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 X wants to access the public domains of an arbitrary number of other objects, which often happens in observer scenarios. To overcome this restriction, we developed so-called *loose* domains which abstract over several *precise* domains. That is, similar to the relation between 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 [21, 11, 19, 10, 7] 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 [20]. 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 [2] 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 *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,

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

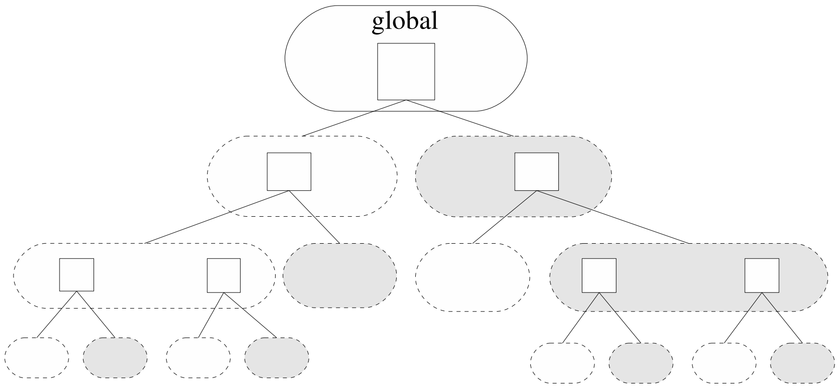

Fig. 1: The ownership and containment relation of objects and domains form a tree rooted by a global domain. Solid rectangles represent objects, dashed rounded rectangles represent domains, where gray rectangles are local domains and white ones are boundary domains. An edge from an object X to a domain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

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 [2]: objects are grouped into distinct *domains*, domains are *owned* by objects, and every object belongs to exactly one domain. Within this paper, we simplify the ownership domain approach of [2] in two ways: Every object owns exactly two domains, namely a *local* domain and a *boundary* domain. Thus, SLOD has no domain declarations. In addition, access permissions between domains are hard-wired, so SLOD needs no *link* declarations.

2.1 Accessibility Properties

Objects that are in the local domain of an object X, belong to the representation of X and are encapsulated. Objects of the boundary domain of X are objects that are accessible from the outside of X, but at the same time are able to access the representation objects of X. In terms of 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 can *not* be accessed, because this complementary relation leads us to a generalization of the owners-as-dominators property. The *domain subtree* of an object X consists of X and, recursively, of all objects that are owned by an object in the domain subtree. An object is *outside* of an object X if it does not belong to the domain subtree of X. The *boundary* of X is the set of objects consisting of X and, recursively, of all objects in the boundary domains owned by an object in the boundary of X. An object is *inside* of X if it belongs to the domain subtree of X, but not to its boundary. With these definitions, SLOD guarantees the following property:

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

This *boundary-as-dominators* property is a generalization of the owners-as-dominators property, as the owners-as-dominators property for an object X can be enforced in SLOD by putting no objects into the boundary domain of X, 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 `this.local T` can only hold references to `T`-objects

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

Fig. 2: Syntax of domain annotations in SLOD.

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 [2], the `Iter` class would need a domain parameter to represent the domain of the `Node` objects.

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

The applications of classes `LinkedList` and `Iter` in `Main` demonstrate further interesting features of SLOD. The variable `it`, for example, is declared with domain annotation `l.boundary`. As `l` is a `final` variable, this is a precise domain annotation. It represents the boundary domain of the `LinkedList` object referenced by variable `l`. Such domain annotations are also supported by the ownership domain approach in [2].

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 = next;
  }
}

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

Fig. 3: A linked list with iterators.

2.3 Loose Domains

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

To demonstrate the enhanced expressivity of loose domains, we use a slightly modified version of an example given in [2] (see Figure 4). It is a model-view system. `Model` objects allow to register `Listener` objects. When an event happens at the model, the model notifies all registered listener objects by calling the method `update(int)`. `View` objects have a state that is updated whenever one of its listeners is notified. Method `listener()` creates new `ViewListener` instances as boundary objects of their view. The example is a simplified version of the observer pattern [13] 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 [2], 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 { ... }
```

```
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) [16] and ClassicJava [12] and is also inspired by several flavors of these type systems which already incorporate ownership information [11, 10, 2, 23].

Like other Java formalizations, we only consider the core feature of the full Java language [14]. 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 parameterization 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 [16]. A bar indicates a sequence: $\overline{L} = L_1, L_2, \ldots, L_n$, where the length is defined as $|\overline{L}| = n$. Similar, $\overline{T}\ \overline{f}$; is equal to $T_1\ f_1; T_2\ f_2; \ldots; T_n\ f_n$. If there is some sequence $\overline{x}$, we write x_i for any element of $\overline{x}$. The empty sequence is donated by $\bullet$.

We use the meta variables P to range over programs; L to range over class declarations; 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

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

Fig. 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 $\langle \overline{L}, C, e \rangle$ of a list of class declarations $\overline{L}$, a class name C, which must exist in $\overline{L}$, 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 $\overline{T}\ \overline{f}$ and a sequence of method declarations $\overline{M}$. In SLODJ every class declaration must have a super class, which can be `Object`. Note that classes have no constructors. Objects are created with all fields initialized to *null*. A method declaration M consists of a result type T, a sequence of formal parameters $\overline{T}\ \overline{x}$ 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 `let` x = e_1 `in` e_2, or a method invocation $e.m(\overline{e})$. 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)
$$\frac{}{\mathit{fields}(\texttt{Object}) = \bullet}$$

(FIELDS NORMAL)
$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}\ \overline{f};\ \overline{M}\} \qquad \mathit{fields}(D) = \overline{U}\ \overline{g}}{\mathit{fields}(C) = \overline{U}\ \overline{g},\ \overline{T}\ \overline{f}}$$

Method type lookup:

(M-TYPE DECL)
$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}\ \overline{f};\ \overline{M}\} \quad U\ m(\overline{U}\ \overline{x})\{\ e\ \} \in \overline{M}}{\mathit{mtype}(m, C) = \overline{U} \rightarrow U}$$

(M-TYPE INHERIT)
$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}\ \overline{f};\ \overline{M}\} \quad m \notin \overline{M}}{\mathit{mtype}(m, C) = \mathit{mtype}(m, D)}$$

Method body lookup:

(M-BODY DECL)
$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}\ \overline{f};\ \overline{M}\} \quad U\ m(\overline{U}\ \overline{x})\{\ e\ \} \in \overline{M}}{\mathit{mbody}(m, C) = \overline{x}.e}$$

(M-BODY INHERIT)
$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}\ \overline{f};\ \overline{M}\} \quad m \notin \overline{M}}{\mathit{mbody}(m, C) = \mathit{mbody}(m, D)}$$

Precise domain annotations:

(ISPRECISE)
$$\frac{}{\mathit{isPrecise}(a.c)}$$

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

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 [16].

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 $\overline{U} \to U$ of a class method. $\overline{U}$ 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 ($m \notin \overline{M}$), it is searched in the super class. If there are no method declarations for a certain method, the function is undefined. In particular, $mtype(m, \texttt{Object})$ is always undefined. The *mbody* function is similar to the *mtype* function, but returns the method body $\overline{x}.e$ of a method, where $\overline{x}$ 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 Γ is a finite mapping from variables to types.

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

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

For any program $\langle \overline{L}, C, e \rangle$, 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):

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

Well-formed Environments and Types:

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

Well-formed Class Names:

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

Well-formed Domains:

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

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 $<:_c$ 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 $<:_c$ is the reflexive, transitive closure of the direct subclass relation given by the class declarations. The relation $<:_d$ is defined on domain annotations. Reflexivity is given by (S-DOMAIN REFL). The rule (S-DOMAIN VAR) states that a domain with a variable as owner, $x.b$, is a subdomain of a domain $d_0.b$ if x is typed with a domain d_x, and that domain is a subdomain of d_0. Note that $<:_d$ is transitive, which we prove later. The subtype relation $<:$ is defined by the relations $<:_c$ and $<:_d$. It is reflexive and transitive.

Subclassing:

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

Subdomaining and Subtyping:

$$\frac{\Gamma \vdash d}{\Gamma \vdash d <:_d d}\ (\text{S-DOMAIN REFL}) \qquad \frac{\begin{array}{cc}\Gamma \vdash x.b & \Gamma \vdash d_0.b \\ \Gamma \vdash x : d_x\ C & \Gamma \vdash d_x <:_d d_0\end{array}}{\Gamma \vdash x.b <:_d d_0.b}\ (\text{S-DOMAIN VAR}) \qquad \frac{\begin{array}{cc}\Gamma \vdash d_1\ C & \Gamma \vdash d_2\ D \\ \Gamma \vdash d_1 <:_d d_2 & \Gamma \vdash C <:_c D\end{array}}{\Gamma \vdash d_1\ C <: d_2\ D}\ (\text{S-TYPE})$$

Fig. 10: Subclassing, Subdomaining, and Subtyping

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

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

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

(T-VAR). The type of a local variable x is the type to which x is mapped in the type environment Γ.

(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.

(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 `this` appears in d_i, the receiver expression e_0 has to be a local variable. This is enforced by $\Gamma \vdash T <: T_f$ 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 `owner` appears in d_i, the domain annotation d 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.

(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.

(T-LET). The type of a let expression `let` $x = e_0$ `in` e_1 is the type of expression e_1 in type environment Γ extended by variable x mapping to the type of e_0. A reassigning of *this* is prevented by rule (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 Γ.

Well-formed Method Declarations:

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

Well-formed Class Declarations and Program Typing:

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

Fig. 11: Method, Class and Program Typing Rules.

Expression Typing:

$$\frac{\begin{array}{c}\Gamma \vdash \diamond \\ \Gamma(x) = T\end{array}}{\Gamma \vdash x : T} \ (\text{T-VAR}) \qquad \frac{\begin{array}{c}\Gamma \vdash e : d\ C \qquad \mathit{fields}(C) = \overline{d}\ \overline{C}\ \overline{f} \\ T_f = \sigma(e, d, d_i)\ C_i \qquad \Gamma \vdash T_f\end{array}}{\Gamma \vdash e.f_i : T_f} \ (\text{T-FIELD})$$

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

$$\frac{\begin{array}{c}\Gamma \vdash e : d\ C \\ \mathit{mtype}(m, C) = \overline{d}\ \overline{C} \to d_u\ C_u \qquad \Gamma \vdash \overline{e} : \overline{U} \qquad \overline{T_s} = \sigma(e, d, \overline{d})\ \overline{C} \qquad \Gamma \vdash \overline{U} <: \overline{T_s} \\ (\exists i.\texttt{owner} \in T_{s_i}) \Rightarrow \mathit{isPrecise}(d) \qquad U_m = \sigma(e, d, d_u)\ C_u \qquad \Gamma \vdash U_m\end{array}}{\Gamma \vdash e.m(\overline{e}) : U_m} \ (\text{T-INVK})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : T_0 \qquad x \notin \mathit{dom}(\Gamma) \\ \Gamma[x \mapsto T_0] \vdash e_1 : T_1 \qquad \Gamma \vdash T_1\end{array}}{\Gamma \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : T_1} \ (\text{T-LET}) \qquad \frac{\begin{array}{c}\Gamma \vdash a.c\ C \\ a \in \{\texttt{this}, \texttt{owner}\}\end{array}}{\Gamma \vdash \texttt{new}\ a.c\ C : a.c\ C} \ (\text{T-NEW})$$

Fig. 12: Expression Typing Rules.

(T-NEW). A new expression `new` $a.c\ C$ has type $a.c\ C$. The domain $a.c$ has the restriction $a \in \{\texttt{this}, \texttt{owner}\}$. Thus it follows that $a.c$ is precise and that the owner of $a.c$ must be `this` or `owner`. So an object can only create new objects in domains that are owned by itself or by its owner.

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

Fig. 13: Dynamic Entities of SLODJ.

Method Declarations, Class Declarations and Program Typing. Figure 11 shows rules for method and class declarations as well as programs. A program $\langle \overline{L}, C, e \rangle$ is typed by typing e in the type environment mapping *this* to `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 $\varnothing \vdash \overline{T}$ and $\varnothing \vdash T_r$ 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 *null*. A runtime domain is a tuple of a value v and a domain tail b. An object state s is a triple $\langle o.b, C, \overline{v} \rangle$, consisting of a precise runtime domain $o.c$ with an object o as owner, a class name C, and a list of field values $\overline{v}$. A store S is a finite mapping from objects o to object states s. A stack frame F is a finite mapping from variable names x to values v.

4.1 Runtime Domains

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 *null*, and b is a sequence of `boundary`, `local` and `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

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

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

$$\frac{S_0, F \vdash e \Rightarrow o, S_1 \qquad S_1(o) = \langle rd, C, \overline{v} \rangle}{S_0, F \vdash e.f_i \Rightarrow v_i, S_1} \ (\text{R-FIELD})$$

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

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

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

Fig. 14: SLODJ Evaluation Rules.

hence a real implementation need not store the actual domain in the object state. Note that objects always belong to runtime domains with objects as owners. We need *null* as owners for runtime domains only to give *null* an owning domain.

4.2 Evaluation Rules

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 to the receiver object o and new store S_1. Then the value v_i of the corresponding field f_i 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_i that corresponds to field f_i 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 $\overline{e}$. Finally, the body expression of method m, e_b, is evaluated with a stack frame mapping *this* to o and the formal parameter names $\overline{x}$ mapping to the evaluated values $\overline{v}$. 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 *null*.

Actual Domain:

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

Owner and Class:

$$\frac{actd(S, v_1) = v_2.c}{owner(S, v_1) = v_2}\ (\text{OWNER}) \qquad \frac{S(o) = \langle \ldots, C, \ldots\rangle}{class(S, o) = C}\ (\text{CLASS})$$

Domain Subset:

$$\frac{}{S \vdash null.c \subseteq v.b}\ (\text{SUBSET NULL}) \qquad \frac{}{S \vdash o.b \subseteq o.b}\ (\text{SUBSET REFL}) \qquad \frac{S \vdash actd(S, o_1) \subseteq o_2.b_2}{S \vdash o_1.b_1 \subseteq o_2.b_2.b_1}\ (\text{SUBSET LOOSE})$$

Runtime Domain:

$$\frac{a \neq x \qquad actd(S, v_1) = v_2.c}{rtd(S, F, v_1, a.b) = [v_1/\mathtt{this}, v_2/\mathtt{owner}, c/\mathtt{same}]\ a.b}\ (\text{RTD THISOWNER})$$

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

Fig. 15: Auxiliary Functions.

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 $o_1.b_1$ is a subset of a runtime domain $o_2.b_2.b_1$ if and only if the actual domain of o_1 is subset of $o_2.b_2$.

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

$$rtd : \textbf{Store} \times \textbf{StackFrame} \times \textbf{Value} \times \textbf{Domain} \rightharpoonup \textbf{RuntimeDomain}$$

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:

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

(T-STORE OBJECT)

$$\frac{\begin{array}{c}\vdash S_0 \qquad \vdash C \qquad S_1 = S_0[o \mapsto \langle \ldots, C, \overline{v} \rangle] \qquad fields(C) = \overline{d}\ \overline{C}\ \overline{f} \\ |\overline{v}| = |\overline{f}| \qquad \forall v_i \in \overline{v}.\ v_i = null \lor S_1 \vdash actd(S_1, v_i) \subseteq rtd(S_1, \varnothing, o, d_i) \land class(S_1, v_i) <:_c C_i\end{array}}{\vdash S_1}$$

Stack Frame Well-Formedness:

$$\frac{}{S, \varnothing \vdash \varnothing} \ (\text{T-STACK } \varnothing) \qquad \frac{S, \Gamma \vdash F}{S, \Gamma[x \mapsto T] \vdash F[x \mapsto null]} \ (\text{T-STACK NULL})$$

$$\frac{S, \Gamma \vdash F \qquad class(S, o) <:_c C \qquad S \vdash actd(S, o) \subseteq rtd(S, F, F(this), d)}{S, \Gamma[x \mapsto d\ C] \vdash F[x \mapsto o]} \ (\text{T-STACK VAR})$$

Fig. 16: Store and Stack Frame Well-Formedness

owner is `this` or `owner` (RTD THISOWNER). In the former case, the variable x is simply replaced by its value in F, in the latter case, `this` is replaced by the receiver value v_1 and `owner` by the corresponding owner value. In both cases, `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 rtd function. After the translation we check that the resulting runtime domain is a superset of the runtime domain of the object. Note that in the case where the value is $null$ we need not check anything.

For the soundness proof we need additional properties for stores and stack frames, which are shown in Figure 16. 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 Γ

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-STORE $*$), and that the types of values of a stack frame F correspond to the types recorded in the type environment Γ (T-STACK $*$).

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

Proof. The proof is by structural induction on the reduction rules of the operational semantics. It can be found in 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 $P = \langle \overline{L}, C, e \rangle$, with $\vdash P : d_e\ C_e$. The initial configuration is given by $\Gamma_{init} = \{this \mapsto \texttt{owner.same}\ C\}$, $S_{init} = \{o \mapsto \langle o.\texttt{boundary}, C, \overline{null} \rangle\}$, $F_{init} = \{this \mapsto o\}$. By (T-PROG) we get $\Gamma_{init} \vdash e : d_e\ C_e$. We have to show the following conditions:

1. $this \in dom(F_{init})$
2. $\vdash S_{init}$
3. $S_{init}, \Gamma_{init} \vdash F_{init}$

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

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 iff

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

Accessibility:

$$\frac{}{S \Vdash o \longrightarrow o.c}\ (\textsc{a-own}) \qquad \frac{S \Vdash o_1 \longrightarrow actd(S, o_2)}{S \Vdash o_1 \longrightarrow o_2.\mathtt{boundary}}\ (\textsc{a-boundary}) \qquad \frac{owner(S, o_1) = o_2}{S \Vdash o_1 \longrightarrow o_2.c}\ (\textsc{a-owner}) \qquad \frac{}{S \Vdash o \longrightarrow null.c}\ (\textsc{a-null})$$

$$\frac{S \Vdash o \longrightarrow actd(S, v)}{S \Vdash o \longrightarrow v}\ (\textsc{a-value})$$

Fig. 17: Accessibility Rules.

Store Accessibility:

$$\frac{}{\Vdash \varnothing}\ (\textsc{a-store } \varnothing) \qquad \frac{\Vdash S_0 \qquad S_1 = S_0[o \mapsto \langle \dots, \overline{v} \rangle] \qquad \forall v_i \in \overline{v}.\ S_1 \Vdash o \longrightarrow v_i}{\Vdash S_1}\ (\textsc{a-store object})$$

Stack Frame Accessibility:

$$\frac{}{S \Vdash \{this \mapsto o\}}\ (\textsc{a-stackframe this}) \qquad \frac{S \Vdash F_0 \qquad F_1 = F_0[x \mapsto v] \qquad S \Vdash F_1(this) \longrightarrow v}{S \Vdash F_1}\ (\textsc{a-stackframe var})$$

Fig. 18: Store and Stack Frame Accessibility

(A-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-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-STORE $*$), and all values of a stack frame must be accessible by the *this*-object (A-STACKFRAME $*$).

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

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

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 [15] and by Almeida with Balloons [4]. The notion of ownership types comes from Clarke [11] as a formalizing the core of Flexible Alias Protection [21]. Ever since, many researchers investigated ownership type systems [9, 19, 7, 3]. Ownership type systems have been used to prevent data-races [8], deadlocks [6, 5], and to allow the modular specification and verification of object-oriented programs [18].

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 [9]. 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 [10] 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 [18].

The second solution [9, 6] is to allow Java's inner member classes [14] 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 modular verification of object-oriented programs. However, this is an ad hoc solution, which Aldrich and Chambers already observed [2]. Interestingly, this approach breaks the owners-as-dominators property, showing that a more general solution is needed.

Ownership types have been combined with type genericity [23], showing that ownership types can be implemented without extending the syntax of a language that already has generic types. This approach can not be directly applied to SLOD, as variables and field names can appear in domain annotations.

Ownership Domains. The basic idea of ownership domains comes from Clarke [9] with ownership contexts. Objects are not directly owned by other objects, but instead are owned by *contexts*. Contexts in turn are owned by objects. While Clarke's formalization was based on the Object Calculus [1], Aldrich and Chambers [2] applied this idea to a subset of Java and extended it by several features. While Clarke proposed the usage of Java's inner classes as a solution to the iterator problem, Ownership Domains is able to handle it without that workaround.

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. The Ownership Domain approach only handles precise ownership information. In that 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 [24]. A more general version of Ownership Domains has been formalized in System F [17].

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

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

[2] Jonathan Aldrich and Criag Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *Proceedings of the 18^{th} European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, June 2004.

[3] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In OOPSLA'02 [22], pages 311–330.

[4] Paulo Sérgio Almeida. Balloon Types: Controlling sharing of state in data types. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11^{th} European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer-Verlag, June 1998.

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

[6] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In OOPSLA'02 [22], pages 211–230.

[7] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proceedings of the 30^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 213–223. ACM Press, January 2003.

[8] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. pages 56–69, October 2001.

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

[10] Dave Clarke and S. Drossopoulou. Ownership, encapsulation, and the disjointness of type and effect. In OOPSLA'02 [22], pages 292–310.

[11] Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13^{th} ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'98)*, pages 48–64. ACM Press, October 1998.

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

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

[14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The $Java^{TM}$ Language Specification – Second Edition*. Addison-Wesley, June 2000.

[15] John Hogg. Islands: Aliasing protection in object-oriented languages. In Andreas Paepcke, editor, *Proceedings of the 6^{th} ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'91)*, pages 271–285. ACM Press, November 1991.

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

[17] Neel Krishnaswami and Jonathan Aldrich. Permission-based ownership: encapsulating state in higher-order typed languages. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming language design and implementation (PLDI'05)*, volume 40, pages 96–106. ACM Press, June 2005.

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

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

[20] James Noble. Iterators and encapsulation. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 431, St. Malo, France, June 2000. IEEE Computer Society.

[21] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *Proceedings of the 12^{th} European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.

[22] *Proceedings of the 17^{th} ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*. ACM Press, November 2002.

[23] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight generic ownership. In *Proceedings of the 33^{rd} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*. ACM Press, January 2006.

[24] Matthew Smith. Towards an effects system for ownership domains. In *7^{th} ECOOP Workshop on Formal Techniques for Java-like Programs - FTfJP'2005*, July 2005.

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 Γ, then T is well-formed in Γ.*

$$\Gamma \vdash e : T \Longrightarrow \Gamma \vdash T$$

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

Case (T-VAR) $\quad e = x \quad \Gamma \vdash \diamond \quad \Gamma(x) = T$.
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) $\quad e = e_0.f_i = e_1 \quad \Gamma \vdash T <: T_f$
1. By (S-TYPE),
   (a) $\Gamma \vdash T$

Case (T-INVK) Immediate.
Case (T-LET) Immediate.
Case (T-NEW) Immediate.

Lemma 2.

1. $\Gamma \vdash d_1 <:_d d_2 \ \wedge \ |d_2| = 2 \Longrightarrow d_1 = d_2 \wedge \Gamma \vdash d_1$
2. $\Gamma \vdash d_1 <:_d d_2 \ \wedge \ d_1 \neq d_2 \Longrightarrow$
   $\exists x, d_x, C_x, b, d_0.\ \Gamma \vdash d_1 \wedge \Gamma \vdash d_2$
   $\wedge\ d_1 = x.b \wedge d_2 = d_0.b \wedge \Gamma \vdash x : d_x C_x \wedge \Gamma \vdash d_x <:_d d_0$

Proof. By definition of S-DOMAIN $*$.

Lemma 3.

$$\Gamma \vdash d_1 <:_d d_2 \wedge \Gamma \vdash d_1.b \wedge \Gamma \vdash d_2.b \Longrightarrow \Gamma \vdash d_1.b <: d_2.b$$

Proof. Assume $\Gamma \vdash d_1 <:_d d_2 \wedge \Gamma \vdash d_1.b \wedge \Gamma \vdash d_2.b$. 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.*

$$S \vdash d_1 <:_d d_2 \wedge S \vdash d_2 <:_d d_3 \Longrightarrow S \vdash d_1 <:_d d_3$$

Proof. By induction on the length $n = |d_1| + |d_2| + |d_3|$.

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

1. Suppose
   (a) $\Gamma \vdash d_1 <:_d d_2 \wedge \Gamma \vdash d_2 <:_d d_3$
2. By Lemma 2,
   (a) $\Gamma \vdash d_1 \wedge \Gamma \vdash d_2 \wedge \Gamma \vdash d_3 \wedge d_1 = d_2 \wedge d_2 = d_3$
3. Hence, by (S-DOMAIN REFL) it follows $\Gamma \vdash d_1 <:_d d_3$, which had to be shown.

Induction Step $n = m$.

1. Suppose
   (a) $\Gamma \vdash d_1 <:_d d_2 \wedge \Gamma \vdash d_2 <:_d d_3$
   for some d_1, d_2, d_3 with $|d_1| + |d_2| + d_3| = m$.
2. Without loss of generality assume $d_1 \neq d_2 \neq d_3$. Hence by Lemma 2 there exist $x, y, b_1, b_2, d_{2_0}, d_{3_0}, d_x, d_y, C_x, D_x$ with
   (a) $d_1 = x.b_1 \wedge d_2 = y.b_2.b_1 \wedge d_3 = d_{3_0}.b_2.b_1$
   (b) $\Gamma \vdash x : d_x\ C_x$
   (c) $\Gamma \vdash y : d_y\ D_x$
   (d) $\Gamma \vdash d_x <:_d y.b_2$
   (e) $\Gamma \vdash d_y <:_d d_{3_0}$
   (f) $\Gamma \vdash d_1 \wedge \Gamma \vdash d_2 \wedge \Gamma \vdash d_3$
3. Note that from (2f) and (2a) it follows $\Gamma \vdash d_{3_0}.b_2$, and from (2d) we get $\Gamma \vdash y.b_2$. Hence with (2c) and (2e) it follows by (S-DOMAIN VAR),
   (a) $\Gamma \vdash y.b_2 <:_d d_{3_0}.b_2$
4. Note that from (2d) it follows that $|d_x| \leq |y.b_2|$. Hence we get $|d_x| + |y.b_2| + |d_{3_0}.b_2| < m$. So we can apply the induction hypothesis to (2d) and (3a), and we get
   (a) $\Gamma \vdash d_x <:_d d_{3_0}.b_2$
5. Hence, with (2b) and (2f) it follows by (S-DOMAIN VAR),
   (a) $\Gamma \vdash x.b_1 <:_d d_{3_0}.b_2.b_1$
6. Thus with (2a) we get $\Gamma \vdash d_1 <:_d d_3$, which had to be shown.

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

$$S \vdash d_1 \subseteq d_2 \wedge S \vdash d_2 \subseteq d_3 \Longrightarrow S \vdash d_1 \subseteq d_3$$

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

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

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 $S \vdash d_1 \subseteq d_2 \wedge S \vdash d_2 \subseteq d_3$. Also assume $d_1 \neq null.c$, 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) $actd(S, v_1) \subseteq v_2.b_2$
2. By $S \vdash d_2 \subseteq d_3$ and (SUBSET LOOSE), $\exists v_3, b_3$ with
(a) $d_3 = v_3.b_3.b_2.b_1$, and
(b) $actd(S, v_2) \subseteq v_3.b_3$
3. Hence by (2b) and (SUBSET LOOSE),
(a) $v_2.b_2 \subseteq v_3.b_3.b_2$
4. By (1c), (3a) and the induction hypothesis (the induction hypothesis can be applied, because $|actd(S, v_1)| = 2$ 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) $actd(S, v_1) \subseteq v_3.b_3.b_2$
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.*

$$S_0 \leq S_1 \equiv \forall\ o, d, C, \overline{f}.\ S_0(o) = \langle d, C, \overline{v_0} \rangle \Longrightarrow \exists\ \overline{v_1}.\ S_1(o) = \langle d, C, \overline{v_1} \rangle$$

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

$$S_0 \leq S_1 \ \wedge\ S_1 \leq S_2 \Longrightarrow S_0 \leq S_2$$

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.*

$$S_0, F \vdash e \Rightarrow v, S_1 \Longrightarrow S_0 \leq S_1$$

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 (F-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) $S_3 = S_2[o \mapsto \langle rd, C, [v/v_i]\overline{v} \rangle]$
2. By (1a), (1b) and the induction hypothesis,
(a) $S_0 \leq S_1$
(b) $S_1 \leq S_2$
3. By Lemma 6,
(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 (3a) and Lemma 6,
(a) $S_0 \leq S_3$

Case (R-NEW) Assume $S_0, F \Vdash \texttt{new}\ d\ C \Rightarrow o, S_1$.
1. By (R-NEW),

(a) $rd = rtd(S_0, F, F(this), d)$
(b) $o \notin dom(S_0)$
(c) $S_1 = S_0[o \mapsto \langle rd, C, \overline{null}\rangle]$
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.*

$$S_0, \Gamma \vdash F \wedge S_0, F \vdash e \Rightarrow v, S_1 \Longrightarrow S_1, \Gamma \vdash F$$

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.*

$$\begin{gathered} this \in dom(F) \wedge \; S, \Gamma \vdash F \;\; \wedge \;\; \Gamma \vdash d_0 <:_d d_1 \\ \Longrightarrow \\ S \vdash rtd(S, F, F(this), d_0) \subseteq rtd(S, F, F(this), d_1) \end{gathered}$$

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 $\Gamma \vdash d_0 <:_d d_1$ and (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 (SUBSET REFL).

Induction Step $n = m$. Assume $\Gamma \vdash d_0 <:_d d_1$. If $d_0 = d_1$ we are done, so asssume $d_0 \neq d_1$.

1. Assume
(a) $this \in dom(F)$
(b) $S, \Gamma \vdash F$
(c) $\Gamma \vdash d_0 <:_d d_1$
2. By (1c) and (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) $\Gamma \vdash d_x <: d_2$
3. As $|d_2| < |d_1|$ and $|d_x| \leq |d_2|$,
(a) $|d_x| + |d_2| < n$
4. By (2d) 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 (T-VAR),
(a) $x \in dom(F)$
6. By (5a), (1b), (T-STACK NULL) and (T-STACK VAR),
(a) $F(x) = null$, or
(b) $S \vdash actd(S, F(x)) \subseteq rtd(S, F, F(this), d_x)$
7. Assume (6a), by (ACTD NULL),
(a) $actd(S, F(x)) = null.\texttt{local}$
8. Hence by (SUBSET NULL),
(a) $S \vdash actd(S, F(x)) \subseteq rtd(S, F, F(this), d_2)$
9. Now assume (6b), by (4a) and Lemma 5,
(a) $S \vdash actd(S, F(x)) \subseteq rtd(S, F, F(this), d_2)$
10. Hence by (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)

I) 1. Assume
   (a) $S, F \vdash x \Rightarrow v, S$
   (b) $\Gamma \vdash x : d\ C$
   (c) $S, \Gamma \vdash F$
2. By (1a) and (R-VAR),
   (a) $F(x) = v$
3. By (1b) and (T-VAR),
   (a) $\Gamma(x) = d\ C$
4. By (T-STACK NULL) and (T-STACK VAR)
   (a) $v = \mathit{null}$, or
   (b) $S \vdash \mathit{actd}(S, v) \subseteq \mathit{rtd}(S, F, F(\mathit{this}), d)\ \wedge\ \mathit{class}(S, v) <:_c C$
   as required.

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

Case (R-LET)

I) 1. Assume
   (a) $S_0, F \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 \Rightarrow v_1, S_2$
   (b) $\Gamma \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : d\ C$
   (c) $S_0, \Gamma \vdash F$
   (d) $\mathit{this} \in \mathit{dom}(F)$
   (e) $\vdash S_0$
2. By (1a) and (R-LET),
   (a) $S_0, F \vdash e_0 \Rightarrow v_0, S_1$
   (b) $S_1, F[x \mapsto v_0] \vdash e_1 \Rightarrow v_1, S_2$
   for some v_0, S_1
3. By (1b) and (T-LET),
   (a) $\Gamma \vdash e_0 : d_0\ C_0$
   (b) $\Gamma[x \mapsto d_0\ C_0] \vdash e_1 : d\ C$
   (c) $x \notin \mathit{dom}(\Gamma)$
   for some d_0, C_0
4. By (3a), (2a), (1c), (1d), (1e) and the induction hypothesis,
   (a) $v_0 = \mathit{null}\ \vee$
   $\mathit{actd}(S_1, v_0) \subseteq \mathit{rtd}(S_1, F, F(\mathit{this}), d_0)\ \wedge\ \mathit{class}(S_1, v_0) <:_c C_0$
   (b) $\vdash S_1$
5. By (1c), (1a) and Lemma 8,
   (a) $S_1, \Gamma \vdash F$
6. With (3a), (4a), (T-STACK NULL) and (T-STACK VAR)
   (a) $S_1, \Gamma[x \mapsto d_0\ C_0] \vdash F[x \mapsto v_0]$
7. Let
   (a) $F' = F[x \mapsto v_0]$
8. By (6a), (4b), (3b), (2b) and the induction hypothesis,
   (a) $v_1 = \mathit{null} \vee S_2 \vdash \mathit{actd}(S_2, v_1) \subseteq \mathit{rtd}(S_2, F', F'(\mathit{this}), d) \wedge \mathit{class}(S_2, v_1) <:_c C$
   (b) $\vdash S_2$
9. By (3c) and (1c),
   (a) $x \notin \mathit{dom}(F)$
10. Hence with (7a),

(a) $\forall y \in dom(F).\ F(y) = F'(y)$

11. With (1d),

    (a) $F'(this) = F(this)$

12. Hence with (8a),

    (a) $v_1 = null \ \vee \ S_2 \vdash actd(S_2, v_1) \subseteq rtd(S_2, F', F(this), d)$

13. Let

    (a) $d = a.b$

14. Case distinction on the shape of a,
15. Assume

    (a) $a \neq y$

16. By (RTD THISOWNER),

    (a) $rtd(S_2, F', F(this), d) = rtd(S_2, F, F(this), d)$

    as F' and F is not used by (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) $y \in dom(\Gamma)$

21. By (1c) and (T-STACK $*$),

    (a) $y \in dom(F)$

22. By (9a),

    (a) $x \neq y$

23. By (7a),

    (a) $F(y) = F'(y)$

24. By (RTD VAR),

    (a) $rtd(S_2, F', F(this), d) = rtd(S_2, F, F(this), d)$

25. Thus, with (16a), (12a) and (8a),

    (a) $v_1 = null \vee S_2 \vdash actd(S_2, v_1) \subseteq rtd(S_2, F, F(this), d) \wedge class(S_2, v_1) <:_c C$

    as required.

II) Shown above by (8b).

Case (R-FIELD)

I) 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) $this \in dom(F)$

   (e) $\vdash S_0$

2. By (1a) and (R-FIELD),

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

   (b) $S_1(o) = \langle rd, C, \overline{v} \rangle$

3. By (1b) and (T-FIELD),

   (a) $\Gamma \vdash e : d\ C$

   (b) $fields(C) = \overline{d_f}\ \overline{C_f}\ \overline{f}$

   (c) $d_i\ C_i = \sigma(e, d, d_{f_i})\ C_{f_i}$

   (d) $\Gamma \vdash d_i\ C_i$

   for some $d, C, \overline{C_f}, \overline{d_f}, \overline{f}, x$

4. By (1c), (1d), (1e), (2a), (3a) and the induction hypothesis,
   (a) $S_1 \vdash actd(S_1, o) \subseteq rtd(S_1, F, F(this), d)$
   (b) $class(S_1, o) <:_c C$
   (c) $\vdash S_1$
5. By (4c) and (T-STORE),
   (a) $v_i = null \vee$
   $S_1 \vdash actd(S_1, v_i) \subseteq rtd(S_1, \varnothing, o, d_{f_i}) \;\wedge\; class(S_1, v_i) <:_c C_i$
6. Assume
   (a) $v_i \neq null$
   otherwise we are done.
7. Hence with (5a),
   (a) $S_1 \vdash actd(S_1, v_i) \subseteq rtd(S_1, \varnothing, o, d_{f_i})$
   (b) $class(S_1, v_i) <:_c C_i$
8. Case distinction on the shape of d_{f_i}. There are two cases:
   (a) $d_{f_i} = \texttt{owner}.b$
   (b) $d_{f_i} = \texttt{this}.b$
   note that by (T-CLASSDECL), d_{f_i} cannot have the form $x.b$.
9. Let
   (a) $d = d_o.c_1$
   (b) $actd(S_1, o) = v_o.c_2$
10. By (4a), (9b) and (SUBSET $*$),
   (a) $c_1 = c_2$

Case $d_{f_i} = \texttt{owner}.b$.

11. Hence by (RTD THISOWNER),
    (a) $rtd(S_1, \varnothing, o, d_{f_i}) = v_o.[c_1/\texttt{same}]b$
12. By (4c),
    (a) $S_1 \vdash actd(S_1, v_i) \subseteq v_o.[c_1/\texttt{same}]b$
13. By (3c) and (8a),
    (a) $d_i = d_o.[c_1/\texttt{same}]b$
14. By (4a), (9b) and (9b),
    (a) $S_1 \vdash v_o.c_1 \subseteq rtd(S_1, F, F(this), d_o.c_1)$
15. Hence by (SUBSET $*$) there are three cases:
    (a) $v_o = null$, or
    (b) $v_o.c_1 = rtd(S_1, F, F(this), d_o.c_1)$, or
    (c) $actd(S_1, v_o) \subseteq rtd(S_1, F, F(this), d_o)$
16. We show that for all three cases the following holds,
    (a) $v_o.[c_1/\texttt{same}]b \subseteq rtd(S_1, F, F(this), d_o.[c_1/\texttt{same}]b)$
    Case (15a), by (SUBSET NULL).
    Case (15b), by (SUBSET REFL), as
    $v_o.[c_1/\texttt{same}]b = rtd(S_1, F, F(this), d_o.[c_1/\texttt{same}]b)$.
    Case (15c), by (SUBSET LOOSE).
17. Thus by (12a), (16a), (13a) and Lemma 5,
    (a) $S_1 \vdash actd(S_1, v_i) \subseteq rtd(S_1, F,\ F(this), d_i)$
    closing the case.

Case $d_{f_i} = \texttt{this}.b$.

18. By (3c),
    (a) $d_i = e.[c/\texttt{same}]b$
19. By (3d), e has to be a local variable,
    (a) $e = x$
20. By (18a),
    (a) $d_i = x.[c/\texttt{same}]b$

for some x.

21. By (9b), (8b), (RTD THISOWNER),
   (a) $rtd(S_1, \varnothing, o, d_{f_i}) = o.[c/\texttt{same}]b$
22. With (4c) and (T-STACK VAR),
   (a) $S_1 \vdash actd(S_1, v_i) \subseteq o.[c/\texttt{same}]b$
23. By (2a), (19a) and (R-VAR),
   (a) $F(x) = o$
24. Hence by (RTD VAR),
   (a) $rtd(S_1, F, F(this), x.[c/\texttt{same}]b) = o.[c/\texttt{same}]b$
25. Thus with (22a) and (20a),
   (a) $S_1 \vdash actd(S_1, v_i) \subseteq rtd(S_1, F, F(this), d_i)$
   closing the case.

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

Case (R-FIELDUP)

I)
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) $this \in dom(\Gamma)$
   (e) $\vdash S_0$
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) $S_2(o) = \langle rd, D, \overline{v} \rangle$
   (d) $S_3 = S_2[o \mapsto \langle rd, D, [v/v_i]\overline{v} \rangle]$
   for some $o, S_1, S_2, rd, D, \overline{v}$.
3. By (1b) and (T-FIELDUP),
   (a) $\Gamma \vdash e_0 : d_0\ C_0$
   (b) $fields(C_0) = \overline{d_f}\ \overline{C_f}\ \overline{f}$
   (c) $T_f = \sigma(e_0, d_0, d_{f_i})\ C_{f_i}$
   (d) $\Gamma \vdash d\ C <: T_f$
   (e) $\texttt{owner} \in d_{f_i} \Rightarrow isPrecise(d_0)$
   for some $d_0, C_0, \overline{d_f}, \overline{C_f}, \overline{f}$.
4. By (1d), (1e), (1c), (3a), (2a) and the induction hypothesis,
   (a) $actd(S_1, o) \subseteq rtd(S_1, F, F(this), d_0)$
   (b) $class(S_1, o) <:_c C_0$
   (c) $\vdash S_1$
5. By (3d) and (S-TYPE),
   (a) $\Gamma \vdash e_1 : d\ C$
6. By (1c) and Lemma 8,
   (a) $S_1, \Gamma \vdash F$
7. With (1d), (4c), (5a), (2b) and the induction hypothesis,
   (a) $v = null\ \vee$
   $(actd(S_2, v) \subseteq rtd(S_2, F, F(this), d)\ \wedge\ class(S_2, v) <:_c C)$
   (b) $\vdash S_2$
8. Assume
   (a) $v \neq null$
   otherwise we are done.
9. Hence with (7a),
   (a) $actd(S_2, v) \subseteq rtd(S_2, F, F(this), d)$
   (b) $class(S_2, v) <:_c C$
10. By (2c) and (2d),

(a) $dom(S_2) = dom(S_3)$
(b) $\forall o_x \in dom(S_2)$.
$class(S_2, o_x) = class(S_3, o_x) \ \wedge$
$actd(S_2, o_x) = actd(S_3, o_x)$

11. By (10b),
(a) $\forall o_x \in dom(S_2)$.
$rtd(S_2, F, F(this), d) = rtd(S_3, F, F(this), d)$
12. Thus with (10a), (10b), (9a) and (9b),
(a) $actd(S_3, v) \subseteq rtd(S_3, F, F(this), d)$
(b) $class(S_3, v) <:_c C$
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 $v = null$ or $S_3 \vdash actd(S_3, v) \subseteq rtd(S_3, \varnothing, o, d_{f_i}) \ \wedge \ class(S_3, v) <:_c C_{f_i}$.

13. Assume
(a) $v \neq null$
otherwise we are done.
14. By (3d), (12b) and (S-CLASS TRANS),
(a) $class(S_3, v) <:_c C_{f_i}$
15. By (3c), (3d) and (S-TYPE),
(a) $\Gamma \vdash d <:_d \sigma(e_0, d_0, d_{f_i})$
16. By (1c) and Lemma 8,
(a) $S_3, \Gamma \vdash F$
17. By (12a), (15a) and Lemma 9,
(a) $S_3 \vdash rtd(S_3, F, F(this), d) \subseteq$
$rtd(S_3, F, F(this), \sigma(e_0, d_0, d_{f_i}))$
18. By (12a) and Lemma 5,
(a) $S_3 \vdash actd(S_3, v) \subseteq rtd(S_3, F, F(this), \sigma(e_0, d_0, d_{f_i}))$
19. We now show that
(a) $rtd(S_3, F, F(this), \sigma(e_0, d_0, d_{f_i})) = rtd(S_3, \varnothing, o, d_{f_i})$.
20. We do a case distinction on the shape of d_{f_i}.

Case $d_{f_i} = \texttt{this}.b$.

21. By (3c),
(a) $T_f = [last(d_0)/\texttt{same}]e_0.b$
22. By (3d) and (S-TYPE),
(a) $\Gamma \vdash T_f$
23. With (3c),
(a) $e_0 = x$
for some x.
24. Hence,
(a) $T_f = [last(d_0)/\texttt{same}]x.b$
25. By (23a), (2a) and (R-VAR),
(a) $F(x) = o$
26. Hence by (RTD VAR),
(a) $rtd(S_3, F, F(this), [last(d_0)/\texttt{same}]x.b) = [last(d_0)/\texttt{same}]o.b$
27. Let
(a) $actd(S_3, o) = o_o.c$
for some o_o, c.
28. By (RTD THISOWNER),
(a) $rtd(S_3, \varnothing, o, d_{f_i}) = [c/\texttt{same}]o.b$
29. Note that

(a) $actd(S_3, o) = actd(S_1, o)$

30. With (4a) and (SUBSET $*$),
    (a) $c = last(d_0)$
31. Thus,
    (a) $rtd(S_3, F, F(this), \sigma(e_0, d_0, d_{f_i})) = rtd(S_3, \varnothing, o, d_{f_i})$
    closing the case.

Case $d_{f_i} = \texttt{owner}.b$.

32. Let
    (a) $actd(S_3, o) = o_o.c$.
    for some o_o, c.
33. By (3e),
    (a) $isPrecise(d_0)$
34. Note that
    (a) $actd(S_1, o) = actd(S_3, o)$
35. 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) $\sigma(e_0, d_0, d_{f_i}) = [c/\texttt{same}]a.b$
38. Thus by (35a),
    (a) $rtd(S_3, F, F(this), \sigma(e_0, d_0, d_{f_i})) = [c/\texttt{same}]o_o.b$
39. By (RTD THISOWNER),
    (a) $rtd(S_3, \varnothing, o, d_{f_i}) = [c/\texttt{same}]o_o.b$
40. Hence,
    (a) $rtd(S_3, F, F(this), \sigma(e_0, d_0, d_{f_i})) = rtd(S_3, \varnothing, o, d_{f_i})$
    closing the case.

41. By (18a), (19a) and Lemma 5,
    (a) $S_3 \vdash actd(S_3, v) \subseteq rtd(S_3, \varnothing, o, d_{f_i})$
42. And finally,
    (a) $\vdash S_3$

Case (R-INVK)

I) 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) $this \in dom(\Gamma)$
    (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) $\cdots$
    (d) $S_n, F \vdash e_n \Rightarrow v_n, S_{n+1}$
    (e) $S_1(o) = \langle \ldots, C, \ldots \rangle$
    (f) $mbody(m, C) = \overline{x}.e_b$
    (g) $S_{n+1}, \{\overline{x} \mapsto \overline{v}, \texttt{this} \mapsto o\} \vdash e_b \Rightarrow v, S_{n+2}$
3. By (1b) and (T-INVK),
    (a) $\Gamma \vdash e : d_e\ C_e$
    (b) $mtype(m, C_e) = \overline{d}\ \overline{C} \to d_u\ C_u$
    (c) $\Gamma \vdash \overline{e} : \overline{d_e}\ \overline{C_e}$
    (d) $\Gamma \vdash \overline{d_e}\ \overline{C_e} <: \sigma(e, d_e, \overline{d})\ \overline{C}$

(e) $(\exists i.\texttt{owner} \in d_i) \Rightarrow isPrecise(d_e)$
(f) $d_m\ C_m = \sigma(e, d_e, d_u)\ C_u$
(g) $\Gamma \vdash d_m\ C_m$

4. We assume,
   (a) $C \vdash d_u\ C_u\ m(\overline{d}\ \overline{C}\ \overline{x})\{\ e_b\ \}$
5. Hence, by (T-METHODDECL),
   (a) $\Gamma_m = \{\ this \mapsto \texttt{owner.same}\ C, \overline{x} \mapsto \overline{d}\ \overline{C}\}$
   (b) $\Gamma_m \vdash e_b : d_b\ C_b$
   (c) $d_b\ C_b <: d_u\ C_u$
   (d) $\varnothing \vdash \overline{d}\ \overline{C}$
   (e) $\varnothing \vdash d_u\ C_u$
   (f) $CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\}$
   (g) if $mtype(m, D) = \overline{U} \to U_r,$ then $\overline{d}\ \overline{C} = \overline{U}$ and $d_u\ C_u = U_r$
   for some $d_b, C_b, \overline{U}, U_r$.
6. We have to show
   (a) $v = null \vee (actd(S_{n+2}, v) \subseteq rtd(S_{n+2}, F, F(this), d_m) \wedge class(S_3, v) <:_c C_m)$
7. By (1e), (1c), (2a) – (2d), (3a), (3c), Lemmas 8, 7 and the induction hypothesizes,
   (a) $S_0, \Gamma \vdash F$ and $\vdash S_1$
   (b) $S_i, \Gamma \vdash F$ and $\vdash S_i$
   (c) $actd(S_1, o) \subseteq rtd(S_1, F, F(this), d_e)\ \wedge\ class(S_1, o) <:_c C_e$
   (d) $v_i = null \vee (actd(S_{i+1}, v_i) \subseteq rtd(S_{i+1}, F, F(this), d_{e_i}) \wedge class(S_{i+1}, v_i <:_c C_{e_i}$
   for $0 < i \leq n+1$.
8. Let
   (a) $F_m = \{this \mapsto o, \overline{x} \mapsto \overline{v}\}$
9. We show
   (a) $S_{n+1}, \Gamma_m \vdash F_m$
10. Let
    (a) $\Gamma_{m_{this}} = \varnothing[this \mapsto \texttt{owner.same}\ C]$
    (b) $F_{m_{this}} = \varnothing[this \mapsto o]$
11. At first we show,
    (a) $S_{n+1}, \Gamma_{m_{this}} \vdash F_{m_{this}}$
12. Let
    (a) $actd(S_{n+1}, o) = this_o.c$
13. Hence
    (a) $rtd(S_{n+1}, F_m, F_m(this), \texttt{owner.same}) = this_o.c$
14. By (SUBSET REFL),
    (a) $actd(S_{n+1}, o) \subseteq rtd(S_{n+1}, F_m, F_m(this), \texttt{owner.same})$
15. Hence, (11a), with (2e) and (T-STACK $\varnothing$).
16. We now show,
    (a) $S_{n+1}, \Gamma_{m_{this}}[\overline{x} \mapsto \overline{d}\ \overline{C}] \vdash F_{m_{this}}[\overline{x} \mapsto \overline{v}]$
17. Let
    (a) $x_i \in \overline{x}$
    (b) $v_i = F_m(x_i)$
18. Assume
    (a) $v_i \neq null$
    otherwise apply (T-STACK NULL).
19. By (5a),
    (a) $\Gamma_m(x_i) = d_i\ C_i$
20. By (7d) and (18a),

(a) $actd(S_{i+1}, v_i) \subseteq rtd(S_{i+1}, F, F(this), d_{e_i})$
(b) $class(S_{i+1}, v_i) <:_c C_{e_i}$

21. By (3d) and (S-CLASS TRANS),
(a) $class(S_{i+1}, v_i) <:_c C_i$

22. By (7b), (7d), (3d) and Lemma 9,
(a) $actd(S_{i+1}, v_i) \subseteq rtd(S_{i+1}, F, F(this), \sigma(e, d_e, d_i))$.

23. We now show that
(a) $rtd(S_{i+1}, F, F(this), \sigma(e, d_e, d_i)) = rtd(S_{i+1}, F_m, F_m(this), d_i)$

24. Case distinction on the shape of d_i. There are three cases.

Case $d_i = \texttt{this}.b$.

25. Hence,
(a) $\sigma(e, d_e, d_i) = [last(d_e)/\texttt{same}]e.b$

26. By (5d),
(a) $e = x$
for some x.

27. Hence
(a) $\sigma(e, d_e, d_i) = [last(d_e)/\texttt{same}]x.b$

28. By (26a), (2a) and (R-VAR),
(a) $F(x) = o$

29. Hence by (RTD VAR),
(a) $rtd(S_{i+1}, F,\ F(this), [last(d_e)/\texttt{same}]x.b) = [last(d_e)/\texttt{same}]o.b.$

30. Note that,
(a) $actd(\ S_{n+1}, o) = this_v.c$
(b) $F_m(this) = o$

31. Hence by (RTD THISOWNER),
(a) $rtd(S_{i+1}, F_m, F_m(this), d_i) = [c/\texttt{same}]o.b.$

32. By (7c),
(a) $c = last(d_e)$

33. Hence by (25a), (29a) and (31a),
(a) $rtd(S_{i+1}, F, F(this), \sigma(e, d_e, d_i)) = rtd(S_{i+1}, F_m, F_m(this), d_i)$
closing the case.

Case $d_i = \texttt{owner}.b$.

34. By (3e),
(a) $isPrecise(d_e)$

35. Note that
(a) $actd(S_1, o) = actd(S_{n+1}, o)$
(b) $actd(S_{n+1}, o) = this_o.c$,

36. Hence by (7c), (34a) and (SUBSET REFL),
(a) $rtd(S_{n+1}, F, F(this), d_e) = this_o.c$

37. Let
(a) $d_e = a.c$

38. Hence,
(a) $\sigma(e, d_e, d_i) = [c/\texttt{same}]a.b$

39. Hence by (36a),
(a) $rtd(S_{i+1}, F, F(this), \sigma(e, d_e, d_i)) = [c/\texttt{same}]this_o.b$

40. As $F_m(this) = o$,
(a) $rtd(S_{i+1}, F_m, F_m(this), d_i) = [c/\texttt{same}]this_o.b$

41. Thus,
(a) $rtd(S_{i+1}, F, F(this), \sigma(e, d_e, d_i)) = rtd(S_{i+1}, F_m, F_m(this), d_i)$
closing the case.

Case $d_i = x.b$.

42. By (5d),

(a) $\varnothing \vdash x.b$

43. By (T-DOMAIN VAR),

(a) $\varnothing \vdash x : T$

for some T.

44. By (T-VAR),

(a) $\varnothing(x) = T$

which is a contradiction.

45. So we have shown

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

46. Note that

(a) $this \in dom(\Gamma_m)$

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

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

(a) $v = null \vee (actd(S_{n+2}, v) \subseteq rtd(S_{n+2}, F_m, F_m(this), d_b) \wedge class(S_{n+2}, v) <:_c C_b$

(b) $\vdash S_{n+2}$

48. Assume

(a) $v \neq null$

otherwise we are done.

49. Hence with (47a),

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

(b) $class(S_{n+2}, v) <:_c C_b$

50. By (49b), (5c), (3f) and (S-CLASS TRANS),

(a) $class(S_{n+2}, v) <:_c C_m$

51. By (45a) and Lemma 8,

(a) $S_{n+2}, \Gamma_m \vdash F_m$.

52. With (5c), (49a), Lemma 9 and Lemma 5,

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

53. We have to show

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

54. Case distinction on the shape of d_u. There are three cases.

55. Let

(a) $d_e = d_{e_1}.c_e$

for some d_{e_1}, c_e.

Case $d_u = \texttt{this}.b$.

56. Hence,

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

57. By (3g),

(a) $e = x$

for some x.

58. Hence,

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

59. By (57a), (2a) and (R-VAR),

(a) $F(x) = o$,

60. Hence,

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

61. By 8a) and (12a),

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

62. By (7c),

(a) $c = c_e$

63. Hence with (58a) and (3f),

(a) $rtd(S_{n+2}, F_m, F_m(this), d_u) =$
$rtd(S_{n+2}, F, F(this), d_m)$

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

Case $d_u = \texttt{owner}.b$.

65. By (12a),
    (a) $actd(S_{n+2}, o) = this_o.c$
66. By (7c)
    (a) $c_e = c$
67. Hence by (RTD THISOWNER),
    (a) $rtd(S_{n+2}, F_m, F_m(this), d_u) = [c_e/\texttt{same}]this_o.b$
68. By (3f),
    (a) $d_m = d_{e_1}.[c_e/\texttt{same}]b$
69. By (7c), (65a) and (SUBSET LOOSE),
    (a) $actd(S_{n+2}, this_o) \subseteq rtd(S_{n+2}, F, F(this), d_{e_1})$.
70. Hence,
    (a) $this_o.[c_e/\texttt{same}]b \subseteq rtd(S_{n+2}, F, F(this), d_{e_1}.[c_e/\texttt{same}]b)$
71. Hence, by (67a) and (68a),
    (a) $rtd(S_{n+2}, F_m, F_m(this), d_u) \subseteq rtd(S_{n+2}, F, F(this), d_m)$
72. Hence, by (53a) and Lemma 5,
    (a) $actd(S_{n+2}, v) \subseteq rtd(S_{n+2}, F, F(this), d_m)$
    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) $S_0, F \vdash \texttt{new}\ d\ C \Rightarrow o, S_1$
    (b) $\Gamma \vdash \texttt{new}\ d\ C : d\ C$
    (c) $\vdash S_0$
2. By (1a) and (R-NEW),
    (a) $rd = rtd(S_0, F, F(this), d)$
    (b) $fields(C) = \overline{T}\ \overline{f}$
    (c) $S_1 = S_0[o \mapsto \langle rd, C, \overline{null}\rangle]$
    (d) $|\overline{null}| = |\overline{f}|$
3. By (1b) and (T-NEW),
    (a) $C \in dom(CT)$
4. Note that,
    (a) $actd(S_0, F(this)) = actd(S_1, F(this))$
5. Hence,
    (a) $rtd(S_0, F, F(this), d) = rtd(S_1, F, F(this), d)$
6. By (2a) and (SUBSET REFL),
    (a) $actd(S_1, o) = rtd(S_0, F, F(this), d)$
7. Hence by (SUBSET REFL),
    (a) $actd(S_1, o) \subseteq rtd(S_1, F, F(this), d)$

II)
8. By (3a) and (T-CLASS DECL),
    (a) $\Gamma \vdash C$
9. With (1c), (2b), (2c), (2d) and (T-STORE OBJECT),
    (a) $\vdash S_1$

A.2 Proof of Theorem 2

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.*

$$S \vdash actd(S, v_1) \subseteq rtd(S, F, v_2, d) \ \wedge\ \Gamma \vdash d \ \wedge\ S \Vdash F \ \wedge\ \Gamma, S \vdash F$$
$$\Longrightarrow$$
$$S \Vdash v_2 \longrightarrow actd(S, v_1)$$

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

Induction Base $n = 2$.

1. Assume
   (a) $S \vdash actd(S, v_1) \subseteq rtd(S, F, v_2, d)$
   (b) $\Gamma \vdash d$
   (c) $S \Vdash F$
   (d) $S, \Gamma \vdash F$
2. By (1b) and (T-DOMAIN $*$) there are two cases:

Case $d = a.c$, where $a \in \{\texttt{owner}, \texttt{this}\}$.

Case $a = \texttt{this}$.

3. By (SUBSET REFL),
   (a) $actd(S, v_1) = v_2.c$.
4. By (A-OWN)
   (a) $S \Vdash v_2 \longrightarrow v_2.c$.

Case $a = \texttt{owner}$.

5. Let
   (a) $actd(S, v_2) = v_o.c_o$
6. Hence,
   (a) $actd(S, v_1) = v_o.[c_o/\texttt{same}]c$
7. Note that
   (a) $owner(S, v_2) = v_o$
8. Hence, by (A-OWNER),
   (a) $S \Vdash v_2 \longrightarrow actd(S, v_1)$

Case $d = x.\texttt{boundary}$.

9. By (1b) and (T-DOMAIN VAR),
   (a) $\Gamma \vdash x : d$
10. By (1d),
    (a) $F(x) = v_x$
    for some v_x.
11. Hence,
    (a) $rtd(S, F, v_2, d) = v_x.\texttt{boundary}$
12. With (1a) and (SUBSET REFL),
    (a) $actd(S, v_1) = v_x.\texttt{boundary}$.
13. By (1c),
    (a) $v_x = null$ or
    (b) $S \Vdash v_2 \longrightarrow v_x$.

Case $v_x = null$.

14. Hence,
    (a) $act(S, v_1) = null.\texttt{boundary}$
15. By (A-NULL),

(a) $S \Vdash v_2 \longrightarrow actd(S, v_1)$.

Case $S \Vdash v_2 \longrightarrow v_x$.

1. Hence by (A-BOUNDARY),
   (a) $S \Vdash v_2 \longrightarrow v_x.\mathtt{boundary}$
2. Thus by (12a),
   (a) $S \Vdash v_2 \longrightarrow actd(S, v_1)$

Induction Step $n = m$.

1. Assume,
   (a) $S \vdash actd(S, v_1) \subseteq rtd(S, F, v_2, d) \ \wedge \ \Gamma \vdash d \ \wedge \ S \Vdash F \ \wedge \ \Gamma, S \vdash F \Longrightarrow S \Vdash v_2 \longrightarrow actd(S, v_1)$
   for all d with $|d| < m$.
2. Let
   (a) $d = a.b$
   with $|d| = m$ and $m > 2$.
3. Assume,
   (a) $S \vdash actd(S, v_1) \subseteq rtd(S, F, v_2, d)$
   (b) $\Gamma \vdash d$
   (c) $S \Vdash F$
   (d) $S, \Gamma \vdash F$
4. By (3b) and (T-DOMAIN $*$),
   (a) $d = a.b_1.\mathtt{boundary}$
   (b) $\Gamma \vdash a.b_1$
   for some b_1.
5. By (3a), (4a) and (SUBSET LOOSE),
   (a) $actd(S, v_1) = v_o.\mathtt{boundary}$
   (b) $S \vdash actd(S, v_o) \subseteq rtd(S, F, v_2, a.b_1)$.
   for some v_o.
6. Clearly,
   (a) $|a.b_1| < m$
7. Hence by (4b), (5b), (3c), (3d) and the induction hypothesis (1a),
   (a) $S \Vdash v_2 \longrightarrow v_o$
8. By (A-BOUNDARY),
   (a) $S \Vdash v_2 \longrightarrow v_o.\mathtt{boundary}$
9. Thus by (5a),
   (a) $S \Vdash v_2 \longrightarrow actd(S, v_1)$

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

$$S_0 \leq S_1 \ \wedge \ S_0 \Vdash o \longrightarrow actd(S_0, v) \Longrightarrow S_1 \Vdash o \longrightarrow actd(S_1, v)$$

Proof.

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

Case (A-OWN)

4. Hence,
   (a) $S_0 \Vdash o \longrightarrow o.c$
   (b) $actd(S_0, v) = o.c$
5. By (A-OWN),
   (a) $S_1 \Vdash o \longrightarrow o.c.$
6. With (2a),
   (a) $S_1 \Vdash o \longrightarrow actd(S_1, v)$

Case (A-BOUNDARY)

7. Hence,
   (a) $actd(S_0, v) = v_2.\texttt{boundary}$
   (b) $S_0 \Vdash o \longrightarrow v_2.\texttt{boundary}$
   (c) $S_0 \Vdash o \longrightarrow actd(S_0, v_2)$
   for some v_2.
8. By (1a), (7c) and the induction hypothesis,
   (a) $S_1 \Vdash o \longrightarrow actd(S_1, v_2)$
9. Hence, by (A-BOUNDARY),
   (a) $S_1 \Vdash o \longrightarrow v_2.\texttt{boundary}$
10. This with (7a) and (2a),
    (a) $S_1 \Vdash o \longrightarrow actd(S_1, v)$

Case (A-OWNER)

11. Hence,
    (a) $actd(S_0, v) = v_2.c$
    (b) $S_0 \Vdash o \longrightarrow v_2.c$
    (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) $S_1 \Vdash o \longrightarrow v_2.c$
14. Thus, by (11a) and (2a),
    (a) $S_1 \Vdash o \longrightarrow actd(S_1, v)$

Lemma 12.

$$S_0 \Vdash F \wedge S_0, F \vdash e \Rightarrow v, S_1 \Longrightarrow S_1 \Vdash F$$

Proof. We have to show $\forall v \in ran(F).\ v = null \vee S_1 \Vdash F(this) \longrightarrow v$. 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) $this \in dom(F)$
   (b) $\vdash S_0$
   (c) $\Gamma, S_0 \vdash F$
   (d) $\Vdash S_0$
   (e) $S_0 \Vdash F$
   for some Γ, S_0, F.
2. The remainder of the proof is by structural induction on the reduction rules of the operational semantics.

Case (R-VAR)

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 (1e),
   (a) $S_0 \Vdash F(\mathit{this}) \longrightarrow v$
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) $S_1(o) = \langle rd, C, \overline{v} \rangle$
9. By (7b) and (T-FIELD),
   (a) $\Gamma \vdash d\ C$
   (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) $v_i = \mathit{null}$, or
   (b) $S_1 \vdash \mathit{actd}(S_1, v_i) \subseteq \mathit{rtd}(S_1, F, F(\mathit{this}), d)$
12. Assume,
   (a) $v_i \neq \mathit{null}$
   otherwise we are done.
13. By (1e),(1c), Lemma 12, and Lemma 8,
   (a) $S_1 \Vdash F$
   (b) $\Gamma, S_1 \Vdash F$
14. With (11b) and Lemma 10,
   (a) $S_1 \Vdash F(\mathit{this}) \longrightarrow v_i$
   closing the case.

Case (R-LET)

15. Assume,
   (a) $S_0, F \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 \Rightarrow v_1, S_2$
   (b) $\Gamma \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : d\ C$
   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) $\Gamma \vdash e_o : d_0\ C_0$
   (b) $x \notin \mathit{dom}(\Gamma)$
   (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) $v_0 = \mathit{null}\ \vee\ S_1 \Vdash F(\mathit{this}) \longrightarrow v_0$
19. By Theorem 1,

(a) $\vdash S_1$

20. Let
    (a) $F_2 = F[x \mapsto v_0]$
    (b) $\Gamma_2 = \Gamma[x \mapsto d_0\ C_0]$
21. With the same argumentation as in case (R-LET) of the proof of Theorem 1, it follows with (1c),
    (a) $\Gamma_2, S_1 \vdash F_2$
22. By Lemma 7,
    (a) $S_0 \leq S_1$
23. By (18b) and (1e),
    (a) $S_1 \Vdash F_2$
24. With (16b), (17c) and the induction hypothesis,
    (a) $\Vdash S_2$
    (b) $S_2 \Vdash F(this) \longrightarrow v_1$
    which had to be shown.

Case (R-FIELDUP)

25. Assume,
    (a) $S_0, F \vdash e_0.f_i = e_1 \Rightarrow v, S_3$
    (b) $\Gamma \vdash e_0.f_i = e_1 : d\ C$
    for some $e_0, f_i, e_1, v, S_3, d, C$.
26. By (25a) 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) $S_2(o) = \langle rd, D, \overline{v} \rangle$
    (d) $S_3 = S_2[o \mapsto \langle rd, D, [v/v_i]\overline{v} \rangle]$
    for some $o, S_1, S_2, rd, D, \overline{v}$.
27. By (25b) and (T-FIELDUP),
    (a) $\Gamma \vdash e_0 : d_0\ C_0$
    (b) $fields(C_0) = \overline{d_f}\ \overline{C_f}\ \overline{f}$
    (c) $\Gamma \vdash e_1 : d\ C$
    for some $d_0, C_0, \overline{d_f}, \overline{C_f}, \overline{f}$.
28. By (27a), (26a) and the induction hypothesis,
    (a) $\Vdash S_1$
    (b) $S_1 \Vdash F(this) \longrightarrow o$
29. By Theorem 1,
    (a) $\vdash S_1$
30. By (1c), (1e), Lemma 8 and 12,
    (a) $S_1, \Gamma \vdash F$
    (b) $S_1 \Vdash F$
31. With (28a), (27c), (26b) and the induction hypothesis,
    (a) $\Vdash S_2$
    (b) $v = null\ \vee\ S_2 \Vdash F(this) \longrightarrow v$
32. By Theorem 1,
    (a) $v = null\ \vee\ actd(S_3, v) \subseteq rtd(S_3, F, F(this), d)$
    (b) $\vdash S_3$
33. By (27c),
    (a) $\Gamma \vdash d$
34. With the same arguments like in the proof of Theorem 1 of case (R-FIELDUP),
    (a) $S_3, \Gamma \vdash F$
35. By Lemma 12,
    (a) $S_2 \Vdash F$
36. As

(a) $\forall o \in dom(S_2).\ actd(S_2, o) = actd(S_3, o)$
(b) $dom(S_2) = dom(S_3)$

37. it follows
(a) $S_3 \Vdash F$
38. Hence with (32a) and Lemma 10,
(a) $v = null \ \vee \ S_3 \Vdash F(this) \longrightarrow v$
showing the first part.
39. We now show
(a) $\Vdash S_3$
40. By (31a) we have $\Vdash S_2$. 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 we have to show
(a) $v = null \ \vee \ S_3 \Vdash o \longrightarrow v$
41. Assume
(a) $v \neq null$
otherwise we are done.
42. By (32b),
(a) $S_3 \vdash actd(S_3, v) \subseteq rtd(S_3, \varnothing, o, d_{f_i})$
43. By (27b) and (T-CLASS),
(a) $\varnothing \vdash d_{f_i}$.
44. Note that
(a) $S_3 \Vdash \varnothing$
(b) $\varnothing, S_3 \vdash \varnothing$
45. With (42a) and Lemma 10,
(a) $S_3 \Vdash o \longrightarrow v$
46. Thus,
(a) $\Vdash S_3$
closing the case.

Case (R-INVK)

47. Assume,
(a) $S_0, F \vdash e.m(\overline{e}) \Rightarrow v, S_{n+2}$
(b) $\Gamma \vdash e.m(\overline{e}) : d\ C$
for some $e, m, \overline{e}, v, S_{n+2}$.
48. By (47a) 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) $\cdots$
(d) $S_n, F \vdash e_n \Rightarrow v_n, S_{n+1}$
(e) $S_1(o) = \langle \ldots, C, \ldots \rangle$
(f) $mbody(m, C) = \overline{x}.e_b$
(g) $S_{n+1}, \{this \mapsto o, \overline{x} \mapsto \overline{v}\} \vdash e_b \Rightarrow v, S_{n+2}$
49. By (47b) and (T-INVK),
(a) $\Gamma \vdash e : d_e\ C_e$
(b) $mtype(m, C_e) = \overline{d}\ \overline{C} \to d_u\ C_u$
(c) $\Gamma \vdash \overline{e} : \overline{d_e}\ \overline{C_e}$
(d) $\Gamma \vdash \overline{d_e}\ \overline{C_e} <: \sigma(e, d_e)\ \overline{d}\ \overline{C}$
50. By Theorem 1),
(a) $\vdash S_{n+2}$
(b) $S_{n+2} \vdash actd(S_{n+2}, v) \subseteq rtd(S_{n+2}, F, F(this), d)$
51. By (48a)–(48e), successively applying the induction hypothesis, Theorem 1, Lemma 12, Lemma 8 and Lemma 11 it follows for all $1 \leq i < n+2$
(a) $S_i, \Gamma \vdash F$
(b) $S_i \Vdash F$

(c) $\Vdash S_i$
(d) $\vdash S_i$
(e) $v_i = null \ \vee \ S_{n+1} \Vdash F(this) \longrightarrow v_i$
(f) $S_{n+1} \Vdash F(this) \longrightarrow o$

52. Let
(a) $F_m = \{this \mapsto o, \overline{x} \mapsto \overline{v}\}$
53. By (51e) and (51f),
(a) $S_{n+1} \Vdash F_m$
54. With the same definitions and arguments of case (R-FIELDUP) of the proof of Theorem 1,
(a) $S_{n+1}, \Gamma_m \vdash F_m$
55. With (48g) and the induction hypothesis,
(a) $\Vdash S_{n+2}$
56. By (1c) and Lemma 12,
(a) $S_{n+2} \Vdash F$
57. By Lemma 8,
(a) $S_{n+2}, \Gamma \vdash F$
58. By (47b) and Lemma 1,
(a) $\Gamma \vdash d$.
59. Thus with (50b) and Lemma 10,
(a) $S_{n+2} \Vdash F(this) \longrightarrow v$
which had to be shown.

Case (R-NEW)

60. Assume,
(a) $S_0, F \vdash \mathtt{new}\ d\ C \Rightarrow o, S_1$
(b) $\Gamma \vdash \mathtt{new}\ d\ C : d\ C$
for some d, C, o, S_1.
61. By (60a) and (R-NEW),
(a) $rd = rtd(S_0, F, F(this), d)$
(b) $fields(C) = \overline{T}\ \overline{f}$
(c) $o \notin dom(S_0)$
(d) $S_1 = S_0[o \mapsto \langle rd, C, \overline{null}\rangle]$
(e) $|\overline{null}| = |\overline{f}|$
for some $\overline{T}, \overline{f}$.
62. By (60b) and (T-NEW),
(a) $d = a.c$
(b) $a \in \{\mathtt{this}, \mathtt{owner}\}$
(c) $C \in dom(CT)$
for some a, c
63. Let
(a) $actd(S_0, F(this)) = v_{this}.c_{this}$
64. Note that
(a) $actd(S_1, o) = rd$
65. We do a case distinction on the shape of a.

Case $a = \mathtt{this}$.

66. Hence by (RTD THISOWNER),
(a) $rd = [c_{this}/\mathtt{same}]F(this).c$.
67. Thus by (A-OWN),
(a) $S_1 \Vdash F(this) \longrightarrow actd(S_1, o)$

Case $a = \mathtt{owner}$.

68. Hence,
(a) $rd = [c_{this}/\mathtt{same}]v_{this}.c$
69. Note that
(a) $owner(S, F(this)) = v_{this}$
70. Hence, by (A-OWNER),
(a) $S_1 \Vdash F(this) \longrightarrow actd(S_1, o)$