Journal of Object Technology

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 5, Special Issue: Workshop on FTfJP, ECOOP 2006, June 2007

A Parameterized Type System for Simple Loose Ownership Domains

Jan Schäfer, TU Kaiserslautern, Germany
Arnd Poetzsch-Heffter, TU Kaiserslautern, Germany

Ownership Domains generalize ownership types. They support programming patterns such as iterators that are not possible with ordinary ownership types. However, they are still too restrictive for cases in which an object X wants to access the public domains of an arbitrary number of other objects, which often happens in observer scenarios. To overcome this restriction, we develop so-called *loose* domains which abstract over several *precise* domains. Similar to the relation between supertypes and subtypes, we get a relation between loose and precise domains. In addition, we simplify ownership domains by reducing the number of domains per object to two and hard-wiring the access permissions between domains. We formalize the resulting type system for an OO core language and prove type soundness and a fundamental accessibility property.

1 INTRODUCTION

Showing the correctness of object-oriented programs is a difficult task. The inherent problem is the combination of aliasing, subtyping, and imperative state changes. Ownership type systems [13, 27, 12, 9] support the encapsulation of objects and guarantee that all objects are reachable from the root only via paths through their owner object. This property is called *owners-as-dominators* [13]. Unfortunately, this property prevents important programming patterns such as the efficient implementation of iterators [28]. Iterators of a linked list, for example, need access to the internal node objects, but must also be accessible by the clients of the linked list.

Ownership domains (OD) [2] generalize ownership types. Objects are not directly owned by other objects. Instead, every object belongs to a certain domain, and domains are owned by objects. Every object can own an arbitrary number of domains, but an object can only belong to a single domain. The programmer specifies with *link* declarations which domains can access which other domains. This indirectly specifies which objects can access which other objects, as objects can only access objects of domains to which its domain has access to. Beside the link declarations, domains can be declared as `public`. If an object X has the right to access an object Y, then X has also the right to access all public domains of Y.

In OD, variables and fields are annotated with domain types. The type rules enforce the following restriction: If a field or variable v holds a reference to an object X with a public domain D, and we want to store an object in D into a variable w, then v has to be `final` and w is annotated by $v.D$. Thus, it is *impossible* with

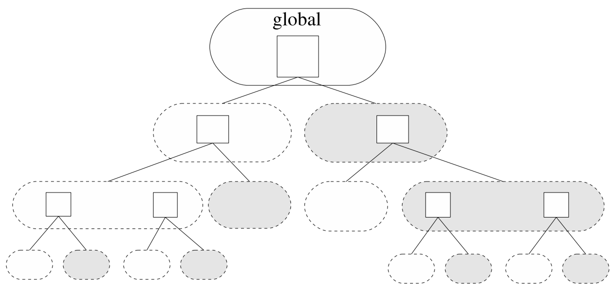

Figure 1: The ownership and containment relation of objects and domains form a tree rooted by a global domain. Solid rectangles represent objects, dashed rounded rectangles represent domains, where gray rectangles are local domains and white ones are boundary domains. An edge from an object X to a domain d means that X owns d.

the OD approach to store an arbitrary number of objects of public domains in an object, as for every object of a public domain there must be a corresponding `final` field, and the number of `final` fields must be known statically.

The problem is that the OD approach requires that the *precise* domain of every object is known statically. But sometimes there are situations in which a programmer does not know the precise domain but only knows a set of possible domains. With our type system it is possible to specify so-called *loose* domains which represent a set of possible domains, allowing to abstract from the precise domain.

The remainder of this paper is as follows. In the next section, we explain our approach together with two examples. In Section 3, we present a formalization of a core object-oriented language that encorparates simple loose ownership domains. The dynamic semantics for that language is given in Section 4. In Section 5 we present the central properties of our language. Section 6 discusses our approach together with related work. We conclude and give an outlook on future work in Section 7.

2 SIMPLE LOOSE OWNERSHIP DOMAINS

The basic idea of Simple Loose Ownership Domains (SLOD) is the same as that of OD [2]: objects are grouped into distinct *domains*, domains are *owned* by objects, and every object belongs to exactly one domain. In this paper, we simplify the ownership domain approach of Aldrich and Chambers [2] in two ways: Every object owns exactly two domains, namely a *local* domain and a *boundary* domain. Thus, SLOD needs no domain declarations. In addition, access permissions between domains are hard-wired, so SLOD needs no *link* declarations.

Accessibility Properties

Objects that are in the local domain of an object X belong to the representation of X and are encapsulated. Objects of the boundary domain of X are objects that are accessible from the outside of X, but at the same time are able to access the representation objects of X. In terms of OD the boundary domain is a *public* domain. The ownership relation of objects and domains forms a hierarchy, where the root of the hierarchy is a special *global* domain (see Figure 1). Furthermore, we call an object X the *owner* of an object Y if X owns the domain of Y.

The domain structure determines which objects can access each other. Let X and Y be objects. We say that X *can access* Y if and only if one of the following conditions is true:

- Y belongs to the global domain.
- X is the owner of Y.
- The owner of X can access Y.
- Y belongs to the boundary domain of an object Z that X can access.

More interesting, however, than the objects that *can* be accessed are the objects that can *not* be accessed, because this complementary relation leads us to a generalization of the owners-as-dominators property which is enforced by ownership type systems [13]. The ownership-as-dominators property states that all access paths from the root to any object go through its owner object.

The *domain subtree* of an object X consists of X and, recursively, of all objects that are owned by an object in the domain subtree. An object is *outside* of an object X if it does not belong to the domain subtree of X. The *boundary* of X is the set of objects consisting of X and, recursively, of all objects in the boundary domains owned by an object in the boundary of X. An object is *inside* of X if it belongs to the domain subtree of X, but not to its boundary. With these definitions, SLOD guarantees the following property:

> All access paths from objects *outside* of X to objects *inside* of X go through X's *boundary.*

This *boundary-as-dominators* property is a generalization of the owners-as-dominators property, as the owners-as-dominators property for an object X can be enforced in SLOD by putting no objects into the boundary domain of X, leading to a boundary of X that only contains X.

$$
\begin{array}{rcl}
\mathit{domain} & ::= & \texttt{global} \mid \texttt{same} \mid d \mid \mathit{owner}.\mathit{kind} \\
\mathit{owner} & ::= & \texttt{this} \mid \texttt{owner} \mid x \mid \mathit{domain} \\
\mathit{kind} & ::= & \texttt{local} \mid \texttt{boundary} \\
d & \in & \textit{domain parameters} \\
x & \in & \textit{final fields and final variables}
\end{array}
$$

Figure 2: Syntax of domain annotations in SLOD.

Domain Annotations

To statically check the boundary-as-dominators property, types in SLOD are augmented with domain annotations. Figure 2 shows the complete syntax of domain annotations.

A domain annotation consists of an owner part and a kind part. The owner part represents the owner object of the domain, the kind part describes the domain kind. For example, the domain annotation `this.local` has owner part `this` and kind part `local`, where `this` represents the current this-object. The `owner` keyword stands for the owner object of the domain in which the current this-object is contained[1]. We also allow `final` fields or `final` variables as owners of domains. Beside the composite domain annotations there are three single domain annotations, these are `global`, `same` and d, where d is a domain parameter. The annotation `global` represents the global domain which is accessible from everywhere and `same` represents the domain in which the this-object is contained in. With domain parameters classes can be made parametric in certain domains.

Domain annotations statically restrict the possible values that a variable or field can hold. For example, a local variable of type `this.local T` can only hold references to `T`-objects that are in the local domain of the current this-object. This subsection introduces the use of domain annotations. The next subsection will explain loose domains in more detail.

We describe domain annotations along with the linked list example in Fig. 3 that in particular illustrates how data structures with iterators can be handled. In the code examples we give a full annotation of all types with domain annotations. However, this annotation overhead can be significantly reduced by using defaults and annotation inference which we describe elsewhere [30].

To make objects, such as the Node objects of the list, inaccessible from the outside of the list, they are placed into the local domain of the owner. Hence, the `head` field of `LinkedList` is annotated with `this.local`. As can be seen in method `add`, this domain annotation is established when `Node` objects are created.

[1]This should not be confused with other ownership type systems such as the one of Clarke et al. [13] where `owner` represents the owning context. In our system the domain in which the this-object is contained in is denoted by `same`, which can be seen as equivalent to the `owner` keyword of Clarke et al.

```
public class LinkedList<d> {
  this.local Node<d> head;
  void add(d Object o) {
    head = new
      this.local Node<d>(o,head);
  }
  this.boundary Iter<d> iter() {
    return new
      this.boundary Iter<d>(head);
  }
}

public class Iter<d> {
  owner.local Node<d> current;
  Iter(owner.local Node<d> head) {
    current = head;
  }
  boolean hasNext() {
    return current != null;
  }
  d Object next() {
    d Object result =
      current.value;
    current = current.next;
    return result;
  }
}
```

```
public class Node<d> {
  d Object value;
  same Node<d> next;
  Node(d Object value,
       same Node<d> n)
  {
    this.value = value;
    this.next = next;
  }
}

public class Main {
  ...
  final this.local
    LinkedList<this.local> l;
  l = new LinkedList<this.local>();
  l.add(new this.local Object());
  // precise domain
  l.boundary Iter<this.local> it;
  it = l.iterator();
  // loose domain
  this.local.boundary
    Iter<this.local> it2;
  it2 = it;
  this.local Object obj = it2.next();
  ...
}
```

Figure 3: A linked list with iterators.

As the Iter objects of the linked list should be accessible from the outside of the linked list and at the same time must be able to access the internal Node objects, the Iter objects are put into the boundary domain of the linked list. Hence, the iterator method of the LinkedList class returns a new this.boundary Iter instance. Within the class Iter, Node objects have domain annotation owner.local indicating that they belong to the local domain of the list object. Thus, the current field is annotated with owner.local. Note that our approach simplifies the use of ownership domains, as in the approach of Aldrich and Chambers [2], the Iter class would need a domain parameter to represent the domain of the Node objects.

In class Node, the next field of Node is annotated with same to indicate that the next object is in the same domain as the current object. In case of the linked list, this is the local domain of the list object (as the Node class is only used for the linked list, we also could have annotated the next field with owner.local). The value field illustrates the use of a domain parameter.

The applications of classes `LinkedList` and `Iter` in `Main` demonstrate further interesting features of SLOD. The variable `it`, for example, is declared with domain annotation `l.boundary`. As `l` is a `final` variable, this is a precise domain annotation. It represents the boundary domain of the `LinkedList` object referenced by variable `l`. Such domain annotations are also supported by OD.

Our approach additionally provides the possibility to use loose domain annotations. A domain annotation is loose if the owner part is a domain, For example, `this.local.boundary` denotes a loose domain representing the set of all boundary domains of all objects that belong to the local domain of the receiver object. Variable `it2` is declared exactly like that. As the domain `l.boundary` is contained in the set of possible domains represented by `this.local.boundary`, it is possible to assign `it` to `it2`. Note that this kind of annotation needs no `final` variable. More details on loose domains are explained in the next subsection.

The `LinkedList`, `Node` and `Iter` classes are parameterized with a domain parameter `d` that represents the domain of the stored data. In the example, the `Main` class instantiates that parameter with `this.local`.

Note that we only allow domains as type parameters and not general types. This is done only to simplify the formalization of our language, it is not a limitation of our approach. Dietl et al. [16] shows how to unify domain parameters with general type parameters, which can also be applied to our language.

Loose Domains

Loose domains allow to abstract from the precise domain of an object. This is a new feature of SLOD compared to the approach of Aldrich and Chambers [2], which increases the flexibility of our system, without loosing any encapsulation properties. In the following, we describe the application and soundness aspects of this feature.

To demonstrate the enhanced expressiveness of loose domains, we use a slightly modified version of an example given by Aldrich and Chambers [2] (see Figure 4). It is a model-view system. `Model` objects allow to register listener (`Lstnr`) objects. When an event happens at the model, the model notifies all registered `Lstnr` objects by calling the method `update(int)`. `View` objects have a state that is updated whenever one of its listeners is notified. Method `lstnr()` creates new `ViewLstnr` instances as boundary objects of their view. The example is a simplified version of the observer pattern [18] and is representative of a category of similar examples.

Loose ownership domains allow more than one `Lstnr` object to be registered at a `Model` object. In the example, the domain parameter of the `Model` object in class `Main` is instantiated with the loose domain `this.local.boundary`. The calls of `m.addLstnr(view.lstnr())` are allowed, because the result domain of `view.lstnr()` is `view.boundary`, and `view` is in domain `this.local`. Thus, `view.boundary` is in the loose domain `this.local.boundary`. In the ownership type system of Aldrich and Chambers [2] this solution is not possible, because

```
interface Lstnr {
   public void update(int data);
}
class View {
  this.local State state;
  this.boundary Lstnr lstnr() {
    return new this.boundary
        ViewLstnr(state);
  }
}

class Model<d> {
  this.local LstnrList<d> lstnrs;
  void addLstnr(d Lstnr l) {
    lstnrs.add(l);
  }
  void notifyAll(int data) {
    for (d Lstnr l : lstnrs) {
      l.update(data);
    }
  }
}
```

```
class ViewLstnr
  implements Lstnr
{
  owner.local State state;
  ViewLstnr(owner.local State s)
  { this.state = s; }
  public void update(int data)
  { /*perform changes on state*/ }
}

class Main {
  ...
  this.local Model<
    this.local.boundary Lstnr> m;
  m = new this.local Model<
    this.local.boundary Lstnr>();
  this.local View view =
    new this.local View();
  m.addLstnr(view.lstnr());
  view = new this.local View();
  m.addLstnr(view.lstnr());
  ...
}
```

Figure 4: A model-view system with listener callbacks.

the parameter of the Model class had to be instantiated with the precise domain view.boundary, where view had to be a final variable. Hence, it would not be possible to add a Lstnr object of a different View object to the Model object.

To guarantee the soundness of our system, we have to restrict the usage of a type that is annotated with a loose domain annotation (a loose type). It is, for example, not possible to update same annotated fields on loose types. In the following code example, the assignment b.f = b2.f is not allowed, even though the static types of b.f and b2.f are the same, as b is a loose type, and thus the precise domain of b.f is not known statically.

```
// ...
this.local A a = new this.local A();
this.local.boundary A b = a.b;
this.local A a2 = new this.local A();
this.local.boundary A b2 = a2.b;
b.f = b2.f; // forbidden
```

```
class A {
  same A f;
  this.boundary A b;
}
```

3 STATIC SEMANTICS

In this section, we present a formalization of the core of SLOD. We call the language Simple Loose Ownership Domain Java (SLODJ). The formalization is based on several existing formal type systems for Java, namely Featherweight Java (FJ) [22] and ClassicJava [17], and is also inspired by several flavors of these type systems which already incorporate ownership information [13, 12, 2, 31, 24].

Domain annotations in SLODJ are similar to those in SLOD. The only differences are that in SLODJ fields cannot be owners of domain annotations and that class parameters are only domain parameters and not type parameters.

Notations. We use similar notations as FJ [22]. A bar indicates a sequence: $\overline{L} = L_1, L_2, \ldots, L_n$, where the length is defined as $|\overline{L}| = n$. Similar, $\overline{T}\ \overline{f}$; is equal to $T_1\ f_1; T_2\ f_2; \ldots; T_n\ f_n$. If there is some sequence $\overline{x}$, we write x_i for any element of $\overline{x}$. The empty sequence is denoted by $\bullet$. We often use sequences as arguments for functions which are only defined on single elements. This means that the function is applied to each element of the sequence.

Syntax and Types

The abstract syntax of SLODJ is shown in Figure 5. Underlined syntactic elements do not belong to the user syntax, but can occur during typing. Square brackets [ ] denote optional elements. A SLODJ program consists of a list of class declarations, a class name, and an expression. A class declaration consists of a class name with a list of domain context parameters, an optional super type, a sequence of field declarations, and a sequence of method declarations.

In contrast to FJ and Java, but like the formalization of Lu and Potter [24], we have no special root class `Object` which is the super class of all classes. This simplifies the formalization and in fact makes the language more general. The Java restriction can be enforced in our language by demanding that all classes except a special `Object` class must have a super class. Classes have no constructors; objects are created with all fields initialized to *null*. One consequence is that the `new` expression takes a type as argument only. Method declarations always have a result type, and a single body expression, which is always the result of the method. A type consists of an owning domain, a class name, and a list of domain parameters. `let` expressions bind variables and are similar to `final` variable declarations in Java. We support field reads and also field updates to get a more realistic model of Java. The domain annotations of a type $T = d\ C\langle\overline{d}\rangle$ consist of *owning domain* d and *domain parameters* $\overline{d}$. Note that the owning domain is not part of the domain parameters. This is different from some ownership type system where the owning context is part of the parameters.

$$
\begin{array}{rcllcl}
P & \in & \textbf{Program} & ::= & (\overline{L}, C, e) \\
L & \in & \textbf{ClassDecl} & ::= & \texttt{class}\ C\langle\overline{\alpha}\rangle\ [\texttt{extends}\ D\langle\overline{\beta}\rangle]\ \{\ \overline{T}\ \overline{f};\ \overline{M}\ \} \\
M & \in & \textbf{MethDecl} & ::= & T\ m\ (\overline{T}\ \overline{x})\{e\} \\
T, U & \in & \textbf{Type} & ::= & d\ C\langle\overline{d}\rangle \\
e & \in & \textbf{Expr} & ::= & \texttt{new}\ T \mid x \mid e.f \mid e.f = e' \mid \\
 & & & & \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' \mid e.m(\overline{e}) \\
d, g & \in & \textbf{Domain} & ::= & a.b \mid \texttt{global} \mid \alpha \mid \texttt{same} \mid \underline{!d} \\
a & \in & \textbf{DomOwner} & ::= & x \mid \texttt{this} \mid \texttt{owner} \mid \underline{?} \\
b & \in & \textbf{DomTail} & ::= & c \mid b.c \\
c & \in & \textbf{DomKind} & ::= & \texttt{local} \mid \texttt{boundary} \\
f & \in & \textbf{FieldName} & & x, y \in \textbf{Variable} \\
m & \in & \textbf{MethName} & & C, D \in \textbf{ClassName} \\
\alpha, \beta & \in & \textbf{DomParam} & &
\end{array}
$$

Figure 5: SLODJ syntax

Lookup Functions

To retrieve methods and fields we define corresponding lookup functions which are shown in Figure 6. As our language does not allow overloading, methods can be looked-up by their name only. If a method is not found in the class definition it is looked-up in the super class (METH-LKP-INHRT).

Auxiliary Functions

We need some auxiliary functions shown in Figure 7. *isPrecise* says whether a domain is precise, i.e. not loose. The `global` domain is precise, domains where the tail only consists of a single element are precise, and the domain `same` is precise. As for domain context parameters it is not possible to say whether they are instantiated by precise or loose domains, we have to assume that they are loose. The function *prcsOwner* returns the precise owner of a given domain. If the domain has no known owner, which is the case for domain context parameters, it returns "?". "?" stands for an invalid owner which leads to an invalid domain and an invalid type. In the case, where a domain has a known owner, but the owner is not precise ($d.c$), the owner is augmented with an exclamation mark "!". Similar, the function *prcsDomain* adds a "!" depending on whether the domain is precise or not. "!" is only used by the *assignable* function. In all other rules we implicitly ignore the "!". *assignable* defines whether a domain is assignable or not. A domain is assignable if and only if it is not augmented with a "!". A type is assignable if all its domain annotations are assignable.

(PARAMS)
$$\frac{\texttt{class}\ C\langle\overline{\alpha}\rangle\ \ldots}{params(C) = \overline{\alpha}}$$

(FIELDS-DRCT)
$$\frac{\texttt{class}\ C\langle\overline{\alpha}\rangle\ \{\overline{T}\ \overline{f};\ \ldots\}}{fields(\_\ C\langle\overline{d}\rangle) = [\overline{d}/\overline{\alpha}]\overline{T}\ \overline{f}}$$

(FIELDS-INHRT)
$$\frac{\texttt{class}\ C\langle\overline{\alpha}\rangle\ \texttt{extends}\ D\langle\overline{\beta}\rangle\ \{\overline{T}\ \overline{f};\ \ldots\}}{fields(\_\ C\langle\overline{d}\rangle) = [\overline{d}/\overline{\alpha}](\overline{T}\ \overline{f}, fields(\_\ D\langle\overline{\beta}\rangle))}$$

(FTYPE)
$$\frac{fields(T) = \ldots, T'\ f, \ldots}{fType(T, f) = T'}$$

(METHOD-DRCT)
$$\frac{\texttt{class}\ C\langle\overline{\alpha}\rangle\ \ldots\ T\ m(\overline{T}\ \overline{x})\{e\}\ \ldots}{method(\_\ C\langle\overline{d}\rangle, m) = [\overline{d}/\overline{\alpha}]\ T'\ m(\overline{T'}\ \overline{x})\{e\}}$$

(METHOD-INHRT)
$$\frac{\texttt{class}\ C\langle\overline{\alpha}\rangle\ \texttt{extends}\ D\langle\overline{\beta}\rangle\ \{\ldots;\overline{M}\}\qquad m \text{ not in } \overline{M}}{method(\_\ C\langle\overline{d}\rangle, m) = [\overline{d}/\overline{\alpha}]\ method(\_\ D\langle\overline{\beta}\rangle, m)}$$

Figure 6: Lookup functions

The Function σ

To translate domain annotations of fields and methods to the calling context we use the function σ. It takes the receiver expression, the receiver type and the type to be translated and returns the translated type. The function replaces the keywords `this`, `owner` and `same`. `this` is replaced by the receiver expression e, `owner` is replaced by the precise owner of domain d as defined by function *prcsOwner*, and `same` is replaced by the domain resulting from the *prcsDomain* function.

Note that this replacement can produce illegal types. For example, domains where `this` is replaced by an expression which is not a variable, or domains where `owner` is replaced by ?. This is intended and these errors are caught by the type system. In addition, it can happen that domains like `global.local` can appear, which are not allowed by our syntax. We solve this by implicitly treating domains of the form `global`.b as `global`.

$$\sigma(d\ C\langle\_\rangle, e) \cdot T = [e/\texttt{this}, prcsOwner(d)/\texttt{owner}, prcsDomain(d)/\texttt{same}]T$$

Type System

The type rules of SLODJ are shown in Figures 12 and 13. We use the judgments shown in Figure 8. The environment Γ is a finite mapping from variables to types.

$$\frac{}{isPrecise(\texttt{global})} \qquad \frac{}{isPrecise(a.c)} \qquad \frac{}{isPrecise(\texttt{same})}$$

$$\frac{}{prcsOwner(\texttt{global}) = \texttt{global}} \qquad \frac{}{prcsOwner(a.c) = a} \qquad \frac{}{prcsOwner(\texttt{same}) = \texttt{owner}}$$

$$\frac{}{prcsOwner(d.c) = !d} \qquad \frac{}{prcsOwner(\alpha) = ?} \qquad \frac{\neg isPrecise(d)}{prcsDomain(d) = !d}$$

$$\frac{isPrecise(d)}{prcsDomain(d) = d} \qquad \frac{\nexists d' : d = !d'}{assignable(d)} \qquad \frac{assignable(d, \overline{d})}{assignable(d\ C\langle\overline{d}\rangle)}$$

Figure 7: Auxiliary functions

$\Gamma; \Delta \vdash \diamond$	Γ is a well-formed environment
$\Gamma; \Delta \vdash T$	T is a well-formed type in Γ
$\Gamma; \Delta \vdash d$	d is a well-formed domain in Γ
$\Gamma; \Delta \vdash T <: U$	T is a subtype of U in Γ
$\Gamma; \Delta \vdash e : T$	e is a well-formed expression of type T in Γ
$\Gamma; \Delta \vdash d \to^{s} d'$	domain d can access domain d'
$C \vdash M$	M is a well-formed method declaration in class C
$\vdash L$	L is a well-formed class declaration
$\vdash P : T$	P is a well-formed program of type T

Figure 8: Judgments for the type system of SLODJ

The context Δ stores a list of valid domain context parameters.

$$\begin{aligned} \Gamma &::= \varnothing \mid \Gamma, x : T \\ \Delta &::= \bullet \mid \Delta, d \end{aligned}$$

For any program $(\overline{L}, C, e)$, we assume an implicitly given fixed class table CT mapping class names to their definitions. All judgments are implicitly parameterized with that class table. The class table is assumed to satisfy the following conditions. (1) $\overline{L} = ran(CT)$; (2) $\forall C \in dom(CT).\ CT(C) = \texttt{class}\ C\ \ldots$; (3) For every class name C appearing anywhere in CT, we assume $C \in dom(CT)$; (4) There are no cycles in the subclass relation induced by CT. In contrast to FJ and Java, we did not model a special `Object` class as a superclass of every class, as this has no effect to the encapsulation properties.

(SA-REFL)
$$\frac{}{\Gamma; \Delta \vdash d \to^s d}$$

(SA-SAME-OWNER)
$$\frac{}{\Gamma; \Delta \vdash a.c \to^s a.c'}$$

(SA-PARAM)
$$\frac{a \in \{\texttt{this}, \texttt{owner}\} \qquad d \in \Delta}{\Gamma; \Delta \vdash a.c \to^s d}$$

(SA-SAME-1)
$$\frac{d \in \Delta}{\Gamma; \Delta \vdash \texttt{same} \to^s d}$$

(SA-SAME-2)
$$\frac{a \in \{\texttt{this}, \texttt{owner}\}}{\Gamma; \Delta \vdash a.c \to^s \texttt{same}}$$

(SA-OWNER-1)
$$\frac{}{\Gamma; \Delta \vdash \texttt{this}.c \to^s \texttt{owner}.c'}$$

(SA-OWNER-2)
$$\frac{}{\Gamma; \Delta \vdash \texttt{same} \to^s \texttt{owner}.c}$$

(SA-BOUNDARY-1)
$$\frac{\Gamma; \Delta \vdash d \to^s d'}{\Gamma; \Delta \vdash d \to^s d'.\texttt{boundary}}$$

(SA-BOUNDARY-2)
$$\frac{}{\Gamma; \Delta \vdash d.\texttt{boundary} \to^s d}$$

(SA-BOUNDARY-VAR)
$$\frac{\Gamma; \Delta \vdash x : d\ C\langle\_\rangle \qquad \Gamma; \Delta \vdash d' \to^s d}{\Gamma; \Delta \vdash d' \to^s x.\texttt{boundary}}$$

(SA-GLOBAL)
$$\frac{}{\Gamma; \Delta \vdash d \to^s \texttt{global}}$$

Figure 9: Static Accessibility Relation

Static Accessibility Relation

Domain annotations represent sets of possible runtime domains. During runtime the accessibility between objects is restricted by their runtime domains. To statically guarantee that during runtime certain accesses may not happen, we define a relation on domain annotations. We call this relation the Static Accessibility Relation. It is written as $\Gamma; \Delta \vdash d \to^s d'$, read: "Domain d can access domain d' under type environment Γ and context Δ". The derivation rules of this relation are shown in Figure 9. For example, the relation states that domains can always access the boundary domain of domains they can access (SA-BOUNDARY-1). The Static Accessibility Relation is only used by the (T-TYPE) rule to ensure that no illegal accesses can be introduced by domain annotations. Note that the relation is not transitive.

Environments and Valid Types

The rules for well-formed environments and valid types are shown in Figure 10. (T-TYPE) ensures that the owning domain of a type can access the domain parameters, and that `this.local` can access all domains of a type. This ensures that no illegal access can be introduced by domain parameters, and that all domains are accessible by the receiver object. For example, the type `owner.local` $C\langle$`this.local`$\rangle$ is not allowed, because domain `owner.local` cannot access domain `this.local`.

$$\text{(T-ENV }\varnothing\text{)}\quad \frac{}{\varnothing;\Delta \vdash \diamond} \qquad\qquad \text{(T-ENV X)}\quad \frac{\Gamma;\Delta \vdash T \qquad x \notin dom(\Gamma)}{\Gamma, x : T;\Delta \vdash \diamond}$$

$$\text{(T-TYPE)}\quad \frac{\Gamma;\Delta \vdash \diamond \qquad \Gamma;\Delta \vdash \texttt{this.local} \to^{s} d, \overline{d} \qquad \Gamma;\Delta \vdash d \to^{s} \overline{d} \qquad |params(C)| = |\overline{d}|}{\Gamma;\Delta \vdash d\ C\langle\overline{d}\rangle}$$

Figure 10: Environments and types

$$\text{(S-DOMAIN REFL)}\quad \frac{}{\Gamma;\Delta \vdash d <:_{d} d} \qquad \text{(S-DOMAIN LOOSE)}\quad \frac{\Gamma;\Delta \vdash x : d\ C\langle\_\rangle}{\Gamma;\Delta \vdash x.c <:_{d} d.c} \qquad \text{(S-DOMAIN EXT)}\quad \frac{\Gamma;\Delta \vdash d <:_{d} d'}{\Gamma;\Delta \vdash d.c <:_{d} d'.c}$$

$$\text{(S-TYPE DOM)}\quad \frac{\Gamma;\Delta \vdash d <:_{d} d'}{\Gamma;\Delta \vdash d\ C\langle\overline{d}\rangle <: d'\ C\langle\overline{d}\rangle} \qquad \text{(S-TYPE CLASS)}\quad \frac{\texttt{class}\ C\langle\overline{\alpha}\rangle\ \texttt{extends}\ D\langle\overline{\beta}\rangle\ \ldots}{\Gamma;\Delta \vdash d\ C\langle\overline{d}\rangle <: d\ D\langle[\overline{d}/\overline{\alpha}]\overline{\beta}\rangle} \qquad \text{(S-TYPE REFL)}\quad \frac{}{\Gamma;\Delta \vdash U <: U}$$

$$\text{(S-TYPE TRANS)}\quad \frac{\Gamma;\Delta \vdash T' <: U \qquad \Gamma;\Delta \vdash U <: T''}{\Gamma;\Delta \vdash T' <: T''}$$

Figure 11: Subtyping

Subtyping

The subdomain and subtype relations are shown in Figure 11. Three rules define the subdomain relation. Reflexivity is given by (S-DOMAIN REFL). The rule (S-DOMAIN LOOSE) states that a precise domain with a variable x as owner and c as kind is a subdomain of the loose domain $d.c$ if d is the domain of x. For example, the domain annotation x.`boundary` is a subdomain of `this.local.boundary` if x is typed with some type `this.local` $\_\langle\_\rangle$. If a domain d is a subdomain of domain d' then extending both domains by the same kind preserves the subdomain relation (S-DOMAIN EXT). Note that $<:_{d}$ is transitive (for the proof cf. [32]).

Beside reflexivity and transitivity, the subtyping relation is defined by two rules. Rule (S-TYPE CLASS) states that a type T is a subtype of type T' if the owning domains are the same and T' is the declared supertype of T', where the domain context parameters are substituted by the corresponding domain parameters of T. Rule (S-TYPE DOM) says that a type T is a subtype of type T' if the owning domain of T is a subdomain of the owning domain of T'. The other domain parameters must be the same.

(T-PROG)
$$\frac{\vdash \overline{L} \qquad \mathit{this} : \texttt{global}\ C\langle\bullet\rangle ; \bullet \vdash e : T}{\vdash (\overline{L}, C, e) : T}$$

(T-CLASS)
$$\frac{\begin{array}{c}\Delta = \overline{\alpha} \qquad T = \texttt{same}\ C\langle\overline{\alpha}\rangle \qquad \Gamma = \mathit{this} : T \qquad \Gamma ; \Delta \vdash \overline{M} \\ \varnothing ; \Delta \vdash \overline{T} \qquad [\overline{\beta} \subseteq \overline{\alpha}] \qquad [\mathit{override}(\_\ D\langle\overline{\beta}\rangle, \overline{M})] \qquad [\mathit{nohiding}(\_\ D\langle\overline{\beta}\rangle, \overline{f})]\end{array}}{\vdash \texttt{class}\ C\langle\overline{\alpha}\rangle\ [\texttt{extends}\ D\langle\overline{\beta}\rangle]\ \{\ \overline{T}\ \overline{f};\ \overline{M}\}}$$

(T-METHOD)
$$\frac{\Gamma' = \Gamma, \overline{x} : \overline{T} \qquad \Gamma' ; \Delta \vdash e : T_e \qquad \Gamma' ; \Delta \vdash T_e <: T_r \qquad \varnothing ; \Delta \vdash \overline{T}, T_r}{\Gamma ; \Delta \vdash T_r\ m(\overline{T}\ \overline{x})\{\ e\ \}}$$

(OVERRIDE)
$$\frac{M = T\ m(\overline{T}\ \_)\{\_\} \qquad \text{if } \mathit{method}(U, m) = T'\ m(\overline{T'}\ \_)\{\_\} \text{ then } T, \overline{T} = T', \overline{T'}}{\mathit{override}(U, M)}$$

(NOHIDING)
$$\frac{\mathit{fields}(U) = \_\ \overline{f} \qquad f \notin \overline{f}}{\mathit{nohiding}(U, f)}$$

Figure 12: Program, class, and method typing

Programs, Classes and Methods

Figure 12 shows the rules for program, class and method typing. A program $(\overline{L}, C, e)$ is typed by typing e in the type environment mapping *this* to `global` $C\langle\bullet\rangle$. We demand that class C has no domain context parameters. In addition, all class declarations must be well-typed. A class declaration is well-typed if all its method declarations are well-typed under the type environment mapping *this* to `same` $C\langle\overline{\alpha}\rangle$, and the types of its fields are well-typed in the empty type environment. This ensures that the domain annotations of fields cannot contain local variables. If the class inherits from another class, all domain parameters of the super type must be domain context parameters of the class itself. In addition, overridden methods must have the same type signature and fields of super classes may not be hidden. A method declaration is well-typed if its body expression is well-typed in the type environment containing *this* and the formal parameters of the method. We demand $\varnothing ; \Delta \vdash T_r, \overline{T}$ to ensure that the domain annotations of formal parameters and the result type do not contain local variables. Note that in principle it would be possible that domain annotations of formal parameters contain other formal parameters as owners, but this feature is omitted for simplicity.

$$\textsc{(t-var)}\quad \frac{\Gamma;\Delta \vdash \diamond}{\Gamma;\Delta \vdash x : \Gamma(x)} \qquad\qquad \textsc{(t-new)}\quad \frac{\Gamma;\Delta \vdash T}{\Gamma;\Delta \vdash \texttt{new}\ T : T}$$

$$\textsc{(t-field)}\quad \frac{\Gamma;\Delta \vdash e : T' \qquad T = \sigma(T', e)\cdot \mathit{fType}(T', f) \qquad \Gamma;\Delta \vdash T}{\Gamma;\Delta \vdash e.f : T}$$

$$\textsc{(t-fieldup)}\quad \frac{\begin{array}{cc}\Gamma;\Delta \vdash e.f : T_f & \mathit{assignable}(T_f)\\ \Gamma;\Delta \vdash e' : T & \Gamma;\Delta \vdash T <: T_f\end{array}}{\Gamma;\Delta \vdash e.f = e' : T} \qquad\qquad \textsc{(t-let)}\quad \frac{\begin{array}{cc}\Gamma;\Delta \vdash e : T & x \notin \mathit{dom}(\Gamma)\\ \Gamma, x : T;\Delta \vdash e' : T' & \Gamma;\Delta \vdash T'\end{array}}{\Gamma;\Delta \vdash \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' : T'}$$

$$\textsc{(t-invk)}\quad \frac{\begin{array}{c}\Gamma;\Delta \vdash e : T_e \qquad \mathit{method}(T_e, m) = T\ m(\overline{T'}\ \_)\ \ldots \qquad T_s, \overline{T'_s} = \sigma(T_e, e)\cdot(T, \overline{T'})\\ \Gamma;\Delta \vdash \overline{e} : \overline{U} \qquad \mathit{assignable}(\overline{T'_s}) \qquad \Gamma;\Delta \vdash \overline{U} <: \overline{T'_s} \qquad \Gamma;\Delta \vdash T_s\end{array}}{\Gamma;\Delta \vdash e.m(\overline{e}) : T_s}$$

Figure 13: Expression typing

Expressions

The expression type rules are shown in Figure 13. Much is standard, so we only explain the highlights of our system. In the (T-FIELD) and (T-INVK) rules, the types of the fields and methods are translated to the calling context by the σ function. This gives the keywords `this`, `owner`, and `same` a meaning in the current context. The rules (T-FIELDUP) and (T-INVK) both test whether the assigned types are assignable. This prevents illegal assignments of loose domains to domains that are loose in the current context, but are precise in the original context. For example, a field update $x.f = e$, where f is declared with an owning domain `owner.boundary`, x has owning domain `this.local.boundary`, and e has owning domain `this.local.boundary`, would be forbidden by rule (T-FIELD). This is because `owner` would be replaced with !`this.local` by the σ function, leading to a non-assignable domain.

4 DYNAMIC SEMANTICS

The dynamic entities of SLODJ are given in Figure 14. A value v is either *null* of an object represented by some object identifier o. A runtime domain is a tuple of a value v and a domain tail b, or `global`. An object state s is a tuple $(\tau, \overline{v})$, consisting of a runtime type τ and a list of field values $\overline{v}$. A store S is a finite mapping from objects o to object states s. A stack frame F is a finite mapping from variable names

$$
\begin{array}{lcll}
o & \in & \textbf{Object} & \\
v & \in & \textbf{Value} & ::= \; null \mid o \\
d & \in & \textbf{Domain} & ::= \; \ldots \mid \delta \\
a & \in & \textbf{DomOwner} & ::= \; \ldots \mid o \\
T & \in & \textbf{Type} & ::= \; \ldots \mid \tau \\
\delta & \in & \textbf{RuntimeDomain} & ::= \; o.b \mid \texttt{global} \\
\tau & \in & \textbf{RuntimeType} & ::= \; \delta\ C\langle\overline{\delta}\rangle \\
s & \in & \textbf{ObjectState} & ::= \; (\tau, \overline{v}) \\
S & \in & \textbf{Store} & ::= \; \overline{o} \mapsto \overline{s} \\
F & \in & \textbf{StackFrame} & ::= \; \overline{x} \mapsto \overline{v}
\end{array}
$$

Figure 14: Dynamic entities of SLODJ

x to values v.

Runtime Types and Runtime Domains

Domains at runtime are modeled as a tuple $o.b$ consisting of an owner object o and a domain tail b, which is a sequence of `boundary` and `local`. Like domain annotations, runtime domains can either be precise or loose. A precise runtime domain has either the form $o.c$ or is `global`, otherwise it is loose. We call types with runtime domains, runtime types.

The domain information at runtime is needed to prove the correctness of our type system. However, the evaluation rules do not depend on the domain information and hence in a real implementation it is not required to store the actual domains in the object state. Note that objects always belong to domain `global` or to runtime domains with objects as owners.

Auxiliary Functions

Figure 15 shows some auxiliary functions. To be able to compare static types with runtime types, we use a translation function *rtd*, which translates a static type into a corresponding runtime type. The function replaces local variables by their values and uses the receiver object to replace syntactic owners by values. In addition, it sets the subscripted value of the runtime type.

Subtype Relation on Runtime Types

Similar to the subtype relation on static types we define a subtype relation $\subset:$ on runtime types (see Figure 16). It uses the subdomain relation $\subset:_d$ to compare runtime domains. $S \vdash \delta \subset:_d \delta'$ states that runtime domain δ is equal to or included

$$\textsc{(t-null)}\ \frac{}{S \vdash null : \tau} \qquad \textsc{(t-obj)}\ \frac{S(o) = (\tau, \_)}{S \vdash o : \tau} \qquad \textsc{(rtd)}\ \frac{S \vdash v : \tau}{rtd_S(F, v, T) = \sigma(\tau, v) \cdot [ran(F)/dom(F)]T}$$

$$\textsc{(actd)}\ \frac{S \vdash v : \delta\ C\langle\overline{\delta}\rangle}{actd_S(v) = \delta} \qquad \textsc{(owner)}\ \frac{actd_S(v) = v'.c}{owner_S(v) = v'}$$

Figure 15: Auxiliary functions

$$\textsc{(sr-domain refl)}\ \frac{}{S \vdash \delta \subset:_d \delta} \qquad \textsc{(sr-domain loose)}\ \frac{S \vdash actd_S(o) \subset:_d o'.b'}{S \vdash o.b \subset:_d o'.b'.b} \qquad \textsc{(sr-type dom)}\ \frac{S \vdash \delta \subset:_d \delta'}{S \vdash \delta\ C\langle\overline{\delta}\rangle \subset: \delta'\ C\langle\overline{\delta}\rangle}$$

$$\textsc{(sr-type class)}\ \frac{\texttt{class}\ C\langle\overline{\alpha}\rangle\ \texttt{extends}\ D\langle\overline{\beta}\rangle\ \ldots}{S \vdash \delta\ C\langle\overline{\delta}\rangle \subset: \delta\ D\langle[\overline{\delta}/\overline{\alpha}]\overline{\beta}\rangle)} \qquad \textsc{(sr-type refl)}\ \frac{}{S \vdash \tau \subset: \tau} \qquad \textsc{(sr-type trans)}\ \frac{S \vdash \tau \subset: \tau' \qquad S \vdash \tau' \subset: \tau''}{S \vdash \tau \subset: \tau''}$$

Figure 16: Runtime subtyping

in runtime domain δ'. This follows the intuition that loose runtime domains can be regarded as sets of precise runtime domains.

Evaluation Rules

The evaluation rules are shown in Figure 17. We use a big-step natural semantics. Local variables are handled by a stack frame F, a store S models the state. The evaluation relation has the form

$$S, F \vdash e \Downarrow v, S'$$

meaning that under store S and stack frame F expression e evaluates to value v and new store S'.

The rules are more or less standard. As in other ownership type system the runtime domains play no role for the evaluation rules, which shows that they are only needed for the soundness proof. The only rule which requires some explanation is (r-new). It shows that an object is created by initializing all fields with *null*. The runtime type τ of the new object is determined by applying the *rtd* function to the static type T.

(R-VAR)
$$\frac{F(x) = v}{S, F \vdash x \Downarrow v, S}$$

(R-LET)
$$\frac{S, F \vdash e \Downarrow v, S' \qquad S', F[x \mapsto v] \vdash e' \Downarrow v', S''}{S, F \vdash \texttt{let}\ x = e\ \texttt{in}\ e' \Downarrow v', S''}$$

(R-FIELD)
$$\frac{S, F \vdash e \Downarrow o, S' \qquad S'(o) = (\tau, \overline{v})}{S, F \vdash e.f_i \Downarrow v_i, S'}$$

(R-FIELDUP)
$$\frac{\begin{array}{c} S, F \vdash e \Downarrow o, S' \\ S', F \vdash e' \Downarrow v, S'' \qquad S''(o) = (\tau, \overline{v}) \end{array}}{S, F \vdash e.f_i = e' \Downarrow v, S''[o \mapsto (\tau, [v/v_i]\overline{v})]}$$

(R-INVK)
$$\frac{\begin{array}{c} S, F \vdash e \Downarrow o, S' \qquad S', F \vdash e_1 \Downarrow v_1, S_1 \ \cdots \ S_{n-1}, F \vdash e_n \Downarrow v_n, S_n \\ type_{S'}(o) = \tau \qquad method(\tau, m) = \_\ m(\_\ \overline{x})\{e_b\} \qquad S_n, \{this \mapsto o, \overline{x} \mapsto \overline{v}\} \vdash e_b \Downarrow v, S'' \end{array}}{S, F \vdash e.m(\overline{e}) \Downarrow v, S''}$$

(R-NEW)
$$\frac{\begin{array}{c} fields(T) = \_\ \overline{f} \\ o \notin dom(S) \qquad \tau = rtd_S(F, F(this), T) \qquad S' = S[o \mapsto (\tau, \overline{null})] \qquad |\overline{null}| = |\overline{f}| \end{array}}{S, F \vdash \texttt{new}\ T \Downarrow o, S'}$$

Figure 17: SLODJ evaluation rules

5 PROPERTIES

We now present two important properties of our formalization, namely the *Subject Reduction Theorem* and the *Accessibility Theorem* that leads to the boundary-as-dominators property.

Type Soundness

In this section we present the Subject Reduction Theorem for SLODJ. We have to show that during the evaluation of a SLODJ program all values can only be of types that correspond to their declared static type. For precise formulation of the theorem, we need additional properties for stores, stack frames, and contexts (Figure 18):

$\vdash S$	Store S is well-formed
$S, \Gamma \vdash F$	Stack frame F is well-formed w.r.t. S and Γ
$T \vdash \Delta$	Context Δ is well-formed w.r.t. T

The judgment $\vdash S$ means that the types of field values of all objects in S correspond to the declared type of the objects' classes, and $S, \Gamma \vdash F$ means that the types of values of a stack frame F correspond to the types recorded in the type environment

$$\text{(T-STORE }\varnothing\text{)}\quad \frac{}{\vdash \varnothing}$$

$$\text{(T-STORE OBJECT)}\quad \frac{\begin{array}{c}\vdash S \qquad S' = S[o \mapsto (\tau, \overline{v})] \\ \overline{T}\ \overline{f} = \mathit{fields}(\tau) \qquad |\overline{v}| = |\overline{f}| \qquad S' \vdash \overline{v} : \overline{\tau} \qquad S' \vdash \overline{\tau} \subset: \mathit{rtd}_{S'}(\varnothing, o, \overline{T})\end{array}}{\vdash S'}$$

$$\text{(T-STACK }\varnothing\text{)}\quad \frac{}{S, \varnothing \vdash \varnothing}$$

$$\text{(T-STACK VAR)}\quad \frac{\begin{array}{c}S, \Gamma \vdash F \qquad S \vdash v : \tau \\ S \vdash \tau \subset: \mathit{rtd}_S(\Gamma, v, T)\end{array}}{S, \Gamma[x \mapsto T] \vdash F[x \mapsto v]}$$

$$\text{(T-CONTEXT)}\quad \frac{\Delta = \overline{d} \qquad \Gamma; \Delta \vdash d \rightarrow^s \overline{d}}{\Gamma; d\ C\langle \overline{d} \rangle \vdash \Delta}$$

Figure 18: Store and stack frame well-formedness

Γ. $T \vdash \Delta$ states that Δ is equal to the list of domain parameters of T, and that the owning domain of T can access all domains in Δ.

The Subject Reduction Theorem states that if an expression e is typed to T and e evaluates to value v, then the runtime type τ of v is a subtype of the runtime representation of T. The theorem also states that the store stays well-formed under the evaluation of e. This is needed by the proof to have a stronger induction hypothesis.

Theorem 1 (Subject Reduction)**.** *If* $\Gamma(\mathit{this}) \vdash \Delta$ *and* $\Gamma; \Delta \vdash e : T$ *and* $S, F \vdash e \Downarrow v, S'$ *and* $\vdash S$ *and* $S, \Gamma \vdash F$ *and* $S' \vdash v : \tau$*, then*

1. $S' \vdash \tau \subset: \mathit{rtd}_{S'}(F, F(\mathit{this}), T)$*, and*
2. $\vdash S'$

Proof. The proof is by structural induction on the reduction rules of the operational semantics and a case analysis for every rule. It uses a main lemma that relates the static subtyping relation with the runtime subtype relation. The crucial cases for the first equation are (R-FIELD), (R-INVK), and (F-FIELDUP). The crucial cases for the well-formedness of the store are (R-FIELDUP) and (R-NEW). The other parts of the proof are by straightforward application of the induction hypothesis. □

Encapsulation Guarantees

To show the encapsulation property of our type system, we first define which accesses are allowed at runtime and then show that our type system guarantees that during the execution of a well-typed program only such accesses can happen.

The accessibility rules in Figure 20 define which domains are accessible by an object at runtime. They are of the form $o \longrightarrow_S \delta$, read "Under store S, object o can access the runtime domain δ". They use the transitive ownership relation shown in Figure 19. These rules give a formal definition of the access properties we described

$$\text{(OWN-DIRECT)}\quad \frac{owner_S(o) = o'}{o \prec_S o'} \qquad \text{(OWN-TRANS)}\quad \frac{o \prec_S o' \qquad o' \prec_S o''}{o \prec_S o''}$$

Figure 19: Ownership relation

$$\text{(A-OWN)}\quad \frac{}{o \longrightarrow_S o.c} \qquad \text{(A-BOUNDARY)}\quad \frac{o \longrightarrow_S actd_S(o')}{o \longrightarrow_S o'.\texttt{boundary}} \qquad \text{(A-OWNER)}\quad \frac{o \prec_S o}{o \longrightarrow_S o.c} \qquad \text{(A-GLOBAL)}\quad \frac{}{o \longrightarrow_S \texttt{global}}$$

Figure 20: Accessibility rules

on Page 3. They state that an object o can access a domain δ if and only if one of the following holds.

- o is the owner of δ (A-OWN)
- δ is the boundary domain of an object o', and o can access the domain that o' belongs to (A-BOUNDARY)
- δ is owned by a transitive owner of o (A-OWNER)
- δ is the global domain (A-GLOBAL)

We write $o \longrightarrow_S v$ as an abbreviation for $o \longrightarrow_S actd_S(v)$. Note that these rules guarantee that an object o can access the local domain of an object o' if and only if $o = o'$, or o' is a transitive owner of o. Thus, it is guaranteed that local objects of an object o can only be accessed by o itself or by objects owned by o.

Similar to the Subject Reduction Theorem we need to define some properties on stores and on stack frames. These are given in Figure 21. All objects of a store must have access to the values of their fields (A-STORE $*$), and all values of a stack frame must be accessible by the *this*-object (A-STACKFRAME $*$).

The Accessibility Theorem states that if an expression e is evaluated to v, and e is well-typed by the type system, then the current receiver object can access v. In addition, all objects of the new store S' can access their field values.

Theorem 2 (Accessibility). *If* $\Gamma(this) \vdash \Delta$ *and* $\Gamma; \Delta \vdash e : T$ *and* $\vdash S$ *and* $\Gamma, S \vdash F$ *and* $\Vdash S$ *and* $S \Vdash F$ *and* $S \vdash e \Downarrow v, S'$ *then*

$$F(this) \longrightarrow_{S'} v \;\wedge\; \Vdash S'$$

$$\textsc{(a-store } \varnothing\textsc{)} \quad \frac{}{\Vdash \varnothing} \qquad \textsc{(a-store object)} \quad \frac{\Vdash S \qquad S' = S[o \mapsto (\_, \overline{v})] \qquad o \longrightarrow_{S'} \overline{v}}{\Vdash S'}$$

$$\textsc{(a-stackframe this)} \quad \frac{}{S \Vdash \{\mathit{this} \mapsto o\}} \qquad \textsc{(a-stackframe var)} \quad \frac{S \Vdash F \qquad F' = F[x \mapsto v] \qquad F'(\mathit{this}) \longrightarrow_S v}{S \Vdash F'}$$

Figure 21: Store and stack frame accessibility

Proof. The proof is by induction on the rules of the operational semantics and a case analysis on each rule. The proof is mainly done by using the results from Theorem 1 and by using the static accessibility relation to show that the accessibility rules at runtime are established. □

Note that this theorem enforces the boundary-as-dominator property described on Page 3. Local objects can only be accessed by the owner object, or objects owned by the owner object. That is, in order to access local objects from the outside, the access path must go through the owner object, or through boundary objects.

6 DISCUSSION AND RELATED WORK

Our work belongs to the category of mechanisms for alias prevention [21] in general, and uses the ownership types idea in particular.

Ownership type systems

The first systems encapsulating objects were proposed by Hogg with Islands [20] and by Almeida with Balloons [4]. *Ownership Types* are a static way to guarantee encapsulation of objects during runtime. The notion of ownership types stems from Clarke [13] to formalize the core of Flexible Alias Protection [29]. Ever since, many researchers investigated ownership type systems [11, 27, 9, 3]. Ownership type systems have been used to prevent data-races [6], deadlocks [7, 5], and to allow the modular specification and verification of object-oriented programs [26, 15]. Lately, ownership types have been combined with type genericity [31].

All the mentioned ownership type systems have one thing in common: They have problems with multi-access ownership contexts. In particular, they cannot handle the iterator problem properly. It turns out that the central property of ownership type systems prevents a solution: the *owners-as-dominators* property. Recently several more or less powerful solutions have been proposed. The first allows the creation of dynamic aliases to owned objects [12], that is, aliases stored in the stack.

The second approach [11, 7] is to allow Java's inner member classes [19] to access the representation objects of their parent objects. Both solutions do not provide the full power to generalize the owners-as-dominators property.

Recently, Lu and Potter [24] presented a type system which separates object ownership and accessibility. Instead of only giving the owner of a type, types are also annotated by their possible accessibility. This results in a system that is in many respects more flexible than ours. However, there are as well programming patterns that are handled better in our system (see below). The iterator pattern can be solved in their system by giving the iterator the accessibility of the owner of the list. For example, the type of the iterator within the `List` class could be `[owner] Iterator<this>`, which specifies that the iterator is owned by the `List` object, but can be accessed by the owner of the list (`owner` is the owner parameter of the `List` class). If the list is typed by the client as `[world] List<this>`, then the iterator would be typed in that context as `[this] Iterator<*>`, where `*` stands for an unknown owner. This allows the client to access the iterator, but the owner of the iterator is completely abstract. This is an important difference to our system, as we do not lose the owner information of the iterator. This allows us to give boundary objects back to their owner, which is not possible in the system of Lu and Potter.

An earlier system by Lu and Potter [25] considers the encapsulation of effects instead of objects. This allows, for example, to access internal representation objects from the outside, but disallows their direct modification. This mechanism is similar to the read-only mechanism of the Universes approach [27], where it is allowed to have read-only references to representation objects. However, this approach forbids programming patterns where boundary objects should be able to directly change the state of representation objects without using the owner object.

Ownership Domains

The basic idea of ownership domains stems from Clarke [11] with ownership contexts. Objects are not directly owned by other objects, but instead are owned by *contexts*. Contexts in turn are owned by objects. While Clarke's formalization was based on the Object Calculus [1], Aldrich and Chambers [2] applied this idea to a subset of Java and extended it with several features. A programmer has the possibility to declare an arbitrary number of domains per object and can define which domains can access which other domains by *link* declarations. So in parts the OD approach is more flexible than our approach, thus it is no surprise that our system can be partly encoded in OD [32].

The iterator problem is solved by OD with so-called *public* domains, which can always be accessed if the owner object can be accessed. However, in OD a public domain must always by attached to a `final` field or variable to unambiguously identify the owner object. This restricts the usage of public domains as the owner object must always be known to the client in order to access its public domain. Our approach solves this by introducing loose domains which can be declared without a `final` field or variable. OD has been combined with an effects system [33]. A more general version of OD has been formalized in System F [23].

Other Related Work

All work that addresses aliasing in object-oriented programming, especially which statically guarantee the absence of aliasing, is related to our work. Confined Types [34] ensure the encapsulation of objects within the boundary of a package. Other approaches enforce the uniqueness of references to prevent aliases [35, 10].

Limitations

As our system is purely static, the domain of an object cannot be changed after its creation. Thus ownership transfer is not possible in SLOD. A variant of ownership transfer, the initialization problem [14], is also not solvable in SLOD, but could be tackled with unique variables [10], for example.

A loose domain in SLOD cannot be turned back into a precise domain. Such a cast needs runtime information, which would introduce a non-negligible space overhead in practice, as every object needs its domain recorded. However, the space overhead might be minimized by only storing runtime information that is really needed to support such casts, similar to the approach by Boyapati et al. [8].

7 CONCLUSION AND FUTURE WORK

Simple Loose Ownership Domains (SLOD) simplifies Ownership Domains by omitting link and domain declarations, but keeping the idea of public and private domains. Hence, we maintain most of the expressiveness of Ownership Domains, while significantly reducing the syntactical overhead. Besides this, SLOD supports so-called *loose* domains, which allow to abstract from precise domains. This enables, for instance, the implementation of model-view systems with an arbitrary number of listener callbacks, which is not possible with standard Ownership Domains. Our system is sound and guarantees a property we call *boundary-as-dominator*, which is a generalization of owners-as-dominators.

We are currently inspecting existing libraries and programs to measure the practicability of our approach. In addition, we are working on the inference of domain annotations in order to reduce the annotation effort. After we did a theoretical work on the inference of domain annotations [30], we are now working on a checking and inference tool for a practical subset of Java. Another interesting aspect is to use domain information at runtime, in order to reduce the annotation effort and to allow casts from loose domains to precise domains. Finally, we are investigating how the encapsulation boundaries of SLOD can be used to give thread-safety guarantees.

REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *Proc. ECOOP 2004*, volume 3086 of *LNCS*, pages 1–25. Springer, June 2004.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proc. OOPSLA 2002*, pages 311–330. ACM Press, Nov. 2002.

[4] P. S. Almeida. Balloon Types: Controlling sharing of state in data types. In *Proc. ECOOP'97*, volume 1241 of *LNCS*, pages 32–59. Springer, June 1998.

[5] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, Feb. 2004.

[6] C. Boyapati and M. Rinard. A parameterized type system for race-free java programs. In *Proc. OOPSLA 2001*, pages 56–69. ACM Press, Oct. 2001.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proc. OOPSLA 2002*, pages 211–230. ACM Press, Nov. 2002.

[8] C. Boyapati, R. Lee, and M. Rinard. Safe runtime downcasts with ownership types. Technical Report TR-853, MIT Laboratory for Computer Science, June 2002.

[9] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proc. POPL '03*, pages 213–223. ACM Press, Jan. 2003.

[10] J. Boyland. Alias burying: Unique variables without destructive reads. *Software – Practice and Experience*, 31(6):533–553, May 2001.

[11] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, July 2001.

[12] D. Clarke and S. Drossopoulou. Ownership, encapsulation, and the disjointness of type and effect. In *Proc. OOPSLA 2002*, pages 292–310. ACM Press, Nov. 2002.

[13] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proc. OOPSLA '98*, pages 48–64. ACM Press, Oct. 1998.

[14] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Research Report 156, Digital Systems Research Center, July 1998. SRC-RR-156.

[15] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[16] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *Foundations and Developments of Object-Oriented Languages (FOOL/WOOD '07)*, Jan. 2007.

[17] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. *Formal Syntax and Semantics of Java*, 1523:241–269, 1999.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[19] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification – Second Edition*. Addison-Wesley, June 2000.

[20] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Proc. OOP-SLA '91*, pages 271–285. ACM Press, Nov. 1991.

[21] J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Messenger*, 3(2):11–16, 1992. ISSN 1055-6400.

[22] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, May 2001.

[23] N. Krishnaswami and J. Aldrich. Permission-based ownership: Encapsulating state in higher-order typed languages. In *Proc. PLDI'05*, pages 96–106. ACM Press, June 2005.

[24] Y. Lu and J. Potter. On ownership and accessibility. In D. Thomas, editor, *Proc. ECOOP 2006*, volume 4067 of *LNCS*, pages 99–123. Springer, July 2006.

[25] Y. Lu and J. Potter. Protecting representation with effect encapsulation. In *In Proc. POPL '06*, pages 359–371. ACM Press, 2006.

[26] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.

[27] P. Müller and A. Poetzsch-Heffter. A type system for controlling representation exposure in Java. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269–5, Fernuniversität Hagen, 2000.

[28] J. Noble. Iterators and encapsulation. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 431, St. Malo, France, June 2000. IEEE Computer Society. ISBN 0-7695-0731-X.

[29] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *Proc. ECOOP'98*, volume 1445 of *LNCS*, pages 158–185. Springer, July 1998.

[30] A. Poetzsch-Heffter, K. Geilmann, and J. Schäfer. Infering ownership types for encapsulated object-oriented program components. In *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*. Springer, 2007. to appear.

[31] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic java. In *Proc. OOPSLA 2006*. ACM Press, 2006.

[32] J. Schäfer and A. Poetzsch-Heffter. Simple loose ownership domains - TR. Technical Report 348/06, Department of Computer Science, University of Kaiserslautern, Germany, P.O. Box 3049, 67653 Kaiserslautern, Germany, Mar. 2006. Available at `http://kluedo.ub.uni-kl.de/volltexte/2006/1941/`.

[33] M. Smith. Towards an effects system for ownership domains. In *ECOOP Workshop - FTfJP 2005*, July 2005.

[34] J. Vitek and B. Bokowski. Confined types in Java. *Software – Practice and Experience*, 31(6):507–532, 2001.

[35] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Working Conference on Programming Concepts and Methods (PROCOMET)*, pages 347–359. North-Holland, Apr. 1990.

ABOUT THE AUTHORS

Jan Schäfer is a PhD student and research assistant at the Software Technology Group at the TU Kaiserslautern, Germany. His research is supported by the Deutsche Forschungsgemeinschaft (German Research Foundation). Contact him at jschaefer@informatik.uni-kl.de.
See also http://softech.informatik.uni-kl.de/˜janschaefer.

Arnd Poetzsch-Heffter is a professor at the Computer Science Department of the TU Kaiserslautern, Germany. He is heading the Software Technology Group. He can be reached at poetzsch@informatik.uni-kl.de. See also http://softech.informatik.uni-kl.de/.