Flexible Ownership Domains

Radu Vanciu **Marwan Abi-Antoun**

August 12, 2012

Department of Computer Science
Wayne State University
Detroit, MI 48202

Abstract

Reasoning about the object structure and aliasing is important, and many ownership type systems have been proposed for that purpose. One such type system, Ownership Domains, further separates the aliasing policy from its mechanism, and the policy is provided by developers using domain links. The base Ownership Domains type system, however, cannot express some common programming idioms, so additional flexibility is needed.
We extend Ownership Domains by incorporating an existential ownership domain that is restricted by the domain links in scope. The resulting type system, Flexible Ownership Domains, can express additional idioms, compared to the base system. We formalize it, prove its soundness, and illustrate it on some worked examples.

Keywords: Ownership Domains, Featherweight Java, existential ownership domain

Contents

List of Figures

List of Symbols

```
class Root {
  domain MODEL, VIEW;
  MODEL Circuit circuit;
  ...
}
class Circuit {
   public domain DB;
   DB Node node;

   private domain OWNED;
   OWNED NodeVector<DB> nodes;
}
class NodeVector<ELTS>  {
   ELTS Node obj;
}
```

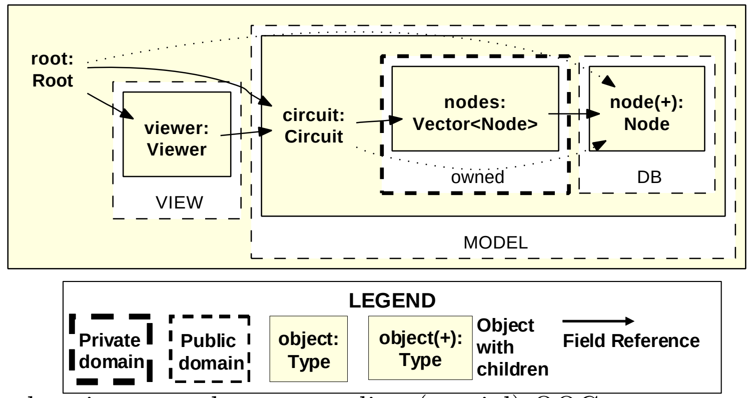


Figure 1: Aphyds: annotated code snippets and corresponding (partial) OOG.

1 Background

Since this paper extends Ownership Domains, we start by reviewing the base type system. We also give some intuition of how an OOG visualizes the object structure, since we use OOGs to clarify some of the examples. Since a deep understanding of OOGs is not needed to understand the type system, the details of the OOG extraction are elsewhere [3, 2].

1.1 Review of Ownership Domains

To organize a large object graph, hierarchy is effective [11]. In an object hierarchy, an object has child objects. More generally, one can introduce a level of indirection, *domains*, which are *named groups of objects*. So, an object has domains, which contain its child objects to form its substructure, and so on.

An ownership domain is a named group of objects with an explicit name that conveys design intent and explicit policies (domain links) that govern how a domain can reference objects in other domains.

There are several key features of Ownership Domains that are crucial for expressing design intent in code. The first is having explicit "contexts" or domains. Other ownership type systems implicitly treat all objects with the same owner as belonging to an implicit context. On the other hand, explicit domains are useful, because developers can define multiple domains per object to express their design intent. Second, Ownership Domains support pushing any object underneath any other object in the ownership hierarchy. In Ownership Domains, a child object may or may not be encapsulated by its parent object. Ownership Domains define two kinds of object hierarchy, logical containment and strict encapsulation, defined below.

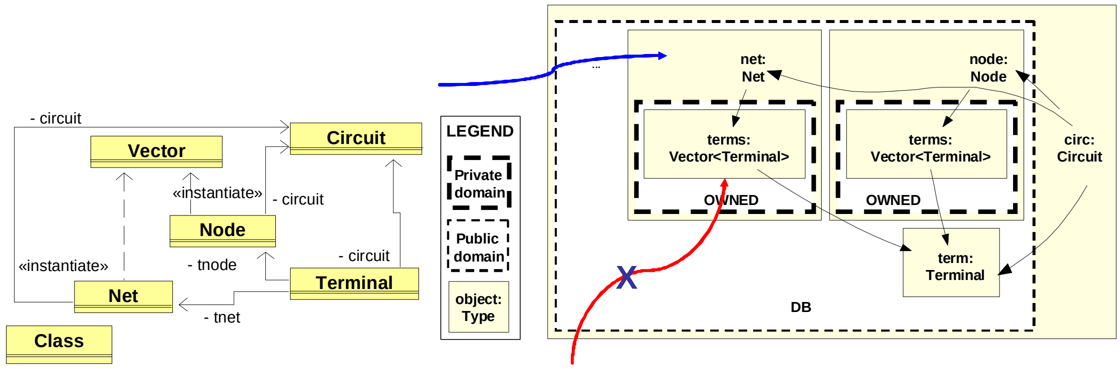
Logical containment means that an object is conceptually *part of* another object, but still accessible to other objects. Logical containment is achieved using a public domain. A child object can still be referenced from outside its owner, if it is part of a public domain of its parent.

Strict Encapsulation means that an object is *owned by* its parent object, and represents domination, i.e., the absence of references to an object from outside the owner. Strict encapsulation is achieved using a private domain. Strict encapsulation requires any access to a child object to go through its owning object. This is the only kind of hierarchy supported by type systems that enforce the *owners-as-dominators* property [8],

Domain declaration. A domain is declared using the `domain` keyword, with a `public` or `private` modifier (Figure 1, line 2). Domains are declared on a class but are treated like fields, in that fresh domains are created for each instance of that class. For a domain `D` declared on a class `C` and two instances `o1` and `o2` of type `C`, the domains `o1.D` and `o2.D` are distinct, for distinct `o1` and `o2`. Each object is assigned to a single ownership domain that does not change at run-time. Developers indicate the domain of an object by annotating each reference to that object in the program. For example, we declare the field `circuit` of type `Circuit` in the domain `MODEL` (line 3).

The two kinds of object hierarchy, logical containment and strict encapsulation, are used to express design intent. As a an example of logical containment, `Circuit` declares a public domain, `DB`, to hold objects of type `Node` (line 7). As an example of strict encapsulation, `Circuit` declares a private domain, `OWNED`, to store the field `nodes:NodeVector` (line 10).

Domain parameters on a type allow objects to share state. Domain parameters are similar to generic type parameters but can be treated independently. For the sake of illustration, we defined a non-generic class, `NodeVector`, which can be considered as the generic type `Vector<Node>`. Typically, a container such as `NodeVector` does not own its elements. Instead, it references them through a domain parameter, `ELTS` (line 13). We ignore how a `NodeVector` is represented, and assume that a `NodeVector` has a reference to its element `Node`, through a "virtual field" `obj`. This



(a) Partial UML class diagram. (b) Sub-structure of the object `circ:Circuit`.

Figure 2: Aphyds: sub-structure of `circ:Circuit` shows distinct instances of `Vector` in different domains.

way, both an instance of `Circuit` and an instance of `NodeVector` can reference the same `Node` object.

1.2 Review of the Ownership Object Graph (OOG)

While ownership annotations show the relationship between two objects or domains that are currently in scope, OOGs show how these relationships compose to form a global hierarchy. The extraction of an OOG from a program with ownership annotations is subtle, because annotations are expressed in terms of locally visible domain names, which may be domain parameters that refer to a domain declared elsewhere. For example, inside the class `NodeVector`, the `Node` field is declared to be in the domain `ELTS`, but `ELTS` is really a domain parameter (declared on line 14). To show the right objects and edges to the `Node` object in the hierarchy, the analysis determines to what declared domain `ELTS` is bound. In this case, `ELTS` is bound to the `DB` domain (line 3) so the OOG shows the edges to the `Node` object in the `DB` domain of the `Circuit` object (Figure 1).

To start the OOG extraction process, the developer picks a root class as a starting point, the `Root` class in the example (Figure 1). The root class is assumed to be instantiated into a root object. The static analysis then uses the architectural hierarchy specified by the annotations in the code to impose a conceptual hierarchy on the objects in the system, starting at the root object.

The OOG shows objects (instances of the classes from a class diagram), and groups related objects into domains. The OOG is hierarchical in that an object can have one or more nested domains, with other objects inside them. As a hierarchical representation, the OOG allows expanding or collapsing individual objects to control the level of visual detail. In Figure 1, we collapse the sub-structure of `circuit` to hide lower-level objects. In Figure 2b, we expand `circuit` to reveal a nested domain, `DB`, which contains objects `node` and `net` of type `Node` and `Net`, respectively. In turn, we also expand the sub-structures of `node` and `net`, to reveal data structures such as instances of the `Vector` class, namely `nodes` and `nets`.

Graphical Notation. Our visualization uses box nesting to indicate containment of objects inside domains and domains inside objects. However, based on user feedback, we may choose a different visualization. Dashed-border, white-filled boxes represent domains. A private domain has a thick, dashed border; a public domain has a thin, dashed border. Solid-filled boxes represent objects. Solid edges represent field references. An object labeled `obj:T` indicates an object reference `obj` of type `T` (a similar convention is used in object diagrams) that we refer to either as the "object `obj`" or the "`T` object" to mean "an instance of the `T` class." A (+) symbol on an object or a domain indicates that it has a collapsed sub-structure.

2 Expressiveness Challenges

In this section, we motivate by example the type system extensions, and informally introduce them.

Challenge #1: Listeners in different domains do not typecheck. We often need to express that an object can be in multiple domains. This challenge occurs in particular, with objects communicating using events. Indeed, such a design encourages a looser coupling between objects. We illustrate this issue on MicroDraw, which we adapted from [1]. MicroDraw follows the Model-View-Controller design pattern [10]. To represent this design intent, the developer defines three top-level domains or tiers, `MODEL`, `VIEW` and `CTRL`, and places objects in these domains.

- `MODEL:` has instances of `Drawing` and `Figure` objects. A `Drawing` is composed of `Figure`s that know their containing `Drawing`. The class `StandardDrawing` implements the `Drawing` interface.
- `VIEW:` has instances of `DrawingEditor` and `DrawingView`. Class `View` implements

DrawingView. JavaDrawApp implements DrawingEditor, which inherits FSListener.

- CTRL: has instances of Command objects. AbstractCommand implements the Command interface, as well as FSListener.

A few code snippets and the associated OOG are in Figure 3. A DrawingView registers itself as an Observer to a DrawingEditor and to a Command. On the OOG, inside a drawingView object, is a fSelectionListeners object of type Vector, which contains references of type FSListener. A FSListener reference can point to either a Command or a DrawingEditor object; this subtyping illustrates one of the features of object-oriented code which makes it challenging to understand.

Consider the method addListener() in Figure 3. The method parameter fsl of type FSListener is problematic. Indeed, the code calls addListener(), once with a Command object, and once with a DrawingEditor object. Both DrawingEditor (which is in VIEW) and AbstractCommand (which is in CTRL) implement FSListener. So one call to addListener() requires the method argument editor to be in the OWNER domain parameter. And the other call requires the method argument of type Command to be in the C domain parameter. So the parameter fsl should support some kind of a union, namely OWNER OR C. In Ownership Domains, using either annotation for the fsl parameter generates an annotation warning, because one or the other method invocation does not typecheck. As this is a fairly common idiom, it is crucial to be able to express it. Furthermore, this limitation in Ownership Domains means that the OOG shows one or the other edge to objects of type FSListener, but not both (the edges in question are shown as thick edges on the OOG). For soundness, it is crucial to show both of these edges. This matters even when not showing the fListeners object. When collapsing the parent's sub-structure, those edges are lifted and appear as going from drawingView to app and command.

Challenge #2: Public domains are hard to use.

Public domains add the important expressiveness of logical containment. For example, we declare a public domain DB on Circuit to make the Node and Net objects conceptually part of the Circuit object. This allow object of type Placer to refer to both the parent object (Circuit), and the child objects (Node, Net). However, defining public domains can make the annotations harder to add and to get them to typecheck. In the Aphyds example (Fig. 4a), using the circ.DB

```
class View<OWNER, M, C> implements DrawingView<OWNER, M, C> ... {
   private domain OWNED;
   assumes OWNER->M, OWNER->C;
   link OWNED->C, OWNED->OWNER;
   // object in OWNED has permission to access CTRL and the parent (in this case VIEW)

   /* The registered list of listeners for selection changes */
   private Vector<OWNED, FSListener<any>> fListeners;

   View() {
      this.fListeners = new Vector<this.OWNED, ???>();

      // DrawingEditor<OWNER,M,C> <: FSListener<any> ?? Yes
      // Vector<OWNED, any>> <: Vector<OWNED, any>>
   }
   void setView(DrawingEditor<OWNER,M,C> editor, ...) {
     ...
     // DrawingEditor implements FSListener
     // editor is in 'OWNER' domain parameter, not 'C'
     addListener(editor);
   }

  /**
   * Add a listener for selection changes.
   * AbstractCommand implements FSListener.
   * Command is in the 'C' domain parameter
   */
   void addListener(FSListener<any> fsl) {
     fListeners.add(fsl);
    }
   ...
}
class Vector<OWNER, T<ELTS>> { // T generic type parame
  assumes OWNER->ELTS;
  private T<ELTS> obj; // Virtual field

  public void add(T<ELTS> element) {
    ...
  }
}

class Listener<OWNER> { }

class FSListener<OWNER> extends Listener<OWNER> { }

class System<OWNER> {
    domain MODEL, VIEW, CTRL;
    link VIEW->MODEL; // Satisfy assumption OWNER->M in new View()
    link VIEW->CTRL; // Satisfy assumption OWNER->C in new View()
    link CTRL->MODEL; // Needed to create the other object cmd, also used in T-Type
    link CTRL->VIEW; // Needed to create the other object cmd, also used in T-Type

    JavaDrawApp<VIEW,MODEL,CTRL> app = new JavaDrawApp<...>();
    View<VIEW,MODEL,CTRL> view = new View<...>();
    view.setView(app);
    Command<CTRL,MODEL,VIEW> cmd = new Command<...>();

    public void init() {
        view.addListener(cmd);
    }

}
```

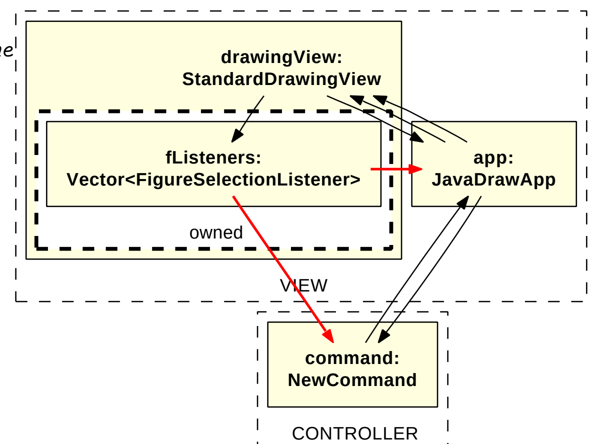


Figure 3: Expressing listeners in multiple domains. Adapted from JHotDraw [2].

```
class Root<OWNER> {
  domain MODEL, UI; // Top-level domains
  link OWNER->UI, OWNER->MODEL
  link UI->MODEL
  Placer<MODEL> placer;
  Viewer<UI,MODEL> viewer;
}
/* The application GUI component */
class Viewer<OWNER,M> {
  assumes OWNER->M
  final Circuit<M> circ; // XXX: Make final to
  Node<circ.DB, M> node; // access public domain DB

  /* Load new Circuit object from file */
  void loadCircuit() {
   // XXX: Cannot assign to final field here
    this.circ = ...;
  }
}
/* A Circuit including Node and Net objects */
class Circuit<OWNER> {
   public domain DB;
   link OWNER->DB, DB->OWNER;
   Node<DB,OWNER> node;
   Net<DB,OWNER> net;

   Node<DB,OWNER> getNodeAt(Point<shared> p) {
      return this.node;
   }
}
class Node<OWNER,M> {
  assumes OWNER->M
  Placer<M> placer;
}
class Placer<OWNER> {
  final Circuit<OWNER> circ;

  // XXX: circ field must be "spec-public"
  Circuit<OWNER> getCircuit() {
    return circ;
  }
  Node<circ.DB,OWNER> getNodeAt(Point<shared> p) {
    return getCircuit().getNodeAt(p);
  }
  void compute(Point<shared> p) {
    final Circuit<OWNER> targ = getCircuit();

   Node<targ.DB,OWNER> node = this.getNodeAt(p);
    // XXX: Need to establish targ == circ
    setPlacement(node);
  }
  void setPlacement(Node<circ.DB,OWNER> n) {
  }
}
```

(a) Ownership Domains (ECOOP'04 system).


```
class Root<OWNER> {
  domain MODEL, UI; // Top-level domains
  link OWNER->UI, OWNER->MODEL
  link UI->MODEL
  Placer<MODEL> placer;
  Viewer<UI,MODEL> viewer;
}
/* The application GUI component */
class Viewer<OWNER,M> {
  assumes OWNER->M
```

```
class Root<OWNER> {
  domain MODEL, UI; // Top-level domains
  link OWNER->UI, OWNER->MODEL
  link UI->MODEL
  Placer<MODEL> placer;
  Viewer<UI,MODEL> viewer;
}
/* The application GUI component */
class Viewer<OWNER,M> {
  assumes OWNER->M
  final Circuit<M> circ; // XXX: Make final to
  Node<circ.DB, M> node; // access public domain DB

  /* Load new Circuit object from file */
  void loadCircuit() {
   // XXX: Cannot assign to final field here
    this.circ = ...;
  }
}
/* A Circuit including Node and Net objects */
class Circuit<OWNER> {
   public domain DB;
   link OWNER->DB, DB->OWNER;
   Node<DB,OWNER> node;
   Net<DB,OWNER> net;

   Node<DB,OWNER> getNodeAt(Point<shared> p) {
      return this.node;
   }
}
class Node<OWNER,M> {
  assumes OWNER->M
  Placer<M> placer;
}
class Placer<OWNER> {
  final Circuit<OWNER> circ;

  // XXX: circ field must be "spec-public"
  Circuit<OWNER> getCircuit() {
    return circ;
  }
  Node<circ.DB,OWNER> getNodeAt(Point<shared> p) {
    return getCircuit().getNodeAt(p);
  }
  void compute(Point<shared> p) {
    final Circuit<OWNER> targ = getCircuit();

   Node<targ.DB,OWNER> node = this.getNodeAt(p);
    // XXX: Need to establish targ == circ
    setPlacement(node);
  }
  void setPlacement(Node<circ.DB,OWNER> n) {
  }
}
```

(a) Ownership Domains (ECOOP'04 system).

```
class Root<OWNER> {
  domain MODEL, UI; // Top-level domains
  link OWNER->UI, OWNER->MODEL
  link UI->MODEL
  Placer<MODEL> placer;
  Viewer<UI,MODEL> viewer;
}
/* The application GUI component */
class Viewer<OWNER,M> {
  assumes OWNER->M
  final Circuit<M> circ; // XXX: Make final to
  Node<circ.DB, M> node; // access public domain DB

  /* Load new Circuit object from file */
  void loadCircuit() {
   // XXX: Cannot assign to final field here
    this.circ = ...;
  }
}
/* A Circuit including Node and Net objects */
class Circuit<OWNER> {
   public domain DB;
   link OWNER->DB, DB->OWNER;
   Node<DB,OWNER> node;
   Net<DB,OWNER> net;

   Node<DB,OWNER> getNodeAt(Point<shared> p) {
      return this.node;
   }
}
class Node<OWNER,M> {
  assumes OWNER->M
  Placer<M> placer;
}
class Placer<OWNER, CDB> {
  Circuit<OWNER> circ;

  // XXX: circ field must be "spec-public"
  Circuit<OWNER> getCircuit() {
    return this.circ;
  }
  Node< CDB ,OWNER> getNodeAt(shared Point p) {
    return getCircuit().getNodeAt(p);
  }
  void compute() {
    final Circuit<OWNER> targ = getCircuit();

    Node<targ.DB,OWNER> node = circ.getNodeAt(p);
    // XXX: Need to establish targ.DB ==  CDB
    setPlacement(node);
  }
  void setPlacement(Node< CDB ,OWNER> n) {
  }
}
```

(b) Attempt to use domain parameters.

Figure 5: Ownership Domains: using domain parameters leads to issues of let bindings.

annotation for the return value of the method `getNodeAt` is problematic, since we need to establish that the local variable `targ` refers to the same object as `circ`. In addition, to be able to refer to the public domain `DB` the variables `circ` and `targ` are required to be final. With the extended type system, we can use `any` to refer to an object in a public domain (Fig. 4b).

One would think that an alternative to `any` is to declare a domain parameter `CDB` and to explicitly thread it through to give objects access to `Circuit`'s public domain `DB`(Fig. 5b). Used in this manner, public domains would lead to a proliferation of domain parameters, which would make the annotations quite verbose. Moreover, used in this way, domain parameters give rise to the problem of let bindings, i.e., we need to establish the correspondence between a domain parameter and the public domain. As a result, the typesystem would be unable to check an assignment of a reference of type `Node` in the `CDB` domain parameter to another reference in the `targ.DB` public domain. In conclusion, domain parameters are not appropriate here, and `any` seems to be a better solution.

3 Requirements

The Flexible Ownership Domains type system satisfies several requirements.

- Allows more programming idioms to be expressed comparing to Ownership Domains. All the restrictions are expressed using domain links. For example, the existential domain allows us express that a collection have objects from different domains.
- Is sound for a subset of Java, Featherweight Domain Java. We prove the standard Preservation and Progress theorems.
- Preserves the Heap Link soundness and Expression Link soundness properties.
- Preserves the Subtyping Link soundness property, without enforcing restrictions on the domain parameters. Each subtyping relation is guaranteed by domain links.

3.1 Domain Links

Similarly to Ownership Domains, our type system ensures *link soundness*, the property that the domain and link declarations in the system conservatively describe all aliasing that could take place at run time. Here we define link soundness in precise but informal language; Section 4 defines link soundness formally and proves that our type system enforces the property.

To state link soundness precisely, we need a few preliminary definitions. First, we say that object o *refers to* object o' if o has a field that points to o', or else a method with receiver o is executing and some expression in that method evaluates to o'.

Second, we say that object *o has permission to access* domain d if one of the following conditions holds:

1. o is part of some domain d', and there is a declaration of the form `link` d' `->` d.

2. o has permission to access some domain d', and d is a public domain declared by some object in domain d'.

3. o is part of domain d.

4. d is a domain declared by o.

We can now define link soundness using the definitions above:

Definition 1 (Link Soundness)

If some object o refers to object o' and o' is in domain d, then o has permission to access domain d.

Discussion. In order for link soundness to be meaningful, we must ensure that objects can't use `link` declarations or auxiliary objects to violate the intent of linking specifications. For example, in Figure 3, the `view` should not be able to give itself access to the `CTRL` domain by declaring `link owner->C`. We can ensure this with the following restriction:

- Each `link` declaration must have a locally-declared domain on one side of the arrow.

Furthermore, even though the `OWNED` domain is local to the `view` object, `view` should not be able to give `OWNED` any privileges that the `view` does not have itself. The `link OWNED->C` is valid only because of the `assume owner->C`. The following rules ensure that local domains obey the same restrictions as their enclosing objects or domains:

- An object can only link a local domain to an external domain d if the `this` object has permission to access d.

- An object can only link an external domain d to a local domain if d has permission to access the `owner` domain.

Finally, the `view` should not be able to get to the `CTRL` domain by creating its own objects in the `CTRL` domain. We formalize this with one final rule:

- An object o can only create objects in domains declared by o, or the `owner` domain of o, or in the `shared` domain.

Similarly to Ownership Domains, and unlike many of its predecessors, our system does not have a rule giving an object permission to access all enclosing domains. This permission can be granted using `link` declarations if needed, but developers can constrain aliasing more precisely relative to previous work by leaving this permission out.

Domain Links and `any`. The typechecker resolves the existential domain `any` locally in the enclosing type based on domain links and assumptions such that any domain that owner has permission to access may substitute `any`. By default, if developer specifies no domain links or assumptions, the typechecker resolves `any` to `OWNER`. In Figure 3, the expression `addListener(editor)` typechecks since the owner of `editor` is `OWNER`, and `FSListener<any>` resolves to `FSListener<OWNER>`.

The invocation `view.addListener(cmd)` in the class `System` is outside the scope of `any`, and requires the link `view.any->CTRL` to typecheck. The link cannot be explicitly declared. Instead, the permissions are granted if the class `View` declares `assumes OWNER->C`, where `C` is the domain parameter substituted by `CTRL`, the owner of `cmd`. Since `any` is resolved locally, the class `View` is the one which requires the permissions. Although `System` declares `link VIEW->CTRL`, it only ensures that the assumption of `View` is satisfied, with no effect on how `any` is resolved.

Since `any` is resolved locally, it has a different meaning in different classes. For example, in the class `Bar`, `any` resolves to `Bar::OWNER, Bar::D,` and `Bar::OWNED`, while in the class `Biff`, `any` resolves to `Biff::OWNER` (Fig. 6). If `any` were a constant domain such as `shared`, the field write expression `this.foo=biff.getFoo()` would typecheck since the return type of `biff.getFoo()` the type of the field `foo` would be `Foo<any>`. Instead, the typechecker distinguishes between `bar.any` and `biff.any` and the field write is invalid. The expression becomes valid only if additional assump-

```
class Main<OWNER> {
  domain DATA, VIEW;
  link OWNER->DATA, OWNER->VIEW;
  link VIEW->DATA;

  void run() {
   Bar<VIEW,DATA> aBar = new Bar<VIEW,DATA>();
   aBar.test();
  }
}

class Biff<OWNER,D> {
  assumes OWNER->OWNER; //default
  //allows biff.getFoo() to be an argument
 // assumes D->OWNER;

  Foo<any> getFoo() {
    return new Foo<OWNER>();
  }
}
class Bar<OWNER,D> {
  domain OWNED;
  assumes OWNER->D;

  //satisfies the assume clause in Biff
 // link OWNER->this.OWNED

  Foo<any> foo;

  public void test() {
    Biff<OWNED,OWNER> biff = new Biff<OWNED,OWNER>();
    biff.getFoo(); //valid invocation
    this.foo = biff.getFoo(); //invalid assignment
  }
}

class Foo<OWNER>{
...
}
```

Figure 6: `any` is not a constant domain like `shared`. Without the commented out `link` and `assumes` statements, the assignment is invalid.

tions and links are declared. First, the type `Foo<biff.any>` need to be well-formed, i.e., `bar.OWNER` has permissions to access `bar.OWNED` which is satisfied by declaring `assumes D->OWNER` in `Biff`. Next, the assumption of `Biff` needs to be satisfied and requires the link `OWNER->this.OWNED` in the class `Bar`.

$$
\begin{array}{rcl}
CL & ::= & \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>} \\
 & & \texttt{assumes}\ \overline{\gamma} \rightarrow \overline{\delta}\ \{\ \overline{D}\ \overline{L}\ \overline{F}\ K\ \overline{M}\ \} \\
D & ::= & [\texttt{public}]\ \texttt{domain}\ d; \\
L & ::= & \texttt{link}\ p \rightarrow p'; \\
F & ::= & T\ f; \\
K & ::= & C(\overline{T'}\ \overline{f'}, \overline{T}\ \overline{f})\ \{\texttt{super}(\overline{f'});\ \overline{\texttt{this}.f} = \overline{f};\} \\
M & ::= & T_R\ m(\overline{T}\ \overline{x})\ T_{\texttt{this}}\ \{\ \texttt{return}\ e;\ \} \\
e & ::= & x \mid \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(\overline{e}) \mid e.f \mid e.f = e' \mid e.m(\overline{e}) \\
 & \mid & (T)e \mid \ell \mid \ell \triangleright e \mid \texttt{error} \\
z & ::= & \texttt{this} \mid \texttt{that} \mid x \\
n & ::= & z \mid v \\
p & ::= & \alpha \mid n.d \mid \texttt{shared} \mid n.\texttt{any} \\
T & ::= & C\texttt{<}\overline{p}\texttt{>} \mid \texttt{ERROR} \\
v, \ell, \ell_0, \theta & \in & \mathit{locations} \\
S & ::= & \ell \mapsto C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) \\
\Gamma & ::= & x \mapsto T \\
\Sigma & ::= & \ell \mapsto T \\
\Delta & ::= & \{\ell_1.d_1 \rightarrow \ell_2.d_2\}
\end{array}
$$

Figure 7: Featherweight Domain Java Abstract Syntax.

4 Formal System

4.1 Syntax

We formally describe Flexible Ownership Domains using Featherweight Domain Java (FDJ), which models a core of the Java language with Ownership Domains [5]. To keep the language simple and easier to reason about, FDJ uses Featherweight Java, which ignores Java language constructs such as interfaces and static code [12]. We adopt the FDJ abstract syntax to which we add field write expressions, and support for existential domains through the keyword `any` (Fig. 7). We *highlight* the key parts where the Flexible Ownership Domains system differs from Ownership Domains.

In FDJ, C ranges over class names; T ranges over types; f ranges over fields; v ranges over values; e ranges over expressions; x ranges over variable names; d over domain names; n ranges over values and variable names; p ranges over domains; S ranges over stores; ℓ range over locations in the store; n_{this} is used to represent the name or the value of `this`; θ is the location corresponding to the value of `this`; α, β, γ, and δ range over formal domain parameters; m ranges over method names. As a shorthand, an overbar is used to represent a sequence. A store S maps locations ℓ to their contents: the class of the object, the actual ownership domain parameters, and the values stored in its fields. $S[\ell]$ denotes the store entry for ℓ; $S[\ell, i]$ to denote the value in the ith field of $S[\ell]$. Adding an entry for location ℓ to the store is abbreviated $S[\ell \mapsto C\langle\overline{p}\rangle(\overline{v})]$. $\ell \triangleright e$ represents a method body e executing with a receiver ℓ. Result of the computation is a location ℓ, which is sometimes referred to as a value v. The set of variables z includes the variable `this` of type $T_{\texttt{this}}$ used to refer to the current context, and `that` used to refer to the context of a receiver. Fixed class table CT mapping classes to their definitions. A program, then, is a tuple (CT, S, e) of a class table, a store, and an expression. The store type Σ maps locations ℓ to a runtime type T. Δ is the union of all links between runtime domains; When necessary, we distinguish between a static type $C\langle\overline{p}\rangle$ and a runtime type $C\langle\overline{\ell'.d}\rangle$.

4.2 Dynamic semantics

The evaluation relation, defined by the dynamic semantics given in Figure 8, is of the form $S;\theta \vdash e \mapsto e', S'$, read "In the context of store S, and receiver θ, expression e reduces to expression e' in one step, producing the new store S'. We write $\mapsto^*$ for the reflexive, transitive closure of $\mapsto$. Most of the rules are standard; the interesting features are how they track ownership domains.

The *R-New* rule reduces an object creation expression to a fresh location. The store is extended at that location to refer to a class with the specified ownership parameters, with the fields set to the values passed to the constructor.

The *R-Read* and *R-Write* rules look up the receiver in the store and identifies the ith field. The result of *R-Read* is the value at field position i in the store. *R-Write* changes the store by replacing the value at the field position i with the value v of the right-hand side of the field write expression, and returns v. In the original rules, *R-Read* and *R-Write* call the $fields$ auxiliary judgements to denote that f_i is the field at position i. Since, in the latest version of $fields$ we added `that`/`this`

$$\frac{\ell \notin domain(S) \qquad S' = S[\ell \mapsto C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v})]}{S;\theta \vdash \texttt{new}\ C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) \mapsto \ell, S'}\ \textit{R-New}$$

$$\frac{S[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) \qquad fields(C\texttt{<}\overline{\ell'.d}\texttt{>}) = \overline{T}\ \overline{f}}{S;\theta \vdash \ell.f_i \mapsto v_i, S}\ \textit{R-Read}$$

$$\frac{S[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) \qquad fields(C\texttt{<}\overline{\ell'.d}\texttt{>}) = \overline{T}\ \overline{f} \qquad S' = S[\ell \mapsto C\texttt{<}\overline{\ell'.d}\texttt{>}([v/v_i]\overline{v})]}{S;\theta \vdash \ell.f_i = v \mapsto v,\ S'}\ \textit{R-Write}$$

$$\frac{S[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) \qquad mbody(m, C\texttt{<}\overline{\ell'.d}\texttt{>}) = (\overline{x}, e_0)}{S;\theta \vdash \ell.m(\overline{v}) \mapsto \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_0, S}\ \textit{R-Invk}$$

$$\frac{S[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) \qquad C\texttt{<}\overline{\ell'.d}\texttt{>} <: T}{S;\theta \vdash (T)\ell \mapsto \ell, S}\ \textit{R-Cast}$$

$$\frac{S[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) \qquad C\texttt{<}\overline{\ell'.d}\texttt{>} \not<: T}{S;\theta \vdash (T)\ell \mapsto \texttt{error}, S}\ \textit{E-Cast}$$

$$\frac{}{S;\theta \vdash \ell \triangleright v \mapsto v, S}\ \textit{R-Context}$$

Figure 8: Dynamic Semantics

substitution, and `this` is not encountered in the dynamic semantics, we removed this call. As an alternative, we could still call *fields*, but we assume that `that`/`this` substitution is ignored if the argument is a runtime type.

As in Java (and FJ), the *R-Cast* rule checks that the cast expression is a subtype of the cast type. Note, however, that in FDJ this check also verifies that the ownership domain parameters match, doing an extra run-time check that is not present in Java. If the run-time check in the cast rule fails, however, then the cast reduces to the `error` expression, following the cast error rule *E-Cast*. This rule shows how the formal system models the exception that is thrown by the full language when a cast fails.

The method invocation rule *R-Invk* looks up the receiver in the store, then uses the *mbody* helper function (defined in Figure 16) to determine the correct method body to invoke. The method invocation is replaced with the appropriate method body. In the body, all occurrences of the formal method parameters and `this` are replaced with the actual arguments and the receiver,

$$\frac{S;\theta \vdash e_i \mapsto e_i', S'}{S;\theta \vdash C\texttt{<}\overline{\ell'.d}\texttt{>}(v_{1..i-1}, e_i, e_{i+1..n}) \mapsto C\texttt{<}\overline{\ell'.d}\texttt{>}(v_{1..i-1}, e_i', e_{i+1..n}), S'} \textit{ RC-New}$$

$$\frac{S;\theta \vdash e \mapsto e', S'}{S;\theta \vdash e.f_i \mapsto e'.f_i, S'} \textit{ RC-Read}$$

$$\frac{S;\theta \vdash e \mapsto e', S'}{S;\theta \vdash e.f_i = e_{arg} \mapsto e'.f_i = e_{arg}, S'} \textit{ RC-RecvWrite}$$

$$\frac{S;\theta \vdash e \mapsto e', S'}{S;\theta \vdash v.f_i = e \mapsto v.f_i = e', S'} \textit{ RC-ArgWrite}$$

$$\frac{S;\theta \vdash e \mapsto e', S'}{S;\theta \vdash e.m(\overline{e}) \mapsto e'.m(\overline{e}), S'} \textit{ RC-RecvInvk}$$

$$\frac{S;\theta \vdash e_i \mapsto e_i', S'}{S;\theta \vdash v.m(v_{1..i-1}, e_i, e_{i+1..n}) \mapsto v.m(v_{1..i-1}, e_i', e_{i+1..n}), S'} \textit{ RC-ArgInvk}$$

$$\frac{S;\theta \vdash e \mapsto e', S'}{S;\theta \vdash (T)e \mapsto (T)e', S'} \textit{ RC-Cast}$$

$$\frac{S;\ell \vdash e \mapsto e', S'}{S;\theta \vdash \ell \triangleright e \mapsto \ell \triangleright e', S'} \textit{ RC-Context}$$

Figure 9: Congruence and Error Rules

respectively. Here, the capture-avoiding substitution of values $\overline{v}$ for variables $\overline{x}$ in e is written $[\overline{v}/\overline{x}]e$. Execution of the method body continues in the context of the receiver location.

When a method expression reduces to a value, the *R-Context* rule propagates the value outside of its method context and into the surrounding method expression.

Figure 9 shows the congruence rules that allow reduction to proceed within an expression in the order of evaluation defined by Java. For example, the rule RC-Read states that an expression $e.f$ reduces to $e'.f$ whenever e reduces to e'. The congruence rule RC-Context shows the semantics of the $\ell \triangleright e$ construct: evaluation of the expression e occurs in the context of the receiver ℓ instead of the receiver θ.

$$\frac{\textit{T-Env-Empty}}{\emptyset; \Sigma; n_{this} \vdash \Diamond} \qquad \frac{\textit{T-Env-X} \quad \Gamma; \Sigma; n_{this} \vdash T \qquad x \notin dom(\Gamma)}{\Gamma, x : T; \Sigma; n_{this} \vdash \Diamond}$$

$$\textit{T-Type} \quad \frac{\Gamma; \Sigma; n_{this} \models n_{this} \rightarrow owner(T) \qquad \Gamma; \Sigma; n_{this} \models assumptions(T)}{\Gamma; \Sigma; n_{this} \vdash T}$$

Figure 10: Well-formed Environment and Types Rules

4.3 Well-formed Environment and Types

The well-formedness rules come in two forms: $\Gamma; \Sigma; n_{this} \vdash \Diamond$ and $\Gamma; \Sigma; n_{this} \vdash T$. The first form of the rule is read: "Given the type environment Γ, the store type Σ, and a name for the current object n_{this}, the type environment is well-formed". The second form is similar, except that the conclusion is "the type T is well-formed" (Fig. 10).

According to *T-Env-Empty* and *T-Env-X*, an empty type environment is well-formed by definition, while a non-empty one is well-formed if whenever Γ is extended with a new variable x, the type T of x is well-formed. Next, according to *T-Type*, a given type T is well-formed if n_{this} has permission to access the owner domain of T, and verifies that the assumptions that T makes of its domain parameters are justified based on the current typing environment. *T-Type* uses the link permission rules defined in Figure 15. We use $\Gamma; \Sigma; n_{this} \models assumptions(C\texttt{<}\overline{p}\texttt{>})$ as a shorthand for $\forall\ (p_1 \rightarrow p_2) \in assumptions(C\texttt{<}\overline{p}\texttt{>}),\ \Gamma; \Sigma; n_{this} \models p_1 \rightarrow p_2$. To avoid duplication, we also use *T-Type* to check that dynamic types $C\texttt{<}\overline{\ell'.d}\texttt{>}$ are well-formed.

4.4 Typing Rules

FDJ's subtyping rules are given in Figure 11. Subtyping is derived from the immediate subclass relation given by the `extends` clauses in the class table CT. The subtyping relation is reflexive and transitive, and it is required that there be no cycles in the relation (other than self-cycles due to reflexivity). The `ERROR` type is a subtype of every type.

We extend the subtyping rules with a reflexive and transitive subtype relation for domains $<:_d$. Every domain p is a subtype domain of the `any` domain. To underline that `any` is an existential

$$\frac{}{p <:_d \texttt{any}}\ \textit{Subtype-Dom-Any} \qquad \frac{}{p <:_d p}\ \textit{Subtype-Dom-Refl} \qquad \frac{p <:_d p' \quad p' <:_d p''}{p <:_d p''}\ \textit{Subtype-Dom-Trans} \qquad \frac{p_i <:_d p'_i}{C\texttt{<}\overline{p}\texttt{>} <: C\texttt{<}\overline{p'}\texttt{>}}\ \textit{Subtype-Dom}$$

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha, \beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>} \ldots}{C\texttt{<}\overline{p, p'}\texttt{>} \ <: \ C'\texttt{<}\overline{p}\texttt{>}}\ \textit{Subtype-Class} \qquad \frac{}{T <: T}\ \textit{Subtype-Reflex} \qquad \frac{T <: T' \quad T' <: T''}{T <: T''}\ \textit{Subtype-Trans}$$

$$\frac{}{\texttt{ERROR} <: T}\ \textit{Subtype-Error}$$

Figure 11: Subtyping Rules

domain, note that the hierarchy of domains has depth one, i.e., there is no subtype domain relation between two domains p and p' if both p and p' are different from `any`. We also introduce a subtyping rule *Subtype-Dom* between two types $C\texttt{<}\overline{p}\texttt{>}$ and $C\texttt{<}\overline{p'}\texttt{>}$ which have the same class C. *Subtype-Dom* ensures that each domain parameter p_i in $\overline{p}$ is a subtype domain of p'_i in $\overline{p'}$.

4.5 Typechecking Rules

Typing judgments are of the form $\Gamma; \Sigma; n_{this} \vdash e : T$, and read "In the type environment Γ, store typing Σ, and receiver instance n_{this} of type T_{this}, expression e has type T." For each rule, in the presence of a subtyping relation $T' <: T$, we check that the every domain parameter of the supertype has permission to access the corresponding domain parameter of the subtype (Fig. 12). This condition allows a variable v of type T' to substitute a variable x of the type T. In the original Ownership Domain, this condition is implicit since $tparams(T') = tparams(T)$. For each rule, we ensure that the subexpressions involved have a well-formed type.

The *T-Var* rule looks up the type of a variable in Γ. The *T-Loc* rule looks up the type of a location in Σ. The *T-New* rule verifies that any assumptions that the class being instantiated makes about its domain parameters are justified based on the current typing environment. It also checks that the parameters to the constructor have types that match the types of that class's fields. Finally, it verifies that the object being created is part of the same domain as n_{this} or else is part of the domains declared by n_{this}, but not `any`.

T-New

$$\frac{\begin{array}{c} p_1 \neq \texttt{this.any} \\ \Gamma;\Sigma;n_{this} \models assumptions(C\texttt{<}\overline{p}\texttt{>}) \qquad \Gamma;\Sigma;n_{this} \vdash \overline{e} : \overline{T'} \\ fields(C\texttt{<}\overline{p}\texttt{>}) = \overline{T}\ \overline{f} \\ \overline{T'_a} \in \Gamma;\Sigma;n_{this} \vdash preciseType(\overline{T'}) \qquad \overline{T'_a} <: \overline{T} \\ \Gamma;\Sigma;n_{this} \models tparams(\overline{T}) \rightarrow tparams(\overline{T'_a}) \\ \Gamma;\Sigma;n_{this} \vdash n_{this}{:}T_{this} \\ owner(C\texttt{<}\overline{p}\texttt{>}) \in (domains(T_{this}) \cup owner(T_{this})) \\ \Gamma;\Sigma;n_{this} \vdash \overline{T'} \qquad \Gamma;\Sigma;n_{this} \vdash T_{this} \end{array}}{\Gamma;\Sigma;n_{this} \vdash \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(\overline{e}) : C\texttt{<}\overline{p}\texttt{>}}$$

T-Read

$$\frac{\begin{array}{c} \Gamma;\Sigma;n_{this} \vdash e_0 : T_0 \qquad fields(T_0) = \overline{T}\ \overline{f} \\ \Gamma;\Sigma;n_{this} \vdash T_0 \end{array}}{\Gamma;\Sigma;n_{this} \vdash e_0.f_i : [e_0/\texttt{that}]T_i}$$

T-Write

$$\frac{\begin{array}{c} \Gamma;\Sigma;n_{this} \vdash e_0 : T_0 \qquad fields(T_0) = \overline{T}\ \overline{f} \qquad T_i \in \overline{T} \qquad \Gamma;\Sigma;n_{this} \vdash e_R : T_R \\ T'_R \in \Gamma;\Sigma;n_{this} \vdash preciseType(T_R) \qquad T'_R <: [e_0/\texttt{that}]T_i \\ \Gamma;\Sigma;n_{this} \models tparams([e_0/\texttt{that}]T_i) \rightarrow tparams(T'_R) \\ \Gamma;\Sigma;n_{this} \vdash T_0 \qquad \Gamma;\Sigma;n_{this} \vdash T_R \end{array}}{\Gamma;\Sigma;n_{this} \vdash e_0.f_i = e_R : T_R}$$

T-Invk

$$\frac{\begin{array}{c} \Gamma;\Sigma;n_{this} \vdash e_0 : T_0 \qquad \Gamma;\Sigma;n_{this} \vdash \overline{e} : \overline{T_a} \\ mtype(m,T_0) = \overline{T} \rightarrow T_R \qquad mbody(m,T_0) = (\overline{x}, e_R) \\ \overline{T'_a} \in \Gamma;\Sigma;n_{this} \vdash preciseType(\overline{T_a}) \qquad \overline{T'_a} <: [\overline{e}/\overline{x}, e_0/\texttt{that}]\overline{T} \\ \Gamma;\Sigma;n_{this} \models tparams([\overline{e}/\overline{x}, e_0/\texttt{that}]\overline{T}) \rightarrow tparams(\overline{T'_a}) \\ \Gamma;\Sigma;n_{this} \vdash T_0 \qquad \Gamma;\Sigma;n_{this} \vdash \overline{T_a} \end{array}}{\Gamma;\Sigma;n_{this} \vdash e_0.m(\overline{e}) : [\overline{e}/\overline{x}, e_0/\texttt{that}]T_R}$$

T-Var

$$\frac{\Gamma(x) = C\texttt{<}\overline{p}\texttt{>}}{\Gamma;\Sigma;n_{this} \vdash x : C\texttt{<}\overline{p}\texttt{>}}$$

T-Loc

$$\frac{\Sigma(\ell) = C\texttt{<}\overline{p}\texttt{>}}{\Gamma;\Sigma;n_{this} \vdash \ell : C\texttt{<}\overline{p}\texttt{>}}$$

T-Cast

$$\frac{\Gamma;\Sigma;n_{this} \vdash e : T' \qquad \Gamma;\Sigma;n_{this} \vdash T}{\Gamma;\Sigma;n_{this} \vdash (T)e : T}$$

T-Context

$$\frac{\Gamma;\Sigma;n_{this} \vdash \Sigma[\ell] \qquad \Gamma;\Sigma;\ell \vdash e : T \qquad \Gamma;\Sigma;\ell \vdash T}{\Gamma;\Sigma;n_{this} \vdash \ell \triangleright e : T}$$

Figure 12: Typechecking rules.

The rule for field reads looks up the declared type of the field T_i using the auxiliary judgment *fields*. The *T-Write* rule looks up the declared type of the field and ensure that the type of the right-hand-side expression is a subtype of T_i. The cast rule simply checks that the expression being

$$\frac{p \neq \texttt{any}}{isPrecise(p)}\ \textit{Aux-isPrecise} \qquad \frac{T = C\texttt{<}\overline{p}\texttt{>}}{tparams(T) = \overline{p}}\ \textit{Aux-TParams}$$

$$\frac{isPrecise(\overline{p'}) \qquad \forall i, (p_0 \rightarrow p'_i) \in linkdecls(C\texttt{<}\overline{p}\texttt{>})}{solveAny(p_0, C\texttt{<}\overline{p}\texttt{>}) = \overline{p'}}\ \textit{Aux-SolveAny}$$

$$\frac{isPrecise(\overline{p})}{\Gamma; \Sigma; n_{this} \vdash preciseType(C\texttt{<}\overline{p}\texttt{>}) = C\texttt{<}\overline{p}\texttt{>}}\ \textit{Aux-PreciseTypeDom}$$

$$\frac{\Gamma; \Sigma; n_{this} \vdash n : T \qquad p_i = n.\texttt{any} \Longrightarrow p'_i = solveAny(owner(T), T)}{\Gamma; \Sigma; n_{this} \vdash preciseType(C\texttt{<}\overline{p}\texttt{>}) = \overline{[p'_i/n.\texttt{any}]C\texttt{<}\overline{p}\texttt{>}}}\ \textit{Aux-PreciseTypeAny}$$

$$\frac{(\ell_0.d_0 \rightarrow \ell''_i.d'_i) \in \Delta}{solveAny(\ell_0.d_0, C\texttt{<}\overline{\ell'.d}\texttt{>}) = \overline{\ell''.d'}}\ \textit{Aux-DynSolveAny}$$

$$\frac{\begin{array}{c} CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha}, \overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \ldots\ \{\ \ldots\ \overline{F}; \ldots\ \} \\ \overline{F} = \overline{T}\ \overline{f} \qquad fields(C'\texttt{<}\overline{\ell'.d'}\texttt{>}) = \overline{T'}\ \overline{f'} \\ \Sigma[\ell] = C\texttt{<}\overline{\ell'.d'}, \overline{\ell''.d''}\texttt{>} \qquad \ell_i.d_i \in solveAny(owner(\Sigma[\ell]), \Sigma[\ell]) \end{array}}{fields(C\texttt{<}\overline{\ell'.d'}, \overline{\ell''.d''}\texttt{>}) = \qquad ([\ell/\texttt{this}][\overline{\ell'.d'}/\overline{\alpha}, \overline{\ell''.d''}/\overline{\beta}][\ell_i.d_i/\texttt{any}]\ \overline{T}\ \overline{f}), \overline{T'}\ \overline{f'}}\ \textit{Aux-DynFields}$$

$$\frac{C'\texttt{<}\overline{p'}, \overline{p''}\texttt{>} <: C\texttt{<}\overline{p}\texttt{>} \qquad |\overline{p'}| = |\overline{p}| \qquad \forall i \in 1..|p|\ \Gamma; \Sigma; n_{this} \models p_i \rightarrow p'_i}{\Gamma; \Sigma; n_{this} \models tparams(C\texttt{<}\overline{p}\texttt{>}) \rightarrow tparams(C'\texttt{<}\overline{p'}, \overline{p''}\texttt{>})}\ \textit{Aux-TParamsLink}$$

Figure 13: Auxiliary judgments specific to `any`. Others judgments that are common to both Flexible Ownership Domains and Ownership Domains are in the technical report.

cast is well-typed; a run-time check will determine if the value that comes out of the expression matches the type of the cast.

Rule *T-Invk* looks up the invoked method's type using the auxiliary judgment *mtype*, and verifies that the actual argument types are subtypes of the method's argument types. Finally, the *T-Context* typing rule for an executing method checks the method's body in the context of the new receiver ℓ.

Resolving `any`. To discuss how Flexible Ownership Domains resolves `any`, we first we introduce additional definitions. A domain p is precise if p is different than `any`. A type $C\texttt{<}\overline{p}\texttt{>}$ is precise if each domain p_i is precise. We introduce the auxiliary judgment *solveAny*, which resolves `any` into a set of precise domains $\overline{p'}$ according to the link declarations of the enclosing type (*Aux-SolveAny*). To resolve a type $C\texttt{<}\overline{p}\texttt{>}$, we introduce the auxiliary judgment *preciseType*. If a type $C\texttt{<}\overline{p}\texttt{>}$ has all its domain p_i different than `any`, *preciseType* simply returns $C\texttt{<}\overline{p}\texttt{>}$. (*Aux-PreciseTypeDom*). Otherwise, if $n.\texttt{any}$ is one of the domain parameters of $C\texttt{<}\overline{p}\texttt{>}$, where n is a name of an object, *preciseType* passes the type T of n to *solveAny*. Next, *preciseType* uses the result p' of *solveAny*, and returns a set of precise types by substituting $n.\texttt{any}$ with the corresponding precise domain (rlAux-PreciseTypeAny). For the $[p'/n.\texttt{any}]$ substitution, there are two alternatives:

$$\exists p' \in solveAny(owner(T), T) \mid [p'/n.\texttt{any}]C\texttt{<}\overline{p}\texttt{>} \tag{1}$$

$$C\texttt{<}\overline{\exists p' \in solveAny(owner(T), T) \mid [p'/n.\texttt{any}]p}\texttt{>} \tag{2}$$

In (1), the same result p' of *solveAny* substitutes all the occurrences $n.\texttt{any}$ in $\overline{p}$. In (2), assuming that *solveAny* returns multiple p', different occurrences of $n.\texttt{any}$ in $\overline{p}$ are substituted with different p'. This provides additional flexibility, but requires more careful reasoning. Since, we expect developers to use `any` rarely only when they cannot decide a precise domain and mostly for the owner domain, we selected the first alternative.

T-New, *T-Write*, *T-Invk* use the result of *preciseType* in the left-hand-side of the subtyping relations, ensuring that the rules distinguish between `any` in different classes. *Aux-DynFields* determines the field types of a location ℓ by substituting formal domain parameters to actual domains, and the context variable `this` to ℓ. Since at runtime `any` resolves to a precise domain, *Aux-DynFields* ensures that one of the results of *solveAny* substitutes `any`.

The typing rules for classes and declarations have the form "class C is OK," and "method/link declaration is OK in C." The class rule checks that the methods and links in the class are well-formed, and that field types in the class are well-formed (Fig. 14).

The *Meth-OK* rule checks that the method body is well typed, and uses the *override* auxiliary judgment to verify that methods are overridden with a method of the same type. It also verifies that the argument types and the return type are well-formed.

$$\textit{ClsOK}\quad \frac{\begin{array}{c}\overline{M}\ \mathit{OK\ in}\ C \qquad \overline{L}\ \mathit{OK\ in}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\\ \mathit{fields}(C'\texttt{<}\overline{\alpha}\texttt{>}) = \overline{T'}\ \overline{g} \qquad \overline{F} = \overline{T}\ \overline{f}\\ \{\texttt{this} : C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\};\ \emptyset;\ \texttt{this} \vdash \overline{T}\\ K = C(\overline{T'}\ \overline{g},\ \overline{T}\ \overline{f})\ \{\ \mathit{super}(\overline{g});\ \mathit{this}.\overline{f} = \overline{f};\ \}\end{array}}{\begin{array}{c}\texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\\ \texttt{assumes}\ \overline{\gamma} \to \overline{\delta}\ \{\ \overline{D};\ \overline{L};\ \overline{F};\ K\ \overline{M};\}\ \mathit{OK}\end{array}}$$

$$\textit{MethOK}\quad \frac{\begin{array}{c}CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ldots\\ \mathit{override}(m, C'\texttt{<}\overline{\alpha}\texttt{>}, \overline{T} \to T_R)\\ \{\overline{x} : \overline{T};\ \texttt{this} : C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\};\ \emptyset;\ \texttt{this} \vdash e : T'_R\\ \{\overline{x} : \overline{T};\ \texttt{this} : C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\};\ \emptyset;\ \texttt{this} \vdash T'_R\\ \{\overline{x} : \overline{T};\ \texttt{this} : C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\};\ \emptyset;\ \texttt{this} \vdash T'_R \qquad T'_R <: T_R\\ \{\overline{x} : \overline{T};\ \texttt{this} : C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\};\ \emptyset;\ \texttt{this} \models \mathit{tparams}(T_R) \to \mathit{tparams}(T'_R)\\ \{\overline{x} : \overline{T};\ \texttt{this} : C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\};\ \emptyset;\ \texttt{this} \vdash \overline{T}\end{array}}{T_R\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}\ \mathit{OK\ in}\ C}$$

$$\textit{LinkOK}\quad \frac{\begin{array}{c}\texttt{any} \notin \{p_1, p_2\} \qquad \{p_1, p_2\} \cap \mathit{domains}(C\texttt{<}\overline{\alpha}\texttt{>}) \neq \emptyset\\ p_1 \notin \mathit{domains}(C\texttt{<}\overline{\alpha}\texttt{>}) \Longrightarrow (\texttt{this} : C\texttt{<}\overline{\alpha}\texttt{>}; \emptyset; \texttt{this} \models p_1 \to \mathit{owner}(C\texttt{<}\overline{\alpha}\texttt{>}))\\ p_2 \notin \mathit{domains}(C\texttt{<}\overline{\alpha}\texttt{>}) \Longrightarrow (\texttt{this} : C\texttt{<}\overline{\alpha}\texttt{>}; \emptyset; \texttt{this} \models \texttt{this} \to p_2)\end{array}}{\texttt{link}\ p_1 \to p_2\ \mathit{OK\ in}\ C\texttt{<}\overline{\alpha}\texttt{>}}$$

$$\textit{T-Assumptions}\quad \frac{\forall \ell \in \mathit{domain}(\Sigma)\quad (\mathit{links}(\Sigma[\ell])) \subseteq \Delta}{\Sigma_\Delta\ \mathit{OK}}$$

$$\textit{T-Store}\quad \frac{\begin{array}{c}\mathit{domain}(S) = \mathit{domain}(\Sigma) \qquad S[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) \iff \Sigma[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>}\\ \mathit{fields}(\Sigma[\ell]) = \overline{T}\ \overline{f} \Longrightarrow (S[\ell, i] = \ell'') \wedge \Sigma[\ell''] <: T_i\\ \mathit{tparams}(T_i) \to \mathit{tparams}(\Sigma[\ell'']) \in \Delta\\ \Sigma_\Delta\ \mathit{OK} \qquad (S[\ell, i] = \ell'') \Longrightarrow (\mathit{owner}(\Sigma[\ell]) \to \mathit{owner}(\Sigma[\ell''])) \in \Delta\end{array}}{\Sigma_\Delta \vdash S}$$

Figure 14: Class, Method and Store Typing.

The *Link-OK* rule verifies that one of the two domains in the link declaration was declared locally, preventing a class from linking two external domains together. The rule also ensures that if the declaration links an internal and an external domain, there is a corresponding linking relationship between `this` and the external domain. None of the domains in a link declaration can be `any`.

The store typing rule ensures that the store type gives a type to each location in the store's domain that is consistent with the classes and ownership parameters in the actual store. For every value ℓ'' of a field in the store, the type of ℓ'' must be a subtype of the declared field type. In the presence of `any`, *T-Store* ensures that the subtyping relation is consistent with domain link declarations.

The check $\Sigma_\Delta\ OK$, defined by the *T-Assumptions* rule, ensures that the links of every type is part of Δ as the union of all links between runtime domains based on actual link declarations in the source code. Finally, the last check of *T-Store* verifies link soundness: if object ℓ refers to object ℓ'' in it's ith field, then the link declarations implied by the store type Σ implies that owner domain of $\Sigma[\ell]$ has permission to access the owner domain of $\Sigma[\ell'']$.

$$\Delta = \bigsqcup_{\ell \in domain(\Sigma)} (links(\Sigma[\ell]))$$

Figure 15 shows the rules for determining whether an object named by n or a domain p has permission to access another domain p'. These rules come in two forms: $\Gamma; \Sigma; n_{this} \models n \to p$ and $\Gamma; \Sigma; n_{this} \models p \to p'$. The first form of rule is read, "Given the type environment Γ, the store type Σ, and a name for the current object n_{this}, the object named by n has permission to access domain p." The second form is similar, except that the conclusion is that any object in domain p has permission to access domain p'. The two forms allow us to reason about access permission both on a per-object basis and on a per-domain basis.

The *T-DynamicLink* rule can be used to conclude that two domains are linked if there is an object in the store that explicitly linked them. The *T-DeclaredLink* rule allows the type system to rely on any links that are declared or assumed in the context of the class of n_{this}. The *T-ChildRef* rule states that any object named by n has permission to access one of its own domains $n.d$. The *T-SelfLink* rule states that every domain can access itself. The *T-LinkRef* rule allows the object named by n to access a domain if the owner of n can access that domain. The *T-PublicLink* and *T-PublicRef* rules allow objects and domains to access the public domain of some object in a domain they already have access to.

$$\textit{T-DynamicLink}\quad \frac{(\ell_1.d_1 \to \ell_2.d_2) \in \Delta}{\Gamma; \Sigma; n_{this} \models \ell_1.d_1 \to \ell_2.d_2}$$

$$\textit{T-DeclaredLink}\quad \frac{\Gamma; \Sigma; n_{this} \vdash n_{this} : T \qquad (p_1 \to p_2) \in \mathit{linkdecls}(T)}{\Gamma; \Sigma; n_{this} \models p_1 \to p_2}$$

$$\textit{T-ChildRef}\quad \frac{}{\Gamma; \Sigma; n_{this} \models n \to n.d}$$

$$\textit{T-SelfLink}\quad \frac{\mathit{isPrecise}(p)}{\Gamma; \Sigma; n_{this} \models p \to p}$$

$$\textit{T-LinkRef}\quad \frac{\Gamma; \Sigma; n_{this} \vdash n : T \qquad \Gamma; \Sigma; n_{this} \models \mathit{owner}(T) \to p}{\Gamma; \Sigma; n_{this} \models n \to p}$$

$$\textit{T-PublicLink}\quad \frac{\Gamma; \Sigma; n_{this} \vdash n : T \qquad \Gamma; \Sigma; n_{this} \models p \to \mathit{owner}(T) \qquad \mathit{public}(d)}{\Gamma; \Sigma; n_{this} \models p \to n.d}$$

$$\textit{T-PublicRef}\quad \frac{\Gamma; \Sigma; n_{this} \vdash n : T \qquad \Gamma; \Sigma; n_{this} \models n_s \to \mathit{owner}(T) \qquad \mathit{public}(d)}{\Gamma; \Sigma; n_{this} \models n_s \to n.d}$$

$$\textit{T-LinkShared}\quad \frac{}{\Gamma; \Sigma; n_{this} \models p \to \texttt{shared}}$$

$$\textit{T-LinkSharedRef}\quad \frac{}{\Gamma; \Sigma; n_{this} \models n \to \texttt{shared}}$$

$$\textit{T-LinkDomAny}\quad \frac{\Gamma; \Sigma; n_{this} \vdash n : T \qquad \mathit{isPrecise}(p) \qquad p' \in \mathit{solveAny}(\mathit{owner}(T), T) \qquad \Gamma; \Sigma; n \models p \to p'}{\Gamma; \Sigma; n_{this} \models p \to n.\texttt{any}}$$

$$\textit{T-LinkAnyDom}\quad \frac{\Gamma; \Sigma; n_{this} \vdash n : T \qquad p' \in \mathit{solveAny}(\mathit{owner}(T), T) \qquad \Gamma; \Sigma; n \models p' \to p}{\Gamma; \Sigma; n_{this} \models n.\texttt{any} \to p}$$

Figure 15: Link permission rules.

Previous link permission rules are the same as in Ownership Domains. For Flexible Ownership Domains, we introduce two additional rules. *T-LinkAnyDom* states that any domain p' that owner of an object named by n has permissions to access is linked to a domain p, if p' is linked to p in the context of n. *T-LinkDomAny* states that domain p is linked to any domain p' that owner of an object named by n has permissions to access if p is linked to p' in the context of n. To avoid a circular definition, *T-LinkDomAny* requires p to be a precise domain. One could argue that we should simply require p to be part of the result of *solveAny*; however, if p is a public domain of an object n', $n.\texttt{any}$ may be resolved to owner of n', in which case $n'.d$ is not part of *solveAny*, but the linking permission holds.

Figure 16 shows the definitions of many auxiliary functions used earlier in the semantics. These definitions are straightforward and in many cases are derived directly from rules in Featherweight Java. The Aux-Public rule checks whether a domain is public. The next few rules define the *domains*, *links*, *assumptions*, and $fields$ functions by looking up the declarations in the class and adding them to the declarations in superclasses. The *linkdecls* function just returns the union of the *links* and *assumptions* in a class, while the *owner* function just returns the first domain parameter (which represents the owning domain in our formal system).

The *mtype* function looks up the type of a method in the class; if the method is not present, it looks in the superclass instead. The *mbody* function looks up the body of a method in a similar way. Finally, the *override* function verifies that if a superclass defines method m, it has the same type as the definition of m in a subclass.

$$CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{assumes}\ \overline{\gamma} \to \overline{\delta}\ \{\ \overline{D};\ \overline{L};\ \overline{F};\ K\ \overline{M};\ \}$$

$$\frac{(\texttt{public domain}\ d) \in \overline{D}}{public(d)}\ \textit{Aux-Public}$$

$$\frac{\overline{D} = \overline{\texttt{public}_{opt}\ \texttt{domain}\ d_C} \qquad domains(C'\texttt{<}\overline{p}\texttt{>}) = \overline{d'}}{domains(C\texttt{<}\overline{p},\overline{p'}\texttt{>}) = \overline{this.d_C}, \overline{d'}}\ \textit{Aux-Domains}$$

$$\frac{}{domains(\texttt{Object}\texttt{<}\overline{\alpha_0}\texttt{>}) = \emptyset}\ \textit{Aux-Domains-Obj}$$

$$\frac{\texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}}{params(C) = \overline{\alpha}}\ \textit{Aux-Params}$$

$$\frac{T = C\texttt{<}\overline{p}\texttt{>}}{tparams(T) = \overline{p}}\ \textit{Aux-TParams}$$

$$\frac{\overline{L} = \overline{\texttt{link}}\ \overline{p_c} \to \overline{p'_c} \qquad links(C'\texttt{<}\overline{p}\texttt{>}) = \overline{p_s} \to \overline{p'_s}}{links(C\texttt{<}\overline{p},\overline{p'}\texttt{>}) = ([\texttt{that}/\texttt{this}][\overline{p}/\overline{\alpha},\overline{p'}/\overline{\beta}]\ (\overline{p_c} \to \overline{p'_c})),\ \overline{p_s} \to \overline{p'_s}}\ \textit{Aux-Links}$$

$$\frac{assumptions(C'\texttt{<}\overline{p}\texttt{>}) = \overline{p_s} \to \overline{p'_s}}{assumptions(C\texttt{<}\overline{p},\overline{p'}\texttt{>}) = ([\texttt{that}/\texttt{this}][\overline{p}/\overline{\alpha},\overline{p'}/\overline{\beta}]\ (\overline{\gamma} \to \overline{\delta})), \overline{p_s} \to \overline{p'_s}}\ \textit{Aux-Assume}$$

$$\frac{\overline{F} = \overline{T}\ \overline{f} \qquad fields(C'\texttt{<}\overline{p}\texttt{>}) = \overline{T'}\ \overline{f'}}{fields(C\texttt{<}\overline{p},\overline{p'}\texttt{>}) = ([\texttt{that}/\texttt{this}][\overline{p}/\overline{\alpha},\overline{p'}/\overline{\beta}]\ \overline{T}\ \overline{f}), \overline{T'}\ \overline{f'}}\ \textit{Aux-Fields}$$

$$\frac{}{fields(\texttt{Object}\texttt{<}\overline{\alpha_0}\texttt{>}) = \emptyset}\ \textit{Aux-Fields-Obj}$$

$$\frac{}{linkdecls(C\texttt{<}\overline{p}\texttt{>}) = links(C\texttt{<}\overline{p}\texttt{>}) \cup assumptions(C\texttt{<}\overline{p}\texttt{>})}\ \textit{Aux-LinkDecls}$$

$$\frac{}{owner(C\texttt{<}\overline{p}\texttt{>}) = p_1}\ \textit{Aux-Owner}$$

$$\frac{(T_R\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}) \in \overline{M}}{mtype(m, C\texttt{<}\overline{p},\overline{p'}\texttt{>}) = [\texttt{that}/\texttt{this}][\overline{p}/\overline{\alpha},\overline{p'}/\overline{\beta}]\ \overline{T} \to T_R}\ \textit{Aux-MType1}$$

$$\frac{m\ is\ not\ defined\ in\ \overline{M}}{mtype(m, C\texttt{<}\overline{p},\overline{p'}\texttt{>}) = mtype(m, C'\texttt{<}\overline{p}\texttt{>})}\ \textit{Aux-MType2}$$

$$\frac{(T_R\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}) \in \overline{M}}{mbody(m, C\texttt{<}\overline{p}\texttt{>}) = [\texttt{that}/\texttt{this}][\overline{p}/\overline{\alpha}]\ (\overline{x},\ e)}\ \textit{Aux-MBody1}$$

$$\frac{m\ is\ not\ defined\ in\ \overline{M}}{mbody(m, C\texttt{<}\overline{p},\overline{p'}\texttt{>}) = mbody(m, C'\texttt{<}\overline{p}\texttt{>})}\ \textit{Aux-MBody2}$$

$$\frac{(mtype(m, C\texttt{<}\overline{p}\texttt{>}) = \overline{T'} \to T') \implies (\overline{T} = \overline{T'} \wedge T = T')}{override(m, C\texttt{<}\overline{p}\texttt{>}, \overline{T} \to T)}\ \textit{Aux-Override}$$

Figure 16: Auxiliary Definitions

5 Flexible Ownership Domains Properties

In this section, we state and prove type soundness and link soundness adapted from Featherweight Domain Java (FDJ). To account for existential domains, we state the subtyping link soundness theorem which ensures that the subtyping relations are consistent with domain link declarations. We also state and prove the standard Progress and Preservation theorems, along with the Super Type Lemma which we use in the proof of the Preservation theorem.

Lemma 2 (Lemma)

If $mtype(m, D) = \overline{T} \rightarrow T_R$ *then* $mtype(m, C) = \overline{T} \rightarrow T_R$ *for all* $C <: D$.

Proof: By induction on the derivation of $C <: D$ and $mtype(m, D)$. ■

Lemma 3 (Substitution Lemma)

If $\Gamma, \overline{x} : \overline{B} \vdash e : D$ *and* $\Gamma \vdash \overline{d} : \overline{A}$ *where* $\overline{A} <: \overline{B}$*, then* $\Gamma \vdash [\overline{d}/\overline{x}]e : C$ *for some* $C <: D$.

Proof: By induction on the typing rules. ■

Lemma 4 (Weakening Lemma)

If $\Gamma \vdash e : C$*, then* $\Gamma, x : D \vdash e : C$.

Proof: By induction on the typing rules. ■

Lemma 5 (Store Lemma)

If $fields(C{<}\overline{\ell'.d}{>}) = \overline{T}\ \overline{f}$ *and* $S[\ell] = C{<}\overline{\ell'.d}{>}(\overline{v})$ *and* $\emptyset; \Sigma; \theta \vdash \overline{v} : \Sigma[\overline{v}]$
then $\Sigma[\overline{v}] <: \overline{T}$*, and* $assumptions(\Sigma[\overline{v}]) \in \Delta$.

Proof: Based on rules T-New and R-New.

$$\frac{\begin{array}{c} p_1 \neq \texttt{this.any} \\ \Gamma;\Sigma;n_{this} \models assumptions(C{<}\overline{p}{>}) \qquad \Gamma;\Sigma;n_{this} \vdash \overline{e} : \overline{T'} \qquad fields(C{<}\overline{p}{>}) = \overline{T}\ \overline{f} \\ \overline{T'_a} \in \Gamma;\Sigma;n_{this} \vdash preciseType(\overline{T'}) \qquad \overline{T'_a} <: \overline{T} \qquad \Gamma;\Sigma;n_{this} \models tparams(\overline{T}) \rightarrow tparams(\overline{T'_a}) \\ \Gamma;\Sigma;n_{this} \vdash n_{this}{:}T_{this} \\ owner(C{<}\overline{p}{>}) \in (domains(T_{this}) \cup owner(T_{this})) \\ \Gamma;\Sigma;n_{this} \vdash \overline{T'} \qquad \Gamma;\Sigma;n_{this} \vdash T_{this} \end{array}}{\Gamma;\Sigma;n_{this} \vdash \texttt{new}\ C{<}\overline{p}{>}(\overline{e}) : C{<}\overline{p}{>}}\,[\textit{T-New}]$$

Take $n_{this} = \theta$, and $n_{that} = \ell$

$S[\ell] = C{<}\overline{p}{>}(\overline{v})$	By hypothesis
$fields(C{<}\overline{p}{>}) = \overline{T}\ \overline{f}$	By hypothesis
$\emptyset;\Sigma;\theta \vdash \overline{v} : \Sigma[\overline{v}]$	By hypothesis
$\emptyset;\Sigma;\theta \vdash \texttt{new}\ C{<}\overline{p}{>}(\overline{v}) : C{<}\overline{p}{>}$	By T-New
$\Sigma[\overline{v}] \in \emptyset;\Sigma;\theta \vdash preciseType(\Sigma[\overline{v}])$	Since $isPrecise(\Sigma[\overline{v}])$
$\Sigma[\overline{v}] <: \overline{T}$	By subderivation of T-New
$\emptyset;\Sigma;\theta \vdash \Sigma[\overline{v}]$	By subderivation of T-New
$\emptyset;\Sigma;\theta \models assumptions(\Sigma[\overline{v}])$	By subderivation of T-New

■

Lemma 6 (Method Lemma)

If $mtype(m, C{<}\overline{p},\overline{p'}{>}) = \overline{T} \rightarrow T_R$

and $mbody(m, C{<}\overline{p},\overline{p'}{>}) = (\overline{x}, e_R)$

then for some $C'{<}\overline{p}{>}$ *with* $C{<}\overline{p},\overline{p'}{>} <: C'{<}\overline{p}{>}$,

there exists $T_0 <: T_R$ *such that* $\overline{x} : \overline{T}, \texttt{this} : C'{<}\overline{p}{>} \vdash e_R : T_0$.

Proof: By induction on *mtype*. ■

Lemma 7 (Super Type Lemma)

If

$\emptyset; \Sigma; \theta \vdash e : T$,

$\emptyset; \Sigma; \theta \vdash T$,

$\Sigma_\Delta \vdash S$,

$S; \theta \vdash e \mapsto \ell \triangleright e', S$

$\emptyset; \Sigma; \theta \vdash \ell \triangleright e' : T'$

$T' <: T$,

$\emptyset; \Sigma; \theta \models tparams(T) \rightarrow tparams(T')$

then

$\Gamma; \Sigma; \theta \vdash T'$

Proof: By induction over the derivation of $\emptyset, \Sigma, \theta \vdash e : T$.

$\emptyset; \Sigma; \theta \vdash e : T$	By induction hypothesis	(1)
$\emptyset; \Sigma; \theta \vdash T$	By induction hypothesis	(2)
$\emptyset; \Sigma; \theta \models \theta \rightarrow owner(T)$	Since (2), by subderivation of T-Type	(3)
$\emptyset; \Sigma; \theta \models assumptions(T)$	Since (2), by subderivation of T-Type	(4)
$assumptions(T) \subseteq \Delta$	by subderivation of T-DynamicLink	(5)
$T' <: T$	by hypothesis	(6)
$tparams(T') <:_d tparams(T)$	By Subtype-Dom	(7)

Subcase $owner(T) \neq$ `this.any` and $owner(T') \neq$ `that.any`. That is, $owner(T) = owner(T')$

$\emptyset; \Sigma; \theta \models \theta \rightarrow owner(T')$	Since $owner(T') = owner(T)$	(8)

Subcase $owner(T) \neq$ `this.any` and $owner(T') =$ `that.any`. `that.any` $\not<:_d$ $owner(T)$ this subcase cannot occur.

Subcase $owner(T) = \texttt{this.any}$. That is $owner(T') <:_d owner(T)$

$$\emptyset; \Sigma; \theta \vdash tparams(T) \to tparams(T') \qquad \text{By hypothesis} \quad (9)$$

$$\emptyset; \Sigma; \theta \vdash owner(T) \to owner(T') \qquad owner(T) \in tparams(T), owner(T') \in tparams(T') \quad (10)$$

$$\emptyset; \Sigma; \theta \vdash \texttt{this.any} \to owner(T') \qquad \text{By subcase hypothesis} \quad (11)$$

$$owner(\Sigma[\theta]) \in solveAny(owner(\Sigma[\theta]), \Sigma[\theta]) \qquad \text{By subderivation of T-LinkAnyDom} \quad (12)$$

$$\emptyset; \Sigma; \theta \models owner(\Sigma[\theta]) \to owner(T') \qquad \text{By subderivation of T-LinkAnyDom} \quad (13)$$

$$\emptyset; \Sigma; \theta \models \theta \to owner(T') \qquad \text{By inversion of T-LinkRef} \quad (14)$$

$$\Gamma; \Sigma; \theta \vdash \ell \triangleright e' : T' \qquad \text{By induction hypothesis} \quad (15)$$

$$\Gamma; \Sigma; \theta \vdash \Sigma[\ell] \qquad \text{By subderivation of T-Context} \quad (16)$$

$$\Gamma; \Sigma; \ell \vdash e' : T' \qquad \text{By subderivation of T-Context} \quad (17)$$

$$\Gamma; \Sigma; \ell \vdash T' \qquad \text{By subderivation of T-Context} \quad (18)$$

$$\Gamma; \Sigma; \theta \models assumptions(\Sigma[\ell]) \qquad \text{Since (16), by subderivation of T-Type} \quad (19)$$

$$assumptions(\Sigma[\ell]) \in \Delta \qquad \text{By subderivation of T-DynamicLink} \quad (20)$$

$$\Gamma; \Sigma; \ell \models assumptions(T') \qquad \text{Since (18), by subderivation of T-Type} \quad (21)$$

$$assumptions(T') \in linkdecls(\Sigma[\ell]) \qquad \text{By subderivation of T-DeclaredLink} \quad (22)$$

$$linkdecls(\Sigma[\ell]) = links(\Sigma[\ell]) \cup assumptions(\Sigma[\ell]) \qquad \text{By Aux-LinkDecls} \quad (23)$$

$$assumptions(T') \in links(\Sigma[\ell]) \cup assumptions(\Sigma[\ell]) \quad (24)$$

$$links(\Sigma[\ell]) \subseteq \Delta \qquad \text{By } \Sigma_\Delta\ OK \quad (25)$$

$$assumptions(\Sigma[\ell]) \subseteq \Delta \qquad \text{By (20)} \quad (26)$$

$$assumptions(T') \subseteq \Delta \quad (27)$$

$$\Gamma; \Sigma; \theta \models assumptions(T') \qquad \text{By inversion of T-DynamicLink} \quad (28)$$

$$\Gamma; \Sigma; \theta \vdash T' \qquad \text{Since (14) and (28), by inversion of T-Type} \quad (29)$$

This proves the lemma. (30)

Theorem 8 (Type Preservation, a.k.a. Subject Reduction)

If

$\emptyset; \Sigma; \theta \vdash e : T,$

$\emptyset; \Sigma; \theta \vdash T,$

$\Sigma_\Delta \vdash S,$

and $S; \theta \vdash e \mapsto e', S'$

then there exists $\Sigma' \supseteq \Sigma,\ \Delta' \supseteq \Delta,$ *and* $T' <: T$ *such that:*

$\emptyset; \Sigma'; \theta \models tparams(T) \to tparams(T'),$

$\emptyset; \Sigma'; \theta \vdash e' : T',$

$\emptyset; \Sigma'; \theta \vdash T',$

and $\Sigma'_{\Delta'} \vdash S'.$

Proof: By induction over the derivation of $S \vdash e \mapsto e', S'$, with a case analysis on the outermost reduction rule used.

Case *R-New*:. Then $e = \texttt{new}\ C{<}\overline{p}{>}(\overline{v}) \qquad e' = \ell$

$$\frac{\ell \notin domain(S) \qquad S' = S[\ell \mapsto C{<}\overline{p}{>}(\overline{v})]}{S; \theta \vdash \texttt{new}\ C{<}\overline{p}{>}(\overline{v}) \mapsto \ell, S'}\ \textit{R-New}$$

To show:

$$\emptyset; \Sigma'; \theta \models tparams(T) \to tparams(T') \tag{1}$$

$$\emptyset; \Sigma'; \theta \vdash e' : T' \tag{2}$$

$$\emptyset; \Sigma'; \theta \vdash T' \tag{3}$$

$$\Sigma'_{\Delta'} \vdash S' \tag{4}$$

By *T-New*, we have:

$$\dfrac{\begin{array}{c} p_1 \neq \texttt{this.any} \\ \Gamma;\Sigma;n_{this} \models assumptions(C\texttt{<}\overline{p}\texttt{>}) \qquad \Gamma;\Sigma;n_{this} \vdash \overline{e} : \overline{T'} \\ fields(C\texttt{<}\overline{p}\texttt{>}) = \overline{T}\ \overline{f} \\ \overline{T'_a} \in \Gamma;\Sigma;n_{this} \vdash preciseType(\overline{T'}) \qquad \overline{T'_a} <: \overline{T} \\ \Gamma;\Sigma;n_{this} \models tparams(\overline{T}) \rightarrow tparams(\overline{T'_a}) \\ \Gamma;\Sigma;n_{this} \vdash n_{this}{:}T_{this} \\ owner(C\texttt{<}\overline{p}\texttt{>}) \in (domains(T_{this}) \cup owner(T_{this})) \\ \Gamma;\Sigma;n_{this} \vdash \overline{T'} \qquad \Gamma;\Sigma;n_{this} \vdash T_{this} \end{array}}{\Gamma;\Sigma;n_{this} \vdash \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(\overline{e}) : C\texttt{<}\overline{p}\texttt{>}}\ [\text{T-New}]$$

By assumption the object creation expression has type $C\texttt{<}\overline{\ell'.d}\texttt{>}$. *R-New* extends the store type to give ℓ the type $C\texttt{<}\overline{\ell'.d}\texttt{>}$, so that the rewritten expression retains the same type. The check that the current object can access the created object is justified by the check that the new object is created inside one of the creating object's domains, or in the same domain as the creating object. *T-New* ensures that the assumptions are satisfied, which shows that the type of the rewritten expression is well-formed.

The store remains well-typed because ℓ is fresh and the values in the fields of ℓ have appropriate types, again by assumptions from the object creation expression typing rule. The assumptions of $C\texttt{<}\overline{\ell'.d}\texttt{>}$ are satisfied by the existing links. The set of links is extended with the links of $C\texttt{<}\overline{\ell'.d}\texttt{>}$ and $\Sigma'_{\Delta'}$ OK is justified.

For dynamic types, the auxiliary judgement $fields$ substitute any with the result of *solveAny* in the scope of $C\texttt{<}\overline{\ell'.d}\texttt{>}$. The highlighted condition of *T-New* ensures that the subtyping is consistent with the links and implies that owner of a field type has permissions to access the owner of values in the fields of ℓ. When the owner of the declared field type is `any`, the owner is substituted by the result of *solveAny*. Also, since the owner of the new expression is part of *solveAny*, implies that owner of $\Sigma[\ell]$ has permissions to access of the owner of each value in the fields of ℓ.

Consider T_i as the declared type of each value v_i in the fields of ℓ. If the owner of T_i is precise, the proof follows the same steps as the proof of Ownership Domains. If the owner of v_i is a local domain of ℓ, the owner of ℓ has permissions to access it. If the owner of v_i is a domain parameter, according to *ClsOK* the domain parameter can be only a domain where the owner has access to. Finally, if the owner of v_i is a public domain of another field v_k, by generalized induction owner of the new location has permission to access owner of v_k and its public domains.

Take $\Sigma' = \Sigma[\ell \mapsto C\texttt{<}\overline{\ell'.d}\texttt{>}]$	
$\emptyset; \Sigma'; \theta \vdash e : C\texttt{<}\overline{\ell'.d}\texttt{>}$	By T-New
$\Sigma'[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>}$	By definition of Σ'
$\emptyset; \Sigma'; \theta \vdash \ell : C\texttt{<}\overline{\ell'.d}\texttt{>}$	By inversion of T-Loc
$\emptyset; \Sigma'; \theta \vdash e' : C\texttt{<}\overline{\ell'.d}\texttt{>}$	By $e' = \ell$
Take $T' = T = C\texttt{<}\overline{\ell'.d}\texttt{>}$	This proves (1), and (2)

$\Sigma[\theta] = T_{this}$	By subderivation of T-New
$owner(C\texttt{<}\overline{\ell'.d}\texttt{>}) \in owner(\Sigma[\theta]) \cup domains(\Sigma[\theta])$	By subderivation of T-New

Subcase $owner(C\texttt{<}\overline{\ell'.d}\texttt{>}) = owner(\Sigma[\theta])$

$\emptyset; \Sigma; \theta \models owner(\Sigma[\theta]) \rightarrow owner(\Sigma[\theta])$ By T-SelfLink

$\emptyset; \Sigma; \theta \models \theta \rightarrow owner(\Sigma[\theta])$ By T-LinkRef

$\emptyset; \Sigma; \theta \models \theta \rightarrow owner(C\texttt{<}\overline{\ell'.d}\texttt{>})$ By subcase hypothesis

$\emptyset; \Sigma; \theta \models assumptions(C\texttt{<}\overline{\ell'.d}\texttt{>})$ By subderivation of T-New

$\emptyset; \Sigma; \theta \vdash C\texttt{<}\overline{\ell'.d}\texttt{>}$ By inversion of T-Type

$T' = T = C\texttt{<}\overline{\ell'.d}\texttt{>}$ This proves (3).

Subcase $owner(C\texttt{<}\overline{\ell'.d}\texttt{>}) \in domains(\Sigma[\theta])$.

Take $\theta.d \in domains(\Sigma[\theta])$, that is $owner(C\texttt{<}\overline{\ell'.d}\texttt{>}) = \theta.d$

$\emptyset; \Sigma; \theta \models \theta \rightarrow \theta.d$ By T-ChildRef

$\emptyset; \Sigma; \theta \models \theta \rightarrow owner(C\texttt{<}\overline{\ell'.d}\texttt{>})$ By subcase hypothesis

$\emptyset; \Sigma; \theta \models assumptions(C\texttt{<}\overline{\ell'.d}\texttt{>})$ By subderivation of T-New

$\emptyset; \Sigma; \theta \vdash C\texttt{<}\overline{\ell'.d}\texttt{>}$ By inversion of T-Type

$T' = T = C\texttt{<}\overline{\ell'.d}\texttt{>}$ This proves (3).

$\Sigma_\Delta \vdash S$ By i. h.

$domain(S) = domain(\Sigma)$ By T-Store

$S[\ell_1] = C{<}\overline{\ell'_1.d}{>}(\overline{v}) \iff \Sigma[\ell_1] = C{<}\overline{\ell'_1.d}{>}$ By T-Store

$fields(\Sigma[\ell_1]) = \overline{T}\ \overline{f} \Longrightarrow (S[\ell_1, i] = \ell''_1) \wedge (\Sigma'[\ell''_1] <: T_i)$ By T-Store

$tparams(T_i) \rightarrow tparams(\Sigma[\ell''_1]) \in \Delta$ By T-Store

$\Sigma_\Delta\ OK$ By T-Store

$(S[\ell_1, i] = \ell''_1) \Longrightarrow (owner(\Sigma[\ell_1]) \rightarrow owner(\Sigma[\ell''_1])) \in \Delta$ By T-Store

$\forall \ell_2 \in domain(\Sigma)\ links(\Sigma[\ell_2]) \subseteq \Delta$ By $\Sigma_\Delta\ OK$

$S' = S[\ell \mapsto C{<}\overline{\ell'.d}{>}(\overline{v})]$ Sub-derivation of *R-New*

$domain(S') = domain(S) \cup \{\ell\}$

$domain(\Sigma') = domain(\Sigma) \cup \{\ell\}$

$domain(S') = domain(\Sigma')$ by T-Store, $domain(S) = domain(\Sigma)$

$S'[\ell] = C{<}\overline{\ell'.d}{>}(\overline{v}) \iff \Sigma'[\ell] = C{<}\overline{\ell'.d}{>}$ by T-Store and def. of S' and Σ'

To show (4):

$$fields(\Sigma[\ell]) = \overline{T}\ \overline{f} \Longrightarrow (S[\ell, i] = \ell'') \wedge (\Sigma'[v_i] <: T_i) \tag{5}$$

$$(tparams(T_i) \rightarrow tparams(\Sigma[\ell''])) \in \Delta \tag{6}$$

$$\Sigma'_{\Delta'}\ OK \tag{7}$$

$$(S'[\ell, i] = \ell'') \Longrightarrow (owner(\Sigma[\ell]) \rightarrow owner(\Sigma'[\ell''])) \in \Delta' \tag{8}$$

$fields(\Sigma'[\ell]) = \overline{T}\ \overline{f}$ By $\Sigma'[\ell] = C\textless\overline{\ell'.d}\textgreater$

$S'[\ell, i] = v_i\ \Sigma'[v_i] = \Sigma[v_i] \implies (S'[\ell, i] = v_i) \wedge (\emptyset; \Sigma'; \theta \vdash \Sigma'[v_i] <: T_i)$ By Store Lemma

Take $\ell'' = v_i,$ this proves (5).

$fields(\Sigma'[\ell]) = \overline{T}\ \overline{f}$ By $\Sigma'[\ell] = C\textless\overline{\ell'.d}\textgreater$

$T_i = [\ell/\texttt{this}][\overline{\ell.d}/\overline{\alpha}][solveAny(owner(\Sigma'[\ell]), \Sigma'[\ell])/\texttt{any}]C_i\textless\overline{p'}\textgreater$ By subderivation of Aux-DynFields

$isPrecise(T_i)$ By substitutions above

$\Sigma'[v_i] <: T_i$ By subderivation of T-New

$\emptyset; \Sigma'; \theta \models tparams(T_i) \to tparams(\Sigma'[v_i])$ By subderivation of T-New

$tparams(T_i) \to tparams(\Sigma[v_i]) \in \Delta$ By Subderivation of T-DynamicLink

Take $\ell'' = v_i,$ this proves (6).

$\emptyset; \Sigma'; \theta \models assumptions(\Sigma'[\ell])$ By subderivation of T-New

$assumptions(\Sigma'[\ell]) \subseteq \Delta$ By subderivation of T-DynamicLink

Take $\Delta' = \Delta \cup (links(C\textless\overline{\ell'.d}\textgreater)$

$links(\Sigma'[\ell]) \in \Delta'$ Since $\Sigma'[\ell] = C\textless\overline{\ell'.d}\textgreater$

This proves (7).

$S'[\ell, i] = v_i$ By $S' = S[\ell \mapsto C\text{<}\overline{p}\text{>}(\overline{v})]$

$\Sigma[v_i] = \Sigma'[v_i]$ By $\Sigma' = \Sigma[\ell \mapsto C\text{<}\overline{\ell'.d}\text{>}]$

$owner(\Sigma'[v_i]) <:_d owner(T_i)$ by $\Sigma'[v_i] <: T_i$

$owner(T_i) \in owner(\Sigma'[\ell]) \cup domains(\Sigma'[\ell]) \cup$

$\cup \{\ell'_j.d_j \in \overline{\ell'.d}\} \cup \{v_k.d_k \mid v_k \in \overline{v}, v_k \neq v_i, public(d_k)\} \cup$

$\cup \{solveAny(owner(\Sigma[\ell]), \Sigma[\ell])\}$ By ClsOK

Subcase: $owner(T_i) = solveAny(owner(\Sigma[\ell]), \Sigma[\ell])$

$\Sigma[v_i] = C_v\text{<}\overline{\ell_v.d_v}\text{>}$ By T-Store

$\Sigma[v_i] \in \emptyset; \Sigma; \theta \vdash preciseType(\Sigma[v_i])$ Since $precise(\Sigma[v_i])$

$\Sigma[v_i] <: T_i$ By subderivation of T-New

$owner(\Sigma[v_i) <:_d owner(T_i)$ By subderivation of Subtype-Dom

$\emptyset; \Sigma; \theta \models tparams(T_i) \rightarrow tparams(\Sigma[v_i])$ By subderivation of T-New

$\emptyset; \Sigma; \theta \models owner(T_i) \rightarrow owner(\Sigma[v_i])$ By subderivation of Aux-TParams

$\emptyset; \Sigma; \theta \models solveAny(owner(\Sigma[\ell]), \Sigma[\ell]) \rightarrow owner(\Sigma[v_i])$ Since $owner(T_i) = solveAny(owner(\Sigma[\ell]), \Sigma[\ell])$

$\emptyset; \Sigma; \theta \models owner(\Sigma[\ell]) \rightarrow owner(\Sigma[v_i])$ Since $owner(\Sigma[\ell]) \in solveAny(owner(\Sigma[\ell]), \Sigma[\ell])$

$owner(\Sigma[\ell]) \rightarrow owner(\Sigma[v_i]) \in \Delta$ By subderivation of T-DynamicLink

Take $\ell'' = v_i$. This proves (8)

Subcase $isPrecise(owner(T_i))$

$\Sigma[v_i] = C_v{<}\overline{\ell_v.d_v}{>}$ By T-Store

$\Sigma[v_i] \in \emptyset; \Sigma; \theta \vdash preciseType(\Sigma[v_i])$ Since $precise(\Sigma[v_i])$

$\Sigma[v_i] <: T_i$ By subderivation of T-New

$owner(\Sigma[v_i) <:_d owner(T_i)$ By subderivation of Subtype-Dom

$owner(\Sigma[v_i]) = owner(T_i)$ Since $isPrecise(owner(T_i))$

$owner(\Sigma[v_i]) \in owner(\Sigma'[\ell]) \cup domains(\Sigma'[\ell]) \cup$

$\cup \{\ell'_j.d_j \in \overline{\ell'.d}\} \cup \{v_k.d_k \mid v_k \in \overline{v}, v_k \neq v_i, public(d_k)\}$ By $owner(\Sigma[v_i]) = owner(T_i)$ and ClsOK

Subcase: $owner(\Sigma'[v_i]) \in owner(\Sigma'[\ell])$

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \to owner(\Sigma'[\ell])$ By T-SelfLink

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \to owner(\Sigma'[v_i])$ By subcase hypothesis

$owner(\Sigma'[\ell]) \to owner(\Sigma'[v_i]) \in \Delta'$ By subderivation of T-DynamicLink

Take $\ell'' = v_i$. This proves (8)

Subcase: $owner(\Sigma'[v_i]) \in domains(\Sigma'[\ell])$

Take $\ell.d \in domains(\Sigma'[\ell])$

$\emptyset; \Sigma'; \theta \models \ell \to \ell.d$ By T-ChildLink

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \to \ell.d$ By subderivation of T-LinkRef

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \to owner(\Sigma'[v_i])$ By subcase hypothesis

$owner(\Sigma'[\ell]) \to owner(\Sigma'[v_i]) \in \Delta'$ By subderivation of T-DynamicLink

Take $\ell'' = v_i$. This proves (8)

Subcase: $owner(\Sigma'[v_i]) \in \{\ell'_j.d_j \in \overline{\ell'.d}\}$

$owner(\Sigma'[\ell]) \rightarrow \ell'_j.d_j \in assumptions(\Sigma'[\ell])$ By Assumptions Owner Lemma

$\emptyset; \Sigma'; \theta \models assumptions(\Sigma'[\ell])$ By subderivation of T-New

$assumptions(\Sigma'[\ell]) \in \Delta'$ By subderivation of T-DynamicLink

$owner(\Sigma'[\ell]) \rightarrow \ell'_j.d_j \in \Delta'$ By transitivity of set inclusion

$owner(\Sigma'[\ell]) \rightarrow owner(\Sigma[v_i]) \in \Delta'$ By subcase hypothesis

Take $\ell'' = v_i$. This proves (8)

Subcase: $owner(\Sigma'[v_i]) \in \{v_k.d_k \mid v_k \in \overline{v}, v_k \neq v_i, public(d_k)\}$

Assume by T-Store: $\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \rightarrow owner(\Sigma[v_k]), \forall k < i$

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \rightarrow owner(\Sigma[v_k]) \quad public(d_k)$ By induction hypothesis

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \rightarrow v_k.d_k$ By inversion of T-PublicLink

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \rightarrow owner(\Sigma'[v_i])$ By subcase hypothesis

$owner(\Sigma'[\ell]) \rightarrow owner(\Sigma'[v_i]) \in \Delta'$ By subderivation of T-DynamicLink

Take $\ell'' = v_i$. This proves (8)

Case *R-Read*: Then $e = \ell.f_i \qquad e' = v_i$

$$\frac{S[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v}) \qquad fields(C\texttt{<}\overline{\ell'.d}\texttt{>}) = \overline{T}\ \overline{f}}{S;\theta \vdash \ell.f_i \mapsto v_i, S}\ \textit{R-Read}$$

To show:

$$\emptyset;\Sigma';\theta \models tparams(T) \rightarrow tparams(T') \tag{9}$$

$$\emptyset;\Sigma';\theta \vdash e' : T' \tag{10}$$

$$\emptyset;\Sigma';\theta \vdash T' \tag{11}$$

$$\Sigma'_{\Delta'} \vdash S' \tag{12}$$

By Rule T-Read, we have:

$$\frac{\begin{array}{c}\Gamma;\Sigma;n_{this} \vdash e_0 : T_0 \qquad fields(T_0) = \overline{T}\ \overline{f} \\ \Gamma;\Sigma;n_{this} \vdash T_0\end{array}}{\Gamma;\Sigma;n_{this} \vdash e_0.f_i : [e_0/\texttt{that}]T_i}\,[\text{T-R}\textsc{ead}]$$

By assumption, the type of field read is the type of the i^{th} field. The type of the rewritten expression is $\Sigma[v_i]$. Since the store remains unchanged, $\Sigma'_{\Delta'} \vdash S'$ is justified by induction hypothesis. By T-Store Lemma, the type of the field value is a subtype of the declared field type, while the permissions rule for the subtyping relation are satisfied by T-Store.

To prove that $\Sigma[v_i]$ is well-formed in the current context θ, we consider subcase when the owner of the declared field type is `any`. That is, owner of the field type is the result of *solveAny* in the context of ℓ. By hypothesis, the type of the field-read is well-formed, which implies that θ has permissions to access the result of *solveAny*. By *T-Store* owner of $\Sigma[\ell]$ has permission to access owner of $\Sigma[v_i]$, and implies that owner of $\Sigma[v_i]$ is part of the result of *solveAny*. Therefore, θ has permissions to access owner of $\Sigma[v_i]$.

When the owner of the declared field type is a precise domain, the owner of the declared field type and the owner of $\Sigma[v_i]$ are equal, and by hypothesis θ has permissions to access the owner of the declared field type.

The second condition requires that the assumptions of $\Sigma[v_i]$ are satisfied. The condition is justified by Store Lemma and concludes the case.

$S[\ell] = C\text{<}\overline{p}\text{>}(\overline{v})$	Sub-derivation of R-Read
$S' = S,$ Take $\Sigma' = \Sigma,\ \Delta' = \Delta$	
$\Sigma_\Delta \vdash S$	By i. h.
This proves (12).	

$\emptyset; \Sigma; \theta \vdash \ell : T_0$	Take $\ell = e_0$
$S[\ell] = C\text{<}\overline{p}\text{>}(\overline{v}) \iff \Sigma[\ell] = C\text{<}\overline{p}\text{>}$	By T-Store
$T_0 = C\text{<}\overline{p}\text{>}$	By T-Store
$fields(T_0) = \overline{T}\ \overline{f}$	By subderivation of T-Read
$\emptyset; \Sigma; \theta \vdash \ell.f_i : [\ell/\texttt{that}]T_i$	By T-Read
$\emptyset; \Sigma; \theta \vdash \overline{v} : \Sigma[\overline{v}],$ where $\Sigma[\overline{v}] <: \overline{T}$	By Store Lemma
$assumptions(\Sigma[\overline{v}]) \subseteq \Delta$	By Store Lemma
$v_i \in \overline{v}$	
$\emptyset; \Sigma; \theta \vdash v_i : \Sigma[v_i]$	By T-Loc
Take $T = [\ell/\texttt{that}]T_i \quad T' = \Sigma[v_i]$	
This proves (10)	
$tparams(T_i) \rightarrow tparams(\Sigma[v_i]) \in \Delta$	By T-Store
$\emptyset; \Sigma; \theta \models tparams(T_i) \rightarrow tparams(\Sigma[v_i])$	By inversion of T-DynamicLink
This proves (9)	

$\emptyset; \Sigma; \theta \vdash \ell.f_i : T$	By induction hypothesis.
$\emptyset; \Sigma; \theta \vdash T$	By induction hypothesis.
$\emptyset; \Sigma; \theta \models \theta \rightarrow \mathit{owner}(T)$	By subderivation of T-Type
$\mathit{assumptions}(\Sigma[v_i]) \subseteq \Delta$	By By Store Lemma
$\emptyset; \Sigma; \theta \models \mathit{assumptions}(\Sigma[v_i])$	By inversion of T-DeclaredLink

Subcase: $\mathit{owner}(T) \neq \mathit{solveAny}(\mathit{owner}(\Sigma[\ell]), \Sigma[\ell])$. That is, $\mathit{owner}(T) = \mathit{owner}(\Sigma[v_i])$

$\emptyset; \Sigma; \theta \models \theta \rightarrow \mathit{owner}(\Sigma[v_i])$	$\mathit{owner}(\Sigma[v_i]) = \mathit{owner}(T)$
$\emptyset; \Sigma; \theta \vdash \Sigma[v_i]$	By inversion of T-Type

This proves (11)

Subcase: $\mathit{owner}(T) = \mathit{solveAny}(\mathit{owner}(\Sigma[\ell]), \Sigma[\ell])$. That is, $\mathit{owner}(\Sigma[v_i]) <:_d \texttt{any}$

$\emptyset; \Sigma; \theta \vdash [\ell/\texttt{that}]T$	By hypothesis
$\emptyset; \Sigma; \theta \models \theta \rightarrow \mathit{solveAny}(\mathit{owner}(\Sigma[\ell]), \Sigma[\ell])$	$\mathit{owner}(T) = \mathit{solveAny}(\mathit{owner}(\Sigma[\ell]), \Sigma[\ell])$
$\emptyset; \Sigma; \theta \models \mathit{owner}(\Sigma[\theta]) \rightarrow \mathit{solveAny}(\mathit{owner}(\Sigma[\ell]), \Sigma[\ell])$	By subderivation of T-LinkRef
$\mathit{owner}(\Sigma[\ell]) \rightarrow \mathit{owner}(\Sigma[v_i]) \in \Delta$	By T-Store
$\mathit{owner}(\Sigma[v_i]) \in \mathit{solveAny}(\mathit{owner}(\Sigma[\ell]), \Sigma[\ell])$	By inversion of Aux-DynSolveAny
$\emptyset; \Sigma; \theta \models \mathit{owner}(\Sigma[\theta]) \rightarrow \mathit{owner}(\Sigma[v_i])$	By above
$\mathit{assumptions}(\Sigma[v_i]) \subseteq \Delta$	By Store Lemma
$\emptyset; \Sigma; \theta \models \mathit{assumptions}(\Sigma[v_i])$	By inversion of T-DeclaredLink
$\emptyset; \Sigma; \theta \vdash \Sigma[v_i]$	By inversion of T-Type

This proves (11)

Case *R-Write*: $e = \ell.f_i = v \quad e' = v$

$$\frac{S[\ell] = C\texttt{<}\overline{p}\texttt{>}(\overline{v}) \qquad S' = S[\ell \mapsto C\texttt{<}\overline{p}\texttt{>}([v/v_i]\overline{v})]}{S; \theta \vdash \ell.f_i = v \mapsto v,\ S'} \ \textit{R-Write}$$

To show:

$$\emptyset; \Sigma'; \theta \models \mathit{tparams}(T) \to \mathit{tparams}(T') \tag{13}$$

$$\emptyset; \Sigma'; \theta \vdash e' : T' \tag{14}$$

$$\emptyset; \Sigma'; \theta \vdash T' \tag{15}$$

$$\Sigma'_{\Delta'} \vdash S' \tag{16}$$

By *T-Write*:

$$\frac{\begin{array}{c}\Gamma; \Sigma; n_{this} \vdash e_0 : T_0 \qquad \mathit{fields}(T_0) = \overline{T}\ \overline{f} \qquad T_i \in \overline{T} \qquad \Gamma; \Sigma; n_{this} \vdash e_R : T_R \\ T'_R \in \Gamma; \Sigma; n_{this} \vdash \mathit{preciseType}(T_R) \qquad T'_R <: [e_0/\texttt{that}]T_i \\ \Gamma; \Sigma; n_{this} \models \mathit{tparams}([e_0/\texttt{that}]T_i) \to \mathit{tparams}(T'_R) \\ \Gamma; \Sigma; n_{this} \vdash T_0 \qquad \Gamma; \Sigma; n_{this} \vdash T_R\end{array}}{\Gamma; \Sigma; n_{this} \vdash e_0.f_i = e_R : T_R} \ [\text{T-WRITE}]$$

The type of the rewritten expression is the type T_R of the right hand side expression e_R. By *T-Write* T_R is well-formed.

The value of the store for the key ℓ changes, while the type context Σ and the set of links Δ remains unchanged. Similar to case R-New, when the owner of the declared field type is `any`, and it has permission to access the owner of $\Sigma[v]$, it implies that owner of ℓ has permission to access owner of $\Sigma[v]$ since owner of ℓ is in *solveAny*.

Next, we consider subcases depending if the owner of the declared type is the owner of ℓ, a locally declared domain of ℓ, a domain parameter of $\Sigma[\ell]$ or a public domain of another field value v_k.

$\emptyset; \Sigma; \theta \vdash \ell.f_i = v : T_R$ By T-Write

$\emptyset; \Sigma; \theta \vdash v : T_R$ By subderivation of T-Write $v = e_R$

Take $T = T' = T_R \quad T' <: T$

This proves (14), and (13)

$\emptyset; \Sigma; \theta \vdash \ell.f_i = v : T_R$ By T-Write

$\emptyset; \Sigma; \theta \vdash v : T_R$ By subderivation of T-Write $v = e_R$

$\emptyset; \Sigma'; \theta \vdash T_R$ By Subderivation of T-Write

$\emptyset; \Sigma'; \theta \vdash \Sigma[v]$ Since $\Sigma[v] = T_R$

This proves (15) Take $T' = T_R$

$S' = S[\ell \mapsto C\texttt{<}\overline{\ell'.d}\texttt{>}([v/v_i]\overline{v})]$ By inversion of R-Write

Take $\Sigma' = \Sigma,\ \Delta = \Delta'$

$\Sigma_\Delta \vdash S$ By i. h.

$domain(S) = domain(\Sigma)$ By T-Store

$S[\ell_1] = C\texttt{<}\overline{\ell'_1.d}\texttt{>}(\overline{v}) \iff \Sigma[\ell_1] = C\texttt{<}\overline{\ell'_1.d}\texttt{>}$ By T-Store

$fields(\Sigma[\ell_1]) = \overline{T}\ \overline{f} \Longrightarrow (S[\ell_1, i] = \ell''_1) \wedge (\Sigma[\ell''_1] <: T_i)$ By T-Store

$solveAny(owner(\Sigma[\ell_1]), \Sigma[\ell_1]) \to owner(\Sigma[\ell''_1] \in \Delta$ By T-Store

$\Sigma_\Delta\ OK$ $\Sigma_\Delta \vdash S$

$(S[\ell_1, i] = \ell''_1) \Longrightarrow (owner(\Sigma[\ell_1]) \to owner(\Sigma[\ell''_1])) \in \Delta$ By T-Store

$\forall \ell_2 \in domain(\Sigma)\ links(\Sigma[\ell_2]) \subseteq \Delta$ By $\Sigma_\Delta\ OK$

$domain(S') = domain(S) = domain(\Sigma) = domain(\Sigma')$

$S'[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>}([v/v_i]\overline{v}) \iff \Sigma[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>}$ Since only $\overline{v}$ changes

$fields(\Sigma'[\ell]) = [\ell/\texttt{that}]\overline{T}\ \overline{f}$ By subderivation of T-Write

$(S'[\ell, i] = v)$

$\Sigma[v] <: [\ell/\texttt{that}]T_i$ By T-Write, since $\Sigma[v] = T_R$

$\emptyset; \Sigma; \theta \models owner([\ell/\texttt{that}]T_i) \to owner(\Sigma[v])$ By subderivation of T-Write

$\emptyset; \Sigma; \theta \models solveAny(owner(\Sigma[\ell]), \Sigma[\ell]) \to owner(\Sigma[v])$ By $owner(T_i) = solveAny(owner(\Sigma[\ell]), \Sigma[\ell])$

$solveAny(owner(\Sigma[\ell])), \Sigma[\ell]) \to owner(\Sigma[v]) \in \Delta$ By subderivation of T-DynamicLink

$\Sigma'_{\Delta'} OK$ Since $\Sigma' = \Sigma,\ \Delta' = \Delta$

$S'[\ell, i] = v$ By $S' = S[\ell \mapsto C\texttt{<}\overline{p}\texttt{>}([v/v_i]\overline{v})]$

$\Sigma[v] = \Sigma'[v]$ By T-Store

$\Sigma'[v] \in \emptyset; \Sigma'; \theta \vdash preciseType(\Sigma'[v])$ By subderivation of T-Write

$\Sigma'[v] <: [\ell/\texttt{that}]T_i$ By subderivation of T-Write

$owner(\Sigma'[v]) <:_d owner([\ell/\texttt{that}]T_i)$ By Subtype-Dom

$owner([\ell/\texttt{that}]T_i) \in owner(\Sigma'[\ell]) \cup domains(\Sigma'[\ell]) \cup$

$\cup \{\ell'_j.d_j \in \overline{\ell'.d}\} \cup \{v_k.d_k \mid v_k \in \overline{v}, v_k \neq v, public(d_k)\} \cup$

$\cup \{solveAny(owner(\Sigma'[\ell]), \Sigma'[\ell])\}$ By and ClsOK

Subcase: $owner([\ell/\texttt{that}]T_i) = solveAny(owner(\Sigma'[\ell]), \Sigma'[\ell])$

$\Sigma'[v] \in \emptyset; \Sigma'; \theta \vdash preciseType(\Sigma'[v])$ By subderivation of T-Write

$\emptyset; \Sigma'; \theta \models owner([\ell/\texttt{that}]T_i) \to owner(\Sigma'[v])$ By subderivation of T-Write

$\emptyset; \Sigma'; \theta \models solveAny(owner(\Sigma'[\ell]), \Sigma'[\ell]) \to owner(\Sigma'[v])$ By subcase hypothesis

$owner(\Sigma'[\ell]) \in solveAny(owner(\Sigma'[\ell]), \Sigma'[\ell])$ Since $owner(\Sigma'[\ell]) \to owner(\Sigma'[\ell]) \in \Delta$

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \to owner(\Sigma'[v])$

$owner(\Sigma'[\ell]) \to owner(\Sigma'[v]) \in \Delta$ By subderivation of T-DynamicLink

The following subcases assume $owner([\ell/\texttt{that}]T_i) \neq solveAny(owner(\Sigma[\ell]), \Sigma[\ell])$. That is, $owner([\ell/\texttt{that}]T_i) = owner(\Sigma'[v])$

Subcase: $owner(\Sigma'[v]) \in owner(\Sigma'[\ell])$

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \to owner(\Sigma'[\ell])$ By T-SelfLink

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \to owner(\Sigma'[v])$ By subcase hypothesis

$owner(\Sigma'[\ell]) \to owner(\Sigma'[v]) \in \Delta'$ By subderivation of T-DynamicLink

Take $\ell'' = v$. This proves (16)

Subcase: $owner(\Sigma'[v]) \in domains(\Sigma'[\ell])$

Take $\ell.d \in domains(\Sigma'[\ell])$

$\emptyset; \Sigma'; \theta \models \ell \to \ell.d$ By T-ChildLink

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \to \ell.d$ By subderivation of T-LinkRef

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \to owner(\Sigma'[v_i])$ By subcase hypothesis

$owner(\Sigma'[\ell]) \to owner(\Sigma'[v]) \in \Delta'$ By subderivation of T-DynamicLink

Take $\ell'' = v$. This proves (16)

Subcase: $owner(\Sigma'[v]) \in \{\ell'_j.d_j \in \overline{\ell'.d}\}$

$owner(\Sigma'[\ell]) \to \ell'_j.d_j \in assumptions(\Sigma'[\ell])$ By Assumptions Owner Lemma

$\emptyset; \Sigma'; \theta \vdash \Sigma[\ell]$ By subderivation of T-Write

$\emptyset; \Sigma'; \theta \models assumptions(\Sigma'[\ell])$ By subderivation of T-Type

$assumptions(\Sigma'[\ell]) \in \Delta'$ By subderivation of T-DeclaredLink

$owner(\Sigma'[\ell]) \to \ell'_j.d_j \in \Delta'$ By transitivity of set inclusion

$owner(\Sigma'[\ell]) \to owner(\Sigma[v]) \in \Delta'$ By subcase hypothesis

Take $\ell'' = v$. This proves (16)

Subcase: $owner(\Sigma'[v]) \in \{v_k.d_k \mid v_k \in \overline{v}, v_k \neq v, public(d_k)\}$

Assume by T-Store: $\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \rightarrow owner(\Sigma[v_k]), \forall k < i$

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \rightarrow owner(\Sigma[v_k]) \quad public(d_k)$ By induction hypothesis

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \rightarrow v_k.d_k$ By inversion of T-PublicLink

$\emptyset; \Sigma'; \theta \models owner(\Sigma'[\ell]) \rightarrow owner(\Sigma'[v_i])$ By subcase hypothesis

$owner(\Sigma'[\ell]) \rightarrow owner(\Sigma'[v]) \in \Delta'$ By subderivation of T-DynamicLink

Take $\ell'' = v$. This proves (16)

Case *R-Invk*: $e = \ell.m(\overline{v}) \quad e' = \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$

$$\frac{S[\ell] = C\texttt{<}\overline{p}\texttt{>}(\overline{v}) \qquad mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x}, e_0)}{S; \theta \vdash \ell.m(\overline{v}) \mapsto \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_0, S} \; \textit{R-Invk}$$

To show:

$$\emptyset; \Sigma'; \theta \vdash tparams(T) \rightarrow tparams(T') \tag{17}$$

$$\emptyset; \Sigma'; \theta \vdash e' : T' \tag{18}$$

$$\emptyset; \Sigma'; \theta \vdash T' \tag{19}$$

$$\Sigma'_{\Delta'} \vdash S' \tag{20}$$

From *T-Invk*:

$$\frac{\begin{array}{c} \Gamma; \Sigma; n_{this} \vdash e_0 : T_0 \qquad \Gamma; \Sigma; n_{this} \vdash \overline{e} : \overline{T_a} \\ mtype(m, T_0) = \overline{T} \rightarrow T_R \qquad mbody(m, T_0) = (\overline{x}, e_R) \\ \overline{T'_a} \in \Gamma; \Sigma; n_{this} \vdash preciseType(\overline{T_a}) \qquad \overline{T'_a} <: [\overline{e}/\overline{x}, e_0/\texttt{that}]\overline{T} \\ \Gamma; \Sigma; n_{this} \models tparams([\overline{e}/\overline{x}, e_0/\texttt{that}]\overline{T}) \rightarrow tparams(\overline{T'_a}) \\ \Gamma; \Sigma; n_{this} \vdash T_0 \qquad \Gamma; \Sigma; n_{this} \vdash \overline{T_a} \end{array}}{\Gamma; \Sigma; n_{this} \vdash e_0.m(\overline{e}) : [\overline{e}/\overline{x}, e_0/\texttt{that}]T_R} \; [\text{T-INVK}]$$

Since the store, Σ and Δ remain unchanged $\Sigma'_{\Delta'} \vdash S'$ is justified by induction hypothesis.

To prove that the type of $\ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_0$ is type T' a subtype of T we use *MethOK*, Method Lemma and Substitution Lemma. The Method Lemmas consider the cases when the method body is defined in a supertype of the receiver, while the type is preserved. By substitution of actual to formal parameters, the return expression might be a subtype of the one declared in the method signature. this allows for example, the type of the return expression e_R to be a precise type, while the return type declared by the method to be imprecise.

Finally, to prove that T' is well-formed we use the Super Type Lemma which states that if the context changes, and owner of the supertype has permissions to access the owner of the subtype, the subtype is still well-formed. The second condition is ensured by the later changes in *T-Invk*.

$S' = S$, Take $\Sigma' = \Sigma$, $\Delta' = \Delta$

$\Sigma_\Delta \vdash S$ By i. h.

This proves (20).

$\emptyset; \Sigma'; \theta \vdash \ell.m(\overline{v}) : T$ By induction hypothesis.

$\{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{\alpha}, \overline{\beta}\texttt{>}\},\ \emptyset,\ \texttt{this} \vdash e_R : T'_R$ By MethOK

$T'_R <: T_R$ By MethOK

$S[\ell] = C\texttt{<}\overline{p}, \overline{p'}\texttt{>}(\overline{v})$ By inversion of R-Invk

$mbody(m, C\texttt{<}\overline{p}, \overline{p'}\texttt{>}) = (\overline{x}, e_R)$ By inversion of R-Invk

$e_0 = \ell$

$\Sigma[\ell] = C\texttt{<}\overline{p}, \overline{p'}\texttt{>} = T_0$ T-Store

$\emptyset; \Sigma'; \theta \vdash e_0 : C\texttt{<}\overline{p}, \overline{p'}\texttt{>}$ By subderivation of T-Invk

$mtype(m, C\texttt{<}\overline{p}, \overline{p'}\texttt{>}) = \overline{T} \to T_R$ By subderivation of T-Invk

$\emptyset; \Sigma'; \theta \vdash \overline{v} : \overline{T_a}$ By subderivation of T-Invk

$\overline{T_a} <: [\overline{v}/\overline{x}, \ell/\texttt{this}]\ \overline{T}$ for some $\overline{T_a}$ and $\overline{T}$

There are some $D\texttt{<}\overline{p}\texttt{>}$ and T'_R s.t. $T'_R <: T_R$

and $C\texttt{<}\overline{p}, \overline{p'}\texttt{>} <: D\texttt{<}\overline{p}\texttt{>}$ s.t. $\{\overline{x} : \overline{T}\}; \emptyset, \texttt{this} : D\texttt{<}\overline{p}\texttt{>} \vdash e_R : T'_R$ By Method Lemma

$\{\overline{x} : \overline{T}\}; \Sigma; \texttt{this} : D\texttt{<}\overline{p}\texttt{>} \vdash e_R : T'_R$ By Weakening

$\Gamma; \Sigma'; \theta \vdash \overline{v} : \overline{T_a}$ and $\overline{T_a} <: [\overline{v}/\overline{x}, \ell/\texttt{this}]\ \overline{T}$ By Weakening

$\Gamma; \Sigma'; \theta \vdash [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R : T_S$ for some $T_S <: [\overline{v}/\overline{x}, \ell/\texttt{this}]T'_R$ By Substitution Lemma

$T_S <: [\overline{v}/\overline{x}, \ell/\texttt{this}]T'_R$ and $[\overline{v}/\overline{x}, \ell/\texttt{this}]T'_R <: [\overline{v}/\overline{x}, \ell/\texttt{this}]T_R$ By above

$T_S <: [\overline{v}/\overline{x}, \ell/\texttt{this}]T_R$ By transitivity of $<:$

$\emptyset; \Sigma; \theta \vdash \Sigma[\ell]$ By subderivation of T-Invk

$\{\overline{x} : \overline{T};\ \texttt{this} : D\texttt{<}\overline{p}\texttt{>}\};\ \emptyset;\ \texttt{this} : D\texttt{<}\overline{p}\texttt{>} \vdash T'_R$ By MethOK

$\Gamma; \Sigma; \ell \vdash T_S$ By Substitution Lemma

$\Gamma; \Sigma; \ell \vdash [\overline{v}/\overline{x}, \ell/\texttt{this}]T'_R$ By Subtype Lemma

$\emptyset; \Sigma; \theta \vdash \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R : T_S$ By inversion of T-Context

Take $T = [\overline{v}/\overline{x}, \ell/\texttt{this}]T_R, T' = T_S$ Preservation

This proves (18).

$\{\overline{x} : \overline{T}\}; \emptyset; \texttt{this} \models \mathit{tparams}(T_R) \rightarrow \mathit{tparams}(T_S)$ By MethOK

$\{\overline{x} : \overline{T}\}; \Sigma; \texttt{this} \models \mathit{tparams}(T_R) \rightarrow \mathit{tparams}(T_S)$ By Weakening

Subcase $\mathit{isprecise}(\mathit{tparams}(T_R))$ and $\mathit{isprecise}(\mathit{tparams}(T_S))$

$\mathit{tparams}(T_R) \rightarrow \mathit{tparams}(T_S) \in \mathit{linkdecls}(\Sigma[\ell])$ By MethOK

$\mathit{linkdecls}(\Sigma[\ell]) \in \Delta$

$\emptyset; \Sigma; \theta \models \mathit{tparams}([\overline{v}/\overline{x}, \ell/\texttt{this}]T_R) \rightarrow \mathit{tparams}([\overline{v}/\overline{x}, \ell/\texttt{this}]T_S)$

Subcase $\mathit{tparams}(T_R)_i = \texttt{any}$ and $\mathit{isprecise}(\mathit{tparams}(T_S))$

$\{\overline{x}:\overline{T}\};\Sigma;\texttt{this} \models tparams(T_R)_i \rightarrow tparams(T_S)_i$ By MethOK

$\{\overline{x}:\overline{T}\};\Sigma;\texttt{this} \models solveAny(owner(\Sigma[\ell]),\Sigma[\ell]) \rightarrow tparams(T_S)_i$ By subderivation of T-LinkAnyDom

$solveAny(owner(\Sigma[\ell]),\Sigma[\ell]) \rightarrow tparams(T_S)_i \in \Delta$ By subderivation of T-DynamicLink

$\emptyset;\Sigma;\theta \models solveAny(owner(\Sigma[\ell]),\Sigma[\ell]) \rightarrow tparams(T_S)_i$ By inversion of T-DynamicLink

Subcase $tparams(T_R)_i \neq \texttt{any}$ and $tparams(T_S)_i = \texttt{any}$ cannot happen since $T_S <: T_R$

Subcase $tparams(T_R)_i = \texttt{any}$ and $tparams(T_S)_i = \texttt{any}$

$\{\overline{x}:\overline{T}\};\Sigma;\texttt{this} \models tparams(T_R)_i \rightarrow tparams(T_S)_i$ By MethOK

$\emptyset;\Sigma;\theta \models solveAny(owner(\Sigma[\ell]),\Sigma[\ell]) \rightarrow solveAny(owner(\Sigma[\ell]),\Sigma[\ell])$ By T-SelfLink

This proves (17)

$\emptyset;\Sigma;\theta \vdash \ell.m(v) : T$ By induction hypothesis.

$\emptyset;\Sigma;\theta \vdash T$ By induction hypothesis.

$S;\theta \vdash \ell.m(\overline{v}) \mapsto \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_0, S$ By induction hypothesis.

$\emptyset;\Sigma;\theta \vdash \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R : T'$ By (18)

$T' <: T$ By (18)

$\emptyset;\Sigma;\theta \models tparams(T) \rightarrow tparams(T')$ By (17)

$\emptyset;\Sigma;\theta \vdash T'$ By Super Type Lemma

Case *R-Context*: $e = \ell \triangleright v$, $e' = v$

$$\frac{}{S;\theta \vdash \ell \triangleright v \mapsto v, S}\ \textit{R-Context}$$

To Show:

$$\emptyset; \Sigma'; \theta \models tparams(T) \rightarrow tparams(T') \tag{21}$$

$$\emptyset; \Sigma'; \theta \vdash e' : T' \tag{22}$$

$$\emptyset; \Sigma'; \theta \vdash T' \tag{23}$$

$$\Sigma'_{\Delta'} \vdash S' \tag{24}$$

From *T-Context*:

$$\frac{\Gamma; \Sigma; n_{this} \vdash \Sigma[\ell] \qquad \Gamma; \Sigma; \ell \vdash e : T \qquad \Gamma; \Sigma; \ell \vdash T}{\Gamma; \Sigma; n_{this} \vdash \ell \triangleright e : T}[\text{T-C}\textsc{ontext}]$$

$S = S'$, Take $\Sigma = \Sigma', \Delta = \Delta'$

$\Sigma_{\Delta} \vdash S$ By induction hypothesis

This proves (24).

$\emptyset; \Sigma'; \theta \vdash e : C{<}\overline{\ell'.d}{>}$ By hypothesis

$\emptyset; \Sigma'; \ell \vdash v : C{<}\overline{\ell'.d}{>}$ By subderivation of T-Context

$\Sigma[v] = C{<}\overline{\ell'.d}{>}$ By subderivation of T-Loc

$\emptyset; \Sigma'; \theta \vdash C{<}\overline{\ell'.d}{>}$ By inversion of T-Loc

This proves (21), and (22). Take $T' = T = C{<}\overline{\ell'.d}{>}$

$\emptyset; \Sigma'; \theta \vdash e : T$ by induction hypothesis

$\emptyset; \Sigma; \theta \vdash T$ by induction hypothesis

$\emptyset; \Sigma; \theta \vdash T'$ Since $T' = T$

This proves (23)

Case *RC-XX*: By the induction hypothesis, types are maintained by the underlying reduction rules since each rule preserves typing whenever its subexpressions are well-typed at the same type or a subtype. The only interesting case is *RC-Context* where the context changes from θ to ℓ. By *T-Context*, the typing is maintained although the context changes. To show that the subtype is well-formed we consider the cases when the owner of the super type is `any` or a precise type. When the owner is precise, it is equal to the owner of the subtype. Otherwise, we use the subderivation of T-Context to show that owner the subtype is in the set of domains that `any` resolves to.

To show that the subtyping relation is consistent with domain links, we use the induction hypothesis to show that links between $tparams(T)$ and $tparams(T')$ are consistent in the context of ℓ. If the domain parameters are precise, since the links are part of Δ, the permissions are also consistent in the context of θ. If a domain parameter is `any`, we use *T-LinkAnyDom* to change the context from ℓ to θ, since the type of ℓ is $\Sigma[\ell]$ no matter what the context is.

Case *RC-Context*: $e = \ell \triangleright e_0, e' = \ell \triangleright e_0'$

$$\frac{S; \ell \vdash e_0 \mapsto e_0', S'}{S; \theta \vdash \ell \triangleright e_0 \mapsto \ell \triangleright e_0', S'} \; \textit{RC-Context}$$

To Show:

$$\emptyset; \Sigma'; \theta \models tparams(T) \rightarrow tparams(T') \tag{25}$$

$$\emptyset; \Sigma'; \theta \vdash e' : T' \tag{26}$$

$$\emptyset; \Sigma'; \theta \vdash T' \tag{27}$$

$$\Sigma'_{\Delta'} \vdash S' \tag{28}$$

From *T-Context*:

$$\frac{\Gamma; \Sigma; n_{this} \vdash \Sigma[\ell] \qquad \Gamma; \Sigma; \ell \vdash e_0 : T \qquad \Gamma; \Sigma; \ell \vdash T}{\Gamma; \Sigma; n_{this} \vdash \ell \triangleright e_0 : T}[\text{T-Context}]$$

$\emptyset; \Sigma; \theta \vdash \ell \triangleright e_0 : T$	By hypothesis
$\emptyset; \Sigma; \theta \vdash \Sigma[\ell]$	By subderivation of T-Context
$\emptyset; \Sigma; \ell \vdash e_0 : T$	By subderivation of T-Context
$\emptyset; \Sigma'; \ell \vdash T$	By subderivation of T-Context
$S, \ell \vdash e_0 \mapsto e_0', S'$	By subderivation of RC-Context
there exists $\Sigma' \supseteq \Sigma,\ \Delta' \supseteq \Delta,$ and $T' <: T$ such that:	By induction hypothesis
$\emptyset; \Sigma'; \ell \models tparams(T) \to tparams(T'),$	
$\emptyset; \Sigma'; \ell \vdash e_0' : T',$	
$\emptyset; \Sigma'; \ell \vdash T',$	
and $\Sigma'_{\Delta'} \vdash S'.$	This proves (28).
$\Sigma[\ell] = \Sigma'[\ell] = C{<}\overline{\ell'.d}{>}$	By T-Store
$\emptyset; \Sigma'; \theta \vdash \Sigma'[\ell]$	Since $\emptyset; \Sigma; \theta \vdash \Sigma[\ell]$
$\emptyset; \Sigma'; \theta \vdash \ell \triangleright e_0' : T'$	By inversion of T-Context

This proves (26).

Subcase $owner(T) \neq \ell.\texttt{any}$

$\emptyset; \Sigma; \theta \vdash T$	By hypothesis
$\emptyset; \Sigma; \theta \models \theta \rightarrow owner(T)$	By subderivation of T-Type
$owner(T) = owner(T')$	By $T' <: T$
$\emptyset; \Sigma; \theta \models \theta \rightarrow owner(T')$	
$\emptyset; \Sigma; \theta \models assumptions(T)$	By subderivation of T-Type
$assumptions(T) \in \Delta$	By subderivation of T-DynamicLink
$\emptyset; \Sigma'; \theta \vdash \ell \triangleright e_0' : T'$	
$\emptyset; \Sigma'; \ell \vdash T'$	By inversion of T-Context
$assumptions(T') \in \Delta'$	By subderivation of T-DynamicLink
$\Delta \subseteq \Delta'$	
$assumptions(T) \subseteq assumptions(T')$	
$\emptyset; \Sigma'; \theta \models assumptions(T')$	By inversion of T-DynamicLink
$\emptyset; \Sigma'; \theta \vdash T'$	By inversion of T-Type

This proves (26).

$\emptyset; \Sigma'; \ell \models tparams(T) \rightarrow tparams(T')$	By induction hypothesis
$tparams(T) \rightarrow tparams(T') \in \Delta'$	By subderivation of T-DynamicLink
$\emptyset; \Sigma'; \theta \models tparams(T) \rightarrow tparams(T')$	By inversion of T-DynamicLink

This proves (25).

Subcase $owner(T) = \ell.\mathtt{any}$

$\emptyset; \Sigma; \theta \vdash T$ By hypothesis

$\emptyset; \Sigma; \ell \vdash T$ By inversion of T-Context

$\emptyset; \Sigma'; \ell \vdash T'$ By inversion of T-Context

$owner(T') \in solveAny(owner(\Sigma[\ell]), \Sigma[\ell])$

$\emptyset; \Sigma; \theta \models \theta \to owner(T)$ By subderivation of T-Type

$\emptyset; \Sigma; \theta \models \theta \to \ell.\mathtt{any}$ By subderivation of T-Type

$owner(T') <:_d owner(T)$ By $T' <: T$

$p' \in solveAny(owner(\Sigma[\ell]), \Sigma[\ell])$

$\emptyset; \Sigma; \theta \models \theta \to p'$ By subderivation of T-LinkDomAny

$\emptyset; \Sigma; \theta \models \theta \to owner(T')$ $owner(T') \in solveAny(owner(\Sigma[\ell]), \Sigma[\ell])$

$\emptyset; \Sigma; \theta \models assumptions(T)$ By subderivation of T-Type

$assumptions(T) \in \Delta$ By subderivation of T-DynamicLink

$\emptyset; \Sigma'; \theta \vdash \ell \triangleright e_0' : T'$

$\emptyset; \Sigma'; \ell \vdash T'$, By inversion of T-Context

$assumptions(T') \in \Delta'$ By subderivation of T-DynamicLink

$\Delta \subseteq \Delta'$

$assumptions(T) \subseteq assumptions(T')$

$\emptyset; \Sigma'; \theta \models assumptions(T')$ By inversion of T-DynamicLink

$\emptyset; \Sigma'; \theta \vdash T'$ By inversion of T-Type

This proves (26).

$\emptyset; \Sigma'; \ell \models tparams(T) \to tparams(T')$ By induction hypothesis

$\emptyset; \Sigma'; \ell \models \ell.\mathtt{any} \to tparams(T')$ By $tparams(T) = \ell.\mathtt{any}$

$\emptyset; \Sigma'; \ell \models solveAny(owner(\Sigma[\ell]), \Sigma[\ell]) \to tparams(T')$ By subderivation of T-LinkAnyDom

$\emptyset; \Sigma'; \theta \models \ell.\mathtt{any} \to tparams(T')$ By inversion of T-LinkAnyDom

This proves (25).

■

Theorem 9 (Progress)

$$
\begin{aligned}
&If\\
&\quad \emptyset; \Sigma; \theta \vdash e : T,\\
&\quad \emptyset; \Sigma; \theta \vdash T,\\
&\quad \textit{and } \Sigma_{\Delta} \vdash S\\
&\quad \textit{— i.e., e is closed and well-typed,}\\
&\textit{then}\\
&\quad \textit{either e is a value, or}\\
&\quad \textit{else } S; \theta \vdash e \mapsto e', S'.
\end{aligned}
$$

Proof: By induction over the derivation of $\emptyset, \Sigma, n_{this} \vdash e : T$. ■

Together, Type Preservation and Progress imply that the type system for FDJ is sound. We also state a link soundness property for FDJ. First, we define link soundness for the heap: if one object refers to another, then it has permission to do so.

Theorem 10 (Heap Link Soundness)

If $\Sigma_\Delta \vdash S$ *and* $S[\ell, i] = \ell''$ *then* $owner(\Sigma[\ell]) \rightarrow owner(\Sigma[\ell'']) \in \Delta$.

Proof: This property is enforced by the store typing rule *T-Store*. ■

In practice, it is important that link soundness hold not only for field references in the system, but also for expressions in methods. The intuition behind expression link soundness is that if a method with receiver object n_{this} is currently executing, it should only be able to compute with objects that n_{this} has permission to access.

Theorem 11 (Expression Link Soundness)

If $\emptyset, \Sigma, n_{this} \vdash e : T$, *and* $\emptyset, \Sigma, n_{this} \vdash T$ *and* $T \neq \texttt{ERROR}$ *then* $\emptyset, \Sigma, n_{this} \models n_{this} \rightarrow owner(T)$.

Proof: This property is enforced by $\emptyset, \Sigma, n_{this} \vdash T$. ■

As a result of link soundness, developers using ownership domains can be confident that the linking specifications are an accurate representation of run time aliasing in the system.

Theorem 12 (Subtyping Link Soundness)

If $\emptyset, \Sigma, n_{this} \vdash e : T'$, $\emptyset, \Sigma, n_{this} \vdash T'$, *and* $T' <: T$ *then* $\emptyset, \Sigma, n_{this} \models tparams(T) \rightarrow tparams(T')$.

Proof: This property is enforced by each typing rule of form $\emptyset, \Sigma, n_{this} \vdash e : T$, where the subtyping relation is guaranteed by domain links. ■

6 Related Work

Several ownership types were proposed to control aliasing and access between objects [18].

6.1 Ownership type systems

Parametric ownership types proposed by Clarke et al. [8] assume an *owners-as-dominators* property, where the internal state of an object is available only through the owner object. The owners-as-dominators property can express strict encapsulation, but cannot express other programming idioms such as iterators. The Universes type system [9] supports the *owners-as-modifiers* property, in which the internal representation of an object is accessible as `read-only`. A separate analysis is required to determine if a method is pure, i.e., that it does not change the internal state of the object.

Similar to Universes, Effective Ownership by Lu and Potter [16] enforces an owners-as-modifiers property. In Effective Ownership, effects are used to enforce encapsulation. The same object hierarchy is used as in ownership types, however, encapsulation is enforced by restricting write effects rather than references. The Effective Ownership type system is more flexible than Universes, but requires more syntactic overhead, and requires knowledge about effects, in addition to ownership.

In Effective Ownership, marking an object reference with the existential owner `any` severely restricts what can be done with that reference, and makes it somewhat similar to a `readonly` reference in Universes (later renamed to `any`). Similarly, Universes requires knowledge about `pure` methods, i.e., ones devoid of side-effects, to use `readonly` references. As [19] pointed out, [16] forbids legitimate cases where objects in public domains directly change the state of representation objects without going through the owner object, as in the listener example. The follow-on system by Lu and Potter, Ownership variance [15], requires specifying both an accessibility modifier and ownership parameters, which would increase further the annotation overhead.

The existential owners proposed by Wrigstad and Clarke [20] are only introduced upon type casting local variables within a method and are not allowed on fields, i.e., they have no statically known relations to other owners. In this paper, we largely ignored issues related to casts.

6.2 Existential ownership

Existential owners have appeared in several ownership type systems [7, 13, 16, 20]. Some of the early proposals used a foundational object calculus [7] or System F [13], which makes them harder to relate to more recent systems formalized in the Featherweight Java style [12].

Existential ownership increased the flexibility of type systems that follow the owners-as-dominators property [7]. Furthermore, Jo∃ extends parametric ownership types with existential quantification of contexts and further increases the flexibility. Cameron and Dietl showed that Universes types are equivalent with a variant of JoE, while JoE is more expressive than Universes [**?**].

Existential ownership was inspired by existential quantifiers in generic types. In Java, generic subtyping is invariant. For example, although Student is a subclass of Person, `List<Students>` is not a subtype of `List<Person>`. Researchers developing existential ownership experienced with different types of variance. For example, types are covariant if a subtype relation between parameters gives a subtype relation between parameterized types. For increased flexibility covariance is preferred as long as the type system preserve soundness. One example of a covariant ownership type system is Effective Ownership [16]. For increase flexibility, we also designed our type system to be covariant, while preserving link soundness.

Notation-wise, the existential ownership adopted in parametric ownership types use an `unknown` context, Universes and Effective Ownership uses `any` and `unknown`, and `?` in multiple ownership type systems. Note that our notation of `any` is similar to `unknown` in parametric ownership types and Universes, and to `any` in Effective Ownership.

Several variants of Ownership Domains were proposed in the literature. For example, Krishnaswami and Aldrich [13] formalized a bounded, existential, ownership domains for System F, called F_{own}. They also introduced domain subtyping in F_{own}. A domain d_1 is a subtype of d_2 if d_1 has more access and create permissions than d_2. F_{own} eliminates domain hierarchy, and permits the creation of new domains at will. For our purposes, we need hierarchy for extracting a hierarchical OOG.

6.3 Simple Loose Ownership Domains (SLOD)

Schäfer and Poetzsch-Heffter [19] proposed a variant of Ownership Domains, called Simple Loose Ownership Domains (SLOD). SLOD simplifies Ownership Domains by hard-coding for each object one private domain and one public domain. It also eliminates domain links, by hard-coding the associated link permission rules into the type well-formedness rules. Domain links are a key feature of Ownership Domains, and eliminating them is a significant departure from the design philosophy behind the system.

SLOD distinguishes between imprecise, i.e., *loose*, and precise domains (and the latter are domain subtypes of the former). SLOD enforces a *boundary-as-dominator* property, whereby objects in private domains objects can only be accessed by the owner object, or objects owned by the owner object. That is, in order to access objects in private domains from the outside, the access must go through the owner object, or through objects in public domains.

More importantly, SLOD prohibits assigning to a field whose type contains an imprecise domain (a type is assignable if all its domain annotations are assignable). In contrast, FOD allows assigning to a field whose type contains `any` , if the domain links in scope allow it.

Finally, some features of our formalization were inspired by the SLOD formalization: we also used well-formed environment and types (Fig. 10) and domain subtyping (Fig. 11).

6.4 Related Techniques

The ownership type systems we previously described assume a fixed, single owner per object that does not change at runtime. For more flexibility, some type systems support multiple ownership [6, 14] or ownership transfer [17]. The increased flexibility means however losing the advantage of having a single hierarchical decomposition, which is important in our application of Ownership Domains for extracting hierarchical OOGs. For example, the Multiple Ownership paper [6] uses diagrams that are related to set diagrams, but that are not very similar to our architectural diagrams. In previous work, we have used these diagrams to analyze architectural conformance [3] as well as security [4].

In Flexible Ownership Domains, we take advantage of the flexibility provided by existential ownership, while preserving a hierarchical organization of objects.

Acknowledgements

References

[1] JHotDraw. `www.jhotdraw.org`, 1996. Version 5.3.

[2] M. Abi-Antoun. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure*. PhD thesis, CMU, 2010.

[3] M. Abi-Antoun and J. Aldrich. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *OOPSLA*, 2009.

[4] M. Abi-Antoun and J. M. Barnes. Analyzing Security Architectures. In *ASE*, 2010.

[5] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.

[6] N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple Ownership. In *OOPSLA*, 2007.

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

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.

[9] W. Dietl and P. Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8), 2005.

[10] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[11] T. Hill, J. Noble, and J. Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *Journal of Visual Languages and Computing*, 13(3), 2002.

[12] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3), 2001.

[13] N. Krishnaswami and J. Aldrich. Permission-Based Ownership: Encapsulating State in Higher-Order Typed Languages. In *PLDI*, 2005.

[14] P. Li, N. Cameron, and J. Noble. More Ownership for Multiple Owners. In *FOOL*, 2010.

[15] Y. Lu and J. Potter. On Ownership and Accessibility. In *ECOOP*, pages 99–123, 2006.

[16] Y. Lu and J. Potter. Protecting Representation with Effect Encapsulation. In *POPL*, 2006.

```
class Listener<OWNER>{}
class View<OWNER,STATE> extends Listener<OWNER>{
  public domain LISTENERS;
  Listener<LISTENERS> listeners(){
    return new ViewListener<LISTENERS,STATE>(state);
  }
}
class View<OWNER,STATE> extends Listener<OWNER>{}

class Model<OWNER,PLIST>{
    void addListener(Listener<PLIST> l){}
}
```

```
class Main{
    public domain VIEW1,VIEW2;
    Model<OWNER,any> model;
    View<VIEW1> view1;
    View<VIEW2> view2;
    public void run(){
      model.addListener(view1);
      model.addListener(view2);
      model.addListener(view1.listener());
      model.addListener(view2.listener());
    }
}
```

Figure 17: . `any` is resolved to `VIEW1` and `VIEW2`, but can it also represent `view1.LISTENERS` and `view2.LISTENERS`?

[17] P. Müller and A. Rudich. Ownership Transfer in Universe Types. In *OOPSLA*, 2007.

[18] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *ECOOP*, 1998.

[19] J. Schäfer and A. Poetzsch-Heffter. A Parameterized Type System for Simple Loose Ownership Domains. *Journal of Object Technology*, 6(5), 2007.

[20] T. Wrigstad and D. Clarke. Existential Owners for Ownership Types. *Journal of Object Technology*, 6(4), 2007.

Appendix

The original FDJ type system assumes that all the domains and types are known, but it lacks expressiveness of some common programming idioms. We consider FDJ type system as a baseline and we extend the rules by *highlighting* the common parts. The extended rules distinguish between precise and imprecise domains and types. A domain is precise if it is different than `any`. A type $C{<}\overline{p}{>}$ is precise, if all p_i are precise domains. The existential domain `any` can be interpreted as: "Any precise domain p in the scope that `OWNER` has permission to access according to the permission links". Since `any` is solved based on the domain links, `any` has a different meaning for different enclosing types. That is why, we changed the original T-SelfLink rule, and require p to be a precise type.

Consider for example `any` that can be solved to `VIEW1` and `VIEW2`, and the method `addListener`, with the argument of type `Listener`. Since `View` extends `Listener`, the actual argument the method `addListener` can be `view1`, `view2`. In this case, p is in turn `VIEW1` and `VIEW2`, and $p \rightarrow$ `any` holds because p is part of *solveAny*. However, `addListener` can also take as an argument `view1.listener()` or `view2.listener()`, in which case p is `view1.LISTENERS` and `view2.LISTENERS`. For the method invocation to typecheck it is necessary that the clause $\Gamma, \Sigma, \texttt{this} : \texttt{Main<OWNER>} \models \texttt{this.any} \rightarrow \texttt{view1.LISTENERS}$ to hold. Indeed, the clause holds since `any` resolves to `VIEW1` and `VIEW1` has permissions to access `view1.LISTENERS`, although `view1.LISTENERS` is not part of *solveAny*.

Subtyping invariant. We could have also relaxed the last part of SubType-Dom by requiring p_i to be a subdomain of p'_i. However, ownership types would no longer be invariant. This is important

```
Class A{
    public domain DOM;
    C<this.DOM> c;
}
```

```
class B{
    final A<OWNER> a;
    C<a.DOM> c1;
    void m(){
        this.c1 = a.c;
    }
}
```

Figure 18: . Supporting example for `that`/`this` substitution.

for subtyping relation,substituting $C < p_1, p_2 >$ with $C < p_1, p'_2 >$, where $p_1 \neq p_2$. For example, `C<MODEL,VIEW>` would be a subtype of `C<MODEL,any>` where `any` may be solved to `VIEW2` rather than `VIEW`, and `C<MODEL,VIEW>` $\not<:$ `C<MODEL,VIEW2>`.

`that`/`this` substitution. We make explicit, several conditions and rules that were implicit in the baseline FDJ, which became relevant while reasoning about `any`. First, we make explicit `that`/`this` substitution, to distinguish `this` in a class definition from `this` occurring in a type parameter, which are not in general the same. `that` refers to the receiver in the context where the expression is being used, and substitutes `this` in type parameters. In the FDJ rules, T-Read, T-Write and T-Invk use the auxiliary judgments to replace `this` with `that`, and then they replace `that` with the receiver e_0. For example, consider the field `c` of class `A` that has the type `C<this.DOM>`. The field is read in the declaration of the class `B` in `this.c1 = a.c`. `this` in `this.DOM` refers to the receiver `a`, and is different from `this` in `this.c1`. While evaluating the type of `a.c`, *fields* perform the [`that`/`this`] substitution: [`that`/`this`]`C<this.DOM>` = `C<that.DOM>`. Next, T-Read replaces `that` with the receiver `a`: [`a`/`that`]`C<that.DOM>` = `C<a.DOM>`.

Lessons from subderivation trees. `any` allows us to express programming idioms that were not possible using the baseline FDJ typechecker. For example, it allowed us to express that the argument of the method is in either `VIEW` or `CONTROLLER` (Fig. 22). However, if the link rules are not specified the type system would return a warning (Fig. 20). `any` also allows us to express that a collection contains objects from different domains (Fig. 24).

```
class CustomerAgent<OWNER,DOMCUSTOMER,DOMBRANCH>{
  assumes OWNER->DOMCUSTOMER, OWNER->DOMBRANCH

  public void doTransaction(final Branch<DOMBRANCH,DOMUSTOMER> branch) {
    // Active teller assigned to customer
    Teller<branch.TELLERS,DOMBRANCH,DOMCUSTOMER> teller = branch.getActiveTeller();

    /* class Teller: assumes owner->DOMBRANCH, owner->DOMCUSTOMER */
    /* NOTE: Could remove that assumption, if we do not really need it */
    /* Here, this requires the following permission(s): */
    /* branch.TELLERS->DOMBRANCH, and branch.TELLERS->DOMCUSTOMER */
    /* Why does: branch.TELLERS->DOMBRANCH hold? Why does: branch.TELLERS->DOMCUSTOMER hold? */
    /* Use n.d -> p rule * /
    /* Because class Branch:: link this.TELLERS->DOMCUSTOMER */
    /* Substitute actuals for formals: branch/this */
  }
}
class Teller<OWNER,DOMBRANCH,DOMCUSTOMER> {
  assumes OWNER->DOMBRANCH, OWNER->DOMCUSTOMER;
}
class Branch<OWNER,DOMCUSTOMER> {
   domain OWNED,VAULTS;
   public domain TELLERS;

   link OWNED->TELLERS
   link TELLERS->OWNER;
   link TELLERS->DOMCUSTOMER;
   assumes OWNER->DOMCUSTOMER;

  Teller<TELLERS,OWNER,DOMCUSTOMER> teller;

  public Teller<TELLERS,OWNER,DOMCUSTOMER> getActiveTeller() {
    return teller;
  }
}
```

Figure 19: Code fragments from the Banking system.

```
class View<OWNER, M, C> ... {
  private domain OWNED;
  assumes OWNER->OWNER, OWNER->M;
  link OWNED->C, OWNED->OWNER;
  ...
  void addListener(FSListener<any> fsl) {
  ...
  }
}

class System<OWNER> {
    domain MODEL, VIEW, CTRL;
    assumes OWNER->MODEL, OWNER->VIEW, OWNER->CTRL;
    link VIEW->MODEL; // Satisfy assumption OWNER->M in new View()
    link VIEW->CTRL; // Satisfy assumption OWNER->C in new View()
    link CTRL->MODEL; // Needed to create the other object cmd, also used in T-Type
    link CTRL->VIEW; // Needed to create the other object cmd, also used in T-Type

    JavaDrawApp<VIEW,MODEL,CTRL> app = new JavaDrawApp<...>();
    View<VIEW,MODEL,CTRL> view = new View<...>(app);
    Command<CTRL,MODEL,VIEW> cmd = new Command<...>();

    public void init() {
          view.addListener(cmd);
    }

}
```

Figure 20: Listener example that does not typecheck. Missing assumes clause `OWNER->C`.

$$\begin{array}{c}
\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{view:View<VIEW,MODEL,CTRL>} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{cmd:Command<CTRL,MODEL,VIEW>} \\
mtype(\texttt{addListener,View<...>}) = [\texttt{that/this}][\texttt{VIEW/OWNER,MODEL/M,CTRL/C}]\texttt{FSListener<}\ \texttt{this.any}\texttt{>} \rightarrow \texttt{void} \\
= \texttt{FSListener<that.any>} \rightarrow \texttt{void}
\end{array}$$

$$\cfrac{\cfrac{\begin{array}{c}
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{this} \rightarrow \texttt{VIEW} \\
assumptions(\texttt{View<VIEW,MODEL,CTRL>}) = [\texttt{that/this}][\texttt{VIEW/OWNER,MODEL/M,CTRL/C}]\texttt{OWNER->OWNER, OWNER->M} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{VIEW} \rightarrow \texttt{VIEW} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{VIEW} \rightarrow \texttt{MODEL}
\end{array}}{\Gamma, \Sigma, \texttt{this:System<OWNER>} \models assumptions(\texttt{View<VIEW,MODEL,CTRL>})}}{\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{View<VIEW,MODEL,CTRL>}}$$

$$\cfrac{\cfrac{\begin{array}{c}
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{this} \rightarrow \texttt{CTRL} \\
assumptions(\texttt{Command<CTRL,MODEL,VIEW>}) = [\texttt{that/this}][\texttt{CTRL/OWNER,MODEL/M,VIEW/V}]\texttt{OWNER->OWNER,OWNER->M,OWNER->V} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{CTRL} \rightarrow \texttt{CTRL} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{CTRL} \rightarrow \texttt{MODEL} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{CTRL} \rightarrow \texttt{VIEW}
\end{array}}{\Gamma, \Sigma, \texttt{this:System<OWNER>} \models assumptions(\texttt{Command<CTRL,MODEL,VIEW>})}}{\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{Command<CTRL,MODEL,VIEW>}}$$

$$\cfrac{isPrecise(\texttt{CTRL,MODEL,VIEW})}{\texttt{Command<CTRL,MODEL,VIEW>} \in \Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash preciseType(\texttt{Command<CTRL,MODEL,VIEW>})}$$

$$\cfrac{\begin{array}{c}
linkdecls(\texttt{View<...>}) = [\texttt{that/this}][\texttt{VIEW/OWNER,MODEL/M,CTRL/C}]\texttt{OWNER} \rightarrow \texttt{OWNER}, \texttt{OWNER} \rightarrow \texttt{M}
\end{array}}{\begin{array}{c}
solveAny(owner(\texttt{View<VIEW,MODEL,CTRL>}), \texttt{View<VIEW,MODEL,CTRL>}) = \{\texttt{VIEW,MODEL}\} \\
\Gamma, \Sigma, \texttt{that : View<VIEW,MODEL,CTRL>} \not\models \texttt{VIEW} \rightarrow \texttt{CTRL} \\
\Gamma, \Sigma, \texttt{that : View<VIEW,MODEL,CTRL>} \not\models \texttt{MODEL} \rightarrow \texttt{CTRL} \\
WARNING!!! \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{that : View<VIEW,MODEL,CTRL>}
\end{array}}$$

$$\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{that.any} \rightarrow \texttt{CTRL}$$

$$\cfrac{\cfrac{\cfrac{\texttt{CTRL} <:_d \texttt{that.any}}{\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{FSListener<CTRL>} <: \texttt{FSListener<that.any>}} \quad \Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{Command<CTRL,MODEL,VIEW>} <: \texttt{FSListener<CTRL>}}{\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{Command<CTRL,MODEL,VIEW>} <: \texttt{FSListener<that.any>}}}{\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{view.addListener(cmd):void}}$$

Figure 21: Sub-derivation for `view.addListener(cmd)` that throws warning. Code is in Fig. 20.

```
class View<OWNER, M, C> ... {
  private domain OWNED;
  assumes OWNER->OWNER, OWNER->M, OWNER->C;
  link OWNED->C, OWNED->OWNER;
 ...
  void addListener(FSListener<any> fsl) {
  ...
  }
}

class System<OWNER> {
    domain MODEL, VIEW, CTRL;
    assumes OWNER->MODEL, OWNER->VIEW, OWNER->CTRL;
    link VIEW->MODEL; // Satisfy assumption OWNER->M in new View()
    link VIEW->CTRL; // Satisfy assumption OWNER->C in new View()
    link CTRL->MODEL; // Needed to create the other object cmd, also used in T-Type
    link CTRL->VIEW; // Needed to create the other object cmd, also used in T-Type

    JavaDrawApp<VIEW,MODEL,CTRL> app = new JavaDrawApp<...>();
    View<VIEW,MODEL,CTRL> view = new View<...>(app);
    Command<CTRL,MODEL,VIEW> cmd = new Command<...>();

    public void init() {
         view.addListener(cmd);
    }

}
```

Figure 22: Listener example that type checks due to the assumes clause `OWNER->C`.

$$
\frac{
\begin{array}{c}
\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{view:View<VIEW,MODEL,CTRL>} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{cmd:Command<CTRL,MODEL,VIEW>} \\
mtype(\texttt{addListener,View<...>}) = [\texttt{that/this}][\texttt{VIEW/OWNER,MODEL/M,CTRL/C}]\texttt{FSListener<this.any>} \rightarrow \texttt{void} \\
= \texttt{FSListener<that.any>} \rightarrow \texttt{void} \\
\\
\dfrac{\dfrac{\begin{array}{c}
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{this} \rightarrow \texttt{VIEW} \\
assumptions(\texttt{View<VIEW,MODEL,CTRL>}) = [\texttt{that/this}][\texttt{VIEW/OWNER,MODEL/M,CTRL/C}]\texttt{OWNER->OWNER, OWNER->M, OWNER-} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{VIEW} \rightarrow \texttt{VIEW} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{VIEW} \rightarrow \texttt{MODEL} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{VIEW} \rightarrow \texttt{CTRL}
\end{array}}{\Gamma, \Sigma, \texttt{this:System<OWNER>} \models assumptions(\texttt{View<VIEW,MODEL,CTRL>})}}{\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{View<VIEW,MODEL,CTRL>}} \\
\\
\dfrac{\dfrac{\begin{array}{c}
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{this} \rightarrow \texttt{CTRL} \\
assumptions(\texttt{Command<CTRL,MODEL,VIEW>}) = [\texttt{that/this}][\texttt{CTRL/OWNER,MODEL/M,VIEW/V}]\texttt{OWNER->OWNER,OWNER->M,OWNER-} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{CTRL} \rightarrow \texttt{CTRL} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{CTRL} \rightarrow \texttt{MODEL} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \models \texttt{CTRL} \rightarrow \texttt{VIEW}
\end{array}}{\Gamma, \Sigma, \texttt{this:System<OWNER>} \models assumptions(\texttt{Command<CTRL,MODEL,VIEW>})}}{\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{Command<CTRL,MODEL,VIEW>}} \\
\\
\dfrac{isPrecise(\texttt{CTRL,MODEL,VIEW})}{\texttt{Command<CTRL,MODEL,VIEW>} \in \Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash preciseType(\texttt{Command<CTRL,MODEL,VIEW>})} \\
\\
\dfrac{\begin{array}{c}
\dfrac{linkdecls(\texttt{View<...>}) = [\texttt{that/this}][\texttt{VIEW/OWNER,MODEL/M,CTRL/C}]\texttt{OWNER} \rightarrow \texttt{OWNER}, \texttt{OWNER} \rightarrow \texttt{M}, \texttt{OWNER} \rightarrow \texttt{C}}{solveAny(owner(\texttt{View<VIEW,MODEL,CTRL>}), \texttt{View<VIEW,MODEL,CTRL>}) = \{\texttt{VIEW,MODEL,CTRL}\}} \\
\Gamma, \Sigma, \texttt{that : View<VIEW,MODEL,CTRL>} \models \texttt{VIEW} \rightarrow \texttt{CTRL} \\
\Gamma, \Sigma, \texttt{that : View<VIEW,MODEL,CTRL>} \models \texttt{CTRL} \rightarrow \texttt{CTRL} \\
\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{that : View<VIEW,MODEL,CTRL>}
\end{array}}{\Gamma, \Sigma, \texttt{that : View<VIEW,MODEL,CTRL>} \models \texttt{that.any} \rightarrow \texttt{CTRL}} \\
\\
\dfrac{\begin{array}{c}
\dfrac{\texttt{CTRL} <:_d \texttt{that.any}}{\Gamma, \Sigma, \texttt{that : View<VIEW,MODEL,CTRL>} \vdash \texttt{FSListener<CTRL>} <: \texttt{FSListener<that.any>}} \\
\Gamma, \Sigma, \texttt{that : View<VIEW,MODEL,CTRL>} \vdash \texttt{Command<CTRL,MODEL,VIEW>} <: \texttt{FSListener<CTRL>}
\end{array}}{\Gamma, \Sigma, \texttt{that : View<VIEW,MODEL,CTRL>} \vdash \texttt{Command<CTRL,MODEL,VIEW>} <: \texttt{FSListener<that.any>}}
\end{array}
}{
\Gamma, \Sigma, \texttt{this:System<OWNER>} \vdash \texttt{view.addListener(cmd):void}
}
$$

Figure 23: Sub-derivation for `view.addListener(cmd)`. Code is in Fig. 22.

```java
class State<OWNER> {
}
class Listener<OWNER> {
   void update(int data){}
}
class View<OWNER> {
  public domain LISTENERS; // Public domain
  domain STATE; // Private domain
  link OWNER->LISTENERS, OWNER->STATE;
  link LISTENERS->STATE;
  link LISTENERS->OWNER; //important
  State<STATE> state;
  Listener<LISTENERS> listener() {
    return new ViewListener<LISTENERS,STATE>(state);
  }
}

class Model<OWNER, PLIST> {
  domain OWNED;
  assumes OWNER->PLIST;
  link OWNED->PLIST;

  Vector<OWNED,Listener<PLIST>> listeners;

  void addListener(Listener<PLIST> listener) {
    listeners.add(listener);
  }

  void notifyAll(int data) {
    for(Listener<PLIST> listener : listeners) {
           listener.update(data);
    }
  }
}
```

```java
class ViewListener<OWNER,PSTATE> extends Listener<OWNER> {
  assumes OWNER -> PSTATE;
  State<PSTATE> state;
  ViewListener(State<PSTATE> s) {
     this.state = s;
  }
  public void update(int data) {
    /* perform changes on state */
  }
}

class Main<OWNER> {
  domain MODEL, VIEW1, VIEW2;
  link OWNER->MODEL, OWNER->VIEW1, OWNER->VIEW2;
  link MODEL->VIEW1, MODEL->VIEW2;

  // Field must be final to access public domain
  final View<VIEW1> view = new View<VIEW1>();
  // this is any can be view.LISTENERS or view2.LISTENERS.
  // Iteration#2: qualify 'any', with explicit domain links:
  // any[MODEL->any]
  // any[any->MODEL]
  // to prevent calling addListener with argument being a Model w
  // assuming Model class also implements Listener interface
  Model<MODEL, any > model = new Model<MODEL,any>();

  public void run() {
     model.addListener(view.listener());

    View<VIEW2> view2 = new View<VIEW2>();
     model.addListener(view2.listener());
  }
}
```

Figure 24: Creating objects in Model with any as a domain parameter.

$$\dfrac{\begin{array}{c}\Gamma,\Sigma,\texttt{this:Main<OWNER>}\vdash\texttt{view:View<this.VIEW1>}\\ \mathit{mtype}(\texttt{listener},\texttt{View<this.VIEW1>})=[\texttt{that}/\texttt{this}][\texttt{this.VIEW1/OWNER}]\texttt{void}\rightarrow\texttt{Listener<this.LISTENERS>}\\ \dfrac{\begin{array}{c}\Gamma,\Sigma,\texttt{this:Main<OWNER>}\models\texttt{this:Main<OWNER>}\rightarrow\mathit{owner}(\texttt{Listener<view.LISTENERS>})\\ \Gamma,\Sigma,\texttt{this:Main<OWNER>}\models\mathit{assumptions}(\texttt{Listener<view.LISTENERS>})\end{array}}{\Gamma,\Sigma,\texttt{this:Main<OWNER>}\vdash\texttt{Listener<view.LISTENERS>}}\end{array}}{\Gamma,\Sigma,\texttt{this:Main<OWNER>}\vdash\texttt{view.listener():Listener<view.LISTENERS>}}$$

Figure 25: Sub derivation for `view.listener()`. An example of an `that/this` substitution. Fig. 24.

$$
\dfrac{\begin{array}{c}
\Gamma, \Sigma, \texttt{this:Main<OWNER>} \vdash \texttt{model:Model<this.MODEL,any>} \\
\Gamma, \Sigma, \texttt{this:Main<OWNER>} \vdash \texttt{view.listener():[view/that]Listener<view.LISTENERS>} \\
\begin{aligned} \mathit{mtype}(\mathit{addListener}, \texttt{Model<this.MODEL,any>}) &= [\texttt{that/this}][\texttt{this.MODEL/OWNER,this.any/PLIST}]\texttt{Listener<PLIST>} \to \texttt{void} \\ &= \texttt{Listener<this.any>} \to \texttt{void} \end{aligned} \\[2ex]
\dfrac{\begin{array}{c}
\dfrac{\dfrac{\begin{array}{c}
\dfrac{\mathit{linkdecls}(\texttt{Main<OWNER>}) = [\texttt{that/this}][\texttt{OWNER/OWNER}]\texttt{OWNER} \to \texttt{MODEL}, \texttt{OWNER} \to \texttt{VIEW1}, \texttt{OWNER} \to \texttt{VIEW2}, \ldots}{\mathit{solveAny}(\mathit{owner}(\texttt{Main<OWNER>}), \texttt{Main<OWNER>}) = \{\texttt{this.MODEL}, \texttt{this.VIEW1}, \texttt{this.VIEW2}\}} \\
\Gamma, \Sigma, \texttt{this:Main<OWNER>} \models \texttt{this.MODEL} \to \texttt{VIEW1} \\
\Gamma, \Sigma, \texttt{this:Main<OWNER>} \models \texttt{this.MODEL} \to \texttt{VIEW2}
\end{array}}{\dfrac{\Gamma, \Sigma, \texttt{this:Main<OWNER>} \models \texttt{this.MODEL} \to \texttt{this.any}}{\mathit{assumptions}(\texttt{Model<this.MODEL,any>}) = [\texttt{that/this}][\texttt{MODEL/OWNER,this.any/PLIST}]\texttt{OWNER->PLIST}}}}{\Gamma, \Sigma, \texttt{this:Main<OWNER>} \models \mathit{assumptions}(\texttt{Model<this.MODEL,this.any>})} \\
\Gamma, \Sigma, \texttt{this:Main<OWNER>} \models \texttt{this} \to \texttt{this.MODEL}
\end{array}}{\Gamma, \Sigma, \texttt{this:Main<OWNER>} \vdash \texttt{Model<this.MODEL,this.any>}} \\[2ex]
\dfrac{\dfrac{\dfrac{\begin{array}{c}
\Gamma, \Sigma, \texttt{this:Main<OWNER>} \vdash \texttt{view:View<this.VIEW1>} \\
\Gamma, \Sigma, \texttt{this:Main<OWNER>} \models \texttt{OWNER} \to \texttt{this.VIEW1} \\
\mathit{public}(\texttt{LISTENERS})
\end{array}}{\Gamma, \Sigma, \texttt{this:Main<OWNER>} \models \texttt{OWNER} \to \texttt{view.LISTENERS}}}{\Gamma, \Sigma, \texttt{this:Main<OWNER>} \models \texttt{this} \to \texttt{view.LISTENERS}}
\quad
\dfrac{\mathit{assumptions}(\texttt{Listener<view.LISTENERS>}) = \emptyset}{\Gamma, \Sigma, \texttt{this:Main<OWNER>} \models \mathit{assumptions}(\texttt{Listener<view.LISTENERS>})}}{\Gamma, \Sigma, \texttt{this:Main<OWNER>} \vdash \texttt{Listener<view.LISTENERS>}} \\[2ex]
\dfrac{\mathit{isPrecise}(\texttt{view.LISTENERS})}{\texttt{Listener<view.LISTENERS>} \in \Gamma, \Sigma, \texttt{this:Main<OWNER>} \vdash \mathit{preciseType}(\texttt{Listener<view.LISTENERS>})} \\[2ex]
\dfrac{\mathit{linkdecls}(\texttt{Main<OWNER>}) = [\texttt{that/this}][\texttt{OWNER/OWNER}]\texttt{OWNER} \to \texttt{this.VIEW1}, \ldots}{\mathit{solveAny}(\texttt{OWNER}, \texttt{Main<OWNER>}) = \{\texttt{this.VIEW1}\}} \\[2ex]
\dfrac{\begin{array}{c}
\dfrac{\begin{array}{c}
\Gamma, \Sigma, \texttt{this:Main<OWNER>} \vdash \texttt{view:View<this.VIEW1>} \\
\Gamma, \Sigma, \texttt{this:Main<OWNER>} \models \texttt{VIEW1} \to \texttt{VIEW1} \\
\mathit{public}(\texttt{LISTENERS})
\end{array}}{\Gamma, \Sigma, \texttt{this:Main<OWNER>} \models \texttt{VIEW1} \to \texttt{view.LISTENERS}} \\
\Gamma, \Sigma, \texttt{this:Main<OWNER>} \vdash \texttt{this:Main<OWNER>}
\end{array}}{\Gamma, \Sigma, \texttt{this:Main<OWNER>} \models \texttt{this.any} \to \texttt{view.LISTENERS}} \\[2ex]
\dfrac{\texttt{view.LISTENERS} <:_d \texttt{this.any}}{\Gamma, \Sigma, \texttt{this:Main<OWNER>} \vdash \texttt{Listener<view.LISTENERS>} <: \texttt{Listener<this.any>}}
\end{array}}{\Gamma, \Sigma, \texttt{this:Main<OWNER>} \vdash \texttt{model.addListener(view.listener());}}
$$

Figure 26: Sub derivation for `model.addListener(view.listener())`. Code is in Fig. 24.

```
class Main<OWNER> {
  domain DATA, VIEW;
  link OWNER->DATA, OWNER->VIEW;
  link VIEW->DATA;

  void run() {
   Bar<VIEW,DATA> aBar = new Bar<VIEW,DATA>();
   aBar.test();
  }
}

class Biff<OWNER,D> {
  assumes OWNER->OWNER; //default
  //allows biff.getFoo() to be an argument
  assumes D->OWNER;

  Foo<any> getFoo() {
    return new Foo<OWNER>();
  }
}
```

```
class Bar<OWNER,D> {
  domain OWNED;
  assumes OWNER->D;

  //allows biff to be created
  //allows biff to be a receiver
  link OWNER->this.OWNED

  Foo<any> foo;

  public void test() {
    Biff<OWNED,OWNER> biff = new Biff<OWNED,OWNER>();
      this.foo = biff.getFoo();
  }
}

class Foo<OWNER>{
...
}
```

Figure 27: `any` is not a constant domain like `shared`. Without the link and assume statements, the evaluation ends with a warning.

$$\dfrac{\begin{array}{c}\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\vdash\mathtt{biff:Biff<this.OWNED,OWNER>}\\ mtype(\mathtt{getFoo},\mathtt{Biff<this.OWNED,OWNER>}) = [\mathtt{that/this}][\mathtt{this.OWNED/OWNER},\mathtt{OWNER/D}]\mathtt{void}\to\mathtt{Foo<this.any>}\\ \\ \dfrac{\begin{array}{c}\dfrac{\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\models\ \mathtt{OWNER}\to\mathtt{this.OWNED}}{\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\models assumptions(\mathtt{Biff<this.OWNED,OWNER>}) = [\mathtt{that/this}][\mathtt{this.OWNED/OWNER},\mathtt{OWNER/D}]\mathtt{OWNER\text{-}>OWNER,\ D\text{-}>OWNER}}\\ \Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\models\mathtt{this}\to\mathtt{this.OWNED}\end{array}}{\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\vdash\mathtt{Biff<this.OWNED,OWNER>}}\end{array}}{\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\vdash\mathtt{biff.getFoo():[biff/that]Foo<that.any>}}$$

$$\begin{array}{c}\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\vdash\mathtt{this}:\mathtt{Bar<OWNER,D>}\\ fields(\mathtt{Bar<OWNER,D>}) = [\mathtt{this/this}][\mathtt{OWNER/OWNER},\mathtt{D/D}]\mathtt{Foo<this.any>\ foo;}\\ \Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\vdash\mathtt{biff.getFoo():Foo<biff.any>}\end{array}$$

$$\dfrac{\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\models\mathtt{this}\to\mathtt{OWNER}\qquad\dfrac{\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\models\mathtt{OWNER}\to\mathtt{D}}{\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\models assumptions(\mathtt{Bar<OWNER,D>}) = \mathtt{OWNER\text{-}>D}}}{\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\vdash\mathtt{Bar<OWNER,D>}}$$

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\dfrac{linkdecls(\mathtt{Biff<this.OWNED,OWNER>}) = [\mathtt{that/this}][\mathtt{OWNED/OWNER},\mathtt{OWNER/D}]\mathtt{OWNER\text{-}>OWNER},\mathtt{D\text{-}>OWNER}}{\Gamma,\Sigma,\mathtt{biff}:\mathtt{Biff<this.OWNED,OWNER>}\models\mathtt{OWNER}\to\mathtt{this.OWNED}}\\ \mathtt{this.OWNED}\in solveAny(\mathtt{Biff<this.OWNED,OWNER>})\end{array}}{\Gamma,\Sigma,\mathtt{this}:\mathtt{Bar<OWNER,D>}\models\mathtt{OWNER}\to\mathtt{biff.any}}}{\Gamma,\Sigma,\mathtt{this}:\mathtt{Bar<OWNER,D>}\models\mathtt{this}\to\mathtt{biff.any}}}{\Gamma,\Sigma,\mathtt{this}:\mathtt{Bar<OWNER,D>}\vdash\mathtt{Foo<biff.any>}}$$

$$\dfrac{\begin{array}{c}\dfrac{linkdecls(\mathtt{Biff<this.OWNED,OWNER>}) = [\mathtt{that/this}][\mathtt{OWNED/OWNER},\mathtt{OWNER/D}]\mathtt{OWNER\text{-}>OWNER},\mathtt{D\text{-}>OWNER}}{\{\mathtt{this.OWNED}\}\in solveAny(\mathtt{this.OWNED},\mathtt{Biff<this.OWNED,OWNER>})}\\ \Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\vdash\mathtt{biff:Biff<this.OWNED,OWNER>}\end{array}}{\{\mathtt{Foo<this.OWNED>}\}\in\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\vdash preciseType(\mathtt{Foo<biff.any>})}$$

$$\dfrac{\begin{array}{c}\dfrac{\begin{array}{c}\dfrac{linkdecls(\mathtt{Bar<OWNER,D>}) = [\mathtt{that/this}][\mathtt{OWNER/OWNER},\mathtt{D/D}]\mathtt{OWNER\text{-}>OWNER},\mathtt{OWNER\text{-}>D},\mathtt{OWNER\text{-}>this.OWNED}}{solveAny(\mathtt{Bar<OWNER,D>}) = \{\mathtt{OWNER,\ D,\ this.OWNED}\}}\\ \Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\vdash\mathtt{this}:\mathtt{Bar<OWNER,D>}\\ \Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\models\mathtt{OWNER}\to\mathtt{this.OWNED}\\ \Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\models\mathtt{this.OWNED}\to\mathtt{this.OWNED}\\ \Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\not\models\mathtt{D}\to\mathtt{this.OWNED}\end{array}}{\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\models\mathtt{this.any}\to\mathtt{this.OWNED}}\\ \\ \dfrac{\mathtt{this.OWNED}<:_d\mathtt{this.any}}{\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\vdash\mathtt{Foo<this.OWNED>}<:\mathtt{Foo<this.any>}}\end{array}}{\Gamma;\Sigma;\mathtt{this}:\mathtt{Bar<OWNER,D>}\vdash\mathtt{this.foo\ =\ biff.getFoo():Foo<OWNED>}}$$

Figure 28: Sub derivation for `this.foo = biff.getFoo()`. Code is in Fig. 27.

$$
\begin{array}{c}
\texttt{void loadCircuit()}\ \mathit{OK\ in\ C} \\
\overline{F} = \texttt{Circuit<M> circ Node<any,M> node}
\end{array}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\mathit{linkdecls}(\texttt{Viewer<OWNER,M>}) = \texttt{OWNER->M}}{\{\texttt{this} : \texttt{Viewer<OWNER,M>}\}; \emptyset; \texttt{this} \models \texttt{OWNER} \to \texttt{M}}}{\{\texttt{this} : \texttt{Viewer<OWNER,M>}\}; \emptyset; \texttt{this} \models \texttt{this} \to \texttt{M}}}{\{\texttt{this} : \texttt{Viewer<OWNER,M>}\}; \emptyset; \texttt{this} \models \texttt{this} \to \mathit{owner}(\texttt{Circuit<M>})}}{\{\texttt{this} : \texttt{Viewer<OWNER,M>}\}; \emptyset; \texttt{this} \vdash \texttt{Circuit<M>}}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\texttt{OWNER}, \texttt{M} \in \mathit{solveAny}(\texttt{OWNER}, \texttt{Viewer<OWNER,M>}) \\ \texttt{OWNER} \to \texttt{M} \in \mathit{assumptions}(\texttt{Viewer<OWNER,M>})\end{array}}{\{\texttt{this} : \texttt{Viewer<OWNER,M>}\}; \emptyset; \texttt{this} \models \texttt{OWNER} \to \texttt{M}}}{\{\texttt{this} : \texttt{Viewer<OWNER,M>}\}; \emptyset; \texttt{this} \models \texttt{OWNER} \to \texttt{any}}}{\{\texttt{this} : \texttt{Viewer<OWNER,M>}\}; \emptyset; \texttt{this} \models \texttt{this} \to \texttt{any}}}{\{\texttt{this} : \texttt{Viewer<OWNER,M>}\}; \emptyset; \texttt{this} \models \texttt{this} \to \mathit{owner}(\texttt{Node<any,M>})}
\quad
\dfrac{\begin{array}{c}\dfrac{\dfrac{\begin{array}{c}\texttt{OWNER}, \texttt{M} \in \mathit{solveAny}(\texttt{OWNER}, \texttt{Viewer<OWNER,M>}) \\ \texttt{OWNER} \to \texttt{M} \in \mathit{assumptions}(\texttt{Viewer<OWNER,M>})\end{array}}{\{\texttt{this} : \texttt{Viewer<OWNER,M>}\}; \emptyset; \texttt{this} \models \texttt{OWNER} \to \texttt{M}}}{\{\texttt{this} : \texttt{Viewer<OWNER,M>}\}; \emptyset; \texttt{this} \models \texttt{any} \to \texttt{M}} \\ \mathit{assumptions}(\texttt{Node<any,M>}) = [\texttt{that/this}][\texttt{any/OWNER,M/M}]\texttt{OWNER->M}\end{array}}{\{\texttt{this} : \texttt{Viewer<OWNER,M>}\}; \emptyset; \texttt{this} \models \mathit{assumptions}(\texttt{Node<any,M>})}}{\{\texttt{this} : \texttt{Viewer<OWNER,M>}\}; \emptyset; \texttt{this} \vdash \texttt{Node<any,M>}}}{\texttt{class Viewer<OWNER,M>\{ link OWNER->M; Circuit<M> circ; Node<any,M> node; ...\}}}
$$

Figure 29: Sub derivation for *ClsOK* `Viewer<OWNER,M>`. Code is in Fig. 4b.

$$\dfrac{\begin{array}{c} \Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{getCircuit():Circuit<OWNER>} \\ \Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{p:Point<shared>} \\ mtype(getNodeAt, \texttt{Circuit<OWNER>}) = [\texttt{that/this}][\texttt{OWNER/OWNER}]\texttt{Point<shared>} \rightarrow \texttt{Node<this.DB,OWNER>} \\ \\ \dfrac{\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \models \texttt{this} \rightarrow \texttt{OWNER}}{\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{Circuit<OWNER>}} \\ \\ \dfrac{\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \models \texttt{this} \rightarrow \texttt{shared}}{\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{Point<shared>}} \\ \texttt{Point<shared>} \in \Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash preciseType(\texttt{Point<shared>}) \\ \Gamma, \Sigma, \texttt{that:Circuit<OWNER>} \vdash \texttt{Point<shared>} <: \texttt{Point<shared>} \end{array}}{\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{getCircuit().getNodeAt(p):[getCircuit()/that]Node<that.DB,OWNER>}}$$

Figure 30: Sub derivation for `getCircuit().getNodeAt(p)`. Code is in Fig. 4b.

$$\dfrac{\begin{array}{c} \Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{this:Placer<OWNER>} \\ \Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{p:Point<shared>} \\ mtype(getNodeAt, \texttt{Placer<OWNER>}) = [\texttt{that/this}][\texttt{OWNER/OWNER}]\texttt{Point<shared>} \rightarrow \texttt{Node<this.any,OWNER>} \\ \\ \dfrac{\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \models \texttt{this} \rightarrow \texttt{OWNER}}{\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{Placer<OWNER>}} \\ \\ \dfrac{\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \models \texttt{this} \rightarrow \texttt{shared}}{\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{Point<shared>}} \\ \texttt{Point<shared>} \in \Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash preciseType(\texttt{Point<shared>}) \\ \Gamma, \Sigma, \texttt{that:Circuit<OWNER>} \vdash \texttt{Point<shared>} <: \texttt{Point<shared>} \end{array}}{\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{this.getNodeAt(p):[getCircuit()/that]Node<that.any,OWNER>}}$$

Figure 31: Sub derivation for `this.getNodeAt(p)`. Code is in Fig. 4b.

$$
\dfrac{
\begin{array}{c}
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{this:Placer<OWNER>} \\
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{node:Node<this.any,OWNER>} \\
\mathit{mtype}(\mathit{setPlacement}, \texttt{this:Placer<OWNER>}) = [\texttt{that/this}][\texttt{OWNER/OWNER}]\texttt{Node<this.any,OWNER>} \to \texttt{void} \\
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{Placer<OWNER>} \\
\\
\dfrac{
\dfrac{
\dfrac{
\begin{array}{c}
\texttt{OWNER} \in \mathit{solveAny}(\mathit{owner}(\texttt{Placer<OWNER>}), \texttt{Placer<OWNER>}) \\
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \models \texttt{OWNER} \to \texttt{OWNER}
\end{array}
}{
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \models \texttt{OWNER} \to \texttt{this.any}
}
}{
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \models \texttt{this} \to \mathit{owner}(\texttt{Node<this.any,OWNER>})
}
\quad
\dfrac{
\begin{array}{c}
\dfrac{
\begin{array}{c}
\texttt{OWNER} \in \mathit{solveAny}(\mathit{owner}(\texttt{Placer<OWNER>}), \texttt{Placer<OWNER>}) \\
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \models \texttt{OWNER} \to \texttt{OWNER}
\end{array}
}{
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \models \texttt{this.any} \to \texttt{OWNER}
} \\
\mathit{assumptions}(\texttt{Node<any,OWNER>}) = [\texttt{that/this}][\texttt{this.any/OWNER,OWNER/M}]\texttt{OWNER}
\end{array}
}{
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \models \mathit{assumptions}(\texttt{Node<this.any,OWNER>})
}
}{
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{Node<this.any,OWNER>}
} \\
\\
\dfrac{
\texttt{OWNER} \in \mathit{solveAny}(\mathit{owner}(\texttt{Placer<OWNER>}), \texttt{Placer<OWNER>})
}{
\texttt{Node<OWNER,OWNER>} \in \Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \mathit{preciseType}(\texttt{Node<any,OWNER>})
} \\
\\
\dfrac{
\begin{array}{c}
\texttt{OWNER} \in \mathit{solveAny}(\mathit{owner}(\texttt{Placer<OWNER>}), \texttt{Placer<OWNER>}) \\
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \models \texttt{OWNER} \to \texttt{OWNER}
\end{array}
}{
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \models \texttt{this.any} \to \texttt{OWNER}
} \\
\\
\dfrac{
\texttt{OWNER} <:_d \texttt{any} \qquad \texttt{OWNER} = \texttt{OWNER}
}{
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{Node<OWNER,OWNER>} <: \texttt{Node<this.any,OWNER>}
}
\end{array}
}{
\Gamma, \Sigma, \texttt{this:Placer<OWNER>} \vdash \texttt{this.setPlacement(node)}:
}
$$

Figure 32: Sub derivation for `this.setPlacement(node)`. Code is in Fig. 4b.