

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 =
sig
  type t

  val new : unit -> t
  val next : t -> int

  val bad : t -> int ref
  val good : t -> int ref
end

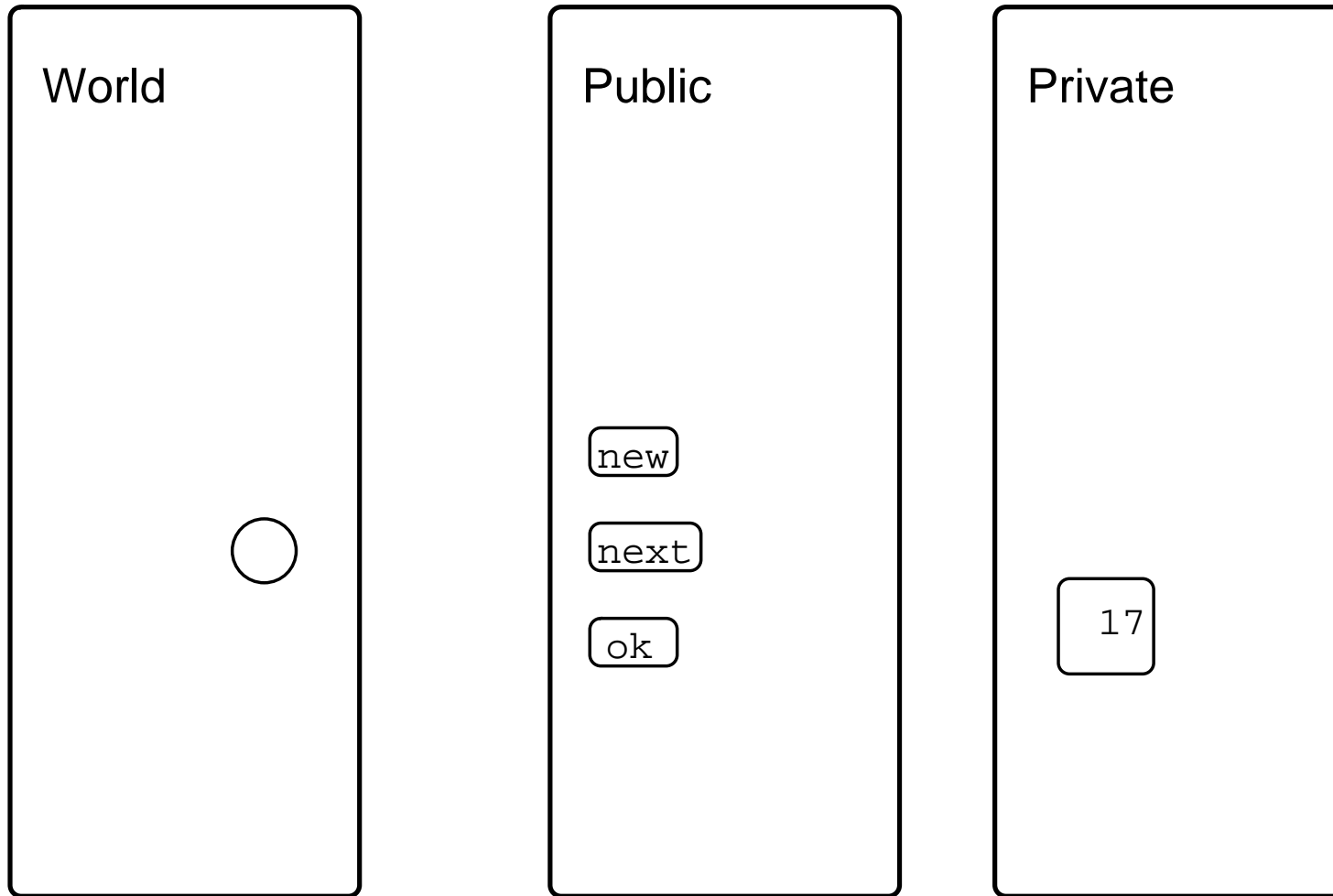
structure Counter :> COUNTER =
struct
  type t = int ref

  fun new() = ref 0
  fun next(c) = (c := !c + 1; !c)

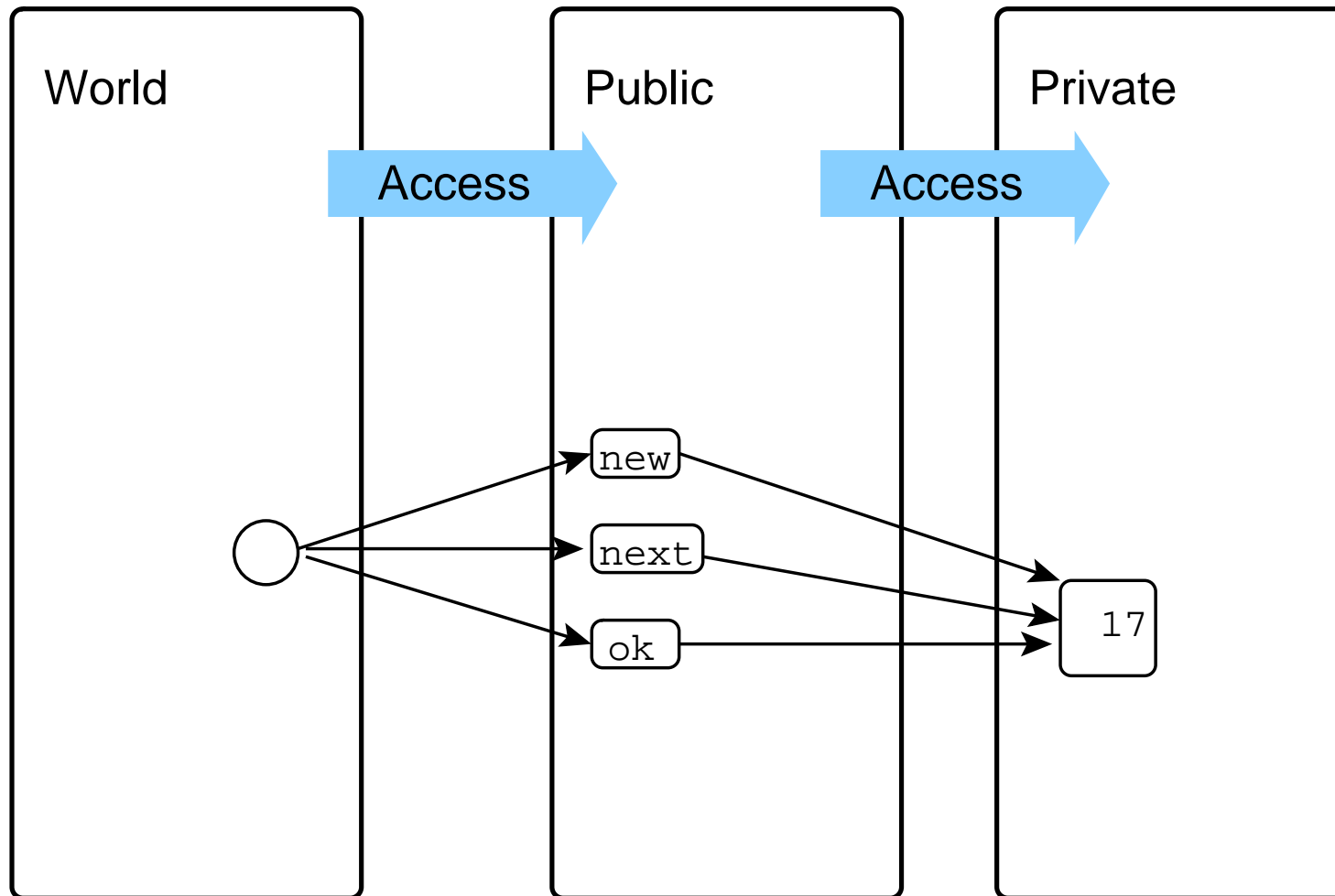
  fun bad(c) = c
  fun good(c) = ref(!c)
end
```

“bad” and “good” show encapsulation and type abstraction are not the same thing!

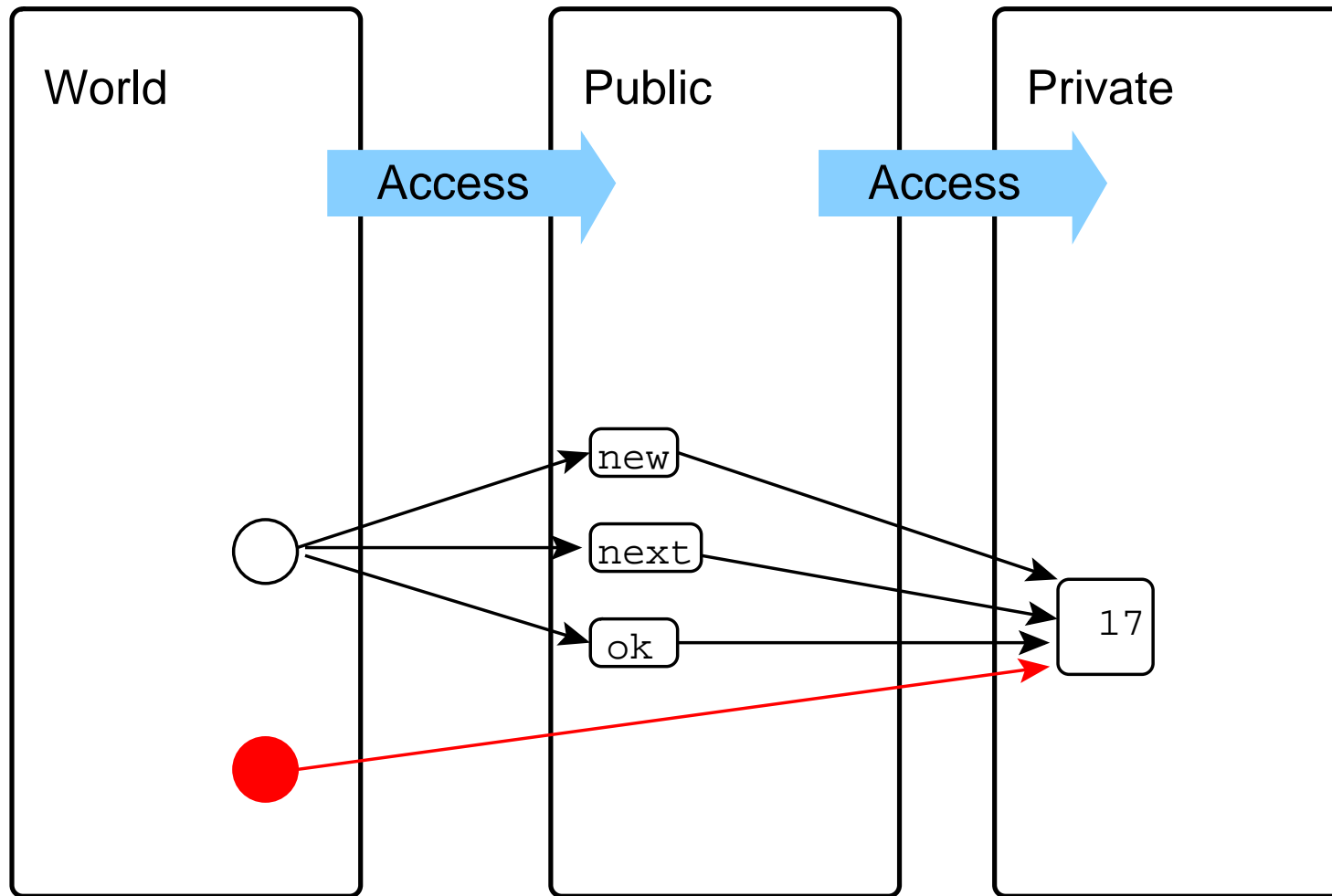
The Shape of the Program



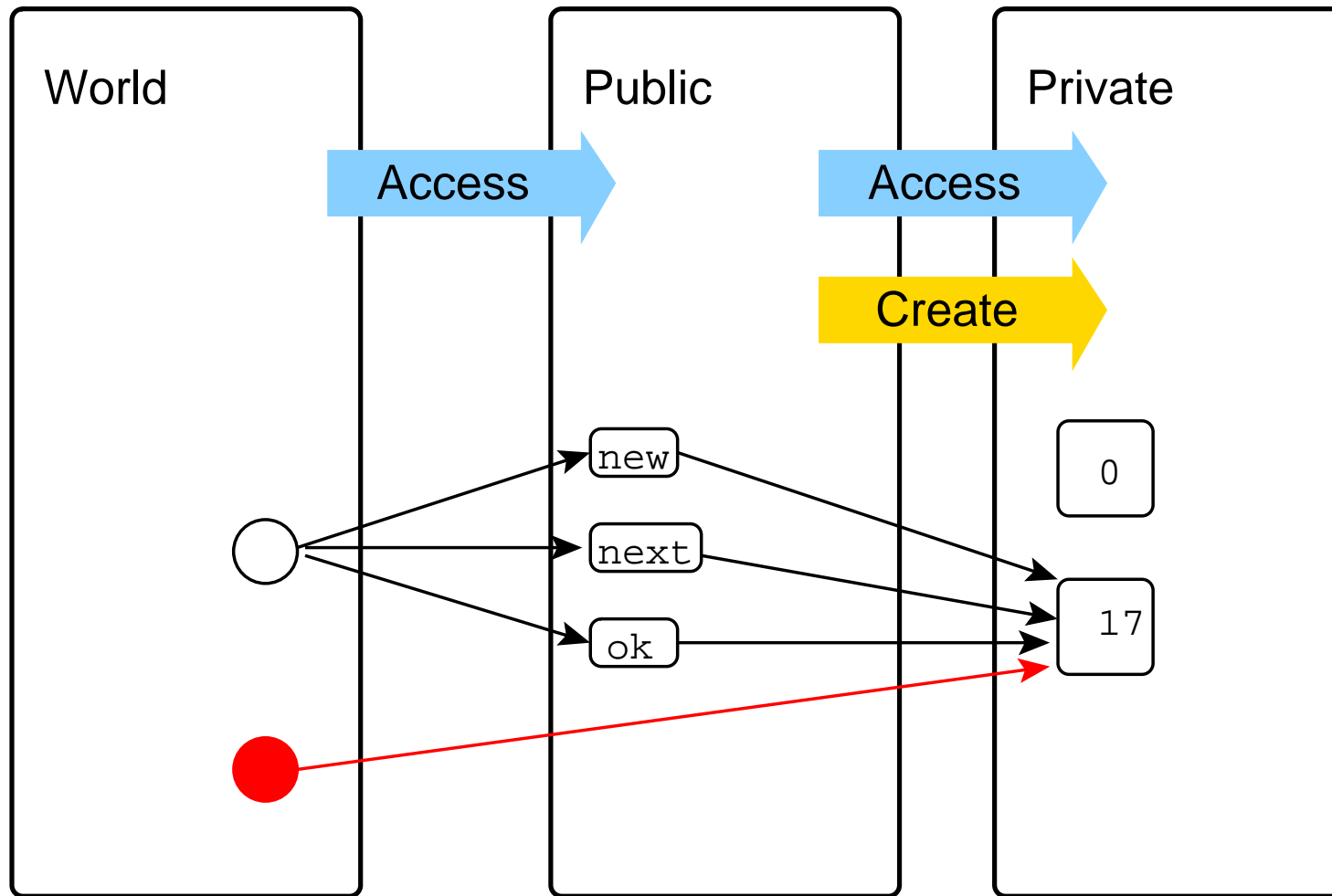
The Access Relationships



The Access Relationships



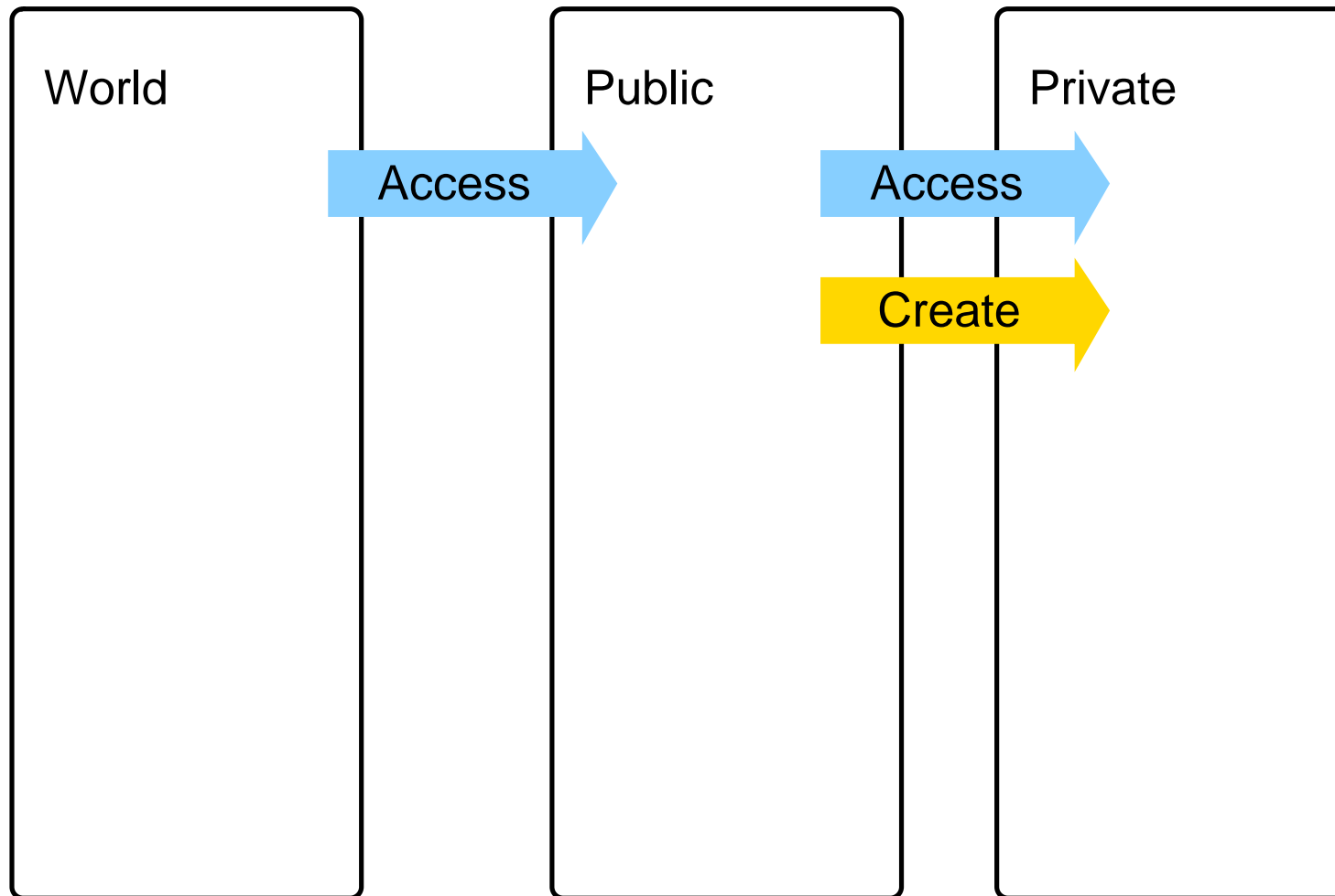
The Creation Relationship



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



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 \equiv

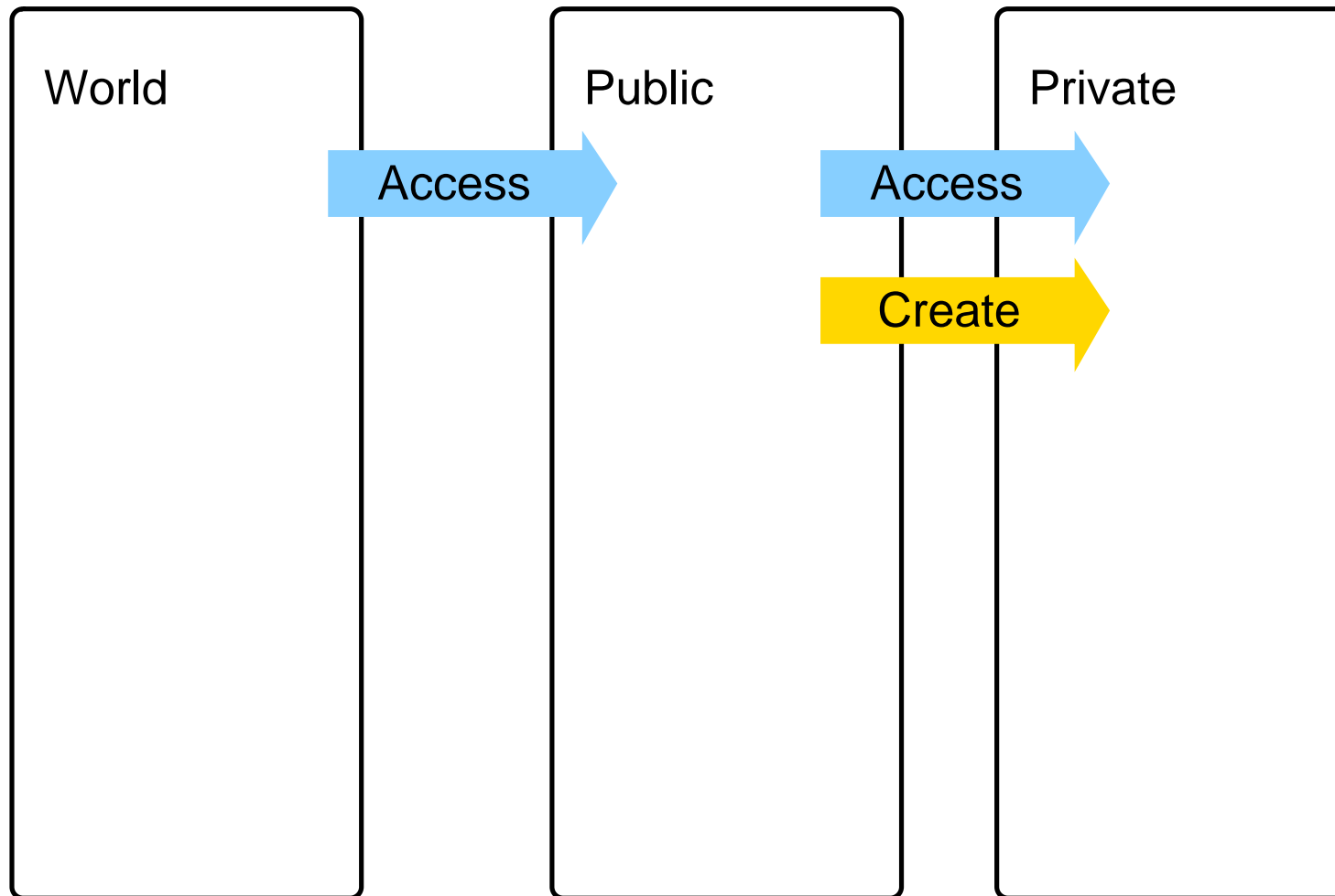
```
{new =  $\lambda_{pub}$  x:unit. ref priv 0;  
next =  $\lambda_{pub}$  c:ref priv int. (c := !c + 1; !c);  
good =  $\lambda_{pub}$  c: ref priv int. ref pub (!c)}
```

Existential Quantification of Domains

COUNTER \equiv

```
pack(pub,  
  pack(ref priv int,  
    {new =  $\lambda_{pub} x:unit. \text{ref } priv \ 0;$   
      next =  $\lambda_{pub} c:\text{ref } priv \ \text{int}. (c := !c + 1; !c);$   
      good =  $\lambda_{pub} c: \text{ref } priv \ \text{int}. \text{ref } pub \ (!c)$ }) as ...  
as  $\exists_{world} public : \text{dom}(world \rightarrow \_, \dots).$   
   $\exists_{world} \alpha : \text{type}.$   
    { new : unit  $\rightarrow_{public} \alpha;$   
      next :  $\alpha \rightarrow_{public} \text{int};$   
      good :  $\alpha \rightarrow_{public} \text{ref } public \ \text{int} \}$ 
```

Domain Structure Diagrammatically



Domain Structure Programatically

COUNTER \equiv

letdom *pub* : (*world* \rightarrow $_$, ...) into

letdom *priv* : (*pub* \rightarrow $_$, *pub* \rightarrow $_$) in

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 \exists_{world} *public* : dom(*world* \rightarrow $_$, ...).

\exists_{world} α : type.

 { new : unit \rightarrow_{public} α ;

 next : α \rightarrow_{public} int;

 good : ref *private* int \rightarrow_{public} ref *public* int }

Catching Errors with Ownership Domains

COUNTER \equiv

```
letdom pub : (world  $\rightarrow$   $\_$ , ...) into
letdom priv : (pub  $\rightarrow$   $\_$ , pub  $\Rightarrow$   $\_$ ) 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);
      bad =  $\lambda_{pub}$  c: ref priv int. c } ) as ...
as  $\exists_{world}$  public : dom(world  $\rightarrow$   $\_$ , ...).
 $\exists_{world}$   $\alpha$  : type.
  { new : unit  $\rightarrow_{public}$   $\alpha$ ;
    next :  $\alpha$   $\rightarrow_{public}$  int;
    bad :  $\alpha$   $\rightarrow_{public}$  ref public int }
```

Encapsulation vs Type Abstraction, revisited

COUNTER \equiv

```
letdom pub : (world  $\rightarrow$   $\_$ , ...) into
letdom priv : (pub  $\rightarrow$   $\_$ , pub  $\Rightarrow$   $\_$ ) 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);
     not_bad =  $\lambda_{pub}$  c: ref priv int. c}) as ...
as  $\exists_{world}$  public : dom(world  $\rightarrow$   $\_$ , ...).
 $\exists_{world}$  private : dom(public  $\rightarrow$   $\_$ , public  $\Rightarrow$   $\_$ ).
{ new : unit  $\rightarrow_{public}$  ref private int;
  next : ref private int  $\rightarrow_{public}$  int;
  not_bad : ref private int  $\rightarrow_{public}$  ref private int }
```

Ownership Prevents Encapsulation Violations

$$\frac{\Gamma \vdash e : \text{ref } d' \ \tau \ @ \ d \quad \Gamma \vdash d \rightarrow d'}{\Gamma \vdash !e : \tau \ @ \ d} \text{Deref}$$

$$\frac{\Gamma \vdash e : \text{ref } \textit{private} \ \textit{int} \ @ \ \textit{world} \quad \Gamma \vdash \textit{world} \not\rightarrow \textit{private}}{\Gamma \vdash !e : \textit{int} \ @ \ \textit{world}}$$

Other Features: Domain Polymorphism

- Polymorphism over both types and domains
- Dual to existential quantification
- Support functions like `map`, and object-oriented iterators

$$\begin{aligned} \text{map} & : \forall d : \text{domain}(_ \rightarrow _). \\ & \quad \forall_d \alpha : \text{type}. \\ & \quad \forall_d \beta : \text{type}. \\ & \quad (\alpha \rightarrow_d \beta) \rightarrow_d \\ & \quad (\text{list}(\alpha) \rightarrow_d \text{list}(\beta)) \end{aligned}$$

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.

$$\frac{\Gamma, x : \tau' \vdash e : \tau \odot d' \quad \Gamma \vdash d \Rightarrow d'}{\Gamma \vdash \lambda_d x : \tau'. e : \tau' \rightarrow_{d'} \tau \odot d} \textit{LambdaProg}$$

$$\langle \lambda_d x : \tau'. e; \sigma \rangle \rightsquigarrow \langle \bar{\lambda}_d x : \tau'. e; \sigma \rangle$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau \odot d'}{\Gamma \vdash \bar{\lambda}_d x : \tau'. e : \tau' \rightarrow_{d'} \tau \odot d} \textit{LambdaVal}$$

Calling Functions and Stack Marks

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow_{d'} \tau \textcircled{d} \quad \Gamma \vdash e_2 : \tau' \textcircled{d} \quad \Gamma \vdash d \rightarrow d'}{\Gamma \vdash e_1 e_2 : \tau \textcircled{d}}$$

$$\langle (\bar{\lambda}_d x : \tau.e)v; \sigma \rangle \rightsquigarrow \langle [v/x]e \text{ at } d; \sigma \rangle$$

$$\frac{\Gamma \vdash e : \tau \textcircled{d'}}{\Gamma \vdash e \text{ at } d' : \tau \textcircled{d}}$$