Simple Loose Ownership Domains

Jan Schäfer* and Arnd Poetzsch-Heffter**

University of Kaiserslautern, Germany
{ jschaefer | poetzsch }@informatik.uni-kl.de

Abstract. Ownership Domains generalize ownership types. They support programming patterns like 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 developed so-called *loose* domains which abstract over several *precise* domains. That is, similar to the relation between supertypes and subtypes we have a relation between loose and precise domains. In addition, we simplified ownership domains by reducing the number of domains per object to two and hard-wiring the access permissions between domains. We formalized the resulting type system for an OO core language and proved 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 [25, 13, 23, 12, 9] support the encapsulation of objects and guarantee that encapsulated objects can only be accessed from the outside by going through their owner. This property is called owners-as-dominators. Unfortunately, this property prevents important programming patterns like the efficient implementation of iterators [24]. 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

* Supported by the Deutsche Forschungsgemeinschaft (German Research Foundation).
** Partially supported by the Rheinland-Pfalz cluster of excellence "Dependable Adaptive Systems and Mathematical Modelling" (DASMOD).

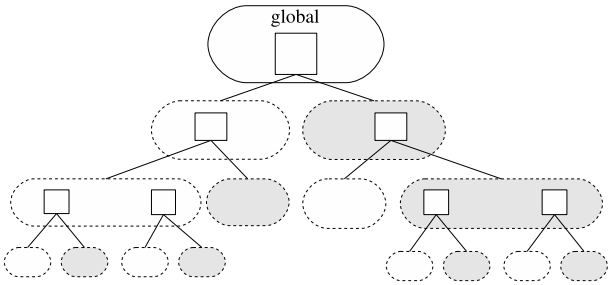

Fig. 1: The ownership and containment relation of objects and domains form a tree rooted by a global domain. Solid rectangles represent objects, dashed rounded rectangles represent domains, where gray rectangles are local domains and white ones are boundary domains. An edge from an object X to a domain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

the 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 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 formalize the type system for loose ownership domains. We define the dynamic semantics in Section 4 and state the central properties of the system in Section 5. The proofs are contained in [28]. 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. Within this paper, we simplify the ownership domain approach of [2] in two ways: Every object owns exactly two domains, namely a *local* domain and a *boundary* domain. Thus, SLOD has no domain declarations. In addition, access permissions between domains are hard-wired, so SLOD needs no *link* declarations.

2.1 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. 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.

2.2 Domain Annotations

To statically check the boundary-as-dominators property, types in SLOD are extended by domain annotations. Figure 2 shows the complete syntax for the

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

Fig. 2: Syntax of domain annotations in SLOD.

annotations. Types together with domain annotations are called domain types. Like ordinary types, domain types 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 types. 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. To make objects inaccessible from the outside, like for example `Node` objects of the list, they are placed into the local domain of the owner. Hence, the `head` field of `LinkedList` is annotated with `local` which is an abbreviation of `this.local`. As can be seen in method `add`, this domain type is established when `Node` objects are created.

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 `boundary Iter` instance (again, `boundary` abbreviates `this.boundary`). Within the class `Iter`, `Node` objects have domain type `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 [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 `data` field illustrates the use of a domain type 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. All domain annotations with domains as owner parts are loose. For

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

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

```
public class Node<T> {
  T data;
  same Node<T> next;
  Node(T data, same Node<T> n) {
    this.data = data;
    this.next = next;
  }
}


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

Fig. 3: A linked list with iterators.

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 parameter `T` that represents the domain type of the stored data. Thus, `T` is not only a place holder for the ordinary type of the data, but also for its domain. In the example, the `Main` class instantiates that parameter with `local Object`.

2.3 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 in [2], which increases the

```
interface Listener {
   public void update(int data);
}
class View {
  local State state;
  boundary Listener listener() {
    return new boundary
        ViewListener(state);
  }
}
class Model<L extends Listener>
{
  local List<L> listeners;
  void addListener(L listener) {
    listeners.add(listener);
  }
  void notifyAll(int data) {
    for (L l : listeners) {
       l.update(data);
    }
  }
}
```

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

class Main {
  ...
  local
    Model<local.boundary Listener> m;
  m = new local
    Model<local.boundary Listener>();
  local View view = new local View();
  m.addListener(view.listener());
  view = new local View();
  m.addListener(view.listener());
  ...
}
```

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

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 expressivity of loose domains, we use a slightly modified version of an example given in [2] (see Figure 4). It is a model-view system. `Model` objects allow to register `Listener` objects. When an event happens at the model, the model notifies all registered listener objects by calling the method `update(int)`. `View` objects have a state that is updated whenever one of its listeners is notified. Method `listener()` creates new `ViewListener` instances as boundary objects of their view. The example is a simplified version of the observer pattern [16] and represents a category of similar implementations.

Loose ownership domains allow to register more than one `Listener` object at a `Model` object. In the example, the type parameter of the `Model` object in class `Main` is instantiated with the loose domain `local.boundary`. The calls of `m.addListener(view.listener())` are allowed, because the result domain of `view.listener()` 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 in [2], this solution is not possible, because 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 `Listener` 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.

```
// ...
local A a = new local A();
local.boundary A b = a.b;
local A a2 = new local A();
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) [20] and ClassicJava [15], and is also inspired by several flavors of these type systems which already incorporate ownership information [13, 12, 2, 27].

Some features of SLOD are left out in SLODJ. These are parameterization of classes with domain annotations, the *global* domain and `final` fields as owners of domain annotations. It is straightforward to extend our formalization with these features, as all these concepts have already been formalized by other ownership type systems. We plan to incorporate them in the future.

3.1 Syntax

The abstract syntax of SLODJ is shown in Figure 5. We use similar notations as FJ [20]. 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 $\bullet$.

A SLODJ program consists of a list of class declarations, a class name, and an expression. A class declarations consists of a class name, a super class, a sequence of field declarations, and a sequence of method declarations. Note that classes have no constructors; objects are created with all fields initialized to *null*. One consequence is that the `new` expression takes a class name 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 a domain annotation

$$
\begin{array}{rll}
P \in \textbf{Program} & ::= & \langle \overline{L}, C, e \rangle \\
L \in \textbf{ClassDecl} & ::= & \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}\ \overline{f};\ \overline{M}\ \} \\
M \in \textbf{MethDecl} & ::= & T\ m\ (\ \overline{T}\ \overline{x}\ )\{\ e\ \} \\
T, U \in \textbf{Type} & ::= & d\ C \\
e \in \textbf{Expression} & ::= & \texttt{new}\ d\ C \mid x \mid e.f \mid e_1.f = e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid e.m(\ \overline{e}\ ) \\
d \in \textbf{Domain} & ::= & a.b \\
a \in \textbf{DomOwner} & ::= & x \mid \texttt{this} \mid \texttt{owner} \\
b \in \textbf{DomTail} & ::= & c \mid b.c \\
c \in \textbf{DomKind} & ::= & \texttt{local} \mid \texttt{boundary} \mid \texttt{same} \\
f \in \textbf{FieldName} & & x, y \in \textbf{Variable} \\
m \in \textbf{MethName} & & C, D \in \textbf{ClassName}
\end{array}
$$

Fig. 5: SLODJ Syntax.

and a class name. `let` expressions bind variables and are similar to `final` variable declarations in Java. We support field updates to get a more realistic model of Java.

Domain annotations in SLODJ are similar to those in SLOD. The only differences are that in SLODJ fields cannot be owners of domains, and that `same` is only a domain kind, instead of a full domain annotation. `same` in SLOD is equal to `owner.same` in SLODJ. For a domain annotation $d = d_1.d_2 \ldots d_n$ the function *front* returns the domain annotation without the last element: $\mathit{front}(d) = d_1.d_2 \ldots d_{n-1}$, *last* returns the last element: $\mathit{last}(d) = d_n$, and *first* returns the first element: $\mathit{first}(d) = d_1$.

3.2 Auxiliary Functions

To capture information from class declarations we need some auxiliary functions. The detailed definitions can be found in the companion report [28], but they are essentially equal to those of FJ [20]. $\mathit{fields}(C)$ returns the list of field declarations of class C; $\mathit{mtype}(m, C)$ returns the signature $\overline{T} \to T$ of method m of class C; $\mathit{mbody}(m, C)$ obtains the names of the formal parameters together with the body expression $(\overline{x}.e)$ of method m of class C; $\mathit{isPrecise}(d)$ is true iff d is a precise domain, i.e. has the form $a.c$.

3.3 Type System

The type rules of SLODJ are shown in Figure 7 and 8. We use the judgments shown in Figure 6. The environment Γ is a finite mapping from variables to types.

For any program $\langle \overline{L}, C, e \rangle$, we assume an implicitly given fixed class table CT mapping class names to their definitions. All judgements are implicitly parameterized with that class table. The class table is assumed to satisfy the following conditions (taken nearly verbatim from FJ):

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash T$	T is a well-formed type in Γ
$\Gamma \vdash d$	d is a well-formed domain in Γ
$\vdash C$	C is a well-defined class name.
$\Gamma \vdash T <: U$	T is a subtype of U in Γ
$\Gamma \vdash e : T$	e is a well-formed expression of type T in Γ
$C \vdash M$	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

Fig. 6: Judgments for the type system of SLODJ.

- $\overline{L} = ran(CT)$;
- $\forall C \in dom(CT).\ CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}\ \overline{f};\ \overline{M}\}$
- $\texttt{Object} \notin dom(CT)$;
- For every class name C except `Object`, appearing anywhere in CT, we assume $C \in dom(CT)$;
- There are no cycles in the subtype relation induced by CT, i.e. the relation $<:_c$ is antisymmetric.

Environments, Types, Class Names, and Domains. Beside the standard rules for well-formed environments, types and class names, there are three rules for well-formed domain annotations. A domain annotation is well-formed iff it is either a precise domain annotation, and the owner is `this` or `owner`, or the owner is a variable x with a well-typed type T and the kind is `boundary`, or its owner part is well-formed and its last element is `boundary`. Note that `same` and `local` can only appear in precise domains annotation with `this` or `owner` as owner part. The well-formedness of domain annotations is important as it guarantees the encapsulation property of our type system.

Subclassing, Subdomaining, and Subtyping. The relation $<:_c$ is the reflexive, transitive closure of the direct subclass relation given by the class declarations. The relation $<:_d$ is defined on domain annotations. Reflexivity is given by (S-DOMAIN REFL). The rule (S-DOMAIN VAR) states that a domain with a variable as owner, $x.b$, is a subdomain of a domain $d_0.b$ if x is typed with a domain d_x, and that domain is a subdomain of d_0. For example, the domain annotation x.`boundary` is a subdomain of `this.local.boundary` iff x is typed with domain annotation `this.local`. $<:_d$ is transitive (for the proof see cf. [28]). The subtype relation $<:$ is defined by the relations $<:_c$ and $<:_d$. It is reflexive and transitive like $<:_c$ and $<:_d$.

Methods, Classes and Programs. A method declaration is well-formed if its body expression is well-typed in the type environment containing *this* and

Environments, Types, Class Names, and Domains

$$\text{(T-ENV } \varnothing\text{)} \quad \frac{}{\varnothing \vdash \diamond} \qquad \text{(T-ENV x)} \quad \frac{\Gamma \vdash T \qquad x \notin dom(\Gamma)}{\Gamma[x \mapsto T] \vdash \diamond} \qquad \text{(T-TYPE)} \quad \frac{\Gamma \vdash \diamond \qquad \Gamma \vdash d \qquad \vdash C}{\Gamma \vdash d\ C}$$

$$\text{(T-CLASS OBJ)} \quad \frac{}{\vdash \texttt{Object}} \qquad \text{(T-CLASS DECL)} \quad \frac{C \in dom(CT)}{\vdash C}$$

$$\text{(T-DOMAIN THIS OWNER)} \quad \frac{a \in \{\texttt{this}, \texttt{owner}\}}{\Gamma \vdash a.c} \qquad \text{(T-DOMAIN VAR)} \quad \frac{\Gamma \vdash x : T}{\Gamma \vdash x.\texttt{boundary}} \qquad \text{(T-DOMAIN BOUNDARY)} \quad \frac{\Gamma \vdash a.b}{\Gamma \vdash a.b.\texttt{boundary}}$$

Subclassing, Subdomaining, and Subtyping

$$\text{(S-CLASS REFL)} \quad \frac{\vdash C}{\Gamma \vdash C <:_c C} \qquad \text{(S-CLASS TRANS)} \quad \frac{\Gamma \vdash C <:_c D \qquad \Gamma \vdash D <:_c E}{\Gamma \vdash C <:_c E} \qquad \text{(S-CLASS DECL)} \quad \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\}}{\Gamma \vdash C <:_c D}$$

$$\text{(S-DOMAIN REFL)} \quad \frac{\Gamma \vdash d}{\Gamma \vdash d <:_d d} \qquad \text{(S-DOMAIN VAR)} \quad \frac{\Gamma \vdash x.b \qquad \Gamma \vdash d_0.b \qquad \Gamma \vdash x : d_x\ C \qquad \Gamma \vdash d_x <:_d d_0}{\Gamma \vdash x.b <:_d d_0.b}$$

$$\text{(S-TYPE)} \quad \frac{\Gamma \vdash d_1\ C \qquad \Gamma \vdash d_2\ D \qquad \Gamma \vdash d_1 <:_d d_2 \qquad \Gamma \vdash C <:_c D}{\Gamma \vdash d_1\ C <: d_2\ D}$$

Methods, Classes, and Programs

$$\text{(T-METHODDECL)} \quad \frac{\begin{array}{c}\Gamma = \{\ this \mapsto \texttt{owner.same}\ C, \overline{x} \mapsto \overline{T}\ \} \\ this \notin \overline{x} \qquad \Gamma \vdash e : T_e \qquad \Gamma \vdash T_e <: T_r \qquad \varnothing \vdash \overline{T} \qquad \varnothing \vdash T_r \\ \texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\} \qquad \text{if } mtype(m, D) = \overline{U} \to U_r, \text{ then } \overline{T} = \overline{U} \text{ and } T_r = U_r\end{array}}{C \vdash T_r\ m(\overline{T}\ \overline{x})\{\ e\ \}}$$

$$\text{(T-CLASSDECL)} \quad \frac{C \vdash \overline{M} \qquad \varnothing \vdash \overline{T}}{\vdash \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}\ \overline{f};\ \overline{M}\}} \qquad \text{(T-PROG)} \quad \frac{\vdash \overline{L} \qquad \vdash C \qquad \{this \mapsto \texttt{owner.same}\ C\} \vdash e : T}{\vdash \langle \overline{L}, C, e \rangle : T}$$

Fig. 7: Various Typing Rules

$$\text{(T-VAR)}\quad \frac{\Gamma \vdash \diamond \qquad \Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(T-NEW)}\quad \frac{\Gamma \vdash a.c\ C}{\Gamma \vdash \texttt{new}\ a.c\ C : a.c\ C}$$

$$\text{(T-FIELD)}\quad \frac{\Gamma \vdash e : d\ C \qquad \mathit{fields}(C) = \overline{d}\ \overline{C}\ \overline{f} \qquad T_f = \sigma(e, d, d_i)\ C_i \qquad \Gamma \vdash T_f}{\Gamma \vdash e.f_i : T_f}$$

$$\text{(T-FIELDUP)}\quad \frac{\Gamma \vdash e_0 : d\ C \qquad \mathit{fields}(C) = \overline{d}\ \overline{C}\ \overline{f} \qquad T_f = \sigma(e_0, d, d_i)\ C_i \qquad \Gamma \vdash e_1 : T \qquad \Gamma \vdash T <: T_f \qquad \texttt{owner} = \mathit{first}(d_i) \Rightarrow \mathit{isPrecise}(d)}{\Gamma \vdash e_0.f_i = e_1 : T}$$

$$\text{(T-LET)}\quad \frac{\Gamma \vdash e_0 : T_0 \qquad x \notin \mathit{dom}(\Gamma) \qquad \Gamma[x \mapsto T_0] \vdash e_1 : T_1 \qquad \Gamma \vdash T_1}{\Gamma \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : T_1}$$

$$\text{(T-INVK)}\quad \frac{\begin{array}{c}\Gamma \vdash e : d\ C \\ \mathit{mtype}(m, C) = \overline{d}\ \overline{C} \to d_u\ C_u \qquad \Gamma \vdash \overline{e} : \overline{U} \qquad \overline{T'} = \sigma(e, d, \overline{d})\ \overline{C} \qquad \Gamma \vdash \overline{U} <: \overline{T'} \\ (\exists i.\ \texttt{owner} = \mathit{first}(d_i)) \Rightarrow \mathit{isPrecise}(d) \qquad U_m = \sigma(e, d, d_u)\ C_u \qquad \Gamma \vdash U_m\end{array}}{\Gamma \vdash e.m(\overline{e}) : U_m}$$

Fig. 8: Expression typing

the formal parameters of the method. We demand $\varnothing \vdash \overline{T}$ and $\varnothing \vdash T_r$ 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. A class declaration is well-formed if all its method declarations are well-formed, and the types of its fields are well-formed in the empty type environment. Note that this ensures that the domain annotations of fields cannot contain local variables. A program $\langle \overline{L}, C, e \rangle$ is typed by typing e in the type environment mapping *this* to `owner.same` C.

Expressions. The expression type rules are shown in Figure 8. Much is standard, so we only explain the highlights of our system.

To translate domain annotations of fields and methods to the calling context we use the function σ.

$$\sigma(e, d_e, d) = [e/\texttt{this}, \mathit{front}(d_e)/\texttt{owner}, \mathit{last}(d_e)/\texttt{same}]\ d$$

Beside the domain d that is translated, σ takes the receiver expression e and its domain d_e as parameters. The substitution replaces `this` by e, `owner` by the *front* of d_e and `same` by the last part of d_e. The typing rules ensure that domain annotations substituted by σ are always well-formed, so ill-formed domain

$$
\begin{array}{rll}
o \in \textbf{Object} & & \\
v \in \textbf{Value} & ::= o \mid \mathit{null} & \\
rd \in \textbf{RuntimeDomain} & ::= v.b & \\
s \in \textbf{ObjectState} & ::= \langle o.b, C, \overline{v} \rangle & \\
S \in \textbf{Store} & ::= \{o \mapsto s\} & \\
F \in \textbf{StackFrame} & ::= \{x \mapsto v\} &
\end{array}
$$

Fig. 9: Dynamic Entities of SLODJ.

annotations with `this` replaced by an arbitrary expression e that is not a local variable, are not accepted by the type system.

Both, the rules (T-FIELDUP) and (T-INVK) have a clause

$$\texttt{owner} = \mathit{first}(d_i) \Rightarrow \mathit{isPrecise}(d)$$

which ensures that if the domain d_i begins with the `owner` keyword, the domain d of the receiver expression is precise. The reason behind this is that the function σ replaces `owner` with the owner part of d. If d is not precise, the owner part is a domain again. If d_i, for example, is `owner.boundary`, and d is a loose domain, then σ would turn d_i into a loose domain, too. This would allow assignments that are not possible in the original context, in which d_i is precise. That is the reason why we have to forbid these cases.

4 Dynamic Semantics

The dynamic entities of SLODJ are given in Figure 9. A value v is either an object o or *null*. A runtime domain is a tuple of a value v and a domain tail b. An object state s is a triple $\langle o.b, C, \overline{v} \rangle$, consisting of a precise runtime domain $o.c$ with an object o as owner, a class name C, 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 x to values v.

4.1 Runtime Domains

Domains at runtime are modeled as a tuple $v.b$ consisting of the owner value v and the 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 the form $v.c$, otherwise it is loose.

In every object state the precise runtime domain of the domain that the object belongs to is stored. This is needed to prove the correctness of our type system. However, it is not needed by the evaluation rules, and hence a real implementation need not store the actual domain in the object state. Note that objects always belong to runtime domains with objects as owners. We need *null* as owners for runtime domains only to give *null* an owning domain.

$$\text{(R-VAR)}\quad \frac{F(x) = v}{S, F \vdash x \Rightarrow v, S} \qquad \text{(R-LET)}\quad \frac{S_0, F \vdash e_0 \Rightarrow v_0, S_1 \qquad S_1, F[x \mapsto v_0] \vdash e_1 \Rightarrow v_1, S_2}{S_0, F \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 \Rightarrow v_1, S_2}$$

$$\text{(R-FIELD)}\quad \frac{S_0, F \vdash e \Rightarrow o, S_1 \quad S_1(o) = \langle rd, C, \overline{v} \rangle}{S_0, F \vdash e.f_i \Rightarrow v_i, S_1} \qquad \text{(R-FIELDUP)}\quad \frac{S_0, F \vdash e_0 \Rightarrow o, S_1 \qquad S_1, F \vdash e_1 \Rightarrow v, S_2 \quad S_2(o) = \langle rd, C, \overline{v} \rangle \qquad S_3 = S_2[o \mapsto \langle rd, C, [v/v_i]\overline{v} \rangle]}{S_0, F \vdash e_0.f_i = e_1 \Rightarrow v, S_3}$$

$$\text{(R-INVK)}\quad \frac{\begin{array}{c} S_0, F \vdash e \Rightarrow o, S_1 \\ S_1, F \vdash e_1 \Rightarrow v_1, S_2 \ \cdots\ S_n, F \vdash e_n \Rightarrow v_n, S_{n+1} \qquad S_1(o) = \langle \ldots, C, \ldots \rangle \\ mbody(m, C) = \overline{x}.e_b \qquad S_{n+1}, \{this \mapsto o, \overline{x} \mapsto \overline{v}\} \vdash e_b \Rightarrow v, S_{n+2} \end{array}}{S_0, F \vdash e.m(\overline{e}) \Rightarrow v, S_{n+2}}$$

$$\text{(R-NEW)}\quad \frac{\begin{array}{c} rd = rtd(S_0, F, F(this), d) \\ fields(C) = \overline{T}\ \overline{f} \qquad o \notin dom(S_0) \qquad S_1 = S_0[o \mapsto \langle rd, C, \overline{null} \rangle] \qquad |\overline{null}| = |\overline{f}| \end{array}}{S_0, F \vdash \texttt{new}\ d\ C \Rightarrow o, S_1}$$

Fig. 10: SLODJ Evaluation Rules.

4.2 Evaluation Rules

The evaluation rules are shown in Figure 10. 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

$$S_1, F \vdash e \Rightarrow v, S_2$$

meaning that under store S_1 and stack frame F expression e evaluates to value v and new store S_2.

The rules are more or less standard. The most interesting fact is that 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 domain *rd* of the new object is determined by applying the *rtd* function to the domain annotation *d*.

4.3 Auxiliary Functions

We need some auxiliary functions which are shown in Figure 11. The function $actd(S, o)$ obtains the actual owning domain of object o in S, it returns *null*.`local` for *null*. The function *owner* returns the owner part of the actual domain of a value, and $class(S, o)$ gives the class of o in S.

$$\frac{}{actd(S, null) = null.\texttt{local}}\ (\text{ACTD NULL}) \qquad \frac{S(o_1) = \langle o_2.c, \ldots \rangle}{actd(S, o_1) = o_2.c}\ (\text{ACTD OBJECT}) \qquad \frac{actd(S, v_1) = v_2.c}{owner(S, v_1) = v_2}\ (\text{OWNER})$$

$$\frac{S(o) = \langle \ldots, C, \ldots \rangle}{class(S, o) = C}\ (\text{CLASS}) \qquad \frac{}{S \vdash null.c \subseteq v.b}\ (\text{SUBSET NULL}) \qquad \frac{}{S \vdash o.b \subseteq o.b}\ (\text{SUBSET REFL})$$

$$\frac{S \vdash actd(S, o_1) \subseteq o_2.b_2}{S \vdash o_1.b_1 \subseteq o_2.b_2.b_1}\ (\text{SUBSET LOOSE}) \qquad \frac{a \neq x \qquad actd(S, v_1) = v_2.c}{rtd(S, F, v_1, a.b) = [v_1/\texttt{this}, v_2/\texttt{owner}, c/\texttt{same}]\ a.b}\ (\text{RTD THISOWNER})$$

$$\frac{F(x) = v_2 \qquad actd(S, v_1) = v_3.c}{rtd(S, F, v_1, x.b) = [c/\texttt{same}]\ v_2.b}\ (\text{RTD VAR})$$

Fig. 11: Auxiliary Functions.

4.4 Domain Subset Relation

Similar to the subdomain relation on domain annotation, we define an inclusion relation on runtime domains. $S \vdash rd_1 \subseteq rd_2$ states that runtime domain rd_1 is equal to or included in runtime domain rd_2. This follows the intuition that loose runtime domains can be regarded as sets of precise runtime domains.

4.5 The *rtd* Function

To be able to compare static domain annotations with runtime domains, we use a semantic interpretation function *rtd*, which translates a domain annotation into a meaningful corresponding runtime domain. The function replaces syntactic owners of a domain by values and `same` with an appropriate kind. The third parameter v of the function is interpreted as the current receiver object. Note that the function also handles *null*, which can be seen as a special kind of receiver.

5 Properties

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

5.1 Type Soundness

In this section we prove the soundness of the type system of SLODJ. We have to show that during the evaluation of a SLODJ program all values can only be of

$$\frac{}{\vdash \varnothing}\ (\textsc{t-store}\ \varnothing)$$

$$\frac{\vdash S_0 \qquad \vdash C \qquad S_1 = S_0[o \mapsto \langle \ldots, C, \overline{v} \rangle] \qquad fields(C) = \overline{d}\ \overline{C}\ \overline{f} \qquad |\overline{v}| = |\overline{f}| \qquad \forall v_i \in \overline{v}.\ v_i = null \vee S_1 \vdash actd(S_1, v_i) \subseteq rtd(S_1, \varnothing, o, d_i) \wedge class(S_1, v_i) <:_c C_i}{\vdash S_1}\ (\textsc{t-store object})$$

$$\frac{}{S, \varnothing \vdash \varnothing}\ (\textsc{t-stack}\ \varnothing) \qquad \frac{S, \Gamma \vdash F}{S, \Gamma[x \mapsto T] \vdash F[x \mapsto null]}\ (\textsc{t-stack null}) \qquad \frac{S, \Gamma \vdash F \qquad class(S, o) <:_c C \qquad S \vdash actd(S, o) \subseteq rtd(S, F, F(this), d)}{S, \Gamma[x \mapsto d\ C] \vdash F[x \mapsto o]}\ (\textsc{t-stack var})$$

Fig. 12: Store and Stack Frame Well-Formedness

a type that corresponds to their declared static type. A type in SLODJ consists of two parts: a class name and a domain annotation. So we have to show that the class of an object is a subtype of the statically declared class, and that the runtime domain of an object corresponds to the static domain annotation. The first part is easy as we can directly use the subclass relation $<:_c$. However, we can not directly compare a runtime domain with a domain annotation. We first have to translate the domain annotation to a meaningful corresponding runtime domain, which is done by the *rtd* function. After the translation we check that the resulting runtime domain is a superset of the runtime domain of the object. Note that in the case where the value is *null* we do not have to check anything.

For precise formulation of the theorem, we need additional properties for stores and stack frames (Fig. 12):

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

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 Γ.

The Subject Reduction Theorem states that if an expression e is typed to $d\ C$ and e evaluates to value v, then v is either *null*, or it is an object, and the actual class of v is a subclass of C, and the actual domain of v is included in the runtime representation of d. 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* $this \in dom(F)$ *and* $\Gamma \vdash e : d\ C$ *and* $S_0, F \vdash e \Rightarrow v, S_1$ *and* $\vdash S_0$ *and* $S_0, \Gamma \vdash F$ *then*

1. $v = null \vee S_1 \vdash actd(S_1, v) \subseteq rtd(S_1, F, F(this), d), \ \wedge \ class(S_1, v) <:_c C$ *and*
2. $\vdash S_1$

Proof. The proof is by structural induction on the reduction rules of the operational semantics. It can be found in [28].

5.2 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.

$$\frac{}{S \Vdash o \longrightarrow o.c}\ (\text{A-OWN}) \qquad \frac{S \Vdash o_1 \longrightarrow actd(S, o_2)}{S \Vdash o_1 \longrightarrow o_2.\texttt{boundary}}\ (\text{A-BOUNDARY}) \qquad \frac{owner(S, o_1) = o_2}{S \Vdash o_1 \longrightarrow o_2.c}\ (\text{A-OWNER}) \qquad \frac{}{S \Vdash o \longrightarrow null.c}\ (\text{A-NULL})$$

Fig. 13: Accessibility Rules.

The accessibility rules in Figure 13 define which domains are accessible by an object at runtime. They are of the form $S \Vdash o \longrightarrow v.c$, read "Under store S, object o can access the runtime domain $v.c$". These rules define that an object o can access a domain d iff

- o is the owner of d (A-OWN)
- d is the boundary domain of an object o_2, and o can access the domain that o_2 belongs to (A-BOUNDARY)
- d is a domain of the owner of o (A-OWNER)
- The owner of d is *null* (A-NULL)

We write $S \Vdash o \longrightarrow v$ to mean $S \Vdash o \longrightarrow actd(S, v)$. Note that these rules guarantee that an object o_1 can only access the local domain of an object o_2 if and only if $o_1 = o_2$, or o_2 is the owner of o_1. 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 14. 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_1 can access their field values.

$$\frac{}{\Vdash \varnothing}\ (\text{A-STORE } \varnothing) \qquad \frac{\Vdash S_0 \qquad S_1 = S_0[o \mapsto \langle \ldots, \overline{v} \rangle] \qquad \forall v_i \in \overline{v}.\ S_1 \Vdash o \longrightarrow v_i}{\Vdash S_1}\ (\text{A-STORE OBJECT})$$

$$\frac{}{S \Vdash \{this \mapsto o\}}\ (\text{A-STACKFRAME THIS}) \qquad \frac{S \Vdash F_0 \qquad F_1 = F_0[x \mapsto v] \qquad S \Vdash F_1(this) \longrightarrow v}{S \Vdash F_1}\ (\text{A-STACKFRAME VAR})$$

Fig. 14: Store and Stack Frame Accessibility

Theorem 2 (Accessibility). *If* $this \in dom(F)$ *and* $\Gamma \vdash e : T$ *and* $\vdash S_0$ *and* $\Gamma, S_0 \vdash F$ *and* $\Vdash S_0$ *and* $S_0 \Vdash F$ *and* $S_0 \vdash e \Rightarrow v, S_1$ *then*

$$S_1 \Vdash F(this) \longrightarrow v \ \wedge\ \Vdash S_1$$

Note that this theorem enforces the boundary-as-dominator property, as objects of the outside of an object o cannot directly access the inside of o, and so have to use objects of the boundary of o.

6 Discussion and Related Work

Our work belongs to the category of mechanisms for alias prevention [19] in general, and uses the ownership types idea in particular.

Ownership type systems. The first systems encapsulating objects were proposed by Hogg with Islands [18] and by Almeida with Balloons [4]. The notion of ownership types stems from Clarke [13] as a formalization the core of Flexible Alias Protection [25]. Ever since, many researchers investigated ownership type systems [11, 23, 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 [22]. Lately, ownership types have been combined with type genericity [27].

All ownership type systems have one thing in common: They cannot handle the iterator problem properly. It turns out that it is an inherent property of ownership type systems that prevents a solution: the *owners-as-dominators* property. Two solutions have been proposed to solve the iterator problem: The first is to allow the creation of dynamic aliases to owned objects [12], the second [11, 7] is to allow Java's inner member classes [17] to access the representation objects of their parent objects. Both solutions are unsatisfactory and break the owners-as-dominators property, showing that a more general solution is needed. With the boundary-as-dominator property, we tackle exactly this problem.

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 by 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 [28].

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 have been combined with an effects system [29]. A more general version of OD has been formalized in System F [21].

Other related work. All work that addresses aliasing in object-oriented programming, especially which statically guarantee the absence of aliasing, is somehow related to our work. One example, beside many others, are Confined Types [30].

Limitations. Like all type systems, SLOD is not complete. That is, it is possible to write programs that are correct at runtime but are refused by our type system.

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 not negligible space overhead in practice, as for every object its domain has to be 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 in [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 plan to extend the formalization of SLOD with domain parameters, and are investigating additional extensions like read-only and unique annotations, as well as immutable objects. We are currently inspecting existing libraries and programs to measure the practicability of our approach. 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. In addition, we will investigate how the encapsulation boundaries of SLOD can be used to give thread-safeness guarantees. Finally, we plan to implement a checking tool for a practical subset of Java.

References

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

[2] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *Proc. ECOOP'04*, volume 3086 of *LNCS*, pages 1–25. Springer-Verlag, June 2004.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In OOPSLA'02 [26], pages 311–330.

[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-Verlag, 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. pages 56–69, Oct. 2001.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In OOPSLA'02 [26], pages 211–230.

[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 OOPSLA'02 [26], pages 292–310.

[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] 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.

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

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

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

[19] 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.

[20] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS '01*, 23(3):396–450, May 2001.

[21] N. Krishnaswami and J. Aldrich. Permission-based ownership: encapsulating state in higher-order typed languages. In *Proc. PLDI'05*, volume 40, pages 96–106. ACM Press, June 2005.

[22] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[23] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programmiersprachen und Grundlagen der Programmierung, Kolloquiumsband '99*, Informatik Berichte 263–1. Fernuniversität Hagen, 2000.

[24] 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.

[25] 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-Verlag, July 1998.

[26] OOPSLA'02. *Proc. OOPSLA'02*, Nov. 2002. ACM Press.

[27] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic java. In *Proc. OOPSLA'06*. ACM Press, 2006.

[28] J. Schäfer and A. Poetzsch-Heffter. Simple loose ownership domains. Technical Report 348/06, Department of Computer Science, University of Kaiserslautern, Germany, Mar. 2006. Available at `http://softech.informatik.uni-kl.de/~janschaefer`.

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

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