Outline

- Object Encapsulation
- Ownership Types
- Upgrades in Persistent Object Stores
Modular Reasoning

- Goal is local reasoning about correctness
  - Prove a class meets its specification, using only specifications but not code of other classes

- Crucial when dealing with large programs

- Requires no interference from code outside the class
  - Objects must be encapsulated
Consider a Set object $s$ implemented using a Vector object $v$

- Local reasoning about $s$ is possible
  - If objects outside $s$ do not access $v$
  - That is, if $v$ is encapsulated within $s$
Encapsulation

- In general, all objects that $s$ depends on must be encapsulated within $s$.

- $s$ depends on $x$ if mutations of $x$ affect behavior of $s$.
  
  - Leino, Nelson (SCR ’00)
  
  - Detlefs, Leino, Nelson (SRC ’98)
Examples

- Rep invariant for Set: no-dups
  - Then, size of vector is size of set
  - Then, remove stops at match

- Clearly, v must be inside s
Examples

- What does no-dups mean?
  - ! e1.equals(e2), for any elements e1 & e2

- So set does not depend on elements if elements are immutable
Iterators and Encapsulation

- Iterators require access to representation
- Okay if violations of encapsulation limited to the same module
Ownership Types

- Goal is to enforce encapsulation statically
  - Programmer can declare s owns v
  - System ensures v is encapsulated in s
Ownership Types

- Every object has an owner
- Owner can be another object or world
- Ownership relation forms a tree
- Owner of an object cannot change
Ownership Types for Encapsulation

- If an object owns objects it depends on
- Then type system enforces encapsulation
  - If v is inside s and o is outside
  - Then o cannot access v
Ownership Types

world

[Diagram with multiple interconnected nodes and arrows]
class TStack {
    TNode head;

    void push(T value) {...}
    T pop() {...}
}

class TNode {
    TNode next;
    T value;
    ...
}

class T {...}
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
}
class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
}
class T<TOwner> {...}
```cpp
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
}
class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
}
class T<TOwner> {...}
```

First owner owns the “this” object
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
}

class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
}

class T<TOwner> {...}

TStack owns the “head” TNode
```cpp
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
}
class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
}
class T<TOwner> {...}
```

The "next" TNode has the same owner as the "this" TNode
All TNodes have the same owner
```cpp
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
}

class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
}

class Client/clientOwner> {
    TStack<this, this> s1;
    TStack<this, world> s2;
    TStack<world, world> s3;
}
```

s1 is an encapsulated stack with encapsulated elements
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
}
class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
}
class Client<clientOwner> {
    TStack<this, this> s1;
    TStack<this, world> s2;
    TStack<world, world> s3;
}

s2 is an encapsulated stack with public elements
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
}

class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
}

class Client<clientOwner> {
    TStack<this, this> s1;
    TStack<this, world> s2;
    TStack<world, world> s3;
}

s3 is a public stack with public elements
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
}
class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
}
class Client/clientOwner> {
    TStack<this, this> s1;
    TStack<this, world> s2;
    TStack<world, world> s3;
    TStack<world, this> s4; // illegal
}

Other owners must be same as or more public than first owner [CD02]
This constraint is necessary to enforce encapsulation with subtyping
Constraints on Owners

```java
class Client<cOwner, sOwner, tOwner> where (sOwner <= tOwner) {
    ...
    TStack<sOwner, tOwner> head;
}
```

This is legal only if tOwner is same as or more public than sOwner

Programmers can constrain owners using where clauses
Iterators

- Consider an Iterator i over Stack s
- If i is encapsulated within s
  - Then i cannot be used outside s
- If i is not encapsulated within s
  - Then i cannot access representation of s
Solution

- Use inner classes
- Gives desired access to representation
- Yet, satisfies our modularity goal
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ....
    class TStackEnum<enumOwner, TOwner> implements TEnum<enumOwner, TOwner> {
        TNode<TStack.this, TOwner> current = TStack.this.head;
        ....
    }
}

Iterators

Inner class objects can access rep of outer class objects
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ....
    class TStackEnum<enumOwner, TOwner> implements TEnum<enumOwner, TOwner> {...}
}

TStackEnum<enumOwner, TOwner> elements<enumOwner>() {
    where (enumOwner <= TOwner) {...}
}
Ownership Types for Encapsulation

- If an object owns objects it depends on
- Then type system enforces encapsulation
  - If v is inside s and o is outside
  - Then o cannot access v
  - Unless o is inner class object of s
Effects Clauses

class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
    T<TOwner> pop() writes (this) {
        if (head == null) return null;
        T<TOwner> value = head.value();
        head = head.next();
        return value;
    }
}

Methods can specify read and write effects
reads(x) means method can read x and its encapsulated objects
writes(x) means method can read/write x and its encapsulated objects
Related Work

- Types augmented with owners
  - Clarke, Potter, Noble (OOPSLA ’98)

- Support for subtyping
  - Clarke, Drossopoulou (OOPSLA ’02)

- Owners combined with effects clauses
  - Boyapati, Rinard (OOPSLA ’01)
Summary

- Ownership types capture dependencies
- Extension for inner class objects allows iterators and wrappers
- Approach provides expressive power, yet ensures modular reasoning
- Effects clauses enhance modular reasoning
Applications

- Safe upgrades in persistent object stores
- Preventing data races and deadlocks
  - Boyapati, Lee, Rinard (OOPSLA ’01) (OOPSLA ’02)
- Safe region-based memory management
  - Boyapati, Salcianu, Beebee, Rinard (MIT ’02)
- Program understanding
  - Aldrich, Kostadinov, Chambers (OOPSLA ’02)
Upgrades in Persistent Object Stores

- Persistent Object Stores store objects
Upgrades in Persistent Object Stores

- Objects are accessed within transactions
  - Transactions support modular reasoning in spite of concurrency and failures
Uses of Upgrades

- Upgrades are needed to
  - Correct errors
  - Improve performance
  - Meet changing requirements

- Upgrades can be
  - Compatible or incompatible
  - Upgrades must be **complete**

- Encapsulation enables safe upgrades
Defining an Upgrade

- An upgrade is a set of class-upgrades
  - Upgrades must be complete

- A class upgrade is
  \[
  \langle \text{old-class}, \text{new-class}, \text{TF} \rangle
  \]

- TF: old-class \(\not\equiv\) new-class
  - TF changes representation of objects
  - System preserves identity of objects
Executing an Upgrade

- Requires: transforming all old-class objects

- Goal: Don’t interfere with applications
  - Don’t stop the world

- Goal: Be efficient in space and time
  - Don’t copy the database
Solution: Lazy, Just in Time

- Applications continue to run
- Objects are transformed just before first access
- Upgrades can run in parallel
Desired Semantics

- Upgrades appear to run when installed
  - Serialized before all later application transactions

- Upgrades appear to run in upgrade order

- Within an upgrade, transforms run as if each were the first to run
Related Work

- **PJama**: Atkinson, Dmitriev, Hamilton (POS ’00)
- **Orion**: Banerjee, Kim, Kim, Korth (Sigmod ’87)
- **O2**: Deux et al (IEEE TKDE ’90)
- **OTGen**: Lerner, Habermann (OOPSLA ’90)
- **Gemstone**: Penney, Stein (OOPSLA ’87)
How System Works

- Objects are transformed just before first access
  - Interrupt the application
  - Run earliest pending transform
  - Transform runs in its own transaction
- Application continues after transform commits
- Transforms can be interrupted too
Example

…;U1;A1;TF1(x);A2;…

U1 is installed
A1 commits
A2 accesses x and is interrupted
TF1(x) commits
A2 commits
Example

...;U1;A1;TF1(x);A2;...

U1 is installed
A1 commits
A2 accesses x and is interrupted
TF1(x) commits
A2 commits

Suppose A1 modifies z and TF1(x) uses z
Example

...;U1;A1;TF1(x);A2;TF1(y);A3; ...

U1 is installed
A1 commits
A2 accesses x and is interrupted
TF1(x) commits
A2 commits
A3 accesses y and is interrupted
TF1(y) commits
A3 commits
Example

...;U1;A1;TF1(x);A2;TF1(y);A3; ...

U1 is installed
A1 commits
A2 accesses x and is interrupted
TF1(x) commits
A2 commits
A3 accesses y and is interrupted
TF1(y) commits
A3 commits

Suppose TF1(y) uses x
Insuring Correct Behavior

- S1: TF(x) only accesses objects x owns
  - Statically enforced by type system
Insuring Correct Behavior

- **S1**: TF(x) only accesses objects x owns
  - Statically enforced by type system

- **S2**: x is transformed before objects x owns
  - No access to owned objects from outside
  - Shared access to owned objects from inner class objects (e.g., iterator)
Insuring Correct Behavior

- S1: TF(x) only accesses objects x owns
- S2: x is transformed before objects x owns
- Plus basic lazy scheme

- Applications don’t interfere with transforms
- Transforms of unrelated objects don’t interfere
- Transforms of related objects run in proper order (owner before owned)
Modular Reasoning

- S1: TF(x) only accesses objects x owns
- S2: x is transformed before objects x owns
- Plus basic lazy scheme

- Ensures modular reasoning: can reason about TF(x) as an extra method of x’s class
Conclusions

- Modular reasoning is key
- Ownership types support modular reasoning
- Software upgrades benefit too
Ownership Types for Object Encapsulation

Barbara Liskov
Chandrasekhar Boyapati
Liuba Shrira

Laboratory for Computer Science
Massachusetts Institute of Technology
{liskov, chandra, liuba}@lcs.mit.edu
Example of Local Reasoning

class IntVector {
    int size() reads (this) {...} ... 
}

class IntStack {
    IntVector<this> vec;
    void push(int x) writes (this) {...} ... 
}

void m (IntStack s, IntVector v) writes (s) reads (v)
    where !(v <= s) !(s <= v) {
        int n = v.size();  s.push(3);  assert( n == v.size() );
    }

Is the condition in the assert true?
Example of Local Reasoning

class IntVector {
    int size() reads (this) {...} ...
}
class IntStack {
    IntVector<this> vec;
    void push(int x) writes (this) {...} ...
}
void m (IntStack s, IntVector v) writes (s) reads (v)

    where !(v <= s) !(s <= v) {
        int n = v.size(); s.push(3); assert( n == v.size() );
    }

s is not encapsulated in v, and v is not encapsulated in s
Example of Local Reasoning

class IntVector {
  int size() reads (this) {...} ...
}

class IntStack {
  IntVector (this) vec;
  void push(int x) writes (this) {...} ...
}

void m (IntStack s, IntVector v) writes (s) reads (v)
  where !(v <= s) !(s <= v) {
    int n = v.size(); s.push(3); assert( n == v.size() );
  }

size only reads v and its encapsulated objects
push only writes s and its encapsulated objects
Example of Local Reasoning

class IntVector {
    int size() reads (this) {...} ...
}
class IntStack {
    IntVector(this) vec;
    void push(int x) writes (this) {...} ...
}
void m (IntStack s, IntVector v) writes (s) reads (v)
    where !(v <= s) !(s <= v) {
        int n = v.size();  s.push(3);  assert( n == v.size() );
    }

So size and push cannot interfere
So the condition in the assert must be true