Ownership Domains: Separating Aliasing Policy from Mechanism

Jonathan Aldrich[1] and Craig Chambers[2]

[1] Carnegie Mellon University, Pittsburgh, PA 15217, USA,
jonathan.aldrich@cs.cmu.edu
[2] University of Washington, Seattle, WA, 98195, USA,
chambers@cs.washington.edu

Abstract. Ownership types promise to provide a practical mechanism for enforcing stronger encapsulation by controlling aliasing in object-oriented languages. However, previous ownership type proposals have tied the aliasing policy of a system to the mechanism of ownership. As a result, these proposals are too weak to express many important aliasing constraints, yet also so restrictive that they prohibit many useful programming idioms.

In this paper, we propose *ownership domains*, which decouple encapsulation policy from the mechanism of ownership in two key ways. First, developers can specify multiple ownership domains for each object, permitting a fine-grained control of aliasing compared to systems that provide only one ownership domain for each object. Second, developers can specify the permitted aliasing between each pair of domains in the system, providing more flexibility compared to systems that enforce a fixed policy for inter-domain aliasing. Because it decouples policy from mechanism, our alias control system is both more precise and more flexible than previous ownership type systems.

1 Introduction

One of the primary challenges in building and evolving large object-oriented systems is reasoning about aliasing between objects. Unexpected aliasing can lead to broken invariants, mistaken assumptions, security holes, and surprising side effects, which in turn may cause defects and complicate software evolution.

Ownership types are one promising approach to addressing the problems of uncontrolled aliasing [23, 13, 10, 4, 8, 11]. With ownership types, the developer of an abstract data type can encapsulate objects used in the internal representation of the ADT, and use static typechecking to ensure that clients of the ADT cannot access its representation.

Despite the potential of ownership types, current ownership type systems have serious limitations, both in the kinds of aliasing constraints they can express and in their ability to support important programming idioms. These limitations can be understood by looking at ownership types as a combination of a *mechanism* for dividing objects into hierarchical groups, and a *policy* for constraining references between objects in those groups.

In previous ownership type systems, each object defines a single group to hold its private state. We will call these groups *ownership domains*. The ownership mechanism is useful for separating the internals of an abstract data type from clients of the ADT, but since each object defines only one ownership domain, ownership types cannot be used to reason about aliasing between different subsystems within an object.

The aliasing policy in previous ownership type systems is fixed: the private state of an object can refer to the outside world, but the outside world may not refer to the private state of the object. This policy is known as owners-as-dominators, because it implies that all paths to an object in a system must go through that object's owner.

This fixed policy is useful for ensuring that clients cannot access the internals of an abstract data type. However, the policy is too restrictive to support common programming idioms such as iterator objects or event callbacks. In these idioms, the iterator or event callback objects must be outside of the ADT so that clients can use them, but they must be able to access the internals of the ADT to do their jobs. Thus, iterators or callbacks create paths to the inside of an ADT that do not go through the ADT object itself, violating owners-as-dominators.

In this paper, we propose ownership domains, an extension of ownership types that separates the alias control policy of a system from the mechanism of ownership. Our system generalizes the mechanism of ownership to permit multiple ownership domains in each object. Each domain represents a logically related set of objects. Thus, developers can use the ownership domain mechanism to divide a system object into multiple subsystems, and can separately specify the policy that determines how those subsystems can interact.

Instead of hard-wiring an aliasing policy into the ownership mechanism, our system allows engineers to specify in detail the permitted aliasing relationships between domains. For example, a Sequence ADT can declare one domain for its internal representation, and a second domain for its iterators. The aliasing policy for the Sequence ADT can be written to allow clients to access the iterators, and to allow the iterators to access the internal representation of the Sequence, while prohibiting clients from accessing the internal representation directly.

As a result of separating the mechanism for dividing objects into domains from the policy of how objects in those domains may interact, our system is both more precise and more flexible than previous ownership type systems. It is more precise in allowing developers to control aliasing between the sub-parts of an object. Furthermore, while our system can be used to statically enforce the owners-as-dominators property, it also supports more flexible alias control policies that permit idioms like iterators or events.

The rest of this paper is organized as follows. In the next section we introduce ownership domains by example, showing how they can express aliasing policies and code idioms that were not expressible in previous systems. We have im-

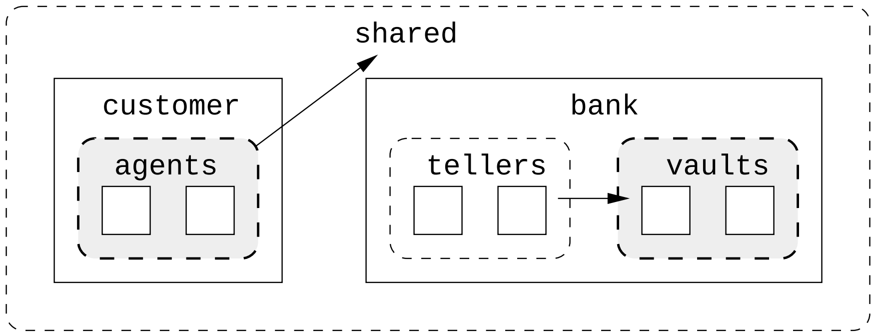

Fig. 1. A conceptual view of AliasJava's ownership domain model. The rounded, dashed rectangles represent domains, with a gray fill for private domains. Solid rectangles represent objects. The top-level `shared` domain contains the highest-level objects in the program. Each object may define one or more domains that in turn contain other objects.

plemented ownership domains in the open-source AliasJava compiler [4], which is freely available at `http://www.archjava.org/`. Section 3 presents our proposal more formally as an extension of Featherweight Java [18], and proves that our type system enforces the aliasing specifications given by the programmer. Section 4 discusses related work, and Section 5 concludes.

2 Ownership Domains

We illustrate ownership domains in the context of AliasJava, an extension to the Java programming language that adds support for a variety of alias specifications [4]. Figure 1 illustrates the ownership domain model used in AliasJava. Every object in the system is part of a single ownership domain. There is a top-level ownership domain denoted by the keyword **`shared`**. In addition, each object can declare one or more domains to hold its internal objects.

The example shows two objects in the **`shared`** domain: a `bank` and a `customer`, denoted by rectangles. The customer declares a domain denoting the customer's agents; the domain has two (unnamed) agent objects in it. The bank declares two domains: one for the tellers in the bank, and one for the bank's vaults. In this example, there are two tellers and two bank vaults.

Each object can declare a policy describing the permitted aliasing among objects in its internal domains, and between its internal domains and external domains. Our system supports two kinds of policy specifications:

– A `link` from one domain to another, denoted with an arrow in the diagram, allows objects in the first domain to access objects in the second domain.
– A domain can be declared `public`, denoted by a thinner dashed rectangle with no shading. Permission to access an object automatically implies permission to access its public domains.

```
class Class {
  domain owned;
  owned List signers;

  /* clients cannot call a method returning an owned list */
  owned List getSigners() {
    return signers;
  }
}
```

Fig. 2. In an early version of the JDK, the `Class.getSigners` method returned the internal list of signers rather than a copy, allowing untrusted clients to pose as trusted code. In an ownership type system, the list could be declared `owned`, and the typechecker would have caught this error at compile time.

For example, the `customer` object declares a link from its `agent` domain to the `shared` domain, allowing the customer's agents to access the bank. The bank declares a link from its `tellers` domain to its `vaults` domain, allowing the tellers to access the vaults. Finally, the bank declares its `tellers` domain to be `public`, allowing the customer and the customer's agents (and any other object that can access the bank) to also access the `tellers`. Note that permissions in our model are *not* transitive, so that the customer and the agents cannot access the bank's vaults directly; they must go through the bank or its tellers.

In addition to the explicit policy specifications mentioned above, our system includes the following implicit policy specifications:

- An object has permission to access other objects in the same domain.
- An object has permission to access objects in the domains that it declares.

The first rule allows the customer to access the bank (and vice versa), while the second rule allows the customer to access its agents and the bank to access its tellers and vaults. Any aliasing relationship not explicitly permitted by one of these rules is prohibited, according to the principle of least privilege.

2.1 Domain Declarations

Figure 2 illustrates ownership domains by showing how they could have been used to catch a security hole in an early release of the JDK, version 1.1. In this somewhat simplified example[1], the security system function `Class.getSigners` returns a pointer to an internal list, rather than a copy. Clients can then modify the list, compromising the Java security model and potentially allowing malicious applets to pose as trusted code.

[1] The real bug was of the same form but involved native code and arrays, not Java code and lists. In an earlier paper, we show how ownership can be integrated with arrays [4].

The code in Figure 2 has been annotated with ownership domain information in order to document the permitted aliasing relationships and prevent bugs like the one described above. Each `Class` instance contains a private domain `owned` that is distinct from the `owned` domain of all other `Class` instances.

In our system, the owner of an object is expressed as part of its type, appearing before the class part of the type. For example, the `signers` field refers to a list that is in the `owned` domain, expressing the fact that the list must not be shared with clients.

According to the typing rules of ownership domains, only objects that have access to the `owned` domain of a `Class` object can access its `signers` field or call the `getSigners` method. Since the `owned` domain is private, only the `Class` object itself can access these members–it would be a typechecking error if client code tried to call `getSigners`.

Although the `signers` field is intended to be private, we would like clients to be able to get the signers of a class, as long as they cannot modify the internal list holding the signers. Thus, a more appropriate return type for `getSigners` would be `shared List`, where the `shared` domain represents a set of globally visible objects. If we were to give `getSigners` this return type and leave the implementation as is, we would get a type error, because the actual list returned by the function has domain `owned`, not **`shared`**. The correct solution is the one used to fix this bug in the actual JDK: allocating a new list each time `getSigners` is called, copying the signers into the new list and returning the result.

This example shows that using ownership domains to protect internal state from clients enforces a stronger invariant than `private` declarations, because the latter only protect the field, not the object in the field. Thus, ownership domains are a useful tool for enforcing the internal invariants of a system, including security invariants like the one in this example.

2.2 Parameterized Types

Figure 3 illustrates how a `Sequence` abstract data type can be expressed with ownership domains. The `Sequence` must have internal references to the elements in the sequence, but the elements are typically part of the domain of some client. Following Flexible Alias Protection [23] and Featherweight Generic Confinement [24], we leverage Java's generics mechanism to specify the ownership information of a type parameter along with the name. Therefore, we parameterize the `Sequence` class by some type `T`, which includes the class of the elements as well as the domain they are in.

Since the sequence must maintain internal pointers to its elements, the compiler must typecheck it assuming that the sequence object has permission to access the domain of `T`. This assumption is expressed with an `assumes` clause stating that the special domain `owner` (meaning the owner of the current object) has permission to access `T.owner`, the domain of `T`. Whenever `Sequence` is instantiated with type parameter `T`, and is placed in some owner domain, this assumption is checked. For example, in Figure 5 a client instantiates the sequence,

```
class Sequence<T> assumes owner -> T.owner /* default */ {
  domain owned;          /* default */
  link owned -> T.owner; /* default */
  owned Cons<T> head;    /* owned is default here */
  void add(T o) {
    head = new Cons<T>(o,head)
  }

  public domain iters;
  link iters -> T.owner, iters -> owned;
  iters Iterator<T> getIter() {
    return new SequenceIterator<T, owned>(head);
  }
}

class Cons<T> assumes owner -> T.owner /* default */  {
  Cons(T obj, owner Cons<T> next) { this.obj=obj; this.next=next; }
  T obj;
  owner Cons<T> next;
}
```

Fig. 3. A `Sequence` abstract data type that uses a linked list for its internal representation. The `Sequence` declares a publicly accessible `iters` domain representing its iterators, as well as a private `owned` domain to hold the linked list. The `link` declarations specify that iterators in the `iter` domain have permission to access objects in the `owned` domain, and that both domains can access owner of the type parameter `T`.

passing in the type `state Object` for the type parameter `T` and placing the sequence in the `state` domain. Since the `state` domain (like every other domain) is considered to be linked to itself, the assumption is valid in this example.

In practice, nearly all objects need `assume` clauses linking their `owner` domain to their domains of their parameter types, so these clauses are defaults in our system and may be omitted.

The code in Figure 3 represents the sequence internally as a linked list. Clients of the `Sequence` should not be able to access the list directly, so the `Sequence` stores the linked list in a private domain called `owned`. Because the links in the list need to be able to refer to the elements of the sequence, the code includes a `link` declaration specifying that objects in the `owned` domain can refer to objects in the `T.owner` domain.

The `Cons` class represents a link in the linked list. In the example, `Cons` is also parameterized by the element type `T`. The cons cell declares that the `next` field is owned by the `owner` of the current cell, so that all the links in the list have the same owning domain. Back in the `Sequence` class, the `head` field has type `owned Cons<T>`, meaning that the field refers to a `Cons` object in the `owned` domain with the type parameter `T`.

```
interface Iterator<T> {
  T next();
  boolean hasNext();
}

class SequenceIterator<T, domain list> implements Iterator<T>
          assumes list -> T.owner {
  SequenceIterator<T, domain list>(list Cons<T> head) { current = head; }
  list Cons<T> current;

  boolean hasNext() { return current != null; }
  T next() {
    T obj = current.obj;
    current = current.next;
    return obj;
  }
}
```

Fig. 4. An iterator interface and a sequence iterator that implements the interface

In our proposed system, not only is the assumption `owner -> T.owner` a default, but every object has a `owned` domain by default that is linked to each of the domains of type parameters (such as `T.owner`). Also, every field of an object is `owned` by default. This means that in Figure 3, most of the ownership declarations may be omitted. The only declarations that are necessary are the `owner` annotations on `next` in `Cons`, and the declarations that have to do with iterators (discussed below). Thus, in the common case where a single domain per object is sufficient, and where domain parameters match the type parameters of Generic Java, there is very little programming overhead to using our system.

2.3 Expressing Iterators

It is typical for abstract data types like `Sequence` to provide a way for clients to iterate over their contents, but expressing iterators in previous ownership type systems presents a problem. If the iterator is part of a client's ownership domain, then it cannot access the links in the list, and so it cannot be implemented. However, if the iterator is part of the internal `owned` domain, the iterator will be useless because clients cannot access it. Previous solutions to this problem have been ad-hoc and often restrictive: for example, allowing iterators on the stack but not on the heap [11] or supporting iterator-like functionality only if the iterators are implemented as inner classes [10, 8].

Intuitively, the iterators are part of the public interface of the sequence: they should be accessible to clients, but they should also be able to access the internals of the sequence [22]. With ownership domains, this intuition can be expressed in a straightforward manner. A second domain, `iters`, is declared to hold the

```
class SequenceClient {
  domain state;
  final state Sequence<state Object> seq = new Sequence<state Object>();

  void run() {
    state Object obj = ...
    seq.add(obj);

    seq.iters Iterator<state Object> i = seq.getIter();
    while (i.hasNext()) {
      state Object cur = i.next();
      doSomething(cur);
    }
  }
}
```

Fig. 5. A client of the Sequence

iterators of the sequence. So that clients can use the iterator objects, we make the `iters` domain public. In our system, permission to access the sequence implies permission to access its public domains.

In order to allow the iterator to access the elements and links in the sequence, we link the `iters` domain to the `T.owner` and `owned` domains. Then we can write a `getIter` method that creates a new `SequenceIterator` object and returns it as part of the `iters` domain.

The definitions of the `Iterator` interface and the concrete `SequenceIterator` class are shown in Figure 4. The `Iterator` interface has a single type parameter `T` to capture the class and owner of the elements over which it iterates. The `SequenceIterator` class has the type parameter `T` and also a *domain parameter* `list`, because its internal implementation must be able to refer to the `Cons` objects in the sequence. The domain parameter is just like a type parameter except it holds only a domain, not a full type. The `list` domain is used within the `SequenceIterator` to refer to the owner of the `Cons` cells.

The `SequenceIterator` class assumes that the `list` domain parameter has permission to refer to objects in the `T.owner` domain. This assumption is needed to fulfill the assumptions that `Cons` makes.

2.4 Using Sequence

Figure 5 shows a client of the `Sequence` ADT. The client declares some domain, `state`, which holds both the sequence and its elements. Thus the `Sequence` type is parameterized by the type `state Object`, meaning objects of class `Object` that are part of the `state` domain. The `run` method creates an object in the `state` domain, and adds it to the sequence. It then calls `getIter` to get an iterator for the sequence. The iterator is in the `iters` domain of the sequence.

Since each sequence has its own `iters` domain, we need to prefix the domain name by the object that declared it. In order to ensure type safety, the program source can only refer to domains of `final` variables such as `seq`–otherwise, we could assign another sequence to the `seq` variable and the type system would lose track of the relationship between a sequence object and its iterators.

2.5 Properties: Link Soundness

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 3.5 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'. We will say that object o declares a domain d or a link between domains d and d' if the class of o declares a domain or a link between domains that, when o is instantiated, refer to d and d'. Finally, we say that object o *has permission to access* domain d if one of the following conditions holds:

1. o is in domain d', and some object declares a link of the form **`link`** d' `->` d.
2. o has permission to access object o', and o' declares a public domain d.
3. o is part of domain d.
4. d is a domain declared by o.

These rules simply state the conditions in the introduction to section 2 more precisely. We can now define link soundness using the definitions above:

Definition 1 (Link Soundness). *If an 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 1, the `customer` should not be able to give itself access to the `bank`'s `vaults` domain. We can ensure this with the following restriction:

– Each `link` declaration must include a locally-declared domain.

Furthermore, even though the `agents` domain is local to the `customer` object, the `customer` should not be able to give the `agents` any privileges that the `customer` does not have itself. 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 customer should not be able to get to the bank's `vaults` domain by creating its own objects in the `tellers` domain:

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

Unlike many previous ownership type systems, 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 by leaving this permission out.

Relation to Previous Work. Previous ownership type systems have enforced the owners-as-dominators property: all paths to an object in a system must go through that object's owner. The link soundness property is more flexible than owners-as-dominators, since it can express examples like the `Iterator` in section 2.3 that violate the owners-as-dominators constraint. However, ownership domains can be used to enforce owners-as-dominators if programmers obey the following guidelines:

– Never declare a public domain.
– Never link a domain parameter to an internal domain.

These guidelines ensure that the rules for link declarations and public domains (rules #1 and #2 above) cannot be used to access internal domains. Rule #3 does not apply, and the only other way to access internal domains is through the object that declared them (rule #4), which is what owners-as-dominators requires.

These guidelines show that previous ownership type systems are essentially a special case of ownership domains. Thus, ownership domains provide a tradeoff between reasoning and expressiveness. Engineers can use ownership domains to enforce owners-as-dominators when this property is needed, but can also use a more flexible alias-control policy in order to express idioms like iterators.

2.6 Listener Callbacks

The listener idiom, an instance of the subject-observer design pattern [14], is very common in object-oriented libraries such as the Java Swing GUI. This pattern is often implemented as shown in Figure 6, where a `Listener` object creates a callback object that is invoked when some event occurs in the event `Generator` being observed. Expressing this idiom is impossible in ownership type systems that enforce owners-as-dominators, since the callback object is visible from the `Generator` but keeps internal references to the state of the `Listener`.

Using ownership domains, we can express this example as shown in Figure 6. The `ListenerSystem` declares domains representing the `generator` and `listener`, and links the `generator` domain to the `listener` domain so that it can pass the listener's callback to the generator. The `Generator` class needs to

```
class ListenerSystem {
   domain generator, listener;
   link generator->listener;
   generator Generator<l.callbacks Callback> s;
   final listener Listener l;
   ... s.callback = l.getCallback(); ...
}

class Generator<CB> {
   CB callback;
   ... callback.notify(data) ...
}

class Listener {
   public domain callbacks;
   domain state;
   link callbacks -> state;
   callbacks Callback getCallback() {
      return new ListenerCB<state>(...)
   }
}

interface Callback { void notify(int data); }
class ListenerCB<domain state> implements Callback {
   void notify(int data) { /* modify state */ }
}
```

Fig. 6. A Listener system.

store a reference to the callback object, so it is parameterized by the callback type `CB`.

Like the `Sequence` class described earlier, the `Listener` declares a private domain for its internal state and a public one for its callback objects, linking the `callback` domain to the `state` domain. The `ListenerCB` object implements the `Callback` interface, storing a reference to the listener's state and performing some action on that state when the `notify` method is invoked.

2.7 Expressing Architectural Constraints

One of our goals in designing ownership domains was to express aliasing constraints between different components in the architecture of a program. Figure 7 shows how the aliasing constraints in two different architectural styles can be expressed with ownership domains. The code in the first example represents a layered architecture [15] by creating an ownership domain for each layer. The link specifications express the constraint that objects in each layer can only refer to objects in the layer below.

```
class LayeredArchitecture {
   domain layer1, layer2, layer3;
   link layer2->layer1, layer3->layer2;
   ...
}

class MediatorArchitecture {
   domain component1, component2, component3;
   domain mediator;
   link component1->mediator, component2->mediator, component3->mediator;
   link mediator->component1, mediator->component2, mediator->component3;
}
```

Fig. 7. A layered architecture and a mediator architecture

The second example shows an architecture in which three different components communicate through a mediator component [26]. Again, the three components and the mediator are represented with domains. However, in this case, the aliasing pattern forms a star with the mediator in the center and the components as the points of the star. The link soundness property can then be used to guarantee that the individual components communicate only indirectly through the mediator. This property is crucial to gain the primary benefit of the mediator style: components in the system can be developed, deployed, and evolved independently from each other.

In both examples, the ability to create multiple ownership domains in one object and to specify aliasing constraints between them is crucial for specifying the architectural structure of the system. The use of ownership domains to specify architectural aliasing complements our earlier work specifying architectural interfaces and control flow in a different extension of Java [3].

2.8 Extensions

The AliasJava compiler includes an implementation of ownership domains as well as a number of useful extensions. A `unique` annotation indicates that there is only one persistent external reference to an object (we allow internal references to unique objects, providing *external uniqueness* [12] in the more flexible setting of ownership domains). Unique objects can later be assigned to a domain, at which point the type system verifies the object's linking assumptions that relate the owner domain to the parameter domains (in order to do this check soundly, we also verify that the domain parameters of a unique object are not "forgotten" by subsumption).

A `lent` annotation indicates a temporary reference to an object. A `unique` or `owned` object can be passed as a `lent` argument of a function, giving that function temporary access to the object but ensuring that the function does not store a persistent reference to the object. AliasJava supports method parameterization (and the corresponding `assumes` clauses) in addition to class parameterization. In the future, we may add support for package-level domains (generalizing confinement [7]) and for readonly types [21].

3 Formalizing Ownership Domains

We would like to use formal techniques to prove that our type system is safe and preserves the intended aliasing invariants. A standard technique, exemplified by Featherweight Java [18], is to formalize a core language that captures the key typing issues while ignoring complicating language details. We have formalized ownership domains as Featherweight Domain Java (FDJ), a core language based on Featherweight Java (FJ).

Featherweight Domain Java makes a number of simplifications relative to the full Java language. As in FJ, the model omits interfaces, inner classes, and some statement and expression forms, since these constructs can be written in terms of more fundamental ones. In order to focus exclusively on ownership domains, FDJ omits other constructs like `shared` that can be modeled with syntactic sugar. These omissions make the formal system simple enough to permit effective reasoning, while still capturing the core constructs of ownership domains.

Although Featherweight Java has been extended with type parameters [18], we model only domain parameters in order to simplify the system. The user-level system can be translated into the formal one by replacing each type parameter with a domain parameter, by replacing uses of the type parameter with the domain parameter applied to class `Object`, and by inserting casts where necessary.

3.1 Syntax

Figure 8 shows the syntax of FDJ. The metavariable 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 and domain names; n ranges over values and variable names; S ranges over stores; ℓ ranges over locations in the store, α and β range over formal ownership domain parameters, and m ranges over method names. As a shorthand, an overbar is used to represent a sequence.

In FDJ, classes are parameterized by a list of ownership domains, and extend another class that has a subsequence of its domain parameters. An `assumes` clause states the linking assumptions that a class makes about its domain parameters. Our formal system does not have the default linking assumptions that are present in the real system; thus all assumptions must be specified explicitly. Each class defines a constructor and sets of fields, domains, link specifications, and methods. The canonical class constructor just assigns the constructor arguments to the fields of the class, while methods use standard Java syntax. We assume a predefined class `Object` that has no fields, domains, or methods.

Source-level expressions e_s include object creation expressions, field reads, casts, and method calls. Although FDJ is a pure language without field assignment, we want to reason about aliasing, and so we use locations to represent

$$
\begin{aligned}
CL &::= \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{assumes}\ \overline{\gamma} \to \overline{\delta}\ \{\ \overline{T}\ \overline{f};\ K\ \overline{D}\ \overline{L}\ \overline{M}\ \}\\
K &::= C(\overline{T'}\ \overline{f'}, \overline{T}\ \overline{f})\{\texttt{super}(\overline{f'}); \texttt{this}.\overline{f} = \overline{f};\}\\
D &::= [\texttt{public}]\ \texttt{domain}\ x;\\
L &::= \texttt{link}\ d \to d';
\end{aligned}
$$

$$
\begin{aligned}
M &::= T_R\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}\\
e_s &::= x \mid \texttt{new}\ C\texttt{<}\overline{\alpha}\texttt{>}(\overline{e})\\
&\quad\mid\ e.f \mid (T)e \mid e.m(\overline{e})\\
e &::= e_s \mid \ell \mid \ell > e \mid \texttt{error}\\
n &::= x \mid v\\
T &::= C\texttt{<}\overline{d}\texttt{>} \mid \texttt{ERROR}\\
d &::= \alpha \mid n.x
\end{aligned}
$$

$$
\begin{aligned}
v, \ell &\in \mathit{locations}\\
S &::= \ell \mapsto C\texttt{<}\overline{\ell.x}\texttt{>}(\overline{v})\\
\Gamma &::= x \mapsto T\\
\Sigma &::= \ell \mapsto T
\end{aligned}
$$

Fig. 8. Featherweight Domain Java Syntax

object identity. 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. We will write $S[\ell]$ to denote the store entry for ℓ and $S[\ell, i]$ to denote the value in the ith field of $S[\ell]$. Adding an entry for location ℓ to the store is abbreviated $S[\ell \mapsto C\texttt{<}\overline{\ell.x}\texttt{>}(\overline{\ell'})]$.

Several method calls may be executing on the stack at once, and to reason about ownership we will need to know the receiver of each executing call. Therefore, there are additional expression forms e that can occur during reduction, including locations ℓ. The expression form $\ell > e$ represents a method body e executing with a receiver ℓ. An explicit **`error`** expression is used to represent the result of a failed cast.

The result of computation is a location ℓ, which is sometimes referred to as a value v. The class of names n includes both values and variables. The set of variables includes the distinguished variable **`this`** used to refer to the receiver of a method. A domain is either one of the domain parameters α of the class, or else a pair of a name n (which can be **`this`**) and a domain name x. Neither the **`error`** expression, nor locations, nor $\ell > e$ expressions may appear in the source text of the program; these forms are only generated during reduction.

A type in FDJ is a class name and a set of actual ownership domain parameters. We simplify the formal system slightly by treating the first domain parameter of a class as its owning domain. We use a slightly different syntax

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

$$\frac{S[\ell] = C\texttt{<}\overline{d}\texttt{>}(\overline{v}) \qquad fields(C\texttt{<}\overline{d}\texttt{>}) = \overline{T}\ \overline{f}}{S \vdash \ell.f_i \mapsto v_i, S} \ \textit{R-Read}$$

$$\frac{S[\ell] = C\texttt{<}\overline{d}\texttt{>}(\overline{v}) \qquad C\texttt{<}\overline{d}\texttt{>} <: T}{S \vdash (T)\ell \mapsto \ell, S} \ \textit{R-Cast}$$

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

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

$$\frac{}{S \vdash \ell > v \mapsto v, S} \ \textit{R-Context}$$

Fig. 9. Dynamic Semantics

in the practical system to emphasize the semantic difference between the owner domain of an object and its domain parameters.

We assume a 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.

Expressiveness. While FDJ has been simplified considerably from the full semantics of ownership domains in Java, it is still quite expressive. The example code in Figures 3-7 can be expressed in FDJ with some minor rewriting. For example, the FDJ `Cons` class below differs from the code in Figure 3 in that the `owner` parameter is explicit; the type parameter `T` is replaced with domain parameter `elemOwner`; and the `extends` clause is explicit. In addition, the owner domain of the field and method argument types is specified as the first parameter instead of appearing before the type name.

```
class Cons<owner, elemOwner> extends Object<owner>
          assumes owner -> elemOwner {
  Cons(Object<elemOwner> obj, Cons<owner,elemOwner> next) {
    this.obj=obj; this.next=next;
  }
  Object<elemOwner> obj;
  Cons<owner,elemOwner> next;
}
```

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha,\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ldots}{C\texttt{<}\overline{d,d'}\texttt{>}\ \ <:\ \ C'\texttt{<}\overline{d}\texttt{>}}\ \textit{Subtype-Class}$$

$$\frac{}{T <: T}\ \textit{Subtype-Reflex} \qquad \frac{T <: T' \qquad T' <: T''}{T <: T''}\ \textit{Subtype-Trans}$$

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

Fig. 10. Subtyping Rules

3.2 Reduction Rules

The evaluation relation, defined by the reduction rules given in Figure 9, is of the form $S \vdash e \mapsto e', S'$, read "In the context of store S, 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* rule looks up the receiver in the store and identifies the ith field using the $fields$ helper function (defined in Figure 14). The result is the value at field position i in the store. 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 14) to determine the correct method body to invoke. The method invocation is replaced with the appropriate method body, where all occurrences of the formal method parameters and **`this`** are replaced with the actual arguments and the receiver, 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. As this rule shows, expressions of the form $\ell > e$ do not affect program execution, and are used only for reasoning about invariants that are necessary for link soundness. The full definition of FDJ, in a companion technical report [2], also includes congruence rules that allow reduction to proceed within an expression in the the order of evaluation defined by Java. For example, the read rule states that an expression $e.f$ reduces to $e'.f$ whenever e reduces to e'.

$$\frac{}{\Gamma, \Sigma, n_{this} \vdash x : \Gamma(x)} \text{ T-Var} \qquad \frac{}{\Gamma, \Sigma, n_{this} \vdash \ell : \Sigma(\ell)} \text{ T-Loc}$$

$$\frac{\begin{array}{c} \Gamma, \Sigma, n_{this} \models assumptions(C\texttt{<}\overline{d}\texttt{>}) \qquad \Gamma, \Sigma, n_{this} \vdash \overline{e} : \overline{T'} \\ fields(C\texttt{<}\overline{d}\texttt{>}) = \overline{T}\ \overline{f} \qquad \overline{T'} <: \overline{T} \qquad \Gamma, \Sigma, n_{this} \vdash n_{this} : T_{this} \\ owner(C\texttt{<}\overline{d}\texttt{>}) \in (domains(T_{this}) \cup owner(T_{this})) \end{array}}{\Gamma, \Sigma, n_{this} \vdash \texttt{new}\ C\texttt{<}\overline{d}\texttt{>}(\overline{e}) : C\texttt{<}\overline{d}\texttt{>}} \text{ T-New}$$

$$\frac{}{\Gamma, \Sigma, n_{this} \vdash \texttt{error} : \texttt{ERROR}} \text{ T-Error}$$

$$\frac{\Gamma, \Sigma, n_{this} \vdash e_0 : T_0 \qquad fields(T_0) = \overline{T}\ \overline{f}}{\Gamma, \Sigma, n_{this} \vdash e_0.f_i : T_i} \text{ T-Read}$$

$$\frac{\Gamma, \Sigma, n_{this} \vdash e : T'}{\Gamma, \Sigma, n_{this} \vdash (T)\ e : T} \text{ T-Cast}$$

$$\frac{\begin{array}{c} \Gamma, \Sigma, n_{this} \vdash e_0 : T_0 \qquad \Gamma, \Sigma, n_{this} \vdash \overline{e} : \overline{T_a} \\ mtype(m, T_0) = \overline{T} \rightarrow T_R \qquad \overline{T_a} <: [\overline{e}/\overline{x}, e_0/\texttt{this}]\ \overline{T} \end{array}}{\Gamma, \Sigma, n_{this} \vdash e_0.m(\overline{e}) : [\overline{e}/\overline{x}, e_0/\texttt{this}]\ T_R} \text{ T-Invk}$$

$$\frac{\Gamma, \Sigma, \ell \vdash e : T}{\Gamma, \Sigma, n_{this} \vdash \ell > e : T} \text{ T-Context}$$

Fig. 11. Typechecking

3.3 Typing Rules

FDJ's subtyping rules are given in Figure 10. 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.

Typing judgments, shown in Figure 11, are of the form $\Gamma, \Sigma, n_{this} \vdash e : T$, read, "In the type environment Γ, store typing Σ, and receiver n_{this}, expression e has type T."

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 object creation rule verifies that any assumptions (see Figure 14) that the class being instantiated makes about its domain parameters are justified based on the current typing environment. The entailment relation $\models$ for linking assumptions will be defined below in Figure 13. The creation rule 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} (the *domains* function is defined in Figure 14, and the *owner* function gets the owner of n_{this} by extracting the first owner parameter from T).

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

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

$$\frac{\begin{array}{c}\{d_1, d_2\} \cap domains(C\texttt{<}\overline{\alpha}\texttt{>}) \neq \emptyset \\ d_1 \notin domains(C\texttt{<}\overline{\alpha}\texttt{>}) \Longrightarrow (\mathtt{this} : C\texttt{<}\overline{\alpha}\texttt{>}, \emptyset, \mathtt{this} \models d_1 \rightarrow owner(C\texttt{<}\overline{\alpha}\texttt{>})) \\ d_2 \notin domains(C\texttt{<}\overline{\alpha}\texttt{>}) \Longrightarrow (\mathtt{this} : C\texttt{<}\overline{\alpha}\texttt{>}, \emptyset, \mathtt{this} \models \mathtt{this} \rightarrow d_2)\end{array}}{\mathtt{link}\ d_1 \rightarrow d_2\ OK\ in\ C\texttt{<}\overline{\alpha}\texttt{>}}\ \textit{LinkOK}$$

$$\frac{\forall \ell \in domain(\Sigma)\ \emptyset, \Sigma, \ell \models assumptions(\Sigma[\ell]))}{\Sigma\ OK}\ \textit{T-Assumptions}$$

$$\frac{\begin{array}{c}domain(S) = domain(\Sigma) \qquad S[\ell] = C\texttt{<}\overline{\ell'.x}\texttt{>}(\overline{v}) \iff \Sigma[\ell] = C\texttt{<}\overline{\ell'.x}\texttt{>} \\ (S[\ell, i] = \ell'') \wedge (fields(\Sigma[\ell]) = \overline{T}\ \overline{f}) \Longrightarrow (\Sigma[\ell''] <: T_i) \qquad \Sigma\ OK \\ (S[\ell, i] = \ell'') \Longrightarrow (\emptyset, \Sigma, \ell \models \ell \rightarrow owner(\Sigma[\ell'']))\end{array}}{\Sigma \vdash S}\ \textit{T-Store}$$

Fig. 12. Class, Method and Store Typing

The typing rule for **error** assigns it the type `ERROR`. The rule for field reads looks up the declared type of the field using the *fields* function defined in Figure 14. The cast rule simply checks that the expression being 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. Our cast rule is simpler than Featherweight Java's in that we omit the check for "stupid casts."

Rule *T-Invk* looks up the invoked method's type using the *mtype* function defined in Figure 14, and verifies that the actual argument types are subtypes of the method's argument types. The method's nominal argument and result types must have actual parameter values substituted for formals, so that domain names that are qualified by a formal parameter are compared properly in the calling context. Finally, the *T-Context* typing rule for an executing method checks the method's body in the context of the new receiver ℓ.

Finally, for each rule of the form $\Gamma, \Sigma, n_{this} \vdash e : T$ we include an implicit check that $T \neq \mathtt{ERROR} \Longrightarrow \Gamma, \Sigma, n_{this} \models n_{this} \rightarrow owner(T)$. This implicit check verifies that the current object named by n_{this} has permission to access the owning domain of the expression.

$$\frac{(d_1 \rightarrow d_2) \in links(\Sigma[\ell])}{\Gamma, \Sigma, n_{this} \models (d_1 \rightarrow d_2)} \; \textit{T-DynamicLink}$$

$$\frac{\Gamma, \Sigma, n_{this} \vdash n_{this} : T \qquad (d_1 \rightarrow d_2) \in linkdecls(T)}{\Gamma, \Sigma, n_{this} \models (d_1 \rightarrow d_2)} \; \textit{T-DeclaredLink}$$

$$\frac{}{\Gamma, \Sigma, n_{this} \models (n \rightarrow n.x)} \; \textit{T-ChildRef} \qquad \frac{}{\Gamma, \Sigma, n_{this} \models (d \rightarrow d)} \; \textit{T-SelfLink}$$

$$\frac{\Gamma, \Sigma, n_{this} \vdash n : T \qquad \Gamma, \Sigma, n_{this} \models (owner(T) \rightarrow d)}{\Gamma, \Sigma, n_{this} \models (n \rightarrow d)} \; \textit{T-LinkRef}$$

$$\frac{\Gamma, \Sigma, n_{this} \vdash n : T \qquad \Gamma, \Sigma, n_{this} \models (d \rightarrow owner(T)) \qquad public(x)}{\Gamma, \Sigma, n_{this} \models (d \rightarrow n.x)} \; \textit{T-PublicLink}$$

$$\frac{\Gamma, \Sigma, n_{this} \vdash n : T \qquad \Gamma, \Sigma, n_{this} \models (n_s \rightarrow owner(T)) \qquad public(x)}{\Gamma, \Sigma, n_{this} \models (n_s \rightarrow n.x)} \; \textit{T-PublicRef}$$

Fig. 13. Link Permission Rules

Figure 12 shows the rules for typing classes, declarations within classes, and the store. 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 the "this" references is allowed to access the domains of the fields in the class.

The rule for methods checks that the method body is well typed, and uses the *override* function (defined in Figure 14) to verify that methods are overridden with a method of the same type. It also verifies that the "this" reference has permission to access the domains of the arguments of the method.

The link 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.

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 in a field in the store, the type of the value must be a subtype of the declared type of the field. The check $\Sigma\ OK$, defined by the *T-Assumptions* rule, verifies that all the linking assumptions made for each object in the store are justified based on actual link declarations in the source code. Finally, the last check verifies link soundness for the store: if object ℓ refers to object ℓ'' in it's ith field, then the link declarations implied by the store type Σ imply that ℓ has permission to access the domain of ℓ''.

Figure 13 shows the rules for determining whether an object named by n or a domain d has permission to access another domain d'. These rules come in two forms: $\Gamma, \Sigma, n_{this} \models (n \rightarrow d)$ and $\Gamma, \Sigma, n_{this} \models (d \rightarrow d')$. 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 d." The second form is similar, except that the conclusion is that any object in domain d has permission to access domain d'. 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 $n.x$. 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.

Figure 14 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.

3.4 Properties

In this section, we state type soundness and link soundness for Featherweight Domain Java. The full proofs are straightforward but tedious, and so we relegate them to a companion technical report [2].

Theorem 1 (Type Preservation).
If $\emptyset, \Sigma, n_{this} \vdash e : T$, $\Sigma \vdash S$, and $S \vdash e \mapsto e', S'$, then there exists $\Sigma' \supseteq \Sigma$ and $T' <: T$ such that $\emptyset, \Sigma', n_{this} \vdash e' : T'$ and $\Sigma' \vdash S'$.

Proof. By induction over the derivation of $S \vdash e \mapsto e', S'$.

Theorem 2 (Progress).
If $\emptyset, \Sigma, n_{this} \vdash e : T$ and $\Sigma \vdash S$ then either e is a value or e has an `error` *subexpression or $S \vdash e \mapsto e', S'$.*

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

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \ldots\ \{\ \overline{T}\ \overline{f};\ \overline{D};\ \overline{L};\ \overline{M};\ \} \qquad (\texttt{public domain}\ x) \in \overline{D}}{public(x)}\ \textit{Aux-Public}$$

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \ldots\ \{\ \overline{T}\ \overline{f};\ \overline{D};\ \overline{L};\ \overline{M};\ \} \qquad \overline{D} = \overline{\texttt{public}_{opt}}\ \overline{\texttt{domain}}\ \overline{x} \qquad domains(C'\texttt{<}\overline{d}\texttt{>}) = \overline{d'}}{domains(C\texttt{<}\overline{d},\overline{d'}\texttt{>}) = \overline{this.x}, \overline{d'}}\ \textit{Aux-Domains}$$

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \ldots\ \{\ \overline{T}\ \overline{f};\ \overline{D};\ \overline{L};\ \overline{M};\ \} \qquad \overline{L} = \overline{\texttt{link}}\ \overline{d_c} \rightarrow \overline{d_c'} \qquad links(C'\texttt{<}\overline{d}\texttt{>}) = \overline{d_s} \rightarrow \overline{d_s'}}{links(C\texttt{<}\overline{d},\overline{d'}\texttt{>}) = ([\overline{d}/\overline{\alpha}, \overline{d'}/\overline{\beta}]\ (\overline{d_c} \rightarrow \overline{d_c'})),\ \overline{d_s} \rightarrow \overline{d_s'}}\ \textit{Aux-Links}$$

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{assumes}\ \overline{\gamma} \rightarrow \overline{\delta} \ldots \qquad assumptions(C'\texttt{<}\overline{d}\texttt{>}) = \overline{d_s} \rightarrow \overline{d_s'}}{assumptions(C\texttt{<}\overline{d},\overline{d'}\texttt{>}) = ([\overline{d}/\overline{\alpha}, \overline{d'}/\overline{\beta}]\ (\overline{\gamma} \rightarrow \overline{\delta})), \overline{d_s} \rightarrow \overline{d_s'}}\ \textit{Aux-Assume}$$

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \ldots\ \{\ \overline{T}\ \overline{f};\ \overline{D};\ \overline{L};\ \overline{M};\ \} \qquad fields(C'\texttt{<}\overline{d}\texttt{>}) = \overline{T'}\ \overline{f'}}{fields(C\texttt{<}\overline{d},\overline{d'}\texttt{>}) = ([\overline{d}/\overline{\alpha}, \overline{d'}/\overline{\beta}]\ \overline{T}\ \overline{f}), \overline{T'}\ \overline{f'}}\ \textit{Aux-Fields}$$

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

$$\frac{}{owner(C\texttt{<}\overline{d}\texttt{>}) = d_1}\ \textit{Aux-Owner}$$

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}\ \ldots\ \{\ \overline{T}\ \overline{f};\ \overline{D};\ \overline{L};\ \overline{M};\ \} \qquad (T_R\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}) \in \overline{M}}{mtype(m, C\texttt{<}\overline{d}\texttt{>}) = [\overline{d}/\overline{\alpha}]\ \overline{T} \rightarrow T_R}\ \textit{Aux-MType1}$$

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \ldots\ \{\ \overline{T}\ \overline{f};\ \overline{D};\ \overline{L};\ \overline{M};\ \} \qquad m\ is\ not\ defined\ in\ \overline{M}}{mtype(m, C\texttt{<}\overline{d},\overline{d'}\texttt{>}) = mtype(m, C'\texttt{<}\overline{d}\texttt{>})}\ \textit{Aux-MType2}$$

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}\ \ldots\ \{\ \overline{T'}\ \overline{f};\ \overline{D};\ \overline{L};\ \overline{M};\ \} \qquad (T_R\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}) \in \overline{M}}{mbody(m, C\texttt{<}\overline{d}\texttt{>}) = [\overline{d}/\overline{\alpha}]\ (\overline{x},\ e)}\ \textit{Aux-MBody1}$$

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \ldots\ \{\ \overline{T}\ \overline{f};\ \overline{D};\ \overline{L};\ \overline{M};\ \} \qquad m\ is\ not\ defined\ in\ \overline{M}}{mbody(m, C\texttt{<}\overline{d},\overline{d'}\texttt{>}) = mbody(m, C'\texttt{<}\overline{d}\texttt{>})}\ \textit{Aux-MBody2}$$

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

Fig. 14. Auxiliary Definitions

Together, Type Preservation and Progress imply that the type system for FDJ is sound. We also wish to 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 3 (Heap Link Soundness).
If $\Sigma \vdash S$ *and* $S[\ell, i] = \ell''$ *then* $\emptyset, \Sigma, \ell \models \ell \rightarrow owner(\Sigma[\ell''])$.

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

Proof. As stated earlier, this condition is implicitly enforced by each typing rule of the form $\emptyset, \Sigma, n_{this} \vdash e : T$.

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.

4 Related Work

Ownership type systems. A number of early research projects, including Islands [17] and Balloons [5], provided a way to encapsulate one object within another. The term "ownership" is due to the Flexible Alias Protection project [23, 13], which added ownership parameters in order to support object-oriented idioms like collection classes. These early systems all enforced the owners-as-dominators property (or even more restrictive properties).

A number of researchers have proposed solutions to the long-recognized problem of expressing iterators in ownership type systems. One solution is to allow dynamic aliases to internal ownership domains [11], breaking the owners-as-dominators property for variables on the stack. Since iterators are generally used only on the stack, this solution is sufficient for most uses of iterators. However, it has two drawbacks: *any* external object–not only trusted iterators–can access objects in private domains. In addition, this solution does not support idioms like event callback objects, which are generally used in a way that requires references to callback objects on the heap.

A more expressive, but somewhat ad-hoc solution was proposed by Clarke [10] and later adopted by Boyapati et al. [8]. This solution allows inner classes to violate the owners-as-dominators property, while enforcing it for all regular classes. This technique supports both iterators and event callbacks, but places some restrictions on implementors, because all iterators and callbacks must be implemented as inner classes (as they often, but not always, are in practice).

Our own previous work, AliasJava, uses a capability-based encapsulation model instead of owners-as-dominators [4]. In this model, the domain parameters of an object are capabilities allowing the object to access the objects in that domain. Thus, developers can reason about access permission by examining the parameterization of objects. Although this solution is more flexible than either of the solutions described above, reasoning about capabilities is not as straightforward as reasoning about object containment.

More recently, Potanin et al. propose a way to provide capability-based encapsulation with no changes to Java's syntax, instead enforcing a stylized use of Java's generics [24]. We build on their ideas (as well as those of Noble et al. [23]) to integrate genericity with ownership, but we introduce some new syntax in order to support stronger and more flexible alias-control policies.

Ownership domains, as presented in this paper, represent the first solution that supports flexible implementations of iterator and event idioms while also preserving clear reasoning about inter-domain aliasing. In addition, the ability to define multiple ownership domains per object and specify a fine-grained policy controlling inter-domain aliasing allows ownership domains to express architectural constraints that cannot be described in previous systems.

Several systems build on the owners-as-dominators property to provide secondary properties including safe concurrency [8], safe memory management [9], reasoning about effects [11], and abstraction [6]. Since ownership domains can be used to enforce owners-as-dominators, our system can support similar kinds of reasoning in a more flexible setting.

Clarke's thesis presents an object calculus that allows multiple *ownership contexts* to be defined for each object, similar to our ownership domains [10]. We build on this work with a concrete language design and increase expressiveness by by specifying aliasing policy separately from containment.

Other related work. Our previous work on ArchJava allows developers to document architectural designs similar to those described in Figure 7 [3]. The original ArchJava system provided a more detailed description of component interactions than ownership domains do, but did not constrain aliasing between components. The first author's dissertation demonstrates adding ownership domains to ArchJava in order to reason about data sharing between components as well [1]. Lam and Rinard express design information using tokens that are somewhat similar to ownership domains, but their system does not support hierarchical designs or important object-oriented constructs like inheritance [19].

Confined types [7] restrict aliases of an object to within a particular package, a weaker but more lightweight notion compared to the object-based encapsulation provided by ownership domains. The Universes system provides both object-based encapsulation and package-based encapsulation [21]. Systems like alias types [27] and separation logic [25] provide a finer control of aliasing compared to ownership domains, but are also much more heavyweight, requiring many more declarations to gain the same level of reasoning about aliasing.

Leino et al.'s data groups [20] and Greenhouse et al.'s regions [16] are similar to ownership domains. Here groups and regions refer to sets of fields rather than sets of objects, and are used to reason about effects rather than aliasing.

5 Conclusion and Future Work

This paper generalizes previous work on ownership type systems to support ownership domains. By separating alias-control policy from the ownership mechanism, we gain two primary benefits. First, programmers can express more flexible aliasing policies that naturally support common object-oriented idioms such as iterators and events. Second, programmers can specify high-level design information by declaring multiple ownership domains per object and specifying the aliasing relationship among these domains. Thus, ownership domains are both more flexible and more precise than previous ownership-based encapsulation mechanisms. In the future, we intend to perform case studies that will provide insights into the usability and benefits of ownership domains.

6 Acknowledgements

We would like to thank Donna Malayeri, John Boyland, Neel Krishnaswami, Aaron Greenhouse, members of the Cecil group, and the anonymous reviewers for their comments and suggestions. This work was supported in part by NSF grants CCR-9970986, CCR-0073379, and CCR-0204047, and gifts from Sun Microsystems and IBM.

References

1. J. Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, August 2003.
2. J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. Carnegie Mellon Technical Report CMU-ISRI-04-110, available at http://www.cs.cmu.edu/~aldrich/papers/, April 2004.
3. J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning with ArchJava. In *European Conference on Object-Oriented Programming*, June 2002.
4. J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *Object-Oriented Programming Systems, Languages, and Applications*, November 2002.
5. P. S. Almeida. Balloon Types: Controlling Sharing of State in Data Types. In *European Conference on Object-Oriented Programming*, June 1997.
6. A. Banerjee and D. A. Naumann. Representation Independence, Confinement, and Access Control. In *Principles of Programming Languages*, January 2002.
7. B. Bokowski and J. Vitek. Confined Types. In *Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

8. C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Object-Oriented Programming Systems, Languages, and Applications*, November 2002.
9. C. Boyapati, A. Salcianu, J. William Beebee, and M. Rinard. Ownership Types for Safe Region-Based Memory Mangement in Real-Time Java. In *Programming Language Design and Implementation*, June 2003.
10. D. Clarke. *Object Ownership & Containment*. PhD thesis, University of New South Wales, July 2001.
11. D. Clarke and S. Drossopoulou. Ownership, Encapsulation, and the Disjointness of Type and Effect. In *Object-Oriented Programming Systems, Languages, and Applications*, November 2002.
12. D. Clarke and T. Wrigstad. External Uniqueness is Unique Enough. In *European Conference on Object-Oriented Programming*, July 2003.
13. D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming Systems, Languages, and Applications*, October 1998.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
15. D. Garlan and M. Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering, I*, 1993.
16. A. Greenhouse and J. Boyland. An Object-Oriented Effects System. In *European Conference on Object-Oriented Programming*, June 1999.
17. J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *Object-Oriented Programming Systems, Languages, and Applications*, October 1991.
18. A. Igarashi, B. Pierce, and P. Wadler. Featherwieght Java: a Minimal Core Calculus for Java and GJ. In *Object-Oriented Programming Systems, Languages, and Applications*, November 1999.
19. P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *European Conference on Object-Oriented Programming*, July 2003.
20. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using Data Groups to Specify and Check Side Effects. In *Programming Language Design and Implementation*, June 2002.
21. P. Muller and A. Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, 1999.
22. J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke. Towards a Model of Encapsulation. In *Intercontinental Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, July 2003.
23. J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *European Conference on Object-Oriented Programming*, 1998.
24. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Featherweight Generic Confinement. In *Foundations of Object-Oriented Languages*, January 2004.
25. J. C. Reynolds. Separation Logic: a Logic for Shared Mutable Data Structures. In *Logic in Computer Science*, July 2002.
26. K. Sullivan and D. Notkin. Reconciling Environment Integration and Software Evolution. *Transactions on Software Engineering and Methodology*, 1(3), July 1992.
27. D. Walker and G. Morrisett. Alias Types for Recursive Data Structures. In *International Workshop on Types in Compilation*, September 2000.