Objects and Aspects: Ownership Types

Neel Krishnaswami

Department of Computer Science
Carnegie Mellon University
`neelk@cs.cmu.edu`

Overview

- The Problem
- An Introduction to Ownership Types
- Evaluating How Ownership Resolves the Problem
- Future Directions

The Problem

- A central idea of OO is to encapsulate state
- But there is no strong language support for this

Aliasing: Threat or Menace?

This is an example from the Java 1.1 JDK:

```
class Class {
  List signers;

  List getSigners() {
    return this.signers;
  }
}
```

Aliasing: Threat or Menace?

This is an example from the Java 1.1 JDK:

```
class Class {
  List signers;

  List getSigners() {
    return this.signers; // clients can mutate signers field!
  }
}
```

Aliasing: Threat or Menace?

```
class JavaClass {
  List signers;

  List getSigners() {
    return this.signers; // clients can mutate signers field!
  }
}
```

Aliasing has caused a failure of encapsulation – the ability to modify an internal field of an object got exposed to a client, because the client received a reference to the object in the instance variable.

An Introduction to Ownership Types

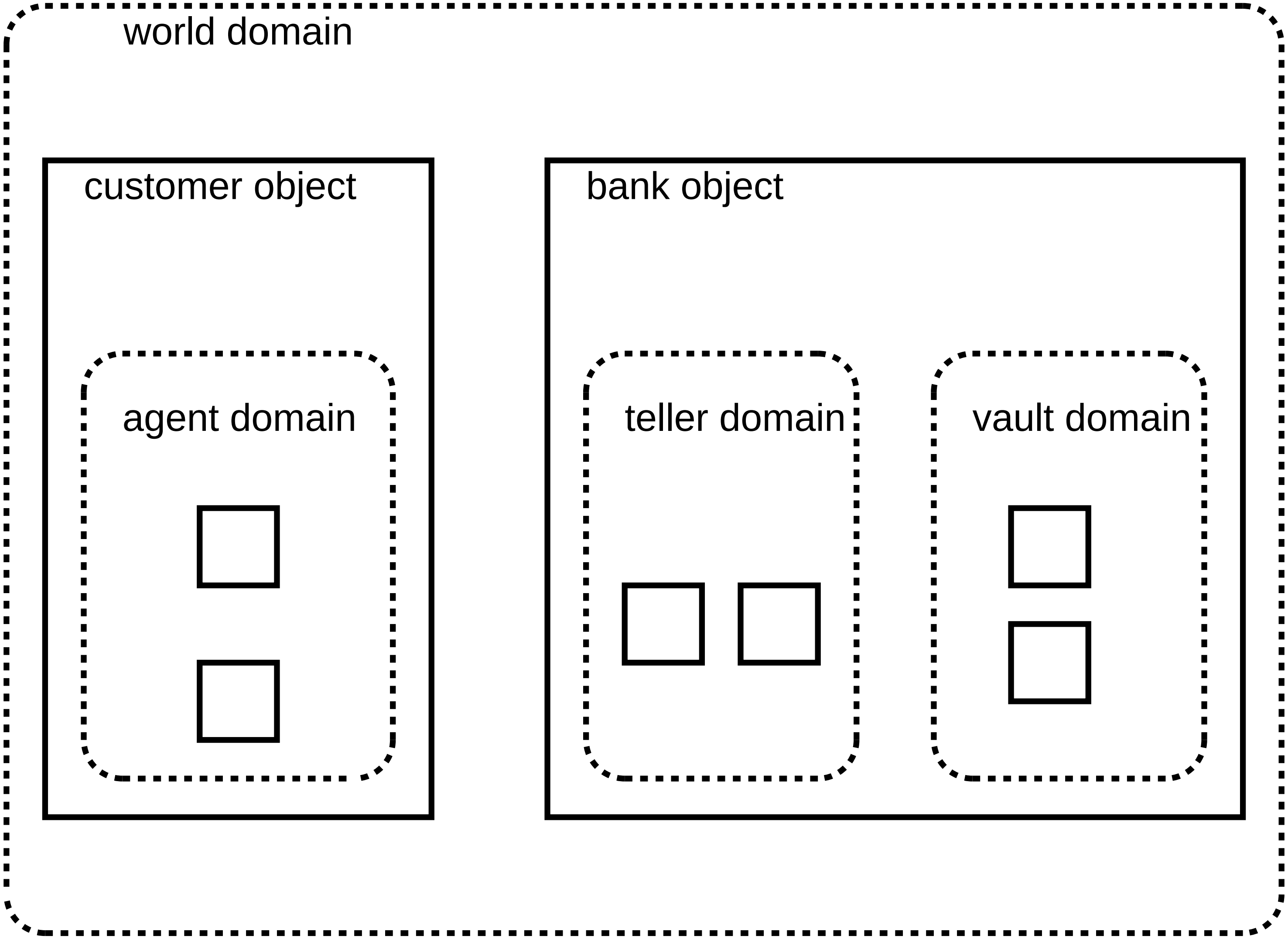
- The Problem
- An Introduction to Ownership Types
- Evaluating How Ownership Resolves the Problem
- Future Directions

The Basic Idea Underlying Ownership

Ownership types represent an attempt to prevent aliasing-based failures of encapsulation.

- Every object itself exists in a *domain*, which is a region of the heap.
- Every object can additionaly create one or more new domains.
- Each field of an object is annotated with the domain it belongs to.

A Graphical View of Ownership



Access Permissions

In order for domains to be useful, we need to define a set of *access permissions* on domains. To *"Access"* a domain d means to:

- Dereference an object field annotated with domain d
- Invoke a method on an object in d
- Receive a value from a method call that is in a domain d.

What May Be Accessed?

An object o in a domain d can access:

- Other objects in the same domain d.
- Other objects in the domains that d is contained in.
- Objects in the domains e, f, g that it declares.
- Objects in domains d' that d has permission to access.

Very important: this is not a transitive relation! If $d \rightarrow e$ and $e \rightarrow f$, then it does *not* follow that $d \rightarrow f$.

Public Domains and Link Annotations

- Objects in domains d' that d has permission to access.

This information comes from *programmer annotations*.

A programmer can mark a declared domain public, in which case that domain may be accessed from any domain that can access the declaring object.

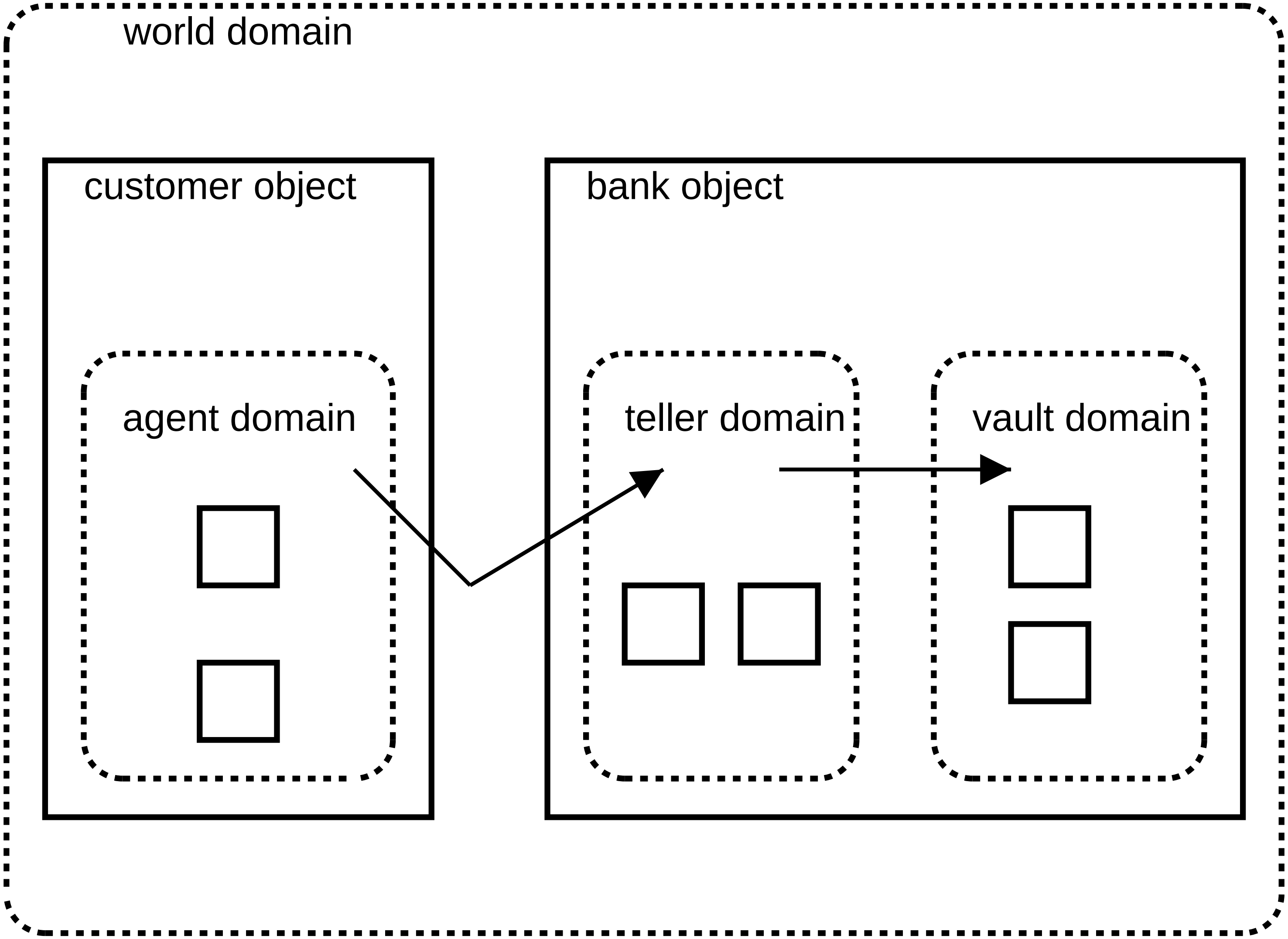
A programmer can declare link specifications, which permit an object to declare access links between the domains it created and domains it can access.

A Code Example

```
class Customer {
  domain agents;
}

class Bank {
  public domain tellers;
  private domain vault;
  link tellers -> vault;
}
```

A Graphical View of Ownership



Link Soundness

This ownership system has a link soundness property. This is a proof that the type system actually enforces the access constraints – that is, if o can access o' and o' is in domain d, then o has permission to access d.

An Introduction to Ownership Types

- The Problem
- An Introduction to Ownership Types
- Evaluating How Ownership Resolves the Problem
- Future Directions

JDK 1.1, revisited

```
class Class {
  private domain internal;
  internal List signers;

  internal List getSigners() { return this.signers; }

  void foo() {
    internal List x = this.getSigners();
    // do stuff using x
  }
}
```

Clients cannot invoke `getSigners`, since the domain `internal` is private and they cannot access it. They can only invoke `foo`.

Making getSigners Available

```
class Class {
  private domain internal;
  internal List signers;

  world List getSigners() {
    world List copy = new List();
    for(int i = 0; i < this.signers.size(); i++) {
      copy.add(this.signers.get(i));
    }
    return copy;
  }
}
```

Generalizing To Iterators, 0/3

Now we will look at a more complex problem – iterator objects. An iterator is an object with access to the internal state of the collection it iterates over, but which does not expose this to the outside world.

Iterators, cont. 1/3

```
class Cons<T> assumes owner -> T.owner {
  Cons(T head, owner Cons<T> tail) {
    this.head = head;
    this.tail = tail;
  }

  T head;
  owner Cons<T> tail;
}
```

`owner` is a keyword to name the owning domain of an object.

Iterators, cont. 2/3

```
class Sequence<T> assumes owner -> T.owner {
  private domain internal;
  link internal -> T.owner;
  internal Cons<T> front;

  void add(T o) { this.front = new Cons<T>(o, this.front); }

  public domain iters;
  link iters -> T.owner,
       iters -> internal;

  iters Iterator<T> getIter() {
    return new SequenceIterator<T, owned>(this.front);
  }
}
```

Iterators, cont. 2/3

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

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

  boolean hasNext() { return current != null; }

  T next() {
    T obj = this.current.head;
    this.current = this.current.tail;
    return obj;
  }
}
```

What Makes This Work

- You can parameterize classes with domains as well as types. *Programmers can write code that works in any domain.*

- Public domains can safely access private ones, because of the lack of transitivity. *Stateful data can now be part of an object's interface without breaking its encapsulation.*

- You can hide "extra" parameterization behind interfaces. *This lets the iterator implementation receive a domain without revealing it to clients.*

An Introduction to Ownership Types

- The Problem
- An Introduction to Ownership Types
- Evaluating How Ownership Resolves the Problem
- Future Directions

Weaknesses With Ownership

- Ownership transfers. How can objects move between domains as the program evolves? (Uniqueness/linearity helps somewhat, but is overkill.)

- Serialization. (This is probably hopeless in the general case.)

- Theoretical complexity – the type system is quite complex, and we've "baked in" a fairly complex set of access rules. It would be nice to simplify this.

Future Work

- Transplant to a mostly-functional setting.
- Characterize what encapsulation really means via studying type abstraction for stateful languages.
- More access modes? Object creation, object update, and object read are quite different conceptually.
- What is the relation to other work? Regions, confinement types, modal logic, etc.