Simple Loose Ownership Domains

Jan Schäfer* and Arnd Poetzsch-Heffter**

Technical Report
No. 348/06

March 31, 2006

Software Technology Group
Department of Computer Science
University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
{jschaefer | poetzsch}@informatik.uni-kl.de
http://softech.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 \textit{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 \textit{loose} domains which abstract over several \textit{precise} domains. That is, similar to the relation between super- 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 \cite{20,10,18,9,6} 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 \cite{19}. 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 \cite{1} is an advancement of 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 \textit{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,
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 $d$.

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 $Y$.

To realize a linked list with an iterator, for example, one can create a nodes domain and a public iters domain and link the iters domain to the nodes domain. Now all objects that can access the linked list can also access the objects of the iters domain, and iterators can access all objects of the nodes domain.

In Ownership Domains, 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 ownership domains 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 ownership domain 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 give an informal introduction into Simple Loose Ownership Domains. We show examples which cannot be handled by ordinary Ownership Domains, but can be handled by our system. In Section 3 we give a formal type system for a subset of Java and prove its correctness in Section 4 by providing a dynamic semantics. In Section 5 we formally define the encapsulation properties that are guaranteed by our system. Section 6 discusses our approach together with related work. We conclude and give a view to the future in Section 7.
2 Simple Loose Ownership Domains

The basic idea of Simple Loose Ownership Domains (SLOD) is the same as that of Ownership Domains [1]: 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 [1] 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 Ownership Domains the boundary domain is a public domain. The ownership relation of objects and domains form 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 whether an object \(X\) can access an object \(Y\). This accessibility relation is the smallest relation satisfying the following conditions:

- \(Y\) 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 cannot 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 \(X\), so that the boundary of \(X\) only contains \(X\).

2.2 Domains 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 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 \texttt{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 ownership domains, as in the approach of [1], 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 the ownership domain approach in [1].

Our approach additionally provides the possibility to use loose domain annotations. All domain annotations with domains as owner parts are loose. 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.

All 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.
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 = n;
    }
}

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.

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 [1], which increases the flexibility of our system, without losing 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 [1] (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 [12] 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 in the loose domain local.boundary, and the model object belongs to the local domain of the Main object. In the ownership type system in [1], this solution is not possible, because the parameter of the Model class had to be
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.

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 soundness in our system, we have to restrict the accessible interface of a type that is annotated with a loose domain annotation (a loose type). On a loose type it is not allowed to assign to a field which has a domain annotation that is same or contains owner, and it is not allowed to invoke a method which has a formal parameter with a domain annotation that is same or contains owner. We have to forbid these cases, because the precise owner of a loose domain is not known statically. To make this more clear we give an example in Figure 5. The assignment b1.b = b2 has to be forbidden, even though the type of b1.b is equal to that of b2, because the domains at runtime can be different.

2.4 Type Parameters

It is possible to parameterize classes, interfaces, or methods with type parameters. A type parameter does not only represent the type but also its domain. We use the term domain parameter to refer to the domain of a type parameter. Domain parameters directly opens a question: Are they loose or precise? This depends on how the domain parameters are instantiated. But this cannot be known locally. To enable modular checking, we assume that domain parameters are in general loose, but can be declared to be precise. The type checker ensures that precise domain parameters can only be instantiated with a precise domain. An exclamation mark declares a domain parameter to be precise:

class C<P!> extends D { ... }

6
class A {
    boundary B b;
    A() { b = new boundary B(); }
}
class B { same B b = new same B(); }
class C {
    void fail() {
        local A a1 = new local A(); local A a2 = new local A();
        local boundary B b1 = a1.b; local boundary B b2 = a2.b;
        b1.b = b2; // Forbidden
    }
}

Fig. 5: Example code that shows a forbidden field assignment on a loose type.

Note that in the most cases a loose type parameter will be sufficient. Only if there is a
field assignment or method invocation on a type parameter where the domain of the
field or the domain of a method parameter is same or contains the owner keyword, it
is required to have a precise domain.

3 Formalization of SLOD

In this section we present a formalization of the core of SLOD. We call the language
Simple Loose Ownership Domain Java (SLODJ). For simplicity we only consider a
subset of Java and the core features of SLOD.

The formalization is based on several existing formal type systems for Java, namely
Featherweight Java (FJ) [15] and CLASSICJAVA [11] and is also inspired by several
flavors of these type systems which already incorporate ownership information [10, 9,
1, 22].

Like other Java formalizations, we only consider the core feature of the full Java
language [13]. The difference to FJ are that we omit cast expressions and constructors,
but we include field updates and a let expression to define local variables. Hence in
contrast to FJ, SLODJ is not a functional language. Like CLASSICJAVA, objects are
created by initializing all fields with null.

We also omit some features of our domain extension. These are the parameter-
ization of classes with domain annotations, the global domain and final fields as
owners of domain annotations. We believe that 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 6. We use similar notations as FJ
[15]. A bar indicates a sequence: $L = L_1, L_2, \ldots, L_n$, where the length is defined as
$|L| = n$. Similar, $T \langle f \rangle$ is equal to $T_1 f_1; T_2 f_2; \ldots; T_n f_n$. If there is some sequence $\pi$, we write $x_i$ for any element of $\pi$. The empty sequence is donated by $\bullet$.

We use the meta variables $P$ to range over programs; $L$ to range over class declara-
tions; $M$ to range over method declarations; $T$ and $U$ to range over types; $e$ to
range over expressions; $d$ to range over domain annotations; $a$ to range over the first
$P \in \text{Program} ::= (\mathcal{L}, C, e)$

$L \in \text{ClassDecl} ::= \text{class } C \text{ extends } D \{ \mathcal{T}, \mathcal{M} \}$

$M \in \text{MethDecl} ::= \mathcal{T} \ m \ (\mathcal{T}) \{ \ e \}$

$T, U \in \text{Type} ::= d \ C$

$e \in \text{Expression} ::= \text{new } d \ C \mid x \mid e.f \mid e_1.f = e_2 \mid$

\hspace{1cm} \text{let } x = e_1 \text{ in } e_2 \mid e.m(\pi)$

$d \in \text{Domain} ::= a.b$

$a \in \text{DomOwner} ::= x \mid this \mid owner$

$b \in \text{DomTail} ::= c \mid b.c$

$c \in \text{DomKind} ::= \text{local} \mid \text{boundary} \mid \text{same}$

$f \in \text{FieldName}$

$x, y \in \text{Variable}$

$m \in \text{MethName}$

$C, D \in \text{ClassName}$

**Fig. 6: SLODJ Syntax.**

element of domain annotations; $b$ to range over the remainder of domain annotations; $c$ to range over domain kinds; $f$ to range over field names; $x$ and $y$ to range over variables; $m$ to range over method names; and $C$ and $D$ to range over class names. Meta variables can be primed or subscripted. So, e.g. $e, e_0, e_1, e', e''$ stand for expressions.

A SLODJ program $P$ is a triple $(\mathcal{L}, C, e)$ of a list of class declarations $\mathcal{L}$, a class name $C$, which must exist in $\mathcal{L}$, and a single expression $e$. A program is executed by creating an instance of $C$ and evaluating $e$ with this referencing the initial instance of $C$. A class declaration $L$ consists of a class name $C$, a super class $D$, a sequence of field declarations $\mathcal{T}$, and a sequence of method declarations $\mathcal{M}$. In SLODJ every class declaration must have a super class, which can be $\text{Object}$. Note that classes have no constructors. Objects are created with all fields initialized to $\text{null}$. A method declaration $M$ consists of a result type $T$, a sequence of formal parameters $\mathcal{T}$, and a single body expression $e$. In SLODJ all methods have a result type, which must be a super type of the type of the body expression $e$. A type $T$ in SLODJ consists of a domain annotation $d$ and a class name $C$. In SLODJ every type must have a domain annotation, there is no default domain annotation. There are also no primitive types like boolean or integer. An expression in SLODJ can either be a new expression, a local variable $x$, a field access $e.f$, a field update $e_1.f = e_2$, a let expression $\text{let } x = e_1 \text{ in } e_2$, or a method invocation $e.m(\pi)$. We support field updates to get a more realistic model of Java. We need a let expression, as in certain situations field updates and method invocations are only allowed on local variables as will be seen later. As we are only reasoning about aliasing SLODJ has no conditional expressions or loops. Nevertheless, like FJ, SLODJ is computational complete.

**Domain Annotations.** As domain annotations in SLODJ are somewhat different from other ownership type systems, we explain them in more detail.

Domain annotations are of the form $a.b$ and consist of an owner part $a$, which is the first element and a sequence of local, boundary and same keywords $b$, representing the kind of the domain. The owner part can either be this, owner or a local variable. The domain annotation this.boundary, for example, represents the boundary domain
Field Lookup:

(FIELDS OBJECT) \[ \text{fields(Object)} = \bullet \]

(FIELDS NORMAL) \[ \begin{align*}
\text{class } C \text{ extends } D & \{ T \not\in; M \} \\
\text{fields}(D) & = U \not\in, T \not\in \\
\text{fields}(C) & = U \not\in, T \not\in
\end{align*} \]

Method type lookup:

(M-TYPE DECL) \[ \begin{align*}
\text{class } C \text{ extends } D & \{ T \not\in; M \} \\
U & m(\overline{U}, \pi) \{ e \} \in M \\
\text{mtype}(m, C) & = U \rightarrow U
\end{align*} \]

(M-TYPE INHERIT) \[ \begin{align*}
\text{class } C \text{ extends } D & \{ T \not\in; M \} \\
m & \notin M \\
\text{mtype}(m, C) & = \text{mtype}(m, D)
\end{align*} \]

Method body lookup:

(M-BODY DECL) \[ \begin{align*}
\text{class } C \text{ extends } D & \{ T \not\in; M \} \\
U & m(\overline{U}, \pi) \{ e \} \in M \\
\text{mbody}(m, C) & = \pi. e
\end{align*} \]

(M-BODY INHERIT) \[ \begin{align*}
\text{class } C \text{ extends } D & \{ T \not\in; M \} \\
m & \notin M \\
\text{mbody}(m, C) & = \text{mbody}(m, D)
\end{align*} \]

Precise domain annotations:

(ISPRECISE) \[ \text{isPrecise}(a.c) \]

Fig. 7: Auxiliary Functions.

owned by the this-object\[1\] As this can also appear in a field declaration, the this-object is in that case the receiver object of the field access.

Note that the meaning of same in SLODJ is different from that of SLOD, as same stands only for the kind of the domain to which the current object belongs. This simplifies the formal system, as there are less syntactically possible domain annotations. owner.same in SLODJ is identical to same in SLOD.

For a domain annotation \(d = d_1.d_2\ldots.d_n\) the function \(\text{front}\) returns the domain annotation without the last element: \(\text{front}(d) = d_1.d_2\ldots.d_{n-1}\), \(\text{last}\) returns the last element: \(\text{last}(d) = d_n\), and \(\text{first}\) returns the first element: \(\text{first}(d) = d_1\).

We distinguish between precise and loose domain annotations. Precise domain annotations exactly represent a single domain, loose domain annotations represent sets of domains. Precise domain annotations consists of exactly two elements. All other domain annotations are loose. E.g. the domain annotation this.local.boundary represents all boundary domains of objects that are in the this.local domain. The concept of loose domain annotations is a unique feature of our type system, as other ownership type systems only have precise ownership information. This increases the flexibility of our system, as it is often not necessary to know the exact domain of an object.

Domain annotations represent sets of possible domains at runtime. So it is possible to define a subdomain relation on domain annotations that resemble the subset relation on domains. For example, the domain annotation x.boundary is a subdomain

\[1\] We call the object to which the this variable points the this-object.
of this.local.boundary iff $x$ is typed with domain annotation this.local, because x.boundary represents the boundary domain of an object $o$ that $x$ points to, which is in the local domain of the this-object, and this.local.boundary represents the set of all boundary domains of all local domains, which certainly includes the boundary domain of $o$. The subdomain relation is defined together with the subclass relation and the subtype relation in Figure 10.

**Auxiliary Functions.** Figure 7 shows some auxiliary functions. Except the isPrecise function, they are taken verbatim from FJ [15].

The fields function gives the fields of a class by looking into the class declaration and adding the fields of its super class. Object has no fields. The mtype function looks up the signature $U \rightarrow U$ of a class method. $U$ are the types of the formal parameters and $U$ is the result type of the method. If a method $m$ does not exist in the method declarations $M$ ($m \notin M$), it is searched in the super class. If there are no method declarations for a certain method, the function is undefined. In particular, $\text{mtype}(m, \text{Object})$ is always undefined. The mbody function is similar to the mtype function, but returns the method body $\mathfrak{T}.e$ of a method, where $\mathfrak{T}$ are the names of the formal parameters of the method, and $e$ is the body expression. Like mtype, mbody is undefined for methods that do not exist. The isPrecise function checks whether or not a domain annotation is precise, which is only true for domain annotations with exactly two elements.

### 3.2 Type System

The type rules of SLODJ are shown in Figure 9, Figure 10, Figure 12 and Figure 11. We use the judgments listed in Figure 8 where the environment $\Gamma$ is a finite mapping from variables to types.

\[
\begin{align*}
\Gamma \vdash \cdot & \quad \text{$\Gamma$ is a well-formed environment} \\
\Gamma \vdash T & \quad T \text{ is a well-formed type in } \Gamma \\
\Gamma \vdash d & \quad d \text{ is a well-formed domain in } \Gamma \\
\vdash C & \quad C \text{ is a well-defined class name.} \\
\Gamma \vdash T <: U & \quad T \text{ is a subtype of } U \text{ in } \Gamma \\
\Gamma \vdash e : T & \quad e \text{ is a well-formed expression of type } T \text{ in } \Gamma \\
C \vdash M & \quad M \text{ is a well-formed method declaration in class } C \\
\vdash L & \quad L \text{ is a well-formed class declaration} \\
\vdash P : T & \quad P \text{ is a well-formed program of type } T
\end{align*}
\]

**Fig. 8:** Judgments for the type system of SLODJ.

For any program $(L, C, e)$, we assume an implicitly given fixed class table $CT$ mapping class names to their definitions. The class table is assumed to satisfy the following conditions (taken nearly verbatim from FJ):

- $L = \text{ran}(CT)$
- $\forall C \in \text{dom}(CT), CT(C) = \text{class } C \text{ extends } D \{ T \; \mathfrak{T}; \; M \}$
- Object $\notin \text{dom}(CT)$
- For every class name $C$ except Object, appearing anywhere in $CT$, we assume $C \in \text{dom}(CT)$. 

10
Well-formed Environments and Types:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T-ENV ∅)</td>
<td>$∅ ⊢ □$</td>
</tr>
<tr>
<td>(T-ENV X)</td>
<td>$Γ ⊢ T$, $x \notin \text{dom}(Γ)$ $Γ[x \mapsto T] ⊢ □$</td>
</tr>
<tr>
<td>(T-TYPE)</td>
<td>$Γ ⊢ o$, $Γ ⊢ d ⊢ C$ $Γ ⊢ d C$</td>
</tr>
</tbody>
</table>

Well-formed Class Names:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T-CLASS OBJECT)</td>
<td>$Γ \vdash \text{Object}$</td>
</tr>
<tr>
<td>(T-CLASS DECL)</td>
<td>$C \in \text{dom}(CT)$ $Γ \vdash C$</td>
</tr>
</tbody>
</table>

Well-formed Domains:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T-DOMAIN THIS OWNER)</td>
<td>$a \in {\text{this, owner}}$ $Γ \vdash a.c$</td>
</tr>
<tr>
<td>(T-DOMAIN VAR)</td>
<td>$Γ \vdash x : T$ $Γ \vdash x.\text{boundary}$</td>
</tr>
<tr>
<td>(T-DOMAIN BOUNDARY)</td>
<td>$Γ \vdash a.b$ $Γ \vdash a.b.\text{boundary}$</td>
</tr>
</tbody>
</table>

Fig. 9: Well-formed Environments, Types, Class Names and Domains.

- There are no cycles in the subtype relation induced by $CT$, i.e. the relation $\prec_c$ is antisymmetric.

Environments, Types, Class Names and Domains. In Figure 9 are rules to ensure the well-formedness of type environments, types, class names and domains. A type environment $Γ$ is well-formed if it is either empty, or a well-formed environment that is extended by a new variable mapping to a well-formed type. A type is well-formed if the domain annotation and the class name is well-defined. A class name is well-defined if it is either Object, or it is contained in the domain of the class table $CT$.

A domain annotation is well-formed if it is either a precise domain annotation where the owner is this or owner, or the owner is a variable $x$ with a well-typed type $T$ and the kind is boundary. Finally, a domain annotation is well-formed if 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.

Subtyping. The subtyping rules are shown in Figure 10. The relation $\prec_c$ is the reflexive, transitive closure of the direct subclass relation given by the class declarations. The relation $\prec_d$ 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 $d_0$. Note that $\prec_d$ is transitive, which we prove later. The subtype relation $\prec$ is defined by the relations $\prec_c$ and $\prec_d$. It is reflexive and transitive.
Subclassing:

\[
\begin{align*}
\text{(s-class refl)} & \quad \Gamma \vdash C \\
\text{(s-class trans)} & \quad \Gamma \vdash C <_{c} D \\
\text{(s-class decl)} & \quad \text{class } C \text{ extends } D \{ \ldots \}
\end{align*}
\]

\[\Gamma \vdash C <_{c} E \]

\[\Gamma \vdash C <_{c} D\]

Subdomaining and Subtyping:

\[
\begin{align*}
\text{(s-domain refl)} & \quad \Gamma \vdash d \\
\text{(s-domain var)} & \quad \Gamma \vdash x.b \quad \Gamma \vdash d_{0}.b \\
\text{(s-type)} & \quad \Gamma \vdash d_{1} C \quad \Gamma \vdash d_{2} D \\
\end{align*}
\]

\[\Gamma \vdash d \prec_{d} d \]

\[\Gamma \vdash x.b \prec_{d} d_{0}.b \]

\[\Gamma \vdash d_{1} \prec_{d} d_{2} \quad \Gamma \vdash C \prec_{c} D\]

Fig. 10: Subclassing, Subdomaining, and Subtyping

Substitution \(\sigma\). To translate domain annotations of fields and methods to the calling context we use the function \(\sigma\).

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

Beside the domain \(d\) that is adapted, \(\sigma\) takes the receiver expression \(e\) and its domain \(d_e\) as parameters. The substitution replaces \textbf{this} by \(e\), \textbf{owner} by the \textit{front} of \(d_e\) and \textbf{same} by the last part of \(d_e\). The typing rules ensure that domain annotations substituted by \(\sigma\) are always well-formed, so ill-formed domain annotations with \textbf{this} replaced by an arbitrary expression \(e\) that is not a local variable, are not accepted by the type system.

Expressions. The expression typing rules are shown in Figure 12.

\(\text{(t-var)}\). The type of a local variable \(x\) is the type to which \(x\) is mapped in the type environment \(\Gamma\).

\(\text{(t-field)}\). The type of a field access \(e.f_i\) is the type of field \(f_i\) of the class of \(e\). The domain annotations of field \(f_i\) are substituted by the context information.

\(\text{(t-fieldup)}\). To be well typed, a field update expression \(e_0.f_i = e_1\) has to follow a number of constraints. The usual constraint is that the type of \(e_1\) must be a subtype of the type \(T_i\) of field \(f_i\). SLODJ has some additional constraints concerning domain annotations. If \textbf{this} appears in \(d_i\), the receiver expression \(e_0\) has to be a local variable. This is enforced by \(\Gamma \vdash T <_{c} T_f\) which implies \(\Gamma \vdash T_f\), which ensures a well-formed domain annotation. Note that this restriction also exists in the Ownership Domains formalization, where it is implicitly demanded by the syntax. The second restriction is the following. If \textbf{owner} appears in \(d_i\), the domain annotation \(d_i\) has to be precise. This is important as otherwise it would be possible to assign an expression with a loose domain to a field with a precise domain.

\(\text{(t-invk)}\). A method invocation has the usual restrictions that the types of the actual parameters must be subtypes of the formal parameters. In addition we demand similar restriction as for the field update expression.

\(\text{(t-let)}\). The type of a let expression \textbf{let} \(x = e_0\) \textbf{in} \(e_1\) is the type of expression \(e_1\) in type environment \(\Gamma\) extended by variable \(x\) mapping to the type of \(e_0\). A reassigning of \textbf{this} is prevented by rule \(\text{(t-env x)}\). Note that we also require \(\Gamma \vdash T_1\), which is important, as otherwise it would be possible that \(T_1\) could contain a domain annotation with the local variable \(x\) as owner, which is not valid in \(\Gamma\).
Well-formed Method Declarations:

\[(T\text{-}\text{methoddecl})\]
\[
\begin{align*}
\Gamma &= \{ \text{this} \mapsto \text{owner}\text{.same } C, \emptyset \mapsto T \} \\
\text{this} \notin \emptyset &\quad \Gamma \vdash e : T_e \quad \Gamma \vdash T_e <; T_r \quad \emptyset \vdash T_r \\
\text{class } C \text{ extends } D \{ \ldots \} &\quad \text{if } \text{mtype}(m, D) = U \rightarrow U_r, \text{ then } T = U \text{ and } T_r = U_r \\
C &\vdash T_r \text{ } m(T\rightarrow\Gamma\{e\})
\end{align*}
\]

Well-formed Class Declarations and Program Typing:

\[(T\text{-}\text{classdecl})\]
\[(T\text{-}\text{prog})\]
\[
\begin{align*}
C &\vdash M \quad \emptyset &\vdash T \\
\vdash \text{class } C \text{ extends } D \{ T_f; M \} \\
C &\vdash T_r m(T\rightarrow\{T_f, M\})
\end{align*}
\]

\textbf{Fig. 11: Method, Class and Program Typing Rules.}

Expression Typing:

\[(T\text{-}\text{var})\]
\[(T\text{-}\text{field})\]
\[(T\text{-}\text{fieldup})\]
\[(T\text{-}\text{invk})\]
\[
\begin{align*}
\Gamma &\vdash \varnothing \quad \Gamma \vdash e : d C \quad \text{fields}(C) = \overline{d} \overline{f} \\
\Gamma(\overline{x}) &= T & \Gamma \vdash e : d_i C_i \quad T_f = \sigma(e, d, d_i) C_i \quad \Gamma \vdash T_f \\
\Gamma \vdash x : T &\quad \Gamma \vdash e.f_i : T_f \\
\Gamma \vdash e_0.f_i = e_1 : T &\quad \Gamma \vdash e_0, f_i = e_1 : T_f \\
\Gamma \vdash e_0, f_i &\quad \text{fields}(C) = \overline{d} \overline{f} \\
\Gamma \vdash e_0, d_i &\quad \Gamma \vdash T_f <; T_f &\quad \text{mtype}(m, C) = \overline{d} \overline{d}' \overline{C} \\
\Gamma \vdash e, d_i &\quad \Gamma \vdash \text{isPrecise}(d) &\quad \Gamma \vdash U <; T_s \\
\exists e, \text{owner} \in T_s &\quad \text{isPrecise}(d) &\quad U_m = \sigma(e, d, d_u) C_u \\
\Gamma &\vdash e.m(\overline{C}) : U_m &\quad \Gamma \vdash e.\overline{m}(\overline{C}) : U_m \\
\end{align*}
\]

\textbf{Fig. 12: Expression Typing Rules.}

\[(T\text{-}\text{new})\]
A new expression \texttt{new } \texttt{a.c } C \text{ has type } \texttt{a.c } C. \text{ The domain } \texttt{a.c } C \text{ has the restriction } a \in \{ \text{this, owner} \}. \text{ Thus it follows that } \texttt{a.c } C \text{ is precise and that the owner of } \texttt{a.c } C \text{ must be } \text{this} \text{ or } \text{owner}. \text{ So an object can only create new objects in domains that are owned by itself or by its owner.}
Method Declarations, Class Declarations and Program Typing. Figure 11 shows rules for method and class declarations as well as programs. A program \( \langle L, C, e \rangle \) is typed by typing \( e \) in the type environment mapping this to \texttt{owner}\_same \( C \). 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 method declaration is well-formed if its body expression is well-typed in the type environment containing this and the formal parameters of the method. We demand \( \emptyset \vdash T \) and \( \emptyset \vdash T_r \) to ensure that the domain annotations of formal parameters and the result type does not contain local variables. Note that theoretically it would be possible that domain annotations of formal parameters contain other formal parameters as owners, but we omitted this feature for simplicity.

4 Dynamic Semantics

To prove the correctness of the type system of SLODJ we define an operational semantics for SLODJ. In contrast to Featherweight Java we use a big-step natural semantics. We handle local variables with a stack frame, instead of substitution, and we model a store, as SLODJ is not a functional language.

The dynamic entities of SLODJ are given in Figure 13. A value \( v \) is either an object \( o \) or \texttt{null}. A runtime domain is a tuple of a value \( v \) and a domain tail \( b \). An object state \( s \) is a triple \( \langle o.b, C, v \rangle \), consisting of a precise runtime domain \( o.c \) with an object \( o \) as owner, a class name \( C \), and a list of field values \( v \). 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

To formally handle domains at runtime, we use runtime domains. A runtime domain has the form \( v.b \). Where \( v \) is the owner of the domain, which can either be an object or \texttt{null}, and \( b \) is a sequence of \texttt{boundary}, \texttt{local} and \texttt{same}. Like domain annotations, runtime domains can either be precise or loose. A precise runtime domain has the form \( v.c \), loose ones are of the form \( v.b.c \). 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, and
The evaluation rules are shown in Figure 14. The rules are of the form

\[ S_0, F \vdash e \Rightarrow v, S_1 \]

read “With store \( S_0 \) and stack frame \( F \), expression \( e \) is evaluated to value \( v \) and the new store \( S_1 \)”.

The rules are more or less standard. A variable is evaluated (r-var), by looking up its value in the stack frame. A let expression is evaluated (r-let), by first evaluating expression \( e_0 \) to a value \( v_0 \) and store \( S_1 \). Then \( e_1 \) is evaluated with the new store \( S_1 \) and the stack frame \( F \) extended by variable \( x \) mapping to \( v_0 \). A field access (r-field) first evaluates \( e_1 \) to the receiver object \( o \) and new store \( S_1 \). Then the value \( v_1 \) of the corresponding field \( f_1 \) is taken from the corresponding object state. A field update (r-fieldup) first evaluates the receiver expression \( e_0 \) to the receiver object \( o \) and then evaluates the right-hand side expression \( e_1 \) to value \( v \). The resulting store \( S_3 \) is updated by replacing the value \( v_3 \) that corresponds to field \( f_1 \) with value \( v \) in the object state corresponding to object \( o \). Method invocations (r-invk) are evaluated by first evaluating the receiver expression \( e \) to object \( o \) and then sequentially evaluating the argument expressions \( \tau \). Finally, the body expression of method \( m \), \( e_0 \), is evaluated with a stack frame mapping \( \text{this} \) to \( o \) and the formal parameter names \( \tau \) mapping to the evaluated values \( \tau \). A new expression (r-new) adds a new object \( o \) to the store.

Perhaps the most interesting rule is the R-New rule, which shows that the domain of an object is determined at its creation time, and that SLODJ has no constructors, but instead initializes the fields of new objects with \( \text{null} \).
4.3 Auxiliary Functions.

We need some auxiliary functions which are shown in Figure 15. To obtain the owning (actual) domain of a value we define the method actd. It returns null.local for null, otherwise it obtains the domain from the object state of the object. The function owner returns the owner part of the actual domain of a value, and class obtains the class name of an object by looking at its object state.

**Domain Subset Relation.** Like the subdomain relation on domain annotation we define a subset relation on runtime domains. It is defined by (subset ∗). A runtime domain with null as owner is subset of any runtime domain (subset null). A runtime domain is subset of itself (subset refl). A runtime domain o₁.b₁ is a subset of a runtime domain o₂.b₂.b₁ if and only if the actual domain of o₁ is subset of o₂.b₂.

**The rtd Function.** To relate domain annotations to runtime domains we use the partial function rtd:

\[ rtd : Store \times StackFrame \times Value \times Domain \rightarrow RuntimeDomain \]

The intention of the function is to replace static syntactic owners of a domain by values and to replace 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. The rtd function distinguishes two cases: one where the domain owner is a variable (rtd var) and one where the domain
Store Well-Formedness:

\[(t\text{-store } \emptyset)\]
\[\vdash \emptyset\]

\[(t\text{-store object})\]
\[\vdash S_0 \quad \vdash C \quad S_1 = S_0[\text{obj} \rightarrow (\ldots, C, \text{value})] \quad \text{fields}(C) = \overline{C} \cap \overline{T}
\]
\[\forall v_i \in \text{var}. \quad v_i = \text{null} \lor S_1 \vdash \text{actd}(S_1, v_i) \subseteq \text{rtd}(S_1, \emptyset, o, d_i) \land \text{class}(S_1, v_i) <_c C_i\]
\[\vdash S_1\]

Stack Frame Well-Formedness:

\[(t\text{-stack } \emptyset)\]
\[(t\text{-stack null})\]
\[(t\text{-stack var})\]
\[S, \Gamma \vdash F\]
\[S, \Gamma \vdash F\]
\[S, \Gamma \vdash F[x \mapsto T] \vdash F[x \mapsto \text{null}]\]
\[S, \Gamma \vdash F[x \mapsto \text{null}]\]
\[S, \Gamma \vdash F[x \mapsto d C] \vdash F[x \mapsto o]\]

Fig. 16: Store and Stack Frame Well-Formedness

owner is \textit{this} or \textit{owner} (\text{rtd thisowner}). In the former case, the variable \(x\) is simply replaced by its value in \(F\), in the latter case, \textit{this} is replaced by the receiver value \(v_1\) and \textit{owner} by the corresponding owner value. In both cases, \textit{same} is replaced by the kind of the owning domain of \(v_1\).

4.4 Type Soundness

In this section we prove the soundness of the type system of SLODJ. We show the general subject reduction theorem and give an initial configuration to apply the theorem to programs.

We have to show that during the evaluation of an SLODJ program all values can only be of 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 \(<_c\). We can not, however, 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 \text{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 \text{null} we need not check anything.

For the soundness proof we need additional properties for stores and stack frames, which are shown in Figure [F]. They add the following new judgments:

\[\vdash S\] Store \(S\) is well-formed
\[S, \Gamma \vdash F\] Stack frame \(F\) is well-formed w.r.t. \(S\) and \(\Gamma\)

Both judgments closely resemble what is needed by the correctness proof, namely that types of field values of objects correspond to the declared type of the objects’ classes (\(t\text{-store }\ast\)), and that the types of values of a stack frame \(F\) correspond to the types recorded in the type environment \(\Gamma\) (\(t\text{-stack }\ast\)).
The Subject Reduction theorem states that if an expression $e$ is typed to $d : C$ and $e$ is evaluated by the operational semantics to value $v$, then $v$ is either $null$, or the actual class of $v$ is a subclass of $C$, and the actual domain of $v$ is a subset of the runtime domain of $d$. In addition, the theorem 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. $v = null \lor S_1 \vdash actd(S_1, v) \subseteq rtld(S_1, F, F(this), d) \land \text{class}(S_1, v) <_c C$ 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 Appendix A.1.

### 4.5 Initial Configuration

To apply the Subject Reduction theorem to a program, we have to give an initial configuration and show that the preconditions of the theorem hold. The question is, in which domain should the first object be contained? As we do not model a global domain, we decided that the first object is contained in its own boundary domain. That is, the owner of the first object is the object itself.

Let $P = (\Gamma, C, e)$, with $\vdash P : d : C$. The initial configuration is given by $\Gamma_{\text{init}} = \{this \mapsto \text{owner}.\text{same} \land C\}$, $S_{\text{init}} = \{o \mapsto (\text{owner}.\text{boundary}, C, \text{null})\}$, $F_{\text{init}} = \{this \mapsto o\}$. By $(\text{t-prog})$ we get $\Gamma_{\text{init}} \vdash e : d : C$. We have to show the following conditions:

1. $\Gamma_{\text{init}} \vdash e : d : C$
2. $\vdash S_{\text{init}}$
3. $S_{\text{init}}, F_{\text{init}} \vdash F_{\text{init}}$

**Proof.**[1][2] holds, because all fields of object $o$ are initialized with $null$. [3] holds, because $class(S_{\text{init}}, o) <_c C$ and $S_{\text{init}} \vdash actd(S_{\text{init}}, o) \subseteq rtld(S_{\text{init}}, F_{\text{init}}, o, \text{owner}.\text{same})$. The last condition holds, because $actd(S_{\text{init}}, o) = o.\text{boundary}$ and $rtld(S_{\text{init}}, F_{\text{init}}, o, \text{owner}.\text{same}) = o.\text{boundary}$.

### 5 Encapsulation Guarantees

In this section we show which encapsulation properties are guaranteed by our system. In order to do this, we first define which accesses should be allowed at runtime and then show that our type system guarantees that during the execution of a well-typed program, only such accesses can appear.

Figure 17 shows access rules that formally define which domains are accessible by an object at runtime. An object $o$ can access a domain $d$ if:

- $o$ is the owner of $d$. ($A\text{-OWN}$)
- $d$ is the boundary domain of an object $o_2$, and $o$ can access the domain that $o_2$ belongs to. ($A\text{-BOUNDARY}$)
- $d$ is a domain of the owner of $o$. ($A\text{-OWNER}$)
- The owner of $d$ is $null$. ($A\text{-NULL}$)
Accessibility:

\[
\begin{align*}
(A\text{-OWN}) & \quad S \models o_1 \rightarrow actd(S, o_2) \\
(A\text{-BOUNDARY}) & \quad S \models o_1 \rightarrow o_2.boundary \\
(A\text{-OWNER}) & \quad S \models o_1 \rightarrow o_2.c \\
(A\text{-NULL}) & \quad S \models o \rightarrow null.c
\end{align*}
\]

\[
\begin{align*}
(A\text{-VALUE}) & \quad S \models o \rightarrow actd(S, v) \\
& \quad S \models o \rightarrow v
\end{align*}
\]

Fig. 17: Accessibility Rules.

Store Accessibility:

\[
\begin{align*}
(A\text{-STORE } \varnothing) & \quad \models \varnothing \\
(A\text{-STORE OBJECT}) & \quad S_0 = S_0[x \mapsto \ldots, y] \\
& \quad \forall v_i \in \mathcal{V}. S_1 \models o \rightarrow v_i
\end{align*}
\]

Stack Frame Accessibility:

\[
\begin{align*}
(A\text{-STACKFRAME } \text{this}) & \quad S \models F_0 \\
& \quad F_1 = F_0[x \mapsto v] \\
& \quad S \models F_1(\text{this}) \rightarrow v
\end{align*}
\]

Fig. 18: Store and Stack Frame Accessibility

\[(A\text{-VALUE})\] defines which values an object can access. It states that an object \(o\) can access a value \(v\) iff \(o\) can access the actual domain of \(v\). Note that in conjunction with \((A\text{-NULL})\) and \((ACTD NULL)\) an object can always access null.

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 \(o_1 \equiv o_2\), or \(o_2\) is the owner of \(o_1\). Thus we get guaranteed that the local objects of an object \(o\) can only be accessed by itself and 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 18. All objects of a store must have access to the values of their fields \((A\text{-STORE } \star)\), and all values of a stack frame must be accessible by the this-object \((A\text{-STACKFRAME } \star)\).

The following 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.

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

\[S_1 \vdash F(\text{this}) \rightarrow v \land \models S_1\]

**Proof.** The proof is by structural induction on the reduction rules of the operational semantics. It can be found in Appendix A.2.
Note that this theorem also enforces the boundary-as-dominator property, as objects of the outside of an object \( o \) cannot directly access the inside of \( o \), but have to use objects of the boundary of \( o \).

6 Discussion and Related Work

Ownership type systems. Encapsulation of objects was first proposed by Hogg with Islands \[14\] and by Almeida with Balloons \[3\]. The notion of ownership types comes from Clarke \[10\] as a formalizing the core of Flexible Alias Protection \[20\]. Ever since, many researchers investigated ownership type systems \[3, 15, 6, 2\]. Ownership type systems have been used to prevent data-races \[7\], deadlocks \[5, 4\], and to allow the modular specification and verification of object-oriented programs \[17\].

All ownership type systems have one 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 so-called owners-as-dominators property \[8\]. It states that all access paths from outside the owner object to its owning objects must go through the owner object itself. Thus preventing iterators to be accessible by the outside and accessing internal objects of a list at the same time. Two solutions have been proposed to solve this problem. The first solution \[9\] is to allow the creation of dynamic aliases to owned objects. Dynamic aliases are variables that lie on the stack. The idea behind this solution is that dynamic alias are gone after a method has exited, and so these aliases are considered to be not as bad as static aliases. However, at least for the modular verification of object-oriented programs, dynamic aliases are as bad as static aliases \[17\].

The second solution \[8, 5\] is to allow Java’s inner member classes \[13\] to access the representation objects of their parent objects. This solution only works, because inner member classes always have an implicit final variable that holds a reference to its parent object. As an inner class is always in a module together with its parent class, this solution does allow the creation of dynamic aliases to owned objects. Dynamic aliases are variables that lie on the stack. The idea behind this solution is that dynamic alias are gone after a method has exited, and so these aliases are considered to be not as bad as static aliases. However, at least for the modular verification of object-oriented programs, dynamic aliases are as bad as static aliases \[17\].

The second solution \[8, 5\] is to allow Java’s inner member classes \[13\] to access the representation objects of their parent objects. This solution only works, because inner member classes always have an implicit final variable that holds a reference to its parent object. As an inner class is always in a module together with its parent class, this solution does allow the creation of dynamic aliases to owned objects. Dynamic aliases are variables that lie on the stack. The idea behind this solution is that dynamic alias are gone after a method has exited, and so these aliases are considered to be not as bad as static aliases. However, at least for the modular verification of object-oriented programs, dynamic aliases are as bad as static aliases \[17\].

Ownership Domains. Ownership domains \[1\] was the first ownership type system that introduced ownership domains. An object is no longer owned by another object, but instead is owned by a domain, that in turn is owned by an object. Domains can either be public or private (corresponding to boundary and local in our system). The ownership domains system was the first ownership type system that could really handle the iterator problem without using the inner classes workaround or dynamic aliases.

In contrast to our system, the number of domains that are owned by an object is not restricted to two, and the access permissions between different domains are not hardwired, but can be specified by the programmer. From this point of view the ownership domains system is more flexible than our system. However, that flexibility comes at a price. As we hardwire our access permissions, we can implement an iterator without passing the owner domain as parameter to the iterator class. This is needed by the ownership domains approach and prevents the iterator class to be visible by
the outside. The solution to this problem is, that the concrete iterator class has to be a subtype of the iterator that can be accessed by the outside. Thus, without subtyping the Ownership Domains approach could not handle the iterator example.

One innovation of our system is the introduction of loose domains. All other ownership type systems only handle precise ownership information. In the ownership domains type system, for example, one is forced to attach a final variable to the type of a public domain to be able to access an object of that domain. Thus, the model-view example can only be implemented with a fixed number of Listener objects, as for every View instance there has to be a final variable to access its Listener object. With our type system it is possible to implement the model-view example with an arbitrary number of Listener objects, as the Subject class can simply be parameterized with a loose domain.

Ownership Domains have been combined with an effects system [23]. A more general version of Ownership Domains has been formalized in System F [16].

Simulating SLOD with Ownership Domains. It is possible to partly encode our system in the ownership domain system by demanding the following conditions:

– Every class declares a private domain local and a public domain boundary.
– Every class declares the links local -> boundary, boundary -> local, local -> owner and boundary -> owner
– For every domain parameter E the class declares the links local -> E and boundary -> E.
– There are no other link declarations and domain declarations.

In addition, the shared domain of ownership domains is called global in our system and the owner keyword in ownership domains is equal to same in SLOD.

Note that this encoding is not equal to our system. There are two important differences:

1. The encoding does not support loose domains.
2. An object X in the boundary domain of an object Y cannot automatically access an object Z in the local domain of Y. To allow this access X has to be parameterized with the local domain of Y. In our system this access is possible without an additional parameter.

The importance of the first difference should be clear, as otherwise programming idioms like the model-view example cannot be implemented. To illustrate the second difference we try to implement the linked list implementation in Figure 3 with the encoded system. In order to implement the Iterator class, we need to give that class an additional domain parameter representing the local domain of the owning list. The linked list must then parameterize the Iterator type as boundary Iterator<T,local>. But this is a problem as such a type is not visible outside the list. The solution is to create a subtype of the Iterator class with the additional domain parameter, but return a super type without that parameter.

7 Conclusion and Future Work

In this paper we presented Simple Loose Ownership domains (SLOD). SLOD significantly simplifies Ownership Domains, by omitting link and domain declarations. Our system introduces loose domains which increases the flexibility opposed to other ownership type systems, as the exact domain need not always be known statically.
This enables, for example, the implementation of model-view systems with an arbitrary number of listener callbacks, which is not possible with standard Ownership Domains. We have shown that our type system is sound and that it guarantees that objects in local domains are encapsulated. Our system guarantees a property we call boundary-as-dominator, which is a generalization of the owners-as-dominator property of ownership type systems.

We plan to extend the formalization of SLOD with domain parameters. We are currently inspecting existing libraries and programs, to measure the practicability of our approach. Further, we are investigating the extension of SLOD by read-only annotations and immutable objects. 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. We will also investigate how SLOD can be used to give thread-safeness guarantees. Finally, we plan to implement a checking tool for a practical subset of Java.
Bibliography


[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


A  Proofs

A.1 Proof of Theorem 1

Helpful Lemmas

Lemma 1 (Type Well-Formedness). If an expression $e$ is typed with type $T$ in $\Gamma$, then $T$ is well-formed in $\Gamma$.

$$\Gamma \vdash e : T \implies \Gamma \vdash T$$

Proof. By case distinction on the shape of $e$. There are 6 cases.

Case (t-var) $e = x$ $\Gamma \vdash \diamond$ $\Gamma(x) = T$.

1. By (t-env $x$),
   (a) $\Gamma' \vdash T$
   for $\Gamma = \Gamma'[x \mapsto T]$.
2. Thus it follows (a) $\Gamma \vdash T$ as for all $y \in dom(\Gamma')$. $\Gamma'(y) = \Gamma(y)$

Case (t-field) Immediate.

Case (t-fieldup) $e = e_0.f_i = e_1$ $\Gamma \vdash T < : T_f$

1. By (s-type),
   (a) $\Gamma \vdash T$

Case (t-invk) Immediate.

Case (t-let) Immediate.

Case (t-new) Immediate.

Lemma 2.

$$\Gamma \vdash d_1 < :_d d_2 \land |d_2| = 2 \implies d_1 = d_2 \land \Gamma \vdash d_1$$

1. $\Gamma \vdash d_1 < :_d d_2 \land d_1 \neq d_2 \implies$

   $\exists x, d_x, C_x, b, d_0. \Gamma \vdash d_1 \land \Gamma \vdash d_2$
   $\land d_1 = x.b \land d_2 = d_0.b \land \Gamma \vdash x : d_x C_x \land \Gamma \vdash d_x < :_d d_0$

2. Applying (s-domain var) to (1c) and (1d) we get
   (a) $\Gamma \vdash x.b_1.b < :_d d_0.b_1.b$

3. With (1a) and (1b) it follows $\Gamma \vdash d_1.b < :_d d_2.b$, which had to be shown.

Lemma 3.

$$\Gamma \vdash d_1 < :_d d_2 \land \Gamma \vdash d_1.b \land \Gamma \vdash d_2.b \implies \Gamma \vdash d_1.b < :_d d_2.b$$

Proof. Assume $\Gamma \vdash d_1 < :_d d_2 \land \Gamma \vdash d_1.b \land \Gamma \vdash d_2.b$. There are two cases.

Case $d_1 = d_2$. Hence $d_1.b = d_2.b$. By (s-domain refl) follows $\Gamma \vdash d_1.b < :_d d_2.b$.

Case $d_1 \neq d_2$.

1. By Lemma 2 there exist $x, b_1, d_0, d_x, C_x$ with
   (a) $d_1 = x.b_1$
   (b) $d_2 = d_0.b_1$
   (c) $\Gamma \vdash x : d_x C_x$
   (d) $\Gamma \vdash d_x < :_d d_0$

2. Applying (s-domain var) to (1c) and (1d) we get
   (a) $\Gamma \vdash x.b_1.b < :_d d_0.b_1.b$

3. With (1a) and (1b) it follows $\Gamma \vdash d_1.b < :_d d_2.b$, which had to be shown.

Lemma 4 (Subdomain Transitivity). The subdomain relation is transitive.

$$S \vdash d_1 < :_d d_2 \land S \vdash d_2 < :_d d_3 \implies S \vdash d_1 < :_d d_3$$
Proof. By induction on the length $n = \lvert d_1 \rvert + \lvert d_2 \rvert + \lvert d_3 \rvert$.

**Induction Base** $n = 6$, as the minimum length of a domain is 2.

1. Suppose
   (a) $\Gamma \vdash d_1 \ll d_2 \land \Gamma \vdash d_2 \ll d_3$
   2. By Lemma 2
      (a) $\Gamma \vdash d_1 \land \Gamma \vdash d_2 \land \Gamma \vdash d_3 \land d_1 = d_2 \land d_2 = d_3$
   3. Hence, by (s-domain refl) it follows $\Gamma \vdash d_1 \ll d_3$, which had to be shown.

**Induction Step** $n = m$.

1. Suppose
   (a) $\Gamma \vdash d_1 \ll d_2 \land \Gamma \vdash d_2 \ll d_3$
   for some $d_1, d_2, d_3$ with $\lvert d_1 \rvert + \lvert d_2 \rvert + \lvert d_3 \rvert = m$.
   2. Without loss of generality assume $d_1 \neq d_2 \neq d_3$. Hence by Lemma 2 there exist $x, y, b_1, b_2, d_{2x}, d_{2x}, d_{x}, d_{y}, C_{x}, D_{x}$ with
      (a) $d_1 = x \cdot b_1 \land d_2 = y \cdot b_2 \cdot b_1 \land d_3 = d_{2x} \cdot d_{2x} \cdot b_1$
      (b) $\Gamma \vdash x : d_x \ C_x$
      (c) $\Gamma \vdash y : d_y \ D_x$
      (d) $\Gamma \vdash d_y \ll d_y \ b_2$
      (e) $\Gamma \vdash d_y \ll d_y \ b_2$
      (f) $\Gamma \vdash d_1 \land \Gamma \vdash d_2 \land \Gamma \vdash d_3$
   3. Note that from (2d) and (2a) it follows $\Gamma \vdash d_{3x} \cdot d_{2x}$, and from (2d) we get $\Gamma \vdash y \cdot b_2$. Hence with (2c) and (2e) it follows by (s-domain var),
      (a) $\Gamma \vdash y \cdot b_2 \ll d_{3x} \cdot d_{2x}$
   4. Note that from (2a) it follows that $\lvert d_x \rvert \leq \lvert y \cdot b_2 \rvert$. Hence we get $\lvert d_x \rvert + \lvert y \cdot b_2 \rvert + \lvert d_{3x} \cdot d_{2x} \rvert < m$. So we can apply the induction hypothesis to (2d) and (2a), and we get
      (a) $\Gamma \vdash d_x \ll d_{3x} \cdot d_{2x} \cdot b_2$
   5. Hence, with (2d) and (2a) it follows by (s-domain var),
      (a) $\Gamma \vdash x \cdot b_1 \ll d_{3x} \cdot d_{2x} \cdot b_1$
   6. Thus with (2a) we get $\Gamma \vdash d_1 \ll d_3$, which had to be shown.

**Lemma 5 (Domain Subset Transitivity).** The domain subset relation is transitive.

$S \vdash d_1 \subseteq d_2 \land S \vdash d_2 \subseteq d_3 \implies S \vdash d_1 \subseteq d_3$

Proof. Let $d_1, d_2$ and $d_3$ be domains and $S \vdash d_1 \subseteq d_2 \land S \vdash d_2 \subseteq d_3$. We show that $S \vdash d_1 \subseteq d_3$ by induction on the sum of the lengths of the domains: $n = \lvert d_1 \rvert + \lvert d_2 \rvert + \lvert d_3 \rvert$.

**Induction Base** $n = 6$. The smallest $n$ is 6, as a domain has at least two elements. We assume $S \vdash d_1 \subseteq d_2 \land S \vdash d_2 \subseteq d_3$, otherwise we are done. As the length of all domains is 2, there are two cases: $d_1 = \text{null.c}$ and $d_1 \neq \text{null.c}$. If $d_1 = \text{null.c}$, we can apply (subset null). So we assume $d_1 \neq \text{null.c}$.

1. By (subset refl),
   (a) $d_1 = d_2$
   (b) $d_2 = d_3$
2. Hence,
   (a) $d_1 = d_3$
3. Thus by (subset refl),
   (a) $S \vdash d_1 \subseteq d_3$

**Induction Step** $n = m$. Assume $S \vdash d_1 \subseteq d_2 \land S \vdash d_2 \subseteq d_3$. Also assume $d_1 \neq \text{null.c}$, otherwise we can apply (subset null), and assume $d_1 \neq d_2 \neq d_3$, otherwise we can apply (subset refl).

1. By $S \vdash d_1 \subseteq d_2$ and (subset loose), $\exists v_1, v_2, b_1, b_2$ with
(a) \( d_1 = v_1.b_1 \), and
(b) \( d_2 = v_2.b_2.b_1 \), and
(c) \( \text{actd}(S, v_1) \subseteq v_2.b_2 \)

2. By \( S \vdash d_2 \subseteq d_3 \) and (subset loose), \( \exists v_3, b_3 \) with
(a) \( d_3 = v_3.b_3.b_1 \), and
(b) \( \text{actd}(S, v_2) \subseteq v_3.b_3 \)

3. Hence by (2b) and (subset loose),
(a) \( v_2.b_2 \subseteq v_3.b_3 \)

4. By (1c), (3a) and the induction hypothesis (the induction hypothesis can be applied, because \( |\text{actd}(S, v_1)| = 2 \) and \( 2 + |v_2.b_2| + |v_3.b_3.b_2| < |v_1.b_1| + |v_2.b_2.b_1| + |v_3.b_3.b_2.b_1| \)),
(a) \( \text{actd}(S, v_1) \subseteq v_3.b_3.b_2 \)

5. Thus, by (subset loose),
(a) \( v_1.b_1 \subseteq v_3.b_3.b_2.b_1 \)

Definition 1 (Store Order). The store order defines an order on stores. A store \( S_1 \) is bigger in this order as a store \( S_0 \) iff all objects of \( S_0 \) also exist in \( S_1 \) and the type and domain of these objects are the same.

\[ S_0 \leq S_1 \equiv \forall o, d, C, f. S_0(o) = \langle d, C, v_0 \rangle \Rightarrow \exists v, S_1(o) = \langle d, C, v \rangle \]

Lemma 6 (Store Order Transitivity). The store order is transitive.

\[ S_0 \leq S_1 \land S_1 \leq S_2 \Rightarrow S_0 \leq S_2 \]

Proof. Clear.

Lemma 7 (Store Only Grows). This lemma states, that objects are never destroyed, and thus the store can only grow. An important fact from this lemma is, that the domain and the type of an existing object never changes during the execution of a program.

\[ S_0, F \vdash e \Rightarrow v, S_1 \Rightarrow S_0 \leq S_1 \]

Proof. This can easily be shown by structural induction on the reduction rules of the operational semantics. The only interesting cases are (r-fieldup) and (r-new).

Case (r-fieldup) Assume \( S_0, F \vdash e_0.f_i = e_1 \Rightarrow v, S_3 \).

1. By (r-fieldup),
   (a) \( S_0, F \vdash e_0 \Rightarrow o, S_1 \)
   (b) \( S_1, F \vdash e_1 \Rightarrow v, S_2 \)
   (c) \( S_2(o) = \langle rd, C, \overline{v} \rangle \)
   (d) \( S_3 = S_2|o \Rightarrow (rd, C, [v/v_i|\overline{v}] \)

2. By (\ref{eq:1}), (\ref{eq:2}) and the induction hypothesis,
   (a) \( S_0 \leq S_1 \)
   (b) \( S_1 \leq S_2 \)

3. By Lemma (\ref{eq:3})
   (a) \( S_0 \leq S_2 \)

4. As \( S_3 \) and \( S_2 \) only differ in the values \( \overline{v} \) of the state of object \( o \), and the domain \( rd \) and the class \( C \) are equal, it follows,
   (a) \( S_2 \leq S_3 \)

5. With (\ref{eq:3}) and Lemma (\ref{eq:3})
   (a) \( S_0 \leq S_3 \)

Case (r-new) Assume \( S_0, F \vdash \text{new } d C \Rightarrow o, S_1 \).

1. By (r-new),
(a) \( rd = rtd(S_0, F, F(this), d) \)
(b) \( o \notin \text{dom}(S_0) \)
(c) \( S_1 = S_0[o \mapsto (rd, C, \text{null})] \)

2. By (1b) and (1c) it follows,
(a) \( S_0 \leq S_1 \)

Lemma 8 (Stack Frame Stays Well-Formed). This lemma states, that no reduction can destroy the well-formedness of a stack frame w.r.t. the store.

\[ S_0, \Gamma \vdash F \land S_0, F \vdash e \Rightarrow v, S_1 \Rightarrow S_1, \Gamma \vdash F \]

Proof. By Lemma 7 we get \( S_0 \leq S_1 \). As the stack frame well-formedness only depends on the domains and the types of objects in \( S_0 \), and \( S_1 \) contains all objects of \( S_0 \) with the same type and domain, stack frame \( F \) is also well-formed w.r.t. store \( S_1 \).

Lemma 9 (Subdomaining relates to Domain Subset). This lemma relates the subdomain relation on domain annotations to the subset relation of runtime domains.

\[ \text{this} \in \text{dom}(F) \land S, \Gamma \vdash F \land \Gamma \vdash d_0 <_{\text{d}} d_1 \]
\[ S \vdash rtd(S, F, F(this), d_0) \subseteq rtd(S, F, F(this), d_1) \]

Proof. We proof this lemma by induction on the sum of the lengths of the domains:
\[ n = |d_0| + |d_1|. \]

Induction Basis \( n = 4 \). The smallest \( n \) is 4, as a domain has at least two elements.

So \( |d_0| = |d_1| = 2 \). By \( \Gamma \vdash d_0 <_{\text{d}} d_1 \) and \( \text{(s-domain refl)} \) it follows that \( d_0 = d_1 \), hence \( S \vdash rtd(S, F, F(this), d_0) \subseteq rtd(S, F, F(this), d_1) \) by \( \text{(subrel}. \text{refl)} \).

Induction Step \( n = m \). Assume \( \Gamma \vdash d_0 <_{\text{d}} d_1 \). If \( d_0 = d_1 \) we are done, so assume \( d_0 \neq d_1 \).

1. Assume
(a) \( \text{this} \in \text{dom}(F) \)
(b) \( S, \Gamma \vdash F \)
(c) \( \Gamma \vdash d_0 <_{\text{d}} d_1 \)

2. By (1c) and \( \text{(s-domain loose)} \), there are \( x, b, d_x, C_x, d_2 \) with
(a) \( d_0 = x.b \)
(b) \( d_1 = d_2.b \)
(c) \( \Gamma \vdash x : d_x C_x \)
(d) \( \Gamma \vdash d_x <_{\text{d}} d_1 \)

3. As \( |d_2| < |d_1| \) and \( |d_x| \leq |d_2| \),
(a) \( |d_x| + |d_2| < n \)

4. By 2c and the induction hypothesis,
(a) \( S \vdash rtd(S, F, F(this), d_x) \subseteq rtd(S, F, F(this), d_2) \)

5. By 2c and \( \text{(t-var)} \),
(a) \( x \in \text{dom}(F) \)

6. By 5a, 1b, \( \text{(t-stack null)} \) and \( \text{(t-stack var)} \),
(a) \( F(x) = \text{null}, \text{or} \)
(b) \( S \vdash \text{actd}(S, F(x)) \subseteq rtd(S, F, F(this), d_x) \)

7. Assume 6a, by \( \text{(actd null)} \),
(a) \( \text{actd}(S, F(x)) = \text{null.local} \)

8. Hence by \( \text{(subset null)} \),
(a) \( S \vdash \text{actd}(S, F(x)) \subseteq rtd(S, F, F(this), d_2) \)

9. Now assume 6b, by 4a and Lemma 5,
(a) \( S \vdash \text{actd}(S, F(x)) \subseteq rtd(S, F, F(this), d_2) \)

10. Hence by \( \text{(subset loose)} \),
(a) \( S \vdash rtd(S, F, F(this), x.b) \subseteq rtd(S, F, F(this), d_2.b) \)

11. Thus with 2a and 2b,
(a) \( S \vdash rtd(S, F, F(this), d_0) \subseteq rtd(S, F, F(this), d_1) \)
Proof of Theorem 1

Proof. The proof is by structural induction on the reduction rules of the operational semantics.

Case (r-var)

1) 1. Assume
(a) \( S, F \vdash x \Rightarrow v, S \)
(b) \( F \vdash x : d \ C \)
(c) \( S, F \vdash F \)
2. By (1a) and (r-var),
(a) \( F(x) = v \)
3. By (1b) and (t-var),
(a) \( F(x) = d \ C \)
4. By (t-stack null) and (t-stack var)
(a) \( v = \text{null}, \) or
(b) \( S \vdash \text{actd}(S, v) \subseteq \text{rtd}(S, F, F(x), d) \) \( \land \) \( \text{class}(S, v) < c \ C \)
as required.

II) The store does not change, so \( \vdash S \) by assumption.

Case (r-let)

1) 1. Assume
(a) \( S_0, F \vdash \text{let } x = e_0 \text{ in } e_1 \Rightarrow v_1, S_2 \)
(b) \( \Gamma \vdash \text{let } x = e_0 \text{ in } e_1 : d \ C \)
(c) \( S_0, \Gamma \vdash F \)
(d) \( \text{this} \in \text{dom}(F) \)
(e) \( \vdash S_0 \)
2. By (1a) and (r-let),
(a) \( S_0, F \vdash e_0 \Rightarrow v_0, S_1 \)
(b) \( S, F[x \mapsto v_0] \vdash e_1 \Rightarrow v_1, S_2 \)
for some \( v_0, S_1 \)
3. By (1b) and (t-let),
(a) \( F \vdash e_0 : d_0 \ C_0 \)
(b) \( F[x \mapsto d_0 C_0] \Rightarrow e_1 : d \ C \)
(c) \( x \notin \text{dom}(F) \)
for some \( d_0, C_0 \)
4. By (3a), (2a), (1c), (1d), (1e) and the induction hypothesis,
(a) \( v_0 = \text{null} \lor \)
\( \text{actd}(S_1, v_0) \subseteq \text{rtd}(S_1, F, F(\text{this}), d_0) \) \( \land \) \( \text{class}(S_1, v_0) < c \ C_0 \)
(b) \( \vdash S_1 \)
5. By (1c), (1a) and Lemma 8
(a) \( S_1, F \vdash F \)
6. With (3a), (4a), (t-stack null) and (t-stack var)
(a) \( S_1, F[x \mapsto d_0 C_0] \vdash F[x \mapsto v_0] \)
7. Let
(a) \( F' = F[x \mapsto v_0] \)
8. By (6a), (4b), (3b), (2b) and the induction hypothesis,
(a) \( v_1 = \text{null} \lor S_2 \vdash \text{actd}(S_2, v_1) \subseteq \text{rtd}(S_2, F', F'(\text{this}), d) \) \( \land \) \( \text{class}(S_2, v_1) < c \ C \)
(b) \( \vdash S_2 \)
9. By (5c) and (1c),
(a) \( x \notin \text{dom}(F) \)
10. Hence with (7a),
(a) \( \forall y \in \text{dom}(F). \ F(y) = F'(y) \)

11. With \(1d\),
   (a) \( F'(this) = F(this) \)

12. Hence with \(8a\),
   (a) \( v_1 = \text{null} \lor S_2 \vdash \text{actd}(S_2, v_1) \subseteq \text{rtd}(S_2, F', F(this), d) \)

13. Let
   (a) \( d = a.b \)

14. Case distinction on the shape of \( a \),

15. Assume
   (a) \( a \neq y \)

16. By (\text{rtd thisowner}),
   (a) \( \text{rtd}(S_2, F', F(this), d) = \text{rtd}(S_2, F, F(this), d) \)
as \( F' \) and \( F \) is not used by (\text{rtd thisowner})

17. Assume
   (a) \( a = y \)
   for some \( y \)

18. By \(1b\), Lemma 1 and (T-TYPE),
   (a) \( \Gamma \vdash d \)

19. By \(17a\) and (T-DOMAIN VAR),
   (a) \( \Gamma \vdash y : T \)
   for some \( T \)

20. By (T-VAR)
   (a) \( y \in \text{dom}(\Gamma) \)

21. By \(1c\) and (T-STACK *),
   (a) \( y \in \text{dom}(\Gamma) \)

22. By \(9a\),
   (a) \( x \neq y \)

23. By \(7a\),
   (a) \( F(y) = F'(y) \)

24. By (\text{rtd var}),
   (a) \( \text{rtd}(S_2, F', F(this), d) = \text{rtd}(S_2, F, F(this), d) \)

25. Thus, with \(16a\), \(12a\) and \(8a\),
   (a) \( v_1 = \text{null} \lor S_2 \vdash \text{actd}(S_2, v_1) \subseteq \text{rtd}(S_2, F, F(this), d) \wedge \text{class}(S_2, v_1) <_{< C} \)
as required.

II) Shown above by \(8a\).

**Case (R-FIELD)**

1) 1. Assume
   (a) \( S_0, F \vdash e. f_i \Rightarrow v_i, S_1 \)
   (b) \( \Gamma \vdash e. f_i : d_i C_i \)
   (c) \( S_0, \Gamma \vdash F \)
   (d) \( \text{this} \in \text{dom}(F) \)
   (e) \( \vdash S_0 \)

2. By \(1a\) and (R-FIELD),
   (a) \( S_0, F \vdash e \Rightarrow o, S_1 \)
   (b) \( S_1(o) = \langle \text{rd}, C, \pi \rangle \)

3. By \(1b\) and (T-FIELD),
   (a) \( \Gamma \vdash e : d C \)
   (b) \( \text{fields}(C) = \overline{d_f} C_f \overline{f} \)
   (c) \( d_i C_i = \sigma(e, d, d_f) C_{f_i} \)
   (d) \( \Gamma \vdash d_i C_i \)
   for some \( d, C, C_f, \overline{d_f}, \overline{f}, x \)
4. By (1c), (1d), (1e), (2a), (3a) and the induction hypothesis,
   (a) \( S_1 \vdash \text{actd}(S_1, o) \subseteq \text{rtd}(S_1, F, F(this), d) \)
   (b) \( \text{class}(S_1, o) <_c C \)
   (c) \( d_i \vdash S_1 \)
5. By (4e) and (T-STORE),
   (a) \( v_i = \text{null} \vee S_1 \vdash \text{actd}(S_1, v_i) \subseteq \text{rtd}(S_1, \emptyset, o, d_f) \wedge \text{class}(S_1, v_i) <_c C_i \)
6. Assume
   (a) \( v_i \neq \text{null} \)
   otherwise we are done.
7. Hence with (5a),
   (a) \( S_1 \vdash \text{actd}(S_1, v_i) \subseteq \text{rtd}(S_1, \emptyset, o, d_f) \)
   (b) \( \text{class}(S_1, v_i) <_c C_i \)
8. Case distinction on the shape of \( d_f \). There are two cases:
   (a) \( d_f = \text{owner}.b \)
   (b) \( d_f = \text{this}.b \)
   note that by (T-CLASSDECL), \( d_f \) cannot have the form \( x.b \).
9. Let
   (a) \( d = d_o.c_1 \)
   (b) \( \text{actd}(S_1, o) = v_o.c_2 \)
10. By (1a), (9b) and (subset *),
   (a) \( c_1 = c_2 \)
   Case \( d_f = \text{owner}.b \).
11. Hence by (rtd THISOWNER),
   (a) \( \text{rtd}(S_1, \emptyset, o, d_f) = v_o.[c_1/\text{same}]b \)
12. By (4a),
   (a) \( S_1 \vdash \text{actd}(S_1, v_i) \subseteq v_o.[c_1/\text{same}]b \)
13. By (4a) and (8a),
   (a) \( d_i = d_o.[c_1/\text{same}]b \)
14. By (4a), (9b) and (9b),
   (a) \( S_1 \vdash v_o.c_1 \subseteq \text{rtd}(S_1, F, F(this), d_o.c_1) \)
15. Hence by (subset *) there are three cases:
   (a) \( v_o = \text{null} \), or
   (b) \( v_o.c_1 = \text{rtd}(S_1, F, F(this), d_o.c_1) \), or
   (c) \( \text{actd}(S_1, v_o) \subseteq \text{rtd}(S_1, F, F(this), d_o) \)
16. We show that for all three cases the following holds,
   (a) \( v_o.[c_1/\text{same}]b \subseteq \text{rtd}(S_1, F, F(this), d_o.[c_1/\text{same}]b) \)
   Case (15a), by (SUBSET NULL).
   Case (15b), by (SUBSET REFL), as
   \( v_o.[c_1/\text{same}]b = \text{rtd}(S_1, F, F(this), d_o.[c_1/\text{same}]b) \).
   Case (15c), by (SUBSET LOOSE).
17. Thus by (12a), (16a), (13a) and Lemma 5,
   (a) \( S_1 \vdash \text{actd}(S_1, v_i) \subseteq \text{rtd}(S_1, F, F(this), d_i) \)
   closing the case.
   Case \( d_f = \text{this}.b \).
18. By (5a),
   (a) \( d_i = e.[c/\text{same}]b \)
19. By (3a), \( e \) has to be a local variable,
   (a) \( e = x \)
20. By (18a),
   (a) \( d_i = x.[c/\text{same}]b \)
for some \( x \).

21. By (8b), (rtd thisowner),
   (a) \( \text{rtd}(S_1, o, d_i) = o.\text{same}b \)

22. With (1c) and (t-stack var),
   (a) \( S_1 \vdash \text{actd}(S_1, v) \subseteq o.\text{same}b \)

23. By (2a), (19a) and (r-var),
   (a) \( F(x) = o \)

24. Hence by (rtd var),
   (a) \( \text{rtd}(S_1, F, F(\text{this}), x.\text{same}b) = o.\text{same}b \)

25. Thus with (22a) and (20a),
   (a) \( S_1 \vdash \text{actd}(S_1, v) \subseteq \text{rtd}(S_1, F, F(\text{this}), d_i) \)
   closing the case.

II) This is already shown above by (4c).

Case (r-fieldup)

1. Assume
   (a) \( S_0, F \vdash e_0, f_i = e_1 \Rightarrow v, S_3 \)
   (b) \( \Gamma \vdash e_0, f_i = e_1 : d C \)
   (c) \( S_0, \Gamma \vdash F \)
   (d) \( \text{this} \in \text{dom}(\Gamma) \)
   (e) \( \Gamma \vdash S_0 \)

2. By (1a) and (r-fieldup),
   (a) \( S_0, F \vdash e_0 \Rightarrow o, S_1 \)
   (b) \( S_1, F \vdash e_1 \Rightarrow v, S_2 \)
   (c) \( S_2(o) = (\text{rd}, D, \pi) \)
   (d) \( S_3 = S_2[o \mapsto (\text{rd}, D, [v/v_i][\pi])] \)
   for some \( o, S_1, S_2, rd, D, \pi \).

3. By (1b) and (t-fieldup),
   (a) \( \Gamma \vdash e_0 : d C_0 \)
   (b) \( \text{fields}(C_0) = d_f C_f \; \Gamma \)
   (c) \( T_f = \sigma(e_0, d_0, d_f) \; C_f \)
   (d) \( \Gamma \vdash d C \; <: \; T_f \)
   (e) \( \text{owner} \in d_f \Rightarrow \text{isPrecise}(d_0) \)
   for some \( d_0, C_0, d_f, C_f, \pi, \Gamma \).

4. By (1d), (1e), (1c), (3a), (2a) and the induction hypothesis,
   (a) \( \text{actd}(S_1, o) \subseteq \text{rtd}(S_1, F, F(\text{this}), d_0) \)
   (b) \( \text{class}(S_1, o) <_c C_0 \)
   (c) \( \Gamma \vdash S_1 \)

5. By (3d) and (s-type),
   (a) \( \Gamma \vdash e_1 : d C \)

6. By (1e) and Lemma 8
   (a) \( S_1, \Gamma \vdash F \)

7. With (1d), (4c), (5a), (2b) and the induction hypothesis,
   (a) \( v = \text{null} \lor \text{actd}(S_2, v) \subseteq \text{rtd}(S_2, F, F(\text{this}), d) \land \text{class}(S_2, v) <_c C) \)
   (b) \( \Gamma \vdash S_2 \)

8. Assume
   (a) \( v \neq \text{null} \)
   otherwise we are done.

9. Hence with (7a),
   (a) \( \text{actd}(S_2, v) \subseteq \text{rtd}(S_2, F, F(\text{this}), d) \)
   (b) \( \text{class}(S_2, v) <_c C \)

10. By (2c) and (2d).
(a) $\text{dom}(S_2) = \text{dom}(S_3)$
(b) $\forall o_x \in \text{dom}(S_2)$. 
   $\text{class}(S_2, o_x) = \text{class}(S_3, o_x) \land$
   $\text{actd}(S_2, o_x) = \text{actd}(S_3, o_x)$
11. By (10b).
   (a) $\forall o_x \in \text{dom}(S_2)$. 
   $\text{rtd}(S_2, F, F(\text{this}), d) = \text{rtd}(S_3, F, F(\text{this}), d)$
12. Thus with (10a), (10b), (10c) and (9f).
   (a) $\text{actd}(S_3, v) \subseteq \text{rtd}(S_3, F, F(\text{this}), d)$
   (b) $\text{class}(S_3, v) \prec C$
which had to be shown.

II) We have to show $\vdash S_3$. By (7b), $\vdash S_2$. The only difference between $S_2$ and $S_3$ is
that the value $v_i$ of field $f_i$ of object $o$ is replaced by $v$. So to show $\vdash S_3$, we have to show $v = \text{null}$ or $S_3 \vdash \text{actd}(S_3, v) \subseteq \text{rtd}(S_3, \varnothing, o, d_{f_i}) \land \text{class}(S_3, v) \prec C_{f_i}$.
13. Assume
   (a) $v \neq \text{null}$
otherwise we are done.
14. By (3d), (12b) and (s-class trans),
   (a) $\text{class}(S_3, v) \prec C_{f_i}$
15. By (3c), (3d) and (s-type),
   (a) $\Gamma \vdash d : \sigma(e_0, d_0, d_{f_i})$
16. By (1c) and Lemma 8
   (a) $S_3, \Gamma \vdash F$
17. By (12a), (15a) and Lemma 9
   (a) $S_3 \vdash \text{rtd}(S_3, F, F(\text{this}), d) \subseteq \text{rtd}(S_3, F, F(\text{this}), \sigma(e_0, d_0, d_{f_i}))$
18. By (12a) and Lemma 9
   (a) $S_3 \vdash \text{actd}(S_3, v) \subseteq \text{rtd}(S_3, F, F(\text{this}), \sigma(e_0, d_0, d_{f_i}))$
19. We now show that
   (a) $\text{rtd}(S_3, F, F(\text{this}), \sigma(e_0, d_0, d_{f_i})) = \text{rtd}(S_3, \varnothing, o, d_{f_i})$.
20. We do a case distinction on the shape of $d_{f_i}$.

Case $d_{f_i} = \text{this.b}$.
21. By (7c).
   (a) $T_f = \text{last}(d_0)/\text{same}[e_0, b$
22. By (5a) and (s-type),
   (a) $\Gamma \vdash T_f$
23. With (3c),
   (a) $e_0 = x$
   for some $x$.
24. Hence,
   (a) $T_f = \text{last}(d_0)/\text{same}[x, b$
25. By (23a), (2a) and (r-var),
   (a) $F(x) = o$
26. Hence by (rtd var),
   (a) $\text{rtd}(S_3, F, F(\text{this}), \text{last}(d_0)/\text{same}[x, b) = \text{last}(d_0)/\text{same}[o, b$
27. Let
   (a) $\text{actd}(S_3, o) = o, c$
   for some $o, c$.
28. By (rtd thisowner),
   (a) $\text{rtd}(S_3, \varnothing, o, d_{f_i}) = [c/\text{same}]o, b$
29. Note that
(a) $actd(S_3, o) = actd(S_1, o)$

30. With (1a) and (subset $\ast$),

(a) $c = \text{last}(d_0)$

31. Thus,

(a) $rtd(S_3, F, F(this), \sigma(e_0, d_0, d_f)) = rtd(S_3, \varnothing, o, d_f)$

closing the case.

Case $d_f = \text{owner}.b$

32. Let

(a) $actd(S_3, o) = a.o.c.$

for some $a, o, c$.

33. By (33a), (4a) and (subset refl),

(a) $o.o.c. = rtd(S_3, F, F(this), d_0)$

36. Let

(a) $d_0 = a.c.$

for some $a$.

37. Hence,

(a) $\sigma(e_0, d_0, d_f) = [c/same]a.b$

38. Thus by (35a),

(a) $rtd(S_3, F, F(this), \sigma(e_0, d_0, d_f)) = [c/same]a.o.b$

39. By (rtd thisowner),

(a) $rtd(S_3, \varnothing, o, d_f) = [c/same]a.o.b$

40. Hence,

(a) $rtd(S_3, F, F(this), \sigma(e_0, d_0, d_f)) = rtd(S_3, \varnothing, o, d_f)$

closing the case.

41. By (18a), (19a) and Lemma 5,

(a) $S_3 \vdash actd(S_3, v) \subseteq rtd(S_3, \varnothing, o, d_f)$

42. And finally,

(a) $\vdash S_3$

Case (r-invk)

1. Assume

(a) $S_0, F \vdash e.m(\overline{e}) \Rightarrow v, S_{n+2}$

(b) $\Gamma \vdash e.m(\overline{e}) : d_m C_m$

(c) $S_0, \Gamma \vdash F$

(d) $\text{this} \in \text{dom}(\Gamma)$

(e) $\vdash S_0$

2. By (1a) and (r-invk),

(a) $S_0, F \vdash e \Rightarrow o, S_1$

(b) $S_1, F \vdash e_1 \Rightarrow v_1, S_2$

(c) $\ldots$

(d) $S_n, F \vdash e_n \Rightarrow v_n, S_{n+1}$

(e) $S_1(o) = \{\ldots, C, \ldots\}$

(f) $\text{mbody}(m, C) = \overline{x.e}$

(g) $S_{n+1}, \{\tau \mapsto \overline{\tau}, \text{this} \mapsto o\} \vdash e_b \Rightarrow v, S_{n+2}$

3. By (1b) and (t-invk),

(a) $\Gamma \vdash e : d_e C_e$

(b) $\text{mtype}(m, C_e) = \overline{d} \overline{e} \rightarrow d_u C_u$

(c) $\Gamma \vdash \overline{e} : d_u C_u$

(d) $\Gamma \vdash d_u \overline{e} <: \sigma(e, d_e, \overline{d}) \overline{C}$

34
(e) (∃i. owner ∈ di) ⇒ isPrecise(di)
(f) dm Cm = σ(e, de, du) C
(g) Γ ⊢ dm Cm

4. We assume,
(a) C ⊢ du Cm m(d C T) ∈ { e_b }

5. Hence, by (T-METHODDECL),
(a) Im = { this → owner.same C, x → d C }
(b) Im ⊢ e_b : db C_b
(c) db C_b < d u C_u
(d) ⊢ d C
(e) ⊢ du C_u
(f) CT(C) = class C extends D {...}
(g) if mtype(m, D) = U → U_r, then d C u = U and d u C u = U_r
for some db, C_b, U, U_r.

6. We have to show
(a) v = null ∨ (actd(S_n+2, v) ⊆ rtd(S_n+2, F(this), d_m) ∧ class(S_3, v) <: C_m)

7. By (1e), (1a), (2a) – (2d), (3a), (3c), Lemmas 8, 7 and the induction hypotheses,
(a) S_0, Γ ⊢ F and ⊢ S_1
(b) S_1, Γ ⊢ F and ⊢ S_2
(c) actd(S_1, o) ⊆ rtd(S_1, F(this), d_c) ∧ class(S_1, o) <: C_c
(d) v_i = null ∨ (actd(S_i+1, v_i) ⊆ rtd(S_i+1, F(this), d_c) ∧ class(S_i+1, v_i) <: C_c

for 0 < i ≤ n + 1.

8. Let
(a) F_m = {this → o, x → v}

9. We show
(a) S_{n+1}, Γ_m ⊢ F_m

10. Let
(a) Γ_{m,this} = [this → owner.same C]
(b) F_{m,this} = [this → o]

11. At first we show,
(a) S_{n+1}, Γ_{m,this} ⊢ F_{m,this}

12. Let
(a) actd(S_{n+1}, o) = this_o.c

13. Hence
(a) rtd(S_{n+1}, F_m, F_m(this), owner.same) = this_o.c

14. By (SUBSET REFL),
(a) actd(S_{n+1}, o) ⊆ rtd(S_{n+1}, F_m, F_m(this), owner.same)

15. Hence, (11a), with (2e) and (T-STACK ∅).

16. We now show,
(a) S_{n+1}, Γ_{m,this}[x → d C] ⊢ F_{m,this}[x → v]

17. Let
(a) x_i ∈ x
(b) v_i = F_m(x_i)

18. Assume
(a) v_i ≠ null
otherwise apply (T-STACK NULL).

19. By (5a),
(a) Γ_m(x_i) = d_i C_i

20. By (7d) and (18a),
(a) \( \text{actd}(S_{i+1}, v_i) \subseteq \text{rtd}(S_{i+1}, F, F(\text{this}), d_i) \)
(b) \( \text{class}(S_{i+1}, v_i) <_c C_e \)
21. By (3d) and (s-class trans),
(a) \( \text{class}(S_{i+1}, v_i) <_c C_i \)
22. By (7d), (7e), (3d) and Lemma 9,
(a) \( \text{actd}(S_{i+1}, v_i) \subseteq \text{rtd}(S_{i+1}, F, F(\text{this}), \sigma(e, d_e, d_i)) \).
23. We now show that
(a) \( \text{rtd}(S_{i+1}, F, F(\text{this}), \sigma(e, d_e, d_i)) = \text{rtd}(S_{i+1}, F_m, F_m(\text{this}), d_i) \)
24. Case distinction on the shape of \( d_i \). There are three cases.

**Case** \( d_i = \text{this}.b \).
25. Hence,
(a) \( \sigma(e, d_e, d_i) = \lfloor \text{last}(d_e) \rfloor / \text{same}c.e.b \)
26. By (5d),
(a) \( e = x \)
for some \( x \).
27. Hence
(a) \( \sigma(e, d_e, d_i) = \lfloor \text{last}(d_e) \rfloor / \text{same}x.b \)
28. By (26a), (2a) and (r-var),
(a) \( F(x) = o \)
29. Hence by (rtd var),
(a) \( \text{rtd}(S_{i+1}, F, F(\text{this}), \lfloor \text{last}(d_e) \rfloor / \text{same}x.b) = \lfloor \text{last}(d_e) \rfloor / \text{same}o.b. \)
30. Note that,
(a) \( \text{actd}(S_{n+1}, o) = \text{this}_o.c \)
(b) \( F_m(\text{this}) = o \)
31. Hence by (rtd thisowner),
(a) \( \text{rtd}(S_{i+1}, F_m, F_m(\text{this}), d_i) = \lfloor c \rfloor / \text{same}o.b. \)
32. By (7a),
(a) \( c = \text{last}(d_e) \)
33. Hence by (25a), (29a) and (31a),
(a) \( \text{rtd}(S_{i+1}, F, F(\text{this}), \sigma(e, d_e, d_i)) = \text{rtd}(S_{i+1}, F_m, F_m(\text{this}), d_i) \)
closing the case.

**Case** \( d_i = \text{owner}.b \).
34. By (6a),
(a) \( \text{isPrecise}(d_e) \)
35. Note that
(a) \( \text{actd}(S_1, o) = \text{actd}(S_{n+1}, o) \)
(b) \( \text{actd}(S_{n+1}, o) = \text{this}_o.c. \)
36. Hence by (7c), (34a) and (subset refl),
(a) \( \text{rtd}(S_{n+1}, F, F(\text{this}), d_e) = \text{this}_o.c \)
37. Let
(a) \( d_e = a.c \)
38. Hence,
(a) \( \sigma(e, d_e, d_i) = \lfloor c \rfloor / \text{same}a.b \)
39. Hence by (36a),
(a) \( \text{rtd}(S_{i+1}, F, F(\text{this}), \sigma(e, d_e, d_i)) = \lfloor c \rfloor / \text{same}this_o.b \)
40. As \( F_m(\text{this}) = o \),
(a) \( \text{rtd}(S_{i+1}, F_m, F_m(\text{this}), d_i) = \lfloor c \rfloor / \text{same}this_o.b \)
41. Thus,
(a) \( \text{rtd}(S_{i+1}, F, F(\text{this}), \sigma(e, d_e, d_i)) = \text{rtd}(S_{i+1}, F_m, F_m(\text{this}), d_i) \)
closing the case.

**Case** \( d_i = x.b \).
42. By (5d),
By \((t\text{-domain var})\),

(a) \(\emptyset \vdash x : T\)

for some \(T\).

By \((t\text{-var})\),

(a) \(\emptyset(x) = T\)

which is a contradiction.

So we have shown

(a) \(S_{n+1}, \Gamma_m \vdash F_m\)

Note that

(b) \(\emptyset \vdash S_{n+1}\)

With \((5b)\), \((2g)\) and the induction hypothesis,

(a) \(v = null \lor (actd(S_{n+2}, v) \subseteq rtd(S_{n+2}, F_m, F_m(this), d_b) \land \text{class}(S_{n+2}, v) <_c C_b)\)

We have to show

(a) \(actd(S_{n+2}, v) \subseteq rtd(S_{n+2}, F, F_m(this), d_u)\)

Case distinction on the shape of \(d_u\). There are three cases.

Let

(a) \(d_u = d_{e_1}c_e\)

for some \(d_{e_1}, c_e\).

**Case** \(d_u = \text{this}.b\)

Hence,

(a) \(\sigma(e, d_e, d_u) = [c_e/\text{same}]e.b\)

By \((4g)\),

(a) \(e = x\)

for some \(x\).

Hence,

(a) \(\sigma(e, d_e, d_u) = [c_e/\text{same}]x.b\)

By \((57a)\), \((2a)\) and \((r\text{-var})\),

(a) \(F(x) = o\)

Hence,

(a) \(rtd(S_{n+2}, F, F(this), [c_e/\text{same}]x.b) = [c_e/\text{same}]o.b\)

By \((8a)\) and \((12a)\).

(a) \(rtd(S_{n+2}, F_m, F_m(this), d_u) = [c/\text{same}]o.b\)

By \((7c)\),

(a) \(c = c_e\)

Hence with \((58a)\) and \((3i)\),

\(c\).
(a) \( \text{rtd}(S_{n+2}, F_m, F_m(\text{this}), d_u) = \text{rtd}(S_{n+2}, F, F(\text{this}), d_m) \)

64. Hence by (52a).
(a) \( \text{actd}(S_{n+2}, v) \subseteq \text{rtd}(S_{n+2}, F, F(\text{this}), d_m) \)
closing the case.

Case \( d_u = \text{owner.}b \).

65. By (12a),
(a) \( \text{actd}(S_{n+2}, o) = \text{this}_o.c \)

66. By (7c)
(a) \( c_e = c \)

67. Hence by (rtd thisowner),
(a) \( \text{rtd}(S_{n+2}, F_m, F_m(\text{this}), d_m) = [c_e/\text{same}] \text{this}_o.b \)

68. By (3f),
(a) \( d_e = d_e[1] \).

69. By (7c), (65a) and (subset loose),
(a) \( \text{actd}(S_{n+2}, \text{this}_o) \subseteq \text{rtd}(S_{n+2}, F, F(\text{this}), d_m) \)

70. Hence,
(a) \( \text{this}_o[c_e/\text{same}]b \subseteq \text{rtd}(S_{n+2}, F, F(\text{this}), d_m) \)

71. Hence, by (67a) and Lemma 5,
(a) \( \text{actd}(S_{n+2}, \text{this}_o) \subseteq \text{rtd}(S_{n+2}, F, F(\text{this}), d_m) \)
closing the case.

Case \( d_u = x.b \). This is a contradiction to (5e).

II) This is already shown above by (47b).

Case (r-new)

I) 1. Assume
(a) \( S_0, F \vdash \text{new} \ d \ C \Rightarrow o, S_1 \)
(b) \( F \vdash \text{new} \ d \ C : \ d \ C \)
(c) \( \vdash S_0 \)

2. By (1a) and (r-new),
(a) \( \text{rd} = \text{rtd}(S_0, F, F(\text{this}), d) \)
(b) \( \text{fields}(C) = T \ l \)
(c) \( S_1 = S_0[o \mapsto (\text{rd}, C, \text{null})] \)
(d) \( \text{null} \) = \( l \)

3. By (1b) and (t-new),
(a) \( C \in \text{dom}(CT) \)

4. Note that,
(a) \( \text{actd}(S_0, F(\text{this})) = \text{actd}(S_1, F(\text{this})) \)

5. Hence,
(a) \( \text{rtd}(S_0, F, F(\text{this}), d) = \text{rtd}(S_1, F, F(\text{this}), d) \)

6. By (2a) and (subset refl),
(a) \( \text{actd}(S_1, o) = \text{rtd}(S_0, F, F(\text{this}), d) \)

7. Hence by (subset refl),
(a) \( \text{actd}(S_1, o) \subseteq \text{rtd}(S_1, F, F(\text{this}), d) \)

II) 8. By (3a) and (t-class decl),
(a) \( F \vdash C \)

9. With (1c), (2b), (2c), (2d) and (t-store object),
(a) \( \vdash S_1 \)

38
A.2 Proof of Theorem

Helper Lemmas

Before we prove the Theorem we show some helpful lemmas.

Lemma 10. This lemma is crucial, as it connects the well-formedness of domains and the domain subset relation with the accessibility of values.

\[
S \vdash \text{actd}(S, v_1) \subseteq \text{rtd}(S, F, v_2, d) \land \Gamma \vdash d \land S \vdash F \land \Gamma, S \vdash F \implies S \vdash v_2 \implies \text{actd}(S, v_1)
\]

Proof. We assume \(v_1 \neq \text{null}\), otherwise \(S \vdash v_2 \implies \text{actd}(S, v_1)\) is directly given by (A-NULL). We do an induction on the length of \(d\): \(n = |d|\).

Induction Base \(n = 2\).

1. Assume
   (a) \(S \vdash \text{actd}(S, v_1) \subseteq \text{rtd}(S, F, v_2, d)\)
   (b) \(\Gamma \vdash d\)
   (c) \(S \vdash F\)
   (d) \(\Gamma, S \vdash F\)

2. By (1b) and (T-DOMAIN *) there are two cases:
   Case \(d = a.c\), where \(a \in \{\text{owner}, \text{this}\}\).
     Case \(a = \text{this}\).  
     3. By (SUBSET REFL),  
        (a) \(\text{actd}(S, v_1) = v_2.c\).  
     4. By (A-OWN)  
        (a) \(S \vdash v_2 \implies v_2.c\).
   Case \(a = \text{owner}\).  
     5. Let  
        (a) \(\text{actd}(S, v_2) = v_o.c_o\)  
     6. Hence,  
        (a) \(\text{actd}(S, v_1) = v_o[c_o/same]c\)  
     7. Note that  
        (a) \(\text{owner}(S, v_2) = v_o\)  
     8. Hence, by (A-OWNER),  
        (a) \(S \vdash v_2 \implies \text{actd}(S, v_1)\)
   Case \(d = x.\text{boundary}\).  
     9. By (1b) and (T-DOMAIN VAR),  
        (a) \(\Gamma \vdash x : d\)  
     10. By (1b),  
        (a) \(F(x) = v_x\) for some \(v_x\).  
     Hence,  
        (a) \(\text{rtd}(S, F, v_2, d) = v_x.\text{boundary}\)  
     12. With (1a) and (SUBSET REFL),  
        (a) \(\text{actd}(S, v_1) = v_x.\text{boundary}\).  
     13. By (1c),  
        (a) \(v_x = \text{null}\) or  
        (b) \(S \vdash v_2 \implies v_x\).
   Case \(v_x = \text{null}\).  
     14. Hence,  
        (a) \(\text{act}(S, v_1) = \text{null}.\text{boundary}\)
     15. By (A-NULL),
\( S \vDash v_2 \rightarrow actd(S, v_1) \).

**Case** \( S \vDash v_2 \rightarrow v_x \).

1. Hence by (A-boundary),
   \( S \vDash v_2 \rightarrow v_x \) boundary
2. Thus by (12a),
   \( S \vDash v_2 \rightarrow actd(S, v_1) \)

**Induction Step** \( n = m \).

1. Assume,
   \( S \vdash actd(S, v_1) \subseteq rtd(S, F, v_2, d) \land \Gamma \vdash d \land S \vdash F \land \Gamma, S \vdash F \implies S \vdash v_2 \rightarrow actd(S, v_1) \)
   for all \( d \) with \( |d| < m \).
2. Let
   \( d = a.b \)
   with \( |d| = m \) and \( m > 2 \).
3. Assume,
   \( S \vdash actd(S, v_1) \subseteq rtd(S, F, v_2, d) \)
   \( \Gamma \vdash d \)
   \( S \vdash F \)
   \( \Gamma, S \vdash F \implies S \vdash v_2 \rightarrow actd(S, v_1) \)
4. By (3a) and (t-domain *),
   \( d = a.b_1 \) boundary
   \( \Gamma \vdash a.b_1 \)
   for some \( b_1 \).
5. By (3a), (1a) and (subset loose),
   \( actd(S, v_1) = v_o \) boundary
   \( S \vdash actd(S, v_o) \subseteq rtd(S, F, v_2, a.b_1) \).
   for some \( v_o \).
6. Clearly,
   \( |a.b_1| < m \)
7. Hence by (4b), (5b), (3c), (3d) and the induction hypothesis (1a),
   \( S \vdash v_2 \rightarrow v_o \)
8. By (A-boundary),
   \( S \vdash v_2 \rightarrow v_o \) boundary
9. Thus by (5a),
   \( S \vdash v_2 \rightarrow actd(S, v_1) \)

**Lemma 11.** This lemma states that a bigger state in our store order, does not change the accessibility of values.

\[ S_0 \leq S_1 \land S_0 \vDash o \implies actd(S_0, v) \implies S_1 \vDash o \implies actd(S_1, v) \]

**Proof.**

1. Assume
   \( S_0 \leq S_1 \)
   \( S_0 \vDash o \rightarrow actd(S_0, v) \)
   for some \( S_0, S_1, o, v \).
2. By (1a),
   \( \forall o \in dom(S_0). actd(S_0, o) = actd(S_1, o) \)
3. The remainder of the proof is by structural induction on the accessibility rules (A+-).

**Case** (A-own)
4. Hence,
   (a) $S_0 \vDash o \rightarrow o.c$
   (b) $actd(S_0, v) = o.c$

5. By (A-OWN),
   (a) $S_1 \vDash o \rightarrow o.c$

6. With (2a),
   (a) $S_1 \vDash o \rightarrow actd(S_1, v)$

Case (A-BOUNDARY)

7. Hence,
   (a) $actd(S_0, v) = v_2.boundary$
   (b) $S_0 \vDash o \rightarrow v_2.boundary$
   (c) $S_0 \vDash o \rightarrow actd(S_0, v_2)$
   for some $v_2$.

8. By (1a), (7a) and the induction hypothesis,
   (a) $S_1 \vDash o \rightarrow actd(S_1, v_2)$

9. Hence, by (A-BOUNDARY),
   (a) $S_1 \vDash o \rightarrow v_2.boundary$

10. This with (7a) and (2a),
    (a) $S_1 \vDash o \rightarrow actd(S_1, v)$

Case (A-OWNER)

11. Hence,
    (a) $actd(S_0, v) = v_2.c$
    (b) $S_0 \vDash o \rightarrow v_2.c$
    (c) $owner(S_0, o) = v_2$
    for some $v_2.c$.

12. By (2a),
    (a) $owner(S_0, o) = owner(S_1, o)$

13. Hence, by (A-OWNER),
    (a) $S_1 \vDash o \rightarrow v_2.c$

14. Thus, by (11a) and (2a),
    (a) $S_1 \vDash o \rightarrow actd(S_1, v)$

Lemma 12.

$S_0 \vDash F \land S_0, F \vdash e \Rightarrow v, S_1 \implies S_1 \vDash F$

Proof. We have to show $\forall v \in \text{ran}(F). v = null \lor S_1 \vDash F(this) \rightarrow v$. By Lemma 7 we get $S_0 \leq S_1$. With the assumption $S_0 \vDash F$ and Lemma 11 it follows $S_1 \vDash F$.

Proof of Theorem [2]

Proof.

1. Assume,
   (a) $this \in \text{dom}(F)$
   (b) $\vdash S_0$
   (c) $\Gamma, S_0 \vDash F$
   (d) $\vdash S_0$
   (e) $S_0 \vDash F$
   for some $\Gamma, S_0, F$.

2. The remainder of the proof is by structural induction on the reduction rules of the operational semantics.

Case (R-VAR)

41
3. Assume,
   (a) $S_0, F \vdash x \Rightarrow v, S_1$
   for some $x, v, S_1$.
4. By (3a) and (r-var),
   (a) $F(x) = v$
   (b) $S_0 = S_1$
5. By (4b),
   (a) $S_0 \vdash F(this) \rightarrow v$
6. By (4b) and (1d),
   (a) $\vdash S_1$

Case (r-field)
7. Assume,
   (a) $S_0, F \vdash e_0 . f_i \Rightarrow v_i, S_1$
   (b) $\Gamma \vdash e_0 . f_i : d C$
   for some $e_0, f_i, v_i, S_1, d, C$.
8. By (7a) and (r-field),
   (a) $S_0, F \vdash e_0 \Rightarrow o, S_1$
   (b) $S_1(o) = \langle rd, C, v \rangle$
9. By (7b) and (t-field),
   (a) $\Gamma \vdash o$
   (b) $\Gamma \vdash e_0 : d_0 C_0$
   for some $d_0, C_0$
10. By (9a), (8a) and the induction hypothesis,
    (a) $\vdash S_1$
11. By the assumptions and Theorem 1
    (a) $v_i = \text{null}$, or
    (b) $S_1 \vdash \text{actd}(S_1, v_i) \subseteq \text{rtd}(S_1, F, F(this), d)$
12. Assume,
    (a) $v_i \neq \text{null}$
    otherwise we are done.
13. By (12a), (10a), Lemma 12 and Lemma 8
    (a) $S_1 \vdash F$
    (b) $\Gamma, S_1 \vdash F$
14. With (12b) and Lemma 10
    (a) $S_1 \vdash F(this) \rightarrow v_i$
    closing the case.

Case (r-let)
15. Assume,
    (a) $S_0, F \vdash \text{let } x = e_0 \text{ in } e_1 \Rightarrow v_1, S_2$
    (b) $\Gamma \vdash \text{let } x = e_0 \text{ in } e_1 : d C$
    for some $x, e_0, e_1, v_1, S_2, d, C$.
16. By (15a) and (r-let),
    (a) $S_0, F \vdash e_0 \Rightarrow v_0, S_1$
    (b) $S_1, F[x \mapsto v_0] \vdash e_1 \Rightarrow v_1, S_2$
17. By (15b) and (t-let),
    (a) $\Gamma \vdash e_0 : d_0 C_0$
    (b) $x \notin \text{dom}(\Gamma)$
    (c) $\Gamma[x \mapsto d_0 C_0] \vdash e_1 : d C$
18. By (17a), (16a) and the induction hypothesis,
    (a) $\vdash S_1$
    (b) $v_0 = \text{null}$ ∨ $S_1 \vdash F(this) \rightarrow v_0$
19. By Theorem 1
(a) ⊢ S₁

20. Let
  (a) F₂ = F[x → v₀]
  (b) Γ₂ = Γ[x → d₀ C₀]

21. With the same argumentation as in case (r-let) of the proof of Theorem 1, it follows with (14a),
  (a) Γ₂, S₁ ⊢ F₂

22. By Lemma 7
  (a) S₀ ≤ S₁

23. By (18a) and (19a),
  (a) S₁ ⊢ F₂

24. With (16b), (17c) and the induction hypothesis,
  (a) ⊢ S₂
  (b) S₂ ⊩ F(this) → v₁
  which had to be shown.

Case (r-fieldup)

25. Assume,
  (a) S₀, F ⊢ e₀, fᵢ = e₁ ⇒ v, S₃
  (b) Γ ⊢ e₀, fᵢ = e₁ : d C
  for some e₀, fᵢ, e₁, v, S₃, d, C.

26. By (25a) and (r-fieldup),
  (a) S₀, F ⊢ e₀ ⇒ o, S₁
  (b) S₁, F ⊢ e₁ ⇒ v, S₂
  (c) S₂(o) = ⟨rd, D, v⟩
  (d) S₃ = S₂[o ⇒ ⟨rd, D, v/vᵢ⟩]
  for some o, S₁, S₂, rd, D, v.

27. By (25b) and (r-fieldup),
  (a) Γ ⊢ e₀ : d₀ C₀
  (b) fields(C₀) = ⟨C₀[G₀ F₁]⟩
  (c) Γ ⊢ e₁ : d C
  for some d₀, C₀, d, C.

28. By (27a), (26a) and the induction hypothesis,
  (a) ⊢ S₁
  (b) S₁ ⊢ F(this) → o

29. By Theorem 1
  (a) ⊢ S₁

30. By (13a), (13b), Lemma 8 and 12
    (a) S₁, Γ ⊢ F
    (b) S₁ ⊢ F

31. With (28a), (27c), (26b) and the induction hypothesis,
    (a) ⊢ S₂
    (b) v = null ∨ S₂ ⊢ F(this) → v

32. By Theorem 1
    (a) v = null ∨ actd(S₃, v) ⊆ rtd(S₃, F, F(this), d)
    (b) ⊢ S₃

33. By (27c),
    (a) Γ ⊢ d

34. With the same arguments like in the proof of Theorem 1 of case (r-fieldup),
    (a) S₃, Γ ⊢ F

35. By Lemma 12
    (a) S₂ ⊢ F

36. As
∀o ∈ dom(S₂), actd(S₂, o) = actd(S₃, o)
(b) dom(S₂) = dom(S₃)

37. it follows
(a) S₁ ⊩ F
38. Hence with (32a) and Lemma 10
(a) v = null ∨ S₃ ⊩ F(this) → v

39. We now show
(a) ⊩ S₃
40. By (31a) we have
(b) ⊩ S₂. The only difference between S₂ and S₃ is that the value vᵢ of field fᵢ of object o is replaced by v. So we have to show
(a) v = null ∨ S₃ ⊩ o → v

41. Assume (a) v = null otherwise we are done.
42. By (32b),
(a) S₃ ⊩ actd(S₃, v) ⊆ rtd(S₃, ∅, o, dfᵢ)
43. By (27b) and (t-class),
(a) ⊩ dfᵢ.
44. Note that
(a) S₃ ⊩ ∅
(b) S₃ ⊩ ∅, S₃ ⊩ ∅
45. With (12a) and Lemma 10
(a) S₃ ⊩ o → v
46. Thus,
(a) ⊩ S₃
closing the case.

Case (r-inv)
47. Assume,
(a) S₀, F ⊩ e.m(π) ⇒ v, Sₙ₊₂
(b) Γ ⊩ e.m(π) : d C
for some e, m, π, v, Sₙ₊₂.
48. By (47a) and (r-inv),
(a) S₀, F ⊩ e ⇒ o, S₁
(b) S₁, F ⊩ e₁ ⇒ v₁, S₂
(c) ... 
(d) Sₙ, F ⊩ eₙ ⇒ vₙ, Sₙ₊₁
(e) S₁(o) = ⟨...C,...⟩
(f) mbody(m, C) = π.eₐ
(g) Sₙ₊₁, {this → o, π → π} ⊩ eₐ ⇒ v, Sₙ₊₂
49. By (47a) and (t-inv),
(a) Γ ⊩ e : dₐ Cₐ
(b) mtype(m, Cₐ) = d C → dₐ Cₐ
(c) Γ ⊩ π : dₐ Cₐ
(d) Γ ⊩ dₐ Cₐ ≺ σ(e, dₐ) d C
50. By Theorem 1,
(a) ⊩ Sₙ₊₂
(b) Sₙ₊₂ ⊩ actd(Sₙ₊₂, v) ⊆ rtd(Sₙ₊₂, F, F(this), d)
51. By (48a)–(48e), successively applying the induction hypothesis, Theorem 1, Lemma 12, Lemma 8 and Lemma 11 it follows for all 1 ≤ i < n + 2
(a) Sᵢ, Γ ⊩ F
(b) Sᵢ ⊩ F
(c) \( \vdash S_i \)  
(d) \( \models S_i \)  
(e) \( v_i = \text{null} \lor S_{n+1} \models F(\text{this}) \rightarrow v_i \)  
(f) \( S_{n+1} \models F(\text{this}) \rightarrow o \)

52. Let 
(a) \( F_m = \{ \text{this} \mapsto o, \tau \mapsto \tau \} \)  
(b) \( S_{n+1} \models F_m \)

53. By (51e) and (51f), 
(a) \( S_{n+1} \models F_m \)

54. With the same definitions and arguments of case (r-fieldup) of the proof of Theorem 1, 
(a) \( S_{n+1}, \Gamma_m \models F_m \)

55. With (48g) and the induction hypothesis, 
(a) \( \models S_{n+2} \)

56. By (1c) and Lemma 12, 
(a) \( S_{n+2} \models F \)

57. By Lemma 8, 
(a) \( S_{n+2}, \Gamma \models F \)

58. By (47b) and Lemma 1, 
(a) \( \Gamma \models d \).

59. Thus with (50b) and Lemma 10, 
(a) \( S_{n+2} \models F(\text{this}) \rightarrow v \)  
which had to be shown.

Case (r-new)  
60. Assume,  
(a) \( S_0, F \vdash \text{new } d \ C \Rightarrow o, S_1 \)  
(b) \( \Gamma \vdash \text{new } d \ C : d \ C \)  
for some \( d, C, o, S_1 \).

61. By (60a) and (r-new), 
(a) \( rd = rtd(S_0, F, F(\text{this}), d) \)  
(b) \( \text{fields}(C) = T \)  
(c) \( o \notin \text{dom}(S_0) \)  
(d) \( S_1 = S_0[o \mapsto \{rd, C, \text{null}\}] \)  
(e) \( \text{null} = 0 \)  
for some \( T, f \).

62. By (60b) and (t-new), 
(a) \( d = a.c \)  
(b) \( a \in \{ \text{this}, \text{owner} \} \)  
(c) \( C \in \text{dom}(CT) \)  
for some \( a, c \).

63. Let 
(a) \( \text{actd}(S_0, F(\text{this})) = v_{\text{this}, c_{\text{this}}} \)

64. Note that 
(a) \( \text{actd}(S_1, o) = rd \)

65. We do a case distinction on the shape of \( a \).

Case \( a = \text{this} \).  
66. Hence by (rtd thisowner), 
(a) \( rd = [c_{\text{this}}/\text{same}]F(\text{this}).c. \)

67. Thus by (a-own), 
(a) \( S_1 \models F(\text{this}) \rightarrow \text{actd}(S_1, o) \)

Case \( a = \text{owner} \).  
68. Hence, 
(a) \( rd = [c_{\text{this}}/\text{same}]v_{\text{this}}.c \)

69. Note that 
(a) \( \text{owner}(S, F(\text{this})) = v_{\text{this}} \)

70. Hence, by (a-owner), 
(a) \( S_1 \models F(\text{this}) \rightarrow \text{actd}(S_1, o) \)

45