Permission-Based Ownership: Encapsulating State in Higher-Order Typed Languages

Neel Krishnaswami
Jonathan Aldrich

Department of Computer Science
Carnegie Mellon University
{neelk, aldrich}@cs.cmu.edu
A Bug in the Java 1.1 JDK

class Class {
    private Object signers[]; // an array of signers

    public Object[] getSigners() {
        return this.signers;
    }
}

The getSigners() method leaks the private internal state of Class object!
The Fix for the Bug

class Class {
    private Object signers[]; // an array of signers

    public Object[] getSigners() {
        return copy(this.signers);
    }
}

We must make a copy to prevent unwanted aliases to the private state.
The Outline of the Talk

- How should information hiding work?
- Ownership and the $F_{own}$ language
- Summary and Future Work
Information Hiding = Type Abstraction + Encapsulation

Abstract types allow the concrete type of a representation to be hidden from the client.

signature COUNTER =

<table>
<thead>
<tr>
<th>sig</th>
<th>type t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>val new : unit -&gt; t</td>
</tr>
<tr>
<td></td>
<td>val next : t -&gt; int</td>
</tr>
<tr>
<td></td>
<td>val bad : t -&gt; int ref</td>
</tr>
<tr>
<td></td>
<td>val good : t -&gt; int ref</td>
</tr>
</tbody>
</table>

structure Counter :> COUNTER =

|  struct | type t = int ref        |
|         | fun new() = ref 0       |
|         | fun next(c) = (c := !c + 1; !c) |
|         | fun bad(c) = c          |
|         | fun good(c) = ref(!c)   |

“bad” and “good” show encapsulation and type abstraction are not the same thing!
The Shape of the Program

World

Public

- new
- next
- ok

Private

17
The Access Relationships

Permission Based Ownership – PLDI 2005
The Access Relationships

Permission Based Ownership – PLDI 2005
The Creation Relationship

Permission Based Ownership – PLDI 2005
Ownership Types Formalize this Picture

• divide program into domains, each of which “owns” parts of it
  – Type phrases with $\Gamma \vdash e : \tau @ d$
  – References have type $\text{ref} \ d \ \tau$

• Check access/creation rights statically
  – access judgement $\Gamma \vdash d \rightarrow d'$
  – creation judgement $\Gamma \vdash d \Rightarrow d'$
The Domain Relationships

World

Access

Public

Access

Create

Private

Permission Based Ownership – PLDI 2005
The $F_{own}$ Language

- How should information hiding work?

- **Ownership and the $F_{own}$ language**

- Summary and Future Work
Previous Ownership Systems

- Designed for OO systems
  - Objects combine type abstraction, encapsulation, and recursive types

- Owners as dominators (aka “deep ownership”)
  - Prohibits too many useful programs
Features of $F_{own}$

$F_{own}$ is a call-by-value version of Girard and Reynolds’ System F (polymorphic lambda calculus) extended with references and a heap, and ownership domains.

- First class functions
- References (can store functions, existential packages)
- Parametric polymorphism
- First-class existential types

Each of these is annotated with the ownership domain it belongs to.
Using Ownership Domains

COUNTER ≡

\{\text{new} = \lambda_{pub}
  \ x: \text{unit}.\ \text{ref}\ priv\ 0;
\text{next} = \lambda_{pub}
  \ c: \text{ref}\ priv\ \text{int}.\ (c := !c + 1; \ !c);
\text{good} = \lambda_{pub}
  \ c:\ \text{ref}\ priv\ \text{int}.\ \text{ref}\ pub\ (!c)\}
Existential Quantification of Domains

COUNTER ≡

pack(pub,  
  pack(ref priv int,  
    {new = λ_{pub} x:unit. ref priv 0;  
      next = λ_{pub} c:ref priv int. (c := !c + 1; !c);  
      good = λ_{pub} c: ref priv int. ref pub (!c)}  
    ) as ...  
  ) as ∃_{world} public : dom(world → _, ...).
  ∃_{world} α : type.  
    {new : unit →_{public} α;  
      next : α →_{public} int;  
      good : α →_{public} ref public int }
Domain Structure Diagramatically

World → Public

Access → Create → Access

Public → Private

Access →
Domain Structure Programatically

COUNTER ≡

letdom pub : \(world \to \_\), ... into

letdom priv : (\(pub \to \_\), \(pub \to \_\)) in

pack(pub,

pack(ref priv int,

\{ new = \(\lambda_{pub} \ x:unit. \ ref \ priv \ 0\);

next = \(\lambda_{pub} \ c:ref \ priv \ int. \ (c := !c + 1; !c)\);

good = \(\lambda_{pub} \ c: ref \ priv \ int. \ ref \ pub \ (!c)\}) \) as ...

as \(\exists_{world} \ public : \text{dom}(world \to \_\), ...\).

\(\exists_{world} \ \alpha : \text{type.}

\{ new : \text{unit} \to_{public} \alpha;

next : \alpha \to_{public} \text{int};

good : \text{ref private} \text{int} \to_{public} \text{ref public} \text{int} \}

Permission Based Ownership – PLDI 2005
COUNTER \equiv

\begin{align*}
& \text{letdom } pub : (world \rightarrow _, \ldots) \text{ into} \\
& \text{letdom } priv : (pub \rightarrow _, pub \Rightarrow _) \text{ in} \\
& \text{pack}(pub, \\
& \quad \text{pack}(\text{ref } priv \text{ int,} \\
& \quad \quad \{ \text{new} = \lambda_{pub} x:unit. \text{ref } priv \ 0; \text{next} = \lambda_{pub} c:ref priv \text{ int.} \ (c := !c + 1; !c); \text{bad} = \lambda_{pub} c: ref priv \text{ int. } c \}) \text{ as …} \\
& \text{as } \exists_{world} public : \text{dom}(world \rightarrow _, \ldots). \\
& \quad \exists_{world} \alpha : \text{type.} \\
& \quad \{ \text{new} : public unit \rightarrow \alpha; \\
& \quad \text{next} : public \alpha \rightarrow \text{int}; \\
& \quad \text{bad} : public \alpha \rightarrow \text{ref } public \text{ int } \}
\end{align*}
Encapsulation vs Type Abstraction, revisited

COUNTER ≡

letdom pub : (world → _, ...) into
letdom priv : (pub → _, pub ⇒ _) in
pack(pub,
  pack(ref priv int,
    {new = λpub x:unit. ref priv 0;
     next = λpub c:ref priv int. (c := !c + 1; !c);
     not_bad = λpub c: ref priv int. c}) as ...
  as ∃_world public : dom(world → _, ...).
exists _world private : dom(public → _, public ⇒ _).
  { new : unit →public ref private int;
  next : ref private int →public int;
  not_bad : ref private int →public ref private int }
Ownership Prevents Encapsulation Violations

\[ \Gamma \vdash e : \text{ref } d' \quad \tau \odot d \quad \Gamma \vdash d \rightarrow d' \]

\[ \begin{array}{c}
\Gamma \vdash e : \text{ref } \tau \odot d \\
\Gamma \vdash d \rightarrow d' \\
\end{array} \]

\text{Deref}

\[ \Gamma \vdash e : \text{ref } \private \text{ int } \odot \text{world} \]

\[ \Gamma \vdash \text{world} \nrightarrow \private \]

\[ \Gamma \vdash e : \text{int } \odot \text{world} \]
Other Features: Domain Polymorphism

- Polymorphism over both types and domains

- Dual to existential quantification

- Support functions like `map`, and object-oriented iterators

\[
\text{map} : \forall \, d : \text{domain}(\_ \rightarrow \_).
\]
\[
\forall_d \, \alpha : \text{type}.
\]
\[
\forall_d \, \beta : \text{type}.
\]
\[
(\alpha \rightarrow_d \beta) \rightarrow_d
\]
\[
(\text{list}(\alpha) \rightarrow_d \text{list}(\beta))
\]
Conclusion

• How should information hiding work?

• The $F_{own}$ language

• **Summary and Future Work**
Related Work

• Proving contextual equivalence for stateful programs
  – Banerjee and Naumann for OO languages
  – Bierman and Parkinson with Reynolds’ separation logic

• Relationship to regions and/or information flow
  – Regions have monadic/lax translation (Fluet and Morrisett)
  – Information flow has box modality (Miyamota and Igarashi)
  – Our judgement $\Gamma \vdash e : \tau @ d$ like explicit worlds
Final Summary

- Information hiding requires type abstraction and encapsulation
- These two properties are largely orthogonal
- $F_{own}$ supports both
Managing Creation and Use

Substitution semantics complicate distinguishing definitions from uses.

\[
\frac{\Gamma, x : \tau' \vdash e : \tau \odot d'}{\Gamma \vdash \lambda_d x : \tau'.e : \tau' \rightarrow_{d'} \tau \odot d} \quad \text{LambdaProg}
\]

\[
< \lambda_d x : \tau'.e; \sigma > \rightsquigarrow < \overline{\lambda_d} x : \tau'.e; \sigma >
\]

\[
\frac{\Gamma, x : \tau' \vdash e : \tau \odot d'}{\Gamma \vdash \overline{\lambda_d} x : \tau'.e : \tau' \rightarrow_{d'} \tau \odot d} \quad \text{LambdaVal}
\]
Calling Functions and Stack Marks

\[ \Gamma \vdash e_1 : \tau' \rightarrow_{d'} \tau \odot d \quad \Gamma \vdash e_2 : \tau' \odot d \quad \Gamma \vdash d \rightarrow d' \]

\[ \Gamma \vdash e_1 \; e_2 : \tau \odot d \]

\[ < (\bar{\lambda}_d x : \tau.e) v; \sigma > \quad \sim \quad < [v/x]e \; \text{at} \; d; \sigma > \]

\[ \Gamma \vdash e : \tau \odot d' \]

\[ \Gamma \vdash e \; \text{at} \; d' : \tau \odot d \]