad \frac{\Gamma, \Theta \vdash \overline{cd} : \mathsf{ok}}{\Gamma, \Theta \vdash P : \mathsf{ok}} \quad \text{where } P = \overline{cd}$$

Fig. 9. Typing Statements

all the methods declared in the class are well-confined, and furthermore, a program is well-confined iff all its classes in it are well-confined, as shown in rules [tp-class] and [tp-program] respectively.

5.3 Typing Rules for Method Invocation

Ownership types specify confinement dependency between the constituents of a class when the class is declared, but instead, our approach delays the process to the moment when the class is used. As a result, more confinement dependency may be introduced for a class in the future. This makes the typing for method invocation more complicated than what in ownership types. Under some circumstance, we have to go back to re-check confinement of the method body being called.

We have two typing rules for method invocation. The first one is for the cases when the method being called is declared with confinement types, or otherwise, both caller and actual arguments are public. Both cases can be checked by rule:

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

where CS_{e_1} is $C\langle\mathsf{this}\rangle$ or some ordinary type T.

As usual, we always need to check that the confinement schemes of actual arguments $\overline{e}$ are subtype to the ones of formal parameters $\overline{y}$, and furthermore, the confinement scheme corresponding to return type TC is subtype to the one for x being assigned. We also require that method m_1 is visible to the caller e_1, represented by $\mathsf{sv}(e_1, TC)$ and $\mathsf{sv}(e_1, \overline{TC_y})$ in the rule. As a result, when m_1 is declared with confinement types, it can only be called by objects of type $C\langle\mathsf{this}\rangle$. In such case, the caller e_1 and the callee m_1 are in the same confinement context corresponding to this, therefore, the method invocation is well-confined. On the other hand, if m_1 is declared without confinement types, it can be called freely by objects in any confinement context. The rule **[tp-methinv]** 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. With this condition, it can be guaranteed that no more confinement dependence for the method body being called will be introduced during the invocation, and therefore, the method invocation is well-confined.

When method m_1 being called is declared without confinement types, and the caller e_1 or actual arguments $\overline{e}$ for m_1 are confined in current class C, we define the typing rule **[tp-methinv']**. Under such circumstance, besides the usual type checking, we need to further check whether the new confinement dependence induced by e_1 and $\overline{e}$ during the invocation to method m_1 will break confinement or not.

In order to solve the problem, given a program P, we introduce its alias summary AS, which maps a pair of class name C and method m to a set of aliasing sets:

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

Each aliasing set A_i is composed of attribute-paths, starting from this, or formal parameters x of m, or the return variable res, followed by a sequence of attribute accesses. For any $i \neq j$, we have $A_i \cap A_j = \emptyset$. The aliasing summary AS records the aliasing relation produced by each method in the program. It can be built statically by structural induction on the body commands for methods. There have been a range of work on this, such as [12, 13]. We demand a preservative partition for the pathes, where any probably aliasing is respected. We assume the existence of AS here and will not detail the construction process, because it is not what the paper concentrates on mainly.

The aliasing summary serves to check confinement of invocation of methods declared without confinement types. We then actually only need to build the set of aliasing sets for methods declared without confinement types in AS. For example, the alias summary for the program in Fig. 4 is defined as follows:

$$\begin{aligned}
AS(Node, setNext) &= \{\{n, \mathsf{this}.next\}\}\\
AS(Node, getNext) &= \{\{\mathsf{res}, \mathsf{this}.next\}\}\\
AS(List, addNode) &= \{\{val, \mathsf{this}.head.next.data\}\}
\end{aligned}$$

Based on these definitions, the second typing rule for method invocation is defined as follows:

[tp-methinv']

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

Given a path p, we use $\delta(p)$ to denote $p[e_1/\mathsf{this}, \overline{e}/\overline{y}, x/\mathsf{res}]$, which substitutes first variable of p with the related instantiation for this, formal parameters $\overline{y}$, and return parameter res in the caller context. In the rule, CS_i denotes the confinement scheme for the substituted path p_i, i.e., $\sigma(\mathsf{ctype}[\delta(p_i)], C)$, and CS_j the same.

As defined in rule **[tp-methinv']**, to check $x := e_1.m_1(\overline{e})$ in method m of class C, except for the usual type checking (as shown 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 care about confinement; second, the method body of m_1 being executed during the invocation should not break confinement of related objects. Because of dynamic binding of method invocation, we consider all possible implementations of m_1 in the subclasses of declared type of e_1, i.e., D in the rule. For method m_1 in class D, any two aliasing paths in the aliasing summary should have compatible confinement schemes during the method invocation. That is, they satisfy subtyping relation, e.g., CS_i and CS_j in the rule, which one is subtype to the other is actually guaranteed by the subtyping relation between their corresponding declared types.

In next section, we will illustrate how to apply the type system for checking program confinement.

6 Case Study

We will use the type system defined above to check the confinement of programs in Section 2.

Example 1 (Pair). First of all, assume the typing environments for the program in Fig. 3 are Γ and Θ. The work to get Γ for programs is routine thus we do not list here. The establishing of Θ can be obtained via the conf clauses.

The method a of $\mathit{Intermediate}$ only consists of a return statement, then following the rule **[tp-method]**, we have

$$\frac{\begin{array}{c}\Gamma,\Theta,\mathit{Intermediate},a\vdash \mathsf{this}:\mathit{Intermediate}\langle\mathsf{this}\rangle\\ \Gamma,\Theta,\mathit{Intermediate},a\vdash \mathsf{this}.p_1:\mathsf{t}_{\Gamma,\Theta}(\mathit{Intermediate}\langle\mathsf{this}\rangle,p_1)\\ \mathsf{t}_{\Gamma,\Theta}(\mathit{Intermediate}\langle\mathsf{this}\rangle,p_1)\preceq_e\sigma(\mathsf{ctype}[p_1],\mathit{Intermediate})\end{array}}{\Gamma,\Theta,\mathit{Intermediate}\vdash\mathsf{ctype}[p_1]\ a()\{\mathsf{return}\,p_1\}:\mathsf{ok}}$$

where the subtyping relation in hypothesis is satisfied, based on the fact that both evaluation on left and right hand sides result in the same scheme, i.e., $\mathit{Pair}\langle\mathit{Intermediate},\mathit{fst}\mapsto X\langle\mathit{Intermediate}\rangle\rangle$. Similar to method a, we can then check methods b, x, y in the same class. We can also check the method update by applying rules **[tp-objcreate]** and **[tp-method]**. The method update does not have a return statement for brevity. By add a statement return null and following the rule **[tp-objcreate]** and **[tp-method]**, we have

$$\frac{\dfrac{\begin{array}{c}\Gamma,\Theta,\mathit{Intermediate},\mathit{update}\vdash p_1:\mathit{Pair}\langle\mathit{Intermediate},\mathit{fst}\mapsto X\langle\mathit{Intermediate}\rangle\rangle\\ \mathsf{t}_{\Gamma,\Theta}(\mathit{Pair}\langle\mathit{Intermediate},\mathit{fst}\mapsto X\langle\mathit{Intermediate}\rangle\rangle,\mathit{fst})=X\langle\mathit{Intermediate}\rangle\\ \mathsf{dtype}(X\langle\mathit{Intermediate}\rangle)\preceq_e X\end{array}}{\begin{array}{c}\Gamma,\Theta,\mathit{Intermediate},\mathit{update}\vdash p_1.\mathit{fst}:=\mathsf{new}\,X():\mathsf{com}\\ \Gamma,\Theta,\mathit{Intermediate},\mathit{update}\vdash\mathsf{Null}\preceq_e\sigma(\mathsf{Null},\mathit{Intermediate})\end{array}}}{\Gamma,\Theta,\mathit{Intermediate}\vdash\mathit{void\ update}()\{...\}:\mathsf{ok}}$$

For class $Main$, by applying rule **[tp-methinv]**, the statement $a:=\mathit{safe}.a()$ in method main is not well-confined. The reason is that method a declared in class $\mathit{Intermediate}$ has return type $\mathsf{ctype}[p_1]$, which is not visible to safe. As a result, $\mathsf{sv}(\mathit{safe},\mathsf{ctype}[p_1])$ is not satisfied in the rule. For the same reason, both statements $b:=\mathit{safe}.b()$ and $x:=\mathit{safe}.x()$ are not well-confined, however, $y:=\mathit{safe}.y()$ is well-confined. Following the rule **[tp-methinv]**, we have

$$\frac{\begin{array}{c}\Gamma,\Theta,\mathit{Main},\mathit{main}\vdash\mathit{safe}:\mathit{Intermediate}\quad\Gamma,\Theta,\mathit{Main},\mathit{main}\vdash y:Y\\ (\mathit{Intermediate},y():Y)\in\Gamma.\mathit{method}\quad\mathsf{sv}(y,Y)\quad Y\preceq_e Y\end{array}}{\Gamma,\Theta,\mathit{Main},\mathit{main}\vdash y:=\mathit{safe}.y():\mathsf{com}}$$

Finally, we proof the call to method update by safe. It is also the case for the rule **[tp-methinv]**. The derivation is as follows:

$$\frac{\begin{array}{c}\Gamma,\Theta,\mathit{Main},\mathit{main}\vdash\mathit{safe}:\mathit{Intermediate}\\ (\mathit{Intermediate},\mathit{update}():\mathsf{Null})\in\Gamma.\mathit{method}\quad\mathsf{Null}\preceq_e\mathsf{Null}\end{array}}{\Gamma,\Theta,\mathit{Main},\mathit{main}\vdash\mathit{safe}.\mathit{update}():\mathsf{com}}$$

□

Example 2 (List). We use the type system to illustrate a concrete typing of a single-linked list in Fig. 4 strictly.

First we inspect the method $\mathit{addNode}$ in class List. $\mathit{temp}:=\mathsf{new}\ \mathit{Node}()$ can be checked by rule **[tp-objcreate]**. Beforehand, the inference a typing expressions inference rule for the confinement scheme of the local variable temp is necessary. By the typing expressions inference rule, we get the confinement scheme of $temp$ is $Node\langle List, next \mapsto +\rangle$.

$$\frac{\begin{array}{c}\Gamma, \Theta, List, addNode \vdash temp : Node\langle List, next \mapsto +\rangle \\ \mathsf{dtype}(Node\langle List, next \mapsto +\rangle) = Node\end{array}}{\Gamma, \Theta, List, addNode \vdash temp \ := \ \mathsf{new}\ Node() : \mathsf{com}}$$

The command $temp.data := val$ can be checked by the rule **[tp-update]**.

$$\frac{\begin{array}{c}\Gamma, \Theta, List, addNode \vdash temp : Node\langle List, next \mapsto +\rangle \\ \mathsf{t}_{\Gamma,\Theta}(Node\langle List, next \mapsto +\rangle, data) = T \quad \Gamma, \Theta, List, addNode \vdash val : T\end{array}}{\Gamma, \Theta, List, addNode \vdash temp.data := val : \mathsf{com}}$$

Method $getNext$ of class $Node$ is declared without any confinement type, and then is called in class $List$ by a confined object $head$, i.e., $temp.next := head.getNext()$. We can apply rule **[tp-methinv']** to check confinement of the statement. First calculate the confinement scheme of $temp.next$, and then apply rule **[tp-methinv']** to perform the checking:

$$\frac{\begin{array}{c}\Gamma, \Theta, List, addNode \vdash temp.next : Node\langle List, next \mapsto +\rangle \\ \Gamma, \Theta, List, addNode \vdash \mathsf{this}.head : CS_{\mathsf{this}.head} \\ (Node, getNext() : Node) \in \Gamma.method \\ Node \preceq_e \mathsf{dtype}(Node\langle List, next \mapsto +\rangle) \\ CS_{\delta(\mathsf{this}.next)} \preceq_e CS_{\delta(\mathsf{res})} \vee CS_{\delta(\mathsf{res})} \preceq_e CS_{\delta(\mathsf{this}.next)}\end{array}}{\Gamma, \Theta, List, addNode \vdash temp.next := head.getNext() : \mathsf{com}}$$

where we use $CS_{\mathsf{this}.head}$ to represent the confinement scheme of $\mathsf{this}.head$, which is, $Node\langle List, next \ \mapsto \ Node\langle List, next \ \mapsto \ +\rangle, data \ \mapsto \ T\langle List\rangle\rangle$. The alias set for method $getNext$ is $\{\{\mathsf{this}.next, res\}\}$, defined by $AS(Node, getNext())$. We then find that the confinement schemes of $\mathsf{this}.next[\mathsf{this}.head/\mathsf{this}]$ and $\mathsf{res}[temp.next/\mathsf{res}]$ after instantiation are the same, so all subtyping conditions in hypothesis are satisfied. According to rule **[tp-methinv']**, the statement is well-confined.

It is the same case to use the rule **[tp-methinv']** for the invocation $setNext(temp)$ by $head$. The derivation is as follows:

$$\frac{\begin{array}{c}\Gamma, \Theta, List, addNode \vdash temp : Node\langle List, next \mapsto +\rangle \\ \Gamma, \Theta, List, addNode \vdash \mathsf{this}.head : CS_{\mathsf{this}.head} \\ (Node, setNext(n : Node) : \mathsf{Null}) \in \Gamma.method \\ \mathsf{dtype}(Node\langle List, next \mapsto +\rangle \preceq_e Node \\ CS_{\delta(\mathsf{this}.next)} \preceq_e CS_{\delta(n)} \vee CS_{\delta(n)} \preceq_e CS_{\delta(\mathsf{this}.next)}\end{array}}{\Gamma, \Theta, List, addNode \vdash head.setNext(temp) : \mathsf{com}}$$

The alias set for method $setNext$ is $\{\{\mathsf{this}.next, n\}\}$, defined by $AS(Node, setNext())$. The subtyping relation for the confinement scheme of $\mathsf{this}.next$ and n with substitutions is needed. Here $CS_{\delta(\mathsf{this}.next)}$ is calculated by $CS_{\mathsf{this}.head.next}$, and $CS_{\delta(n)}$ is calculated by CS_{temp}, both are $Node\langle List, next \ \mapsto \ +\rangle$. All subtyping conditions in hypothesis are satisfied. According to rule **[tp-methinv']**, the statement is well-confined.

The method $getHead$ is checked by the rule **[tp-method]** for both the confinement scheme of the return value $head$ and $\mathsf{ctype}[head]$ is the same.

The method $violate$ could not pass the rule **[tp-method]** for the command $head := ls.head$ is not well-confined. We can not get the confinement scheme of $ls.head$ because the confinement scheme calculation for attributes requires $head \in \mathsf{va}(List)$ when the confinement scheme of ls is $List$. But actually, $head \notin \mathsf{va}(List)$ is the truth. □

Example 3 (Subtype). Finally, we proof the example in Fig. 6 with subtyping by our typing rules. In the example, the attribute a_1 is confined in class A, so what visible for the other classes is only the attribute a_2, i.e. $\mathsf{va}(A) = \{a_2\}$. For class B, $\mathsf{va}(B) = \{a_2, b_2\}$. The conf clauses in the three classes are checked to be valid. The confinement tree for class $Main$ is of the third one in Fig. 7. Using the typing expressions rules, we can get the confinement schemes of variables in method $main$ of class $Main$.

$$\frac{\begin{array}{c}\Gamma.locvar(Main, main, x) = \mathsf{ctype}[a] \\ \sigma(\mathsf{ctype}[a], Main) = A\langle Main,\ a_2 \mapsto T_2\langle Main\rangle\rangle\end{array}}{\Gamma, \Theta, Main, main \vdash x : A\langle Main,\ a_2 \mapsto T_2\langle Main\rangle\rangle}$$

$$\frac{\begin{array}{c}\Gamma.locvar(Main, main, y) = \mathsf{ctype}[b] \\ \sigma(\mathsf{ctype}[b], Main) = B\langle Main,\ b_2 \mapsto S_2\langle Main\rangle\rangle\end{array}}{\Gamma, \Theta, Main, main \vdash y : B\langle Main,\ b_2 \mapsto S_2\langle Main\rangle\rangle}$$

The confinement scheme of x is $A\langle Main,\ a_2 \mapsto T_2\langle Main\rangle\rangle$, which is not subtype to $B\langle Main,\ b_2 \mapsto S_2\langle Main\rangle\rangle$, the confinement scheme of y , so the command $x := y$ is not well-confined. □

7 Related Work

Because of the importance in software security, the object aliasing problem [14] has attracted a lot of research and discussion for many years. Some early attempts include Hogg's Islands [15] and Almeida's Balloon types [16], which both enforce *full encapsulation*, preventing all objects encapsulated in an object from being exposed outside the object. However, this restriction is too strong such that excludes the possibility to separate the internal representation from shared objects in a collection. Later approaches, including ours, allow specifying what needs to be confined while leaving other objects to be accessible, thus provide more flexible object encapsulation.

The most notable work in this area is *Flexible Alias Protection* proposed by Noble, Vitek, and Potter *et al.* [17], known as *ownership types*, and then it is formalized by Clarke *et al.* in [3, 9]. With the notions of ownership contexts and owners, the ownership relation between objects are structured by parameterized annotating classes. At the beginning, for each object can have at most one owner, it obtains the *owner-as-dominators* property. As the first static type system providing flexible object encapsulation, ownership types have been extended into two mainstreams.

In the first direction, people attempt to enhance ownership types to support inheritance and other features while keeping the owner-as-dominators property. Clarke and Drossopoulou [18] enforce computational effects to describe how to exploit the strong encapsulation for OO languages. Boyapati et al. [10] propose a way to implement the important programming idioms such as iterators by extending ownerships with inner classes, where objects of classes defined in the same modular have the privilege of accessing each other's representation. *Ownership Domains* [5] supports more precise and flexible aliasing protection, where *domain* stands for a logically related set of objects that are accessible between each other. Programmers can specify multiple domains in one object. An object can be in one shared or owned domain, and *links* between domains specify which objects can be accessed by objects in other domains.

The second line of attempts aims at minimizing annotations in programs, and meanwhile, providing flexible aliasing control. *Universe* [19, 4] takes rep and peer modifiers to build the topological ownership structures on a set of objects, and provides read-only references to enforce *owner-as-modifier* property, in the sense that other objects can read an object but only its owner can modify it. This work has been integrated into JML notation for Java specifications [20]. *Universe Types* [21] is later proposed to support ownership transfer. *Generic Universe Types* has been formalized in [22], and a compiler for Java with ownership types and generic universe types is implemented in [23]. A recent work on minimal annotations of ownership is *Pedigree Types* [24] inspired by universe types, where ownership information is represented with an explicit shaping *Pedigree*, *etc*. Our work also aims at simplifying annotations in programs while keeping the expressive power.

Most of the mechanisms discussed above have the restriction that one object can only have one owner. Cameron *et al.* develop *Multiple Ownership* [25], where objects can have more than one owner. It is suitable for implementing many modern program idioms such as design patterns. In other direction, *Confined Types* [1, 2] adopts a package-level encapsulation to provide better security.

As numerous aliasing control schemes with their own usages and limitations have been proposed, people also want to have a unified understanding of the confinement problem, and have published some results [26–28].

8 Conclusion

In this paper, we present a new framework for specifying and reasoning about confinement of OO programs. Our basic idea is inspired by the *ownership types*. In our approach, a class is not responsible for specifying future confinement requirement of other employing classes. If a class wants to confine its representation, it has to express this requirement entirely in its declaration by itself. In this sense, we say that the object confinement scheme proposed in this paper is built from outside. Thus, our framework is based on an intricate different idea from what of ownership types, although they have similar intentions and abilities.

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

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 future work is to define confinement types visible for multiple classes by connecting internal domain of these classes.

References

1. Vitek, J., Bokowski, B.: Confined types in Java. Software Practice and Experience **31** (2000) 507–532
2. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. In: Proc.of OOPSLA'01, ACM Press (2001)
3. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: Proc.of OOPSLA'98, ACM Press (1998)
4. Müller, P.: Modular specification and verification of object-oriented programs. PhD thesis, FernUniversit at Hagen, *LNCS 2262*, Springer (2002)
5. Aldrich, J., Chambers, C.: Ownership domains: Separating alias policy from mechanism. In: Proc.of ECOOP'04, Springer (2004)
6. Boyland, J.: Alias burying: Unique variables without reads. Software Practice and Experience **31**(6) (2001) 533–553
7. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: Proc.of OOPSLA'02, ACM Press (2002)
8. Clarke, D., Wrigstad, T.: External uniqueness. In: Proc.of FOOL'03, ACM Press (2003)
9. Clarke, D.: Ownership types and containment. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Australia (2001)
10. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: Proc.of POPL'03, ACM Press (2003)
11. Qiu, Z., Wang, S., Quan, L.: Sequential μJava: Formal foundations. In: Proc.of AWSF'07. (2007) Avaliable at: www.math.pku.edu.cn:8000/en/preindex.php.
12. Meyer, B.: The theory and calculus of aliasing. CoRR **abs/1001.1610** (2010)
13. Naeem, N.A., Lhoták, O.: Faster alias set analysis using summaries. In: Proc.of CC'11. Volume 6601 of Lecture Notes in Computer Science., Springer (2011) 82–103
14. 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
15. Hogg, J.: Islands: Aliasing protection in object-oriented languages. In: Proc. of OOPSLA'91, ACM Press (1991)
16. Almeida, P.S.: Balloon types: Controlling sharing of state in data types. In: Proc. of ECOOP'97, Springer (1997)
17. Noble, J., Vitek, J., Potter, J.: Flexible alias protection. In: Proc.of ECOOP'98, Springer (1998)
18. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: Proc.of OOPSLA'02, ACM Press (2002)
19. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for controlling representation exposure. Technical Report 263, FernUniversitat Hagen (1999)
20. Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Journal of Object Technology **4** (2005) 5–32

21. Müller, P., Rudich, A.: Ownership transfer in universe types. In: Proc.of OOPSLA'07, ACM Press (2007)
22. Dietl, W., Drossopoulou, S., Müller, P.: Generic universe types. In: Proc.of ECOOP'07, Springer (2007)
23. Cameron, N.R., Noble, J.: Encoding ownership types in Java. In: TOOLS (48), Springer (2010)
24. Liu, Y.D., Smith, S.F.: Pedigree types. In: Proc.of IWACO'08, Springer (2008)
25. Cameron, N.R., Drossopoulou, S., Noble, J., Smith, M.J.: Multiple ownership. In: Proc.of OOPSLA'07, ACM Press (2007)
26. Noble, J., Biddle, R., Tempero, E., Potanin, A., Clarke, D.: Towards a model of encapsulation. In: Proc.of IWACO'03, Darmstadt, Germany (2003)
27. Zhao, Y., Boyland, J.: A fundamental permission interpretation for ownership types. In: Proc.of TASE'08, IEEE CS (2008)
28. 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

A Soundness

To demonstrate the soundness of our type system, we follow the strategy [3] to prove subject reduction. First we give the operational semantics associated with a state comprised of a store and an object pool for CμJava. Then we introduce a definition for the well-confinedness on states. Finally, we prove a subject reduction theorem, which applies to commands with a given well confined state, to get the soundness of our type system.

A.1 Operational Semantics

Similar as [28], we define a program state as a pair composed of a stack and store. Formally,

$$\begin{array}{lcl}\text{Store} & \widehat{=} & \text{Variables} \rightharpoonup \text{Ref} \\ \text{Opool} & \widehat{=} & \text{Ref} \rightharpoonup \text{Attributes} \rightharpoonup \text{Ref} \\ \text{States} & \widehat{=} & \text{Store} \times \text{Opool}\end{array}$$

where a store maps variables (Variables) to values (Ref) and an object pool maps pairs of object references and attributes names (Attributes) to corresponding values.

The operational semantics given in Fig. 10 is expressed in terms of a transition relation on a state (σ, O), a command c and another state (σ', O'), written as $\langle c, (\sigma, O)\rangle \rightsquigarrow (\sigma', O')$ or $\langle c, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$, in which $\sigma \in \text{Store}, O \in \text{Opool}$. We use $\triangle(C, m)$ to get the signature of the method m in class C. For well-typed programs, we omitted the part of operational semantics on states that go wrong in the execution.

A.2 Well-confined State

Definition 8 (Confinement Context). *The confinement context in a method m of class C applies to the opool. The object pool O can be partitioned to several protected domains since the confinement status depends on many objects with their own representations. Assume there are n separated protected domains: $\{pd_{i=1}^n\}$, each domain consists of three separated parts: the confined attributes part of interface objects (ϕ_{1i}), the* [op-skip] $$\frac{}{\langle \mathsf{skip}, (\sigma, O)\rangle \rightsquigarrow (\sigma, O)}$$

[op-assign] $$\frac{}{\langle x := e, (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto \sigma e\}, O)}$$

[op-update] $$\frac{\langle \sigma e, a\rangle \in \mathsf{dom}_2\, O}{\langle e.a := x, (\sigma, O)\rangle \rightsquigarrow (\sigma, O \oplus \{\langle \sigma e, a\rangle \mapsto \sigma x\})}$$

[op-lookup] $$\frac{\langle \sigma e, a\rangle \in \mathsf{dom}_2\, O}{\langle x := e.a, (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto O(\sigma e)(a)\}, O)}$$

[op-asncast] $$\frac{\mathit{type}(\sigma e) \preceq T}{\langle x := (T)e, (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto \sigma e\}, O)}$$

[op-objcreate]
$$\frac{r \notin \mathsf{dom}\, O \wedge \{\overline{f} = \mathsf{attr}(C)\}}{\langle x := \mathsf{new}\, T(), (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto r\}, O \oplus \{\overline{\langle r, f\rangle \mapsto \mathsf{null}}\})}$$

[op-methinv]
$$\frac{\begin{array}{c}\mathit{type}(\sigma e_1) = D, \triangle(D, m_1) = \lambda(\overline{y})\{\mathsf{var}\ \overline{x}; c; \mathsf{return}\, e\} \\ \langle \overline{x} := \overline{\mathit{nil}}; c; \mathsf{return}\, e[(D)e_1/\mathsf{this}, \overline{e}/\overline{y}], (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')\end{array}}{\langle x := e_1.m_1(\overline{e}), (\sigma, O)\rangle \rightsquigarrow^* (\sigma \oplus \{x \mapsto \sigma' \mathsf{res}\}, O')}$$

[op-sequence]
$$\frac{\langle c_1, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O'), \langle c_2, (\sigma', O')\rangle \rightsquigarrow^* (\sigma'', O'')}{\langle c_1; c_2, (\sigma, O)\rangle \rightsquigarrow^* (\sigma'', O'')}$$

Fig. 10. Operational Semantics

confined attributes part of confined objects (ϕ_{2i}), and non-confined attributes part of confined objects (ϕ_{3i}). The confined objects in ϕ_{1i} might nest their own confinement contexts. We use $\theta_{C,m} \mathrel{\widehat{=}} \{(\phi_{1i}, \phi_{2i}, \phi_{3i}) | i = 1..n\}$ to denote the confinement context for method m in C, in which we use $(\phi_{11}, \phi_{21}, \phi_{31})$ to denote the partition decided by current object this. □

Definition 9 (well-confined State). *Assume $\theta_{C,m} = \{(\phi_{1i}, \phi_{2i}, \phi_{3i}) | i = 1..n\}$. (σ, O) is a* well-confined *state under $\theta(C, m)$ iff the following conditions are satisfied:*

- *Let $[\![\phi_{ji}]\!] (\sigma, O) = O_{ji}$ and $O_i = O_{1i} \cup O_{2i} \cup O_{3i}$ for $j = 1..3$ and $i = 1..n$, (O_{11}, O_{21}, O_{31}) denotes the partition decided by current object* this*, then*

$$\begin{array}{l} O_{1i}, O_{2i}, O_{3i} \textit{ are disjoint for } i = 1..n; \\ \textit{for each } O_i, (O_{1i} * O_{2i}) \triangleright (O_{2i} * O_{3i}); \\ (O \setminus \odot_{i=1}^{n}(O_{1i} * O_{2i})) \triangleright (O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i})) \end{array}$$

- *For each variable or return value $x \in \mathsf{dom}\,\sigma$, if the confinement scheme of x (denoted by CS_x) is $T\langle C, \ldots\rangle$, then $\sigma x \in \mathsf{dom}(O_{21} * O_{31})$; else $\sigma x \in \mathsf{dom}(O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$.*

We say O is well-confined *under $\theta_{C,m}$, if the first condition above holds.* □

Theorem 1 (Soundness Theorem). *If $\Gamma, \Theta \vdash P : \mathsf{ok}$, and P starts with a state $(\sigma_{init}, O_{init})$ that is well-confined, then if P is terminated in a state (σ', O'), (σ', O') is* *a well confined state under its confinement context, where $\sigma_{init} = \emptyset$, $O_{init} = \{root \mapsto$ $\mathsf{null}\}$.* □

For the soundness of our type system, we have to prove that if program P is well-typed under the type system then P is well-confined under its confinement context. Since P is a set of classes in CμJava, we finally must prove that for each command getting through the related typing rules in methods of P is well-confined under the confinement context of the related method.

Theorem 2 (Subject Reduction). *If $\Gamma, \Theta, C, m \vdash c : \mathsf{com}$, the execution of command c starts from any well-confined state (σ, O), then if the execution of c is terminated in a state (σ', O'), the state (σ', O') is well-confined.* □

Notice that here in the theorem of [Subject Reduction], (σ, O) is an any given well-confined states. However, if we prove the soundness under the reachable state $(\sigma_{C,m}, O_{C,m})$, it grantees that the soundness can be also kept under (σ, O). Since the unreachable part of the state would never be changed, $(\sigma, O) = (\sigma_{C,m} * \mathsf{true}, O_{C,m} * \mathsf{true})$. So we only prove the soundness on the reachable stack and opool.

Definition 10 (Reachable Opool). *For a call to method m of class C, the reachable opool $O_{C,m}$ contains all the objects can be accessed by the method, including the newly created objects, the refers through the caller's attributes and the objects referred by real parameters.* □

A.3 Soundness Proof

Now we prove the subject reduction theorem 2. Assume that the confinement context is $\theta(C, m)$ and the state is (σ, O) before the command execution, $[\![\theta(C, m)]\!] = \{(O_{1i}, O_{2i}, O_{3i}) | i = 1..n\}$ satisfies the conditions in definition 9 and each command satisfies the corresponding typing rule. The unreachable states from the current object obviously satisfy the confinement conditions of the current context, so we take no account of the garbage collection in the proof.

[skip] skip
Proof: It is trivial to prove that the statement skip preserves the well-confinedness.

[assign] $x := e$
Proof: By induction on the confinement typing derivation.

Case: $x : T \Rightarrow CS_x = T$

1. (a) By assumptions, $x := e$ satisfies the rule **[tp-assign]**
   (b) By operational semantics, $\langle x := e, (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto \sigma e\}, O)$
   (c) By assumptions, (σ, O) satisfies definition 9
2. (a) By $CS_x = T$, $\sigma x \in \mathsf{dom}(O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$
   (b) By (1.a) and $CS_x = T$, $\exists S.CS_e = S \preceq_e T$, $\sigma e \in \mathsf{dom}(O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$
   (c) By (1.a) and (1.b), $O' = O$, then by (1.c) all the partitions are still well-confined. And by (2.b), $\sigma' x = \sigma e \in \mathsf{dom}(O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$

Case: $x : \mathsf{ctype}[p] \Rightarrow CS_x = T\langle C \cdots\rangle$

1. (a), (b) and (c) are the same as (1.a), (1.b) and (1.c) stated in the previous case
2. (a) By $x : \mathsf{ctype}[p]$, $\sigma x \in \mathsf{dom}(O_{21} * O_{31})$
   (b) By (1.a), $CS_e = S\langle C \cdots\rangle \preceq_e CS_x$, $\sigma e \in \mathsf{dom}(O_{21} * O_{31})$ and moreover, the confined part of $O_{21} * O_{31}$ that σx and σe can access has the same confinement paths, eg., if $\exists(\sigma x, f, \sigma(x)(f)) \in (O_{21} * O_{31}) \Rightarrow$ $(\sigma e, f, \sigma(e)(f)) \in (O_{21} * O_{31})$
   (c) By (1.b), $O' = O$, by (2.b), all the partitions are still well-confined.
   (d) By (2.b) and (2.c), $\sigma' x = \sigma e \in \mathsf{dom}(O'_{21} * O'_{31})$

Consequently, for the two cases, by (2.c) in the first case and (2.c), (2.d) in the second case, (σ', O') under $\theta(C, m)$ satisfies the definition 9. The assignment statement keeps the well-confinedness.

[update] $e.a := x$
Proof: By induction on the confinement typing derivation.

Case: $x : T \Rightarrow CS_x = T$

1. (a) By assumptions, $e.a := x$ satisfies the rule **[tp-update]**
   (b) By operational semantics, $\langle \sigma e, a\rangle \in \mathsf{dom}_2\, O \Rightarrow \langle e.a := x, (\sigma, O)\rangle \rightsquigarrow (\sigma, O \oplus \{\langle \sigma e, a\rangle \mapsto \sigma x\})$
   (c) By assumptions, (σ, O) satisfies definition 9
2. (a) By $CS_x = T$, $\sigma x \in \mathsf{dom}(O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$
   (b) By (1.a) and $CS_x = T$, $\exists S. CS_{e.a} = S \preceq_e T \Rightarrow \sigma(e)(a) \in \mathsf{dom}(O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$
   (c) By (2.b), $(\sigma e, a) \in \mathsf{dom}_2\, O_{31} \vee (\sigma e, a) \notin \mathsf{dom}_2 \odot_{i=1}^{n}(O_{1i} * O_{2i} * O_{3i})$
      i. Case $(\sigma e, a) \in \mathsf{dom}_2\, O_{31} \Rightarrow (O'_{11}, O'_{21}, O'_{31}) = (O_{11}, O_{21}, O_{31} \oplus \{\langle \sigma e, a\rangle \mapsto \sigma x\})$, thus $(O'_{11} * O'_{21}) \rhd (O'_{21} * O'_{31})$. For this case, without changing any other partitions, $(O_{1i} * O_{2i}) \rhd (O_{2i} * O_{3i}), i = 2..n$ is preserved. Moreover, by (1.b) $O' = O \oplus \{\langle \sigma e, a\rangle \mapsto \sigma x\}$, $O' \setminus \odot_{i=1}^{n}(O'_{2i} * O'_{3i}) = O \oplus \{\langle \sigma e, a\rangle \mapsto \sigma x\} \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}), O' \setminus \odot_{i=1}^{n}(O'_{2i} * O'_{3i}) = O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}) \Rightarrow (O' \setminus \odot_{i=1}^{n}(O'_{1i} * O'_{2i})) \rhd (O' \setminus \odot_{i=1}^{n}(O'_{2i} * O'_{3i}))$
      ii. Case $(\sigma e, a) \notin \mathsf{dom}_2 \odot_{i=1}^{n}(O_{1i} * O_{2i} * O_{3i})$, all the partitions are not changed, thus $(O_{1i} * O_{2i}) \rhd (O_{2i} * O_{3i}), i = 1..n$ is preserved. Moreover, by (1.b) $O' = O \oplus \{\langle \sigma e, a\rangle \mapsto \sigma x\}, O' \setminus \odot_{i=1}^{n}(O'_{1i} * O'_{2i}) \rhd O' \setminus \odot_{i=1}^{n}(O'_{2i} * O'_{3i})$
   (d) By (2.b), $\sigma' x = \sigma x \in \mathsf{dom}(O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$

Case: $x : \mathsf{ctype}[p] \Rightarrow CS_x = T\langle C \cdots\rangle$

1. (a), (b) and (c) are the same as (1.a), (1.b) and (1.c) stated in the previous case
2. (a) By $x : \mathsf{ctype}[p]$, $\sigma x \in \mathsf{dom}(O_{21} * O_{31})$
   (b) By (1.a), $CS_{e.a} = S\langle C \cdots\rangle \preceq_e CS_x$, thus $(\sigma e, a) \in \mathsf{dom}_2(O_{11} * O_{21})$
      i. Case $(\sigma e, a) \in \mathsf{dom}_2\, O_{11} \Rightarrow CS_e = C\langle \mathsf{this}\rangle \wedge a \in \mathsf{cpath}(C)$, thus $(O'_{11}, O'_{21}, O'_{31}) = (O_{11} \oplus \{\langle \sigma e, a\rangle \mapsto \sigma x\}, O_{21}, O_{31})$, thus $(O'_{11} * O'_{21}) \rhd (O'_{21} * O'_{31})$. For this case, without changing any other partitions, $(O_{1i} * O_{2i}) \rhd (O_{2i} * O_{3i}), i = 2..n$ is preserved. Moreover, by (1.b) $O' = O \oplus \{\langle \sigma e, a\rangle \mapsto \sigma x\}$, $(O' \setminus \odot_{i=1}^{n}(O'_{1i} * O'_{2i})) \rhd (O' \setminus \odot_{i=1}^{n}(O'_{2i} * O'_{3i}))$

ii. Case $(\sigma e, a) \in \mathsf{dom}_2\, O_{21} \Rightarrow (O'_{11}, O'_{21}, O'_{31}) = (O_{11}, O_{21} \oplus \{\langle \sigma e, a\rangle \mapsto \sigma x\}, O_{31})$,thus $(O'_{11} * O'_{21}) \rhd (O'_{21} * O'_{31})$. Moreover, by (1.b) $O' = O \oplus \langle \sigma e, a\rangle \mapsto \sigma x\}$, $O' \setminus \odot_{i=1}^{n}(O'_{1i} * O'_{2i}) = O \setminus \odot_{i=1}^{n}(O_{1i} * O_{2i}) \wedge O' \setminus \odot_{i=1}^{n}(O'_{2i} * O'_{3i}) = O \setminus \odot_{i=1}^{n}(O_{1i} * O_{2i})$, thus $(O' \setminus \odot_{i=1}^{n}(O'_{1i} * O'_{2i})) \rhd (O' \setminus \odot_{i=1}^{n}(O'_{2i} * O'_{3i}))$

(c) By (1.b), $\sigma' x = \sigma x \in \mathsf{dom}(O_{21} * O_{31})$

Consequently, for the two cases, (σ', O') under $\theta(C, m)$ satisfies the definition 9. The well-confinedness is preserved.

[lookup] $x := e.a$
Proof: By induction on the confinement typing derivation.

Case: $x : T \Rightarrow CS_x = T$

1. (a) By assumptions, $x := e.a$ satisfies the rule **[tp-lookup]**
   (b) By operational semantics, $\langle \sigma e, a\rangle \in \mathsf{dom}_2\, O \Rightarrow \langle x := e.a, (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{\sigma x \mapsto O(\sigma e)(a)\}, O)$
   (c) By assumptions, (σ, O) satisfies definition 9
2. (a) By $CS_x = T$, $\sigma x \in \mathsf{dom}(O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$
   (b) By (1.a) and $CS_x = T$, $\exists S. CS_{e.a} = S \preceq_e T$, thus we get $O(\sigma e)(a) \in \mathsf{dom}(O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$
   (c) By (1.b) and (2.b), all the partitions in the confinement context are not changed, and $\sigma' x = O(\sigma e)(a) \in \mathsf{dom}(O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$, thus (σ', O') satisfies definition 9

Case: $x : \mathsf{ctype}[p] \Rightarrow CS_x = T\langle C \cdots\rangle$

1. (a), (b) and (c) are the same as (1.a), (1.b) and (1.c) stated in the previous case
2. (a) By $x : \mathsf{ctype}[p]$, $\sigma x \in \mathsf{dom}(O_{21} * O_{31})$
   (b) By (1.a), $CS_{e.a} = S\langle C \cdots\rangle \preceq_e CS_x$, thus $O(\sigma e)(a) \in \mathsf{dom}(O_{21} * O_{31})$
   (c) By (1.b), we get $O' = O$ and $(O'_{11}, O'_{21}, O'_{31}) =$
   $(O_{11}, O_{21}[O(\sigma e)(a)/\sigma x], O_{31}[O(\sigma e)(a)/\sigma x])$
   (d) By (1.c) and (2.c), $(O_{1i} * O_{2i}) \rhd (O_{2i} * O_{3i}), i = 1..n$
   (e) By $O' \setminus \odot_{i=1}^{n}(O'_{1i} * O'_{2i}) \rhd (O' \setminus \odot_{i=1}^{n}(O'_{2i} * O'_{3i}))$
   (f) By (1.b), $\sigma' x = O(\sigma e)(a) \in \mathsf{dom}(O_{21} * O_{31})$

Consequently, for the two cases, (σ', O') under $\theta(C, m)$ satisfies the definition 9. The well-confinedness is preserved.

[asncast] $x := (T)e$
Proof: By induction on the confinement typing derivation.

1. By assumptions, $x := (T)e$ satisfies the rule **[tp-asncast]** that only allows the upcasting
2. By the operational semantics,$type(\sigma e) \preceq T \Rightarrow \langle x := (T)e, (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto \sigma e\}, O)$
3. By the above, the proof is similar as $x := e$.

[objcreate] $x := \mathsf{new}\, T()$
Proof: By induction on the confinement typing derivation.

Case: $x : T \Rightarrow CS_x = T$

1. (a) By assumptions, $x := \mathsf{new}\,T()$ satisfies the rule **[tp-objcreate]**
   (b) By operational semantics, $\langle x := \mathsf{new}\,T(), (\sigma, O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto r\}, O \oplus \{\overline{\langle r, f\rangle \mapsto \mathsf{null}}\})$ Where $r \notin \mathsf{dom}\,O \wedge \{\overline{f}\} = \mathsf{attr}(C)$
   (c) By assumptions, (σ, O) satisfies definition 9
2. (a) By $CS_x = T$, we get $\sigma x \in \mathsf{dom}(O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$
   (b) By (1.a) and (1.b), $(O_{1i} * O_{2i}) \rhd (O_{2i} * O_{3i}), i = 1..n$ and $O' = O \oplus \{\overline{\langle r, f\rangle \mapsto \mathsf{null}}\}$, thus $O' \setminus \odot_{i=1}^{n}(O'_{1i} * O'_{2i}) = O \oplus \{\overline{\langle r, f\rangle \mapsto \mathsf{null}}\} \setminus \odot_{i=1}^{n}(O_{1i} * O_{2i}, O' \setminus \odot_{i=1}^{n}(O'_{2i} * O'_{3i}) = O \oplus \{\overline{\langle r, f\rangle \mapsto \mathsf{null}}\} \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}$, so we know $(O \setminus \odot_{i=1}^{n}(O_{1i} * O_{2i})) \rhd (O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$ is satisfied.
   (c) By 1.b) and 2.b), $r \notin \mathsf{dom}\,O \Rightarrow \sigma' x = \in \mathsf{dom}(O' \setminus \odot_{i=1}^{n}(O'_{2i} * O'_{3i}))$

Case: $x : \mathsf{ctype}[p] \Rightarrow CS_x = T\langle C \cdots \rangle$

1. (a), (b) and (c) are the same as (1.a), (1.b) and (1.c) stated in the previous case
2. (a) By $x : \mathsf{ctype}[p]$, $\sigma x \in \mathsf{dom}(O_{21} * O_{31})$
   (b) By (1.a) and (1.b), $(O'_{11}, O'_{21}, O'_{31}) = (O_{11}, O_{21}[r/\sigma x], O_{31}[r/\sigma x]) \wedge O' = O \oplus \{\overline{\langle r, f\rangle \mapsto \mathsf{null}}\}$, because for each $\{\overline{f}\} = \mathsf{attr}(C)$ refer to null, thus $(O_{11} * O_{21}[r/\sigma x]) \rhd (O_{31}[r/\sigma x])$, i.e., $(O'_{11} * O'_{21}) \rhd (O'_{21} * O'_{31})$. Besides, for the other partitions are preserved, $(O'_{1i} * O'_{2i}) \rhd (O'_{2i} * O'_{3i}), i = 2..n$, so $(O \setminus \odot_{i=1}^{n}(O_{1i} * O_{2i})) \rhd (O \setminus \odot_{i=1}^{n}(O_{2i} * O_{3i}))$
   (c) By (1.b) and (2.b), $\sigma' x = r \in \mathsf{dom}(O'_{21} * O'_{31})$

Consequently, for the two cases, (σ', O') under $\theta(C, m)$ satisfies the definition 9. The well-confinedness is preserved.

[methodinv] $x := e_1.m_1(\overline{e})$
Proof: From the operational semantics, we have

$$\frac{\begin{array}{c} type(\sigma e_1) = D, \triangle(D, m_1) = \lambda(\overline{y})\{\mathsf{var}\ \overline{x}; c; \mathsf{return}\ e\} \\ \langle \overline{x := nil}; c; \mathsf{return}\ e[(D)e_1/\mathsf{this}, \overline{e}/\overline{y}], (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O') \end{array}}{\langle x := e_1.m_1(\overline{e}), (\sigma, O)\rangle \rightsquigarrow^* (\sigma \oplus \{x \mapsto \sigma' \mathsf{res}\}, O')}$$

We denote the confinement context of the method m invocation by $\mathcal{C}$, and the one where m_1 is defined by $\mathcal{D}$. We need to prove that, if $x := e_1.m_1(\overline{e})$ satisfies the related rule for the method invocation, then if (σ, O) is well-confined in $\mathcal{C}$, the resulted state $(\sigma \oplus \{x \mapsto \sigma' \mathsf{res}\}, O')$ is well-confined in $\mathcal{C}$.

From that O is well-confined in $\mathcal{C}$, if $\mathcal{C} \neq \mathcal{D}$, it can be partitioned into the following parts:

$$(O_{1C} * O_{1D}, O_{2C} * O_{2D}, O_{3C} * O_{3D})$$

where (O_{1C}, O_{2C}, O_{3C}) represents the three parts for the confinement context $\mathcal{C}$ and (O_{1D}, O_{2D}, O_{3D}) represents the three parts for $\mathcal{D}$. If the confinement context of $\mathcal{C}$and $\mathcal{D}$relate to the same this, then $\mathcal{C} = \mathcal{D}$, O_{1i} and O_{2i} are the same one, thus O is partitioned into (O_{1C}, O_{2C}, O_{3C}).

We will prove this fact by considering three cases for the separated premises on method invocation typing rules.

The first is for the case when **[tp-methinv]** is used. We need to prove it is sound. Here we will prove a stronger fact (1), which says that for any command s in $\overline{x} :=$

$\overline{nil}; c;$ return $e[(D)e_1/\mathsf{this}, \overline{e}/\overline{y}]$, if it ends at (σ', O') starting from (σ, O) and O is well-confined in $\mathcal{C}$, then the executing of s would not break the well-confinedness of $\mathcal{D}$. Moreover, O' is still well-confined in $\mathcal{C}$.

1. Case $CS_{e_1} : C\langle\mathsf{this}\rangle$, then $\mathcal{C} = \mathcal{D}$, proof for O' is the same for basic statements proved ahead; proof for $\sigma \oplus \{x \mapsto \sigma'\mathsf{res}\}$, res can refer to any part of O'.
2. Case $CS_{e_1} : T$, then $\mathcal{C} \neq \mathcal{D}$. From e_1 is not confined in $\mathcal{C}$, then e_1 and its attributes will not locate at $O_{1C} * O_{2C} * O_{3C}$. Plus the premise $\mathsf{sv}(e_1, TC)$ and $\overline{\mathsf{sv}(e_1, TC_y)}$, we know that TC and $\overline{TC_y}$ are all ordinary types. From $\overline{CS_e \preceq_e CS_y}$, $\overline{CSy}$ are all ordinary types too. Therefore, $\overline{e}$ are not confined in $\mathcal{C}$. Besides, by $\sigma(TC, C) \preceq_e CS_x$, CS_x is some ordinary type, then x is not confined in $\mathcal{C}$ either. By induction on the commands s in $\mathcal{D}$,
   (a) Case x:=e: when x and e are of confinement type ctype[•], by $\mathcal{C} \neq \mathcal{D}$, x, e are both confined in $\mathcal{D}$, then x, e will not change $O_{iC}, i = 1..3$ at all. From the well-confinedness of assignment $x := e$ in $\mathcal{D}$, we can see the assignment will preserve the well-confinedness of $O_{iD}, i = 1..3$ in $\mathcal{D}$. So O' is still well-confined in $\mathcal{C}$. On the other hand, if x and e are of type T, then the assignment will not change the partition $(O_{1C} * O_{1D}, O_{2C} * O_{2D}, O_{3C} * O_{3D})$ at all, obviously O' is well-confined in $\mathcal{C}$.
   (b) Case $x := e.a$: when x is of confinement type ctype[•], $x, e.a$ are confined in $\mathcal{D}$. After the lookup, $O_{1D} * O_{2D} * O_{3D}$ will be changed to a still well-confined partition $O_{1D} * O'_{2D} * O'_{3D}$. $O_{iC}, i = 1..3$ are not changed at all. From the well-confinedness of assignment $x := e.a$ in $\mathcal{D}$, we can see the assignment will preserve the well-confinedness of $O_{iD}, i = 1..3$ in $\mathcal{D}$. So O' is still well-confined in $\mathcal{C}$. On the other hand, if x is of confinement type T, then $e.a$ is of confinement scheme T, the assignment will not change the partition (O_{1D}, O_{2D}, O_{3D}) at all. In addition, both e_1 and $\overline{e}$ are not confined in $\mathcal{C}$, then the well-confinedness of $O_{iD}, i = 1..3$ in $\mathcal{D}$ is preserved. Thus O' is well-confined in $\mathcal{C}$.
   (c) Case $e.a := x$: similar as the previous case $x := e.a$.
   (d) Case $x := (T)e$: similar as first case $x := e$.
   (e) Case $x := $ new $T()$: when x is of confinement type ctype[•], $O'_{2D} \oplus \{\overline{\langle r, f\rangle \mapsto \mathsf{null}}\}$ where $r \notin \mathsf{dom}\, O \wedge \{\overline{f}\}$ are the confined attributes of T and
   $O'_{3D} \oplus \{\overline{\langle r, f\rangle \mapsto \mathsf{null}}\}$, where $r \notin \mathsf{dom}\, O \wedge \{\overline{f}\}$ are the
   non-confined attributes of T, the partition is still preserved, thus O' is well-confined in $\mathcal{C}$.
   (f) $x := e.m(\overline{e})$: proved by induction.
   (g) return e: it will not change the opool at all.
   So the fact (1) is proved.
   Now we need to prove that $\sigma \oplus \{x \mapsto \sigma'\mathsf{res}\}$ is well-confined in $\mathcal{C}$. From $\mathcal{C} \neq \mathcal{D}$ and the premise $\sigma(TC, \mathsf{dtype}(CS)) \preceq_e CS_x$, x is not confined in $\mathcal{C}$. We need to prove $\sigma'\mathsf{res}$ is not confined in $\mathcal{C}$and $\mathcal{D}$. From the operational semantics, $\sigma'\mathsf{res} = \sigma' e$, and the rule **[tp-method]** for m_1, $CS_e \preceq_e \sigma(TC, CS_{e_1})$, by TC is an ordinary type, we get the return e is not confined in $\mathcal{C}$and $\mathcal{D}$. The fact holds.

The second is for the case that the caller e_1 or actual arguments $\overline{e}$ for m_1 are confined in $\mathcal{C}$, and m_1 being called is declared without confinement types. For this case, we apply the typing rule **[tp-methinv']**, and we need to prove its soundness. From the typing rule, we have the facts that all the aliases for the command c; $\mathsf{return}\ e$ in the body of m_1 have the compatible confinement schemes with the related instantiation substituted, we can get the fact that all the alias in $c; \mathsf{return}\ e[(D)e_1/\mathsf{this}, \overline{e}/\overline{y}]$ have the compatible confinement schemes. Therefore, after executing $c; \mathsf{return}\ e[(D)e_1/\mathsf{this}, \overline{e}/\overline{y}]$, (σ', O') is still well-confined. The fact is proved.

[sequence] $c_1; c_2$;
Proof: By induction on the structure of the command c_1 and c_2. By assumptions, before executing c_1, the state (σ, O) is well-confined. By operational semantics, after the executing c_1, the state is (σ', O'). For each kind of commands, we have proved the well-confinedness maintained, so that the state (σ', O') is well-confined. Similarly, after executing c_2, we get that the state (σ'', O'') is well-confined.