Tracing sharing and immutability in OO languages

Paola Giannini (Univ. Piemonte Orientale)

joint work with Tim Richter (Univ. Potsdam), Marco Servetto (Victoria Univ. of Wellington) and Elena Zucca (Univ. Genova)

HVL, Bergen, 16 August 2018

Outline

Outline

Context and motivation

- Design of imperative (object oriented) programming languages[1]

[1] Marco Servetto is working on the language 42 see http://l42.is/

Context and motivation

- Design of imperative (object oriented) programming languages[1]
- Key issue is sharing/aliasing of variables: a change to *x* affects *y* as well (in object oriented languages variables refer to objects!)

[1] Marco Servetto is working on the language 42 see `http://l42.is/`

Context and motivation

- Design of imperative (object oriented) programming languages[1]
- Key issue is sharing/aliasing of variables: a change to *x* affects *y* as well (in object oriented languages variables refer to objects!)
- unwanted sharing relations are common bugs: inconsistent state, invalidation of invariants

[1] Marco Servetto is working on the language 42 see `http://l42.is/`

Context and motivation

- Design of imperative (object oriented) programming languages[1]
- Key issue is sharing/aliasing of variables: a change to *x* affects *y* as well (in object oriented languages variables refer to objects!)
- unwanted sharing relations are common bugs: inconsistent state, invalidation of invariants
- particularly important in the concurrent/multithreaded case in which the "heap" containing objects is shared by threads

[1] Marco Servetto is working on the language 42 see `http://l42.is/`

Context and motivation

- Design of imperative (object oriented) programming languages[1]
- Key issue is sharing/aliasing of variables: a change to x affects y as well (in object oriented languages variables refer to objects!)
- unwanted sharing relations are common bugs: inconsistent state, invalidation of invariants
- particularly important in the concurrent/multithreaded case in which the "heap" containing objects is shared by threads
- hence: interest in type systems which statically detect (and control) sharing and mutation

[1] Marco Servetto is working on the language 42 see `http://l42.is/`

Outline

Simplified Java syntax

e	$::=$		expression
		x	variable
		$\mid e.f$	field access
		$\mid e.m(e_1, \ldots, e_n)$	method call
		$\mid e.f{=}e'$	field assignment
		$\mid$ **new** $C(e_1, \ldots, e_n)$	object creation
		$\mid \{T_1\ x_1{=}e_1; \ldots\ T_n\ x_n{=}e_n;\ e\}$	block
T	$::=$	$C \mid$ **int** $\mid \ldots$	type
md	$::=$	$T\ m(T'_1\ y_1, \ldots, T'_k\ y_k)\{T_1\ x_1{=}e_1; \ldots\ T_n\ x_n{=}e_n;\ e\}$	method definition
cd	$::=$	**class** $C\ \{T_1\ f_1; \ldots\ T_n\ f_n;\ md_1 \ldots md_k\}$	class definition

Simplified Java syntax

e	$::=$		expression
		x	variable
		$\mid e.f$	field access
		$\mid e.m(e_1, \ldots, e_n)$	method call
		$\mid e.f{=}e'$	field assignment
		$\mid$ **new** $C(e_1, \ldots, e_n)$	object creation
		$\mid \{T_1\ x_1{=}e_1; \ldots\ T_n\ x_n{=}e_n;\ e\}$	block
T	$::=$	$C \mid$ **int** $\mid \ldots$	type
md	$::=$	$T\ m(T'_1\ y_1, \ldots, T'_k\ y_k)\{T_1\ x_1{=}e_1; \ldots\ T_n\ x_n{=}e_n;\ e\}$	method definition
cd	$::=$	**class** $C\ \{T_1\ f_1; \ldots\ T_n\ f_n;\ md_1 \ldots md_k\}$	class definition

We assume classes have a constructor initialising all their fields:

$$C(T_1\ f_1, \ldots, T_n\ f_n)\{\textbf{this}.f_1 = f_1; \ldots \textbf{this}.f_n = f_n;\}$$

Simplified Java syntax

e	$::=$		expression
		x	variable
		$\mid e.f$	field access
		$\mid e.m(e_1, \ldots, e_n)$	method call
		$\mid e.f{=}e'$	field assignment
		$\mid$ **new** $C(e_1, \ldots, e_n)$	object creation
		$\mid \{T_1\ x_1{=}e_1; \ldots\ T_n\ x_n{=}e_n;\ e\}$	block
T	$::=$	$C \mid$ **int** $\mid \ldots$	type
md	$::=$	$T\ m(T'_1\ y_1, \ldots, T'_k\ y_k)\{T_1\ x_1{=}e_1; \ldots\ T_n\ x_n{=}e_n;\ e\}$	method definition
cd	$::=$	**class** $C\ \{T_1\ f_1; \ldots\ T_n\ f_n;\ md_1 \ldots md_k\}$	class definition

We assume classes have a constructor initialising all their fields:

$$C(T_1\ f_1, \ldots, T_n\ f_n)\{\texttt{this}.f_1 = f_1; \ldots \texttt{this}.f_n = f_n;\}$$

in the examples we sometimes omit the outermost block.

Sharing between variables

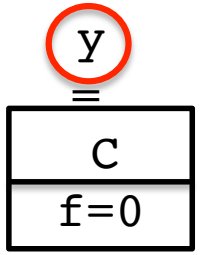
```
class B { C f; }
```

```
class C { int f;}
```

Sharing between variables

```
class B { C f; }     class C { int f;}
```

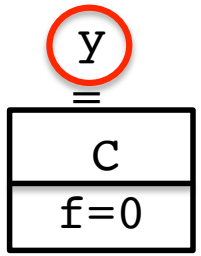
```
C y = new C(0);
```



Sharing between variables

```
class B { C f; }     class C { int f;}
```

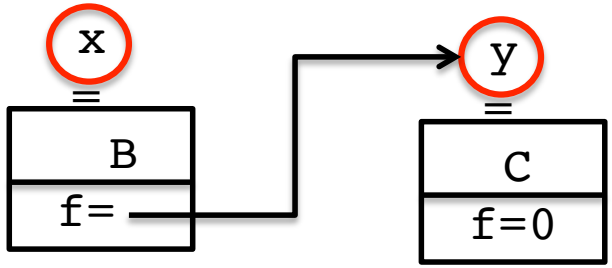
```
C y = new C(0); B x= new B(y);
```



Sharing between variables

```
class B { C f; }     class C { int f;}
```

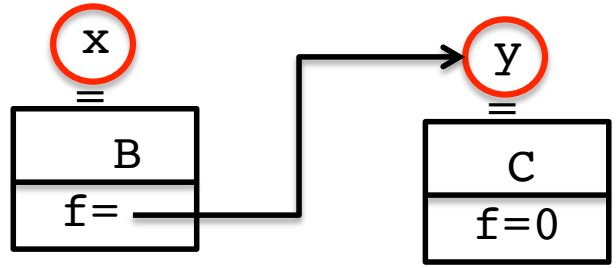
```
C y = new C(0);   B x= new B(y);
```



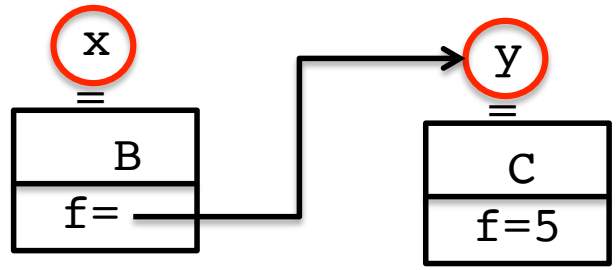
Sharing between variables

```
class B { C f; }     class C { int f;}
```

```
C y = new C(0);   B x= new B(y);
```



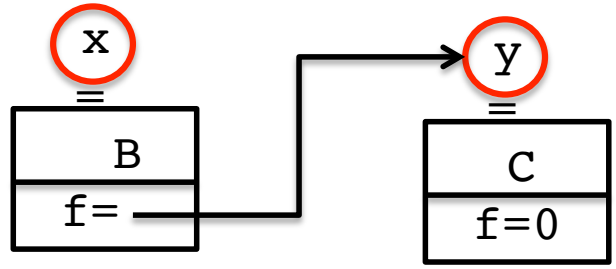

```
y.f = 5; // modifies also x.f.f
```



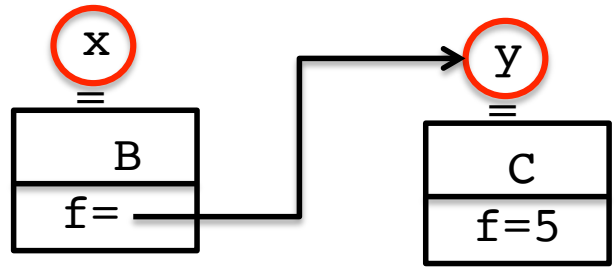
Sharing between variables

```
class B { C f; }     class C { int f;}
```

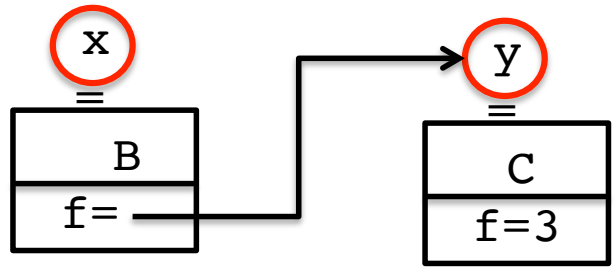
```
C y = new C(0);   B x= new B(y);
```




```
y.f = 5; // modifies also x.f.f
```




```
x.f.f = 3; // modifies also y.f
```



Uniqueness and immutability

We focus on the following properties of a reference x:

Uniqueness and immutability

We focus on the following properties of a reference x:

- *Uniqueness*: x denotes an isolated portion of store, called a capsule

Uniqueness and immutability

We focus on the following properties of a reference x:

- *Uniqueness*: x denotes an isolated portion of store, called a capsule

  = the reachable subgraph cannot be reached through other (non immutable) references

Uniqueness and immutability

We focus on the following properties of a reference x:

- *Uniqueness*: x denotes an isolated portion of store, called a capsule

  = the reachable subgraph cannot be reached through other (non immutable) references

  so x denotes mutable state that can be safely handled by a thread

Uniqueness and immutability

We focus on the following properties of a reference x:

- *Uniqueness*: x denotes an isolated portion of store, called a capsule

  = the reachable subgraph cannot be reached through other (non immutable) references

  so x denotes mutable state that can be safely handled by a thread
- *Immutability*: x denotes an immutable portion of store

Uniqueness and immutability

We focus on the following properties of a reference x:

- *Uniqueness*: x denotes an isolated portion of store, called a capsule

  = the reachable subgraph cannot be reached through other (non immutable) references

  so x denotes mutable state that can be safely handled by a thread
- *Immutability*: x denotes an immutable portion of store

  = the reachable subgraph cannot be modified through any reference

Uniqueness and immutability

We focus on the following properties of a reference x:

- *Uniqueness*: x denotes an isolated portion of store, called a capsule

  = the reachable subgraph cannot be reached through other (non immutable) references

  so x denotes mutable state that can be safely handled by a thread
- *Immutability*: x denotes an immutable portion of store

  = the reachable subgraph cannot be modified through any reference

  x can be safely shared in a multithreading environment

Uniqueness and immutability

We focus on the following properties of a reference x:

- *Uniqueness*: x denotes an isolated portion of store, called a capsule

  = the reachable subgraph cannot be reached through other (non immutable) references

  so x denotes mutable state that can be safely handled by a thread
- *Immutability*: x denotes an immutable portion of store

  = the reachable subgraph cannot be modified through any reference

  x can be safely shared in a multithreading environment

In the following nodes in red refer to mutable references in green to immmutable references and in blu to unique/capsule references.

Immutability

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x = new C(1); C y = new C(2);  new D(x,y,z1);}
}
```

Immutability

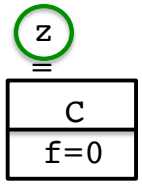
```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x = new C(1); C y = new C(2);  new D(x,y,z1);}
}
```

```
C z = new C(0);
D w = m(z);
```

Immutability

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x = new C(1); C y = new C(2);  new D(x,y,z1);}
}
```

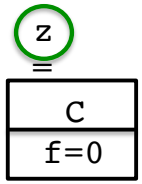
```
C z = new C(0); //no z.f=… in the code
D w = m(z);
```



Immutability

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x = new C(1); C y = new C(2);  new D(x,y,z1);}
}
```

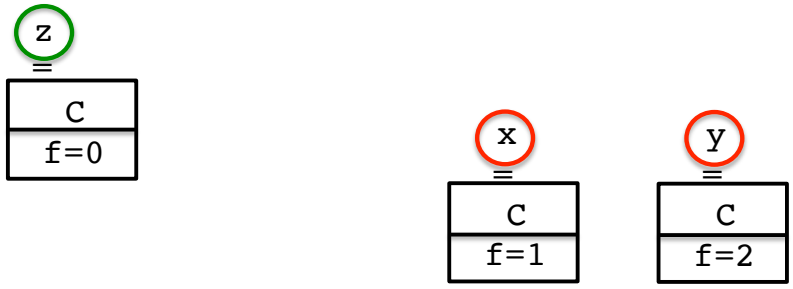
```
C z = new C(0);
D w = {C x=new C(1);C y=new C(2); new D(x,y,z)}
```



Immutability

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x = new C(1); C y = new C(2);  new D(x,y,z1);}
}
```

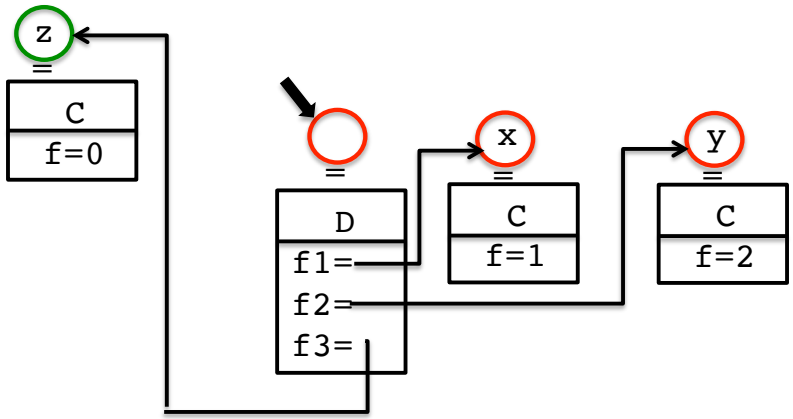
```
C z = new C(0);
D w ={C x=new C(1);C y=new C(2); new D(x,y,z)}
```



Immutability

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x = new C(1); C y = new C(2);  new D(x,y,z1);}
}
```

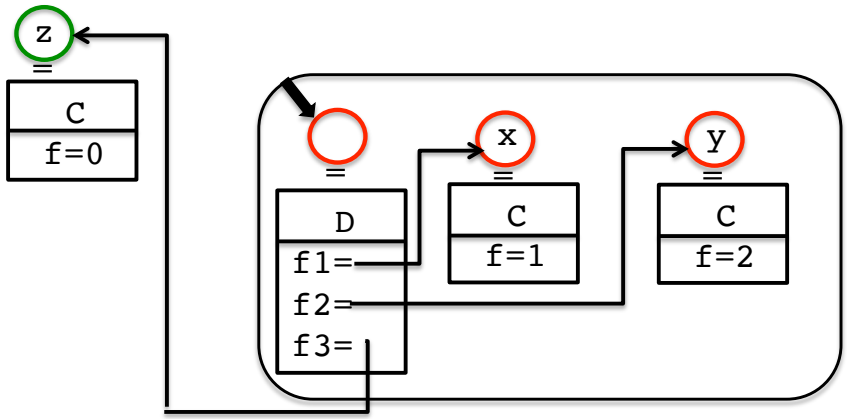
```
C z = new C(0);
D w ={C x=new C(1);C y=new C(2); new D(x,y,z)}
```



Immutability

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x = new C(1); C y = new C(2);  new D(x,y,z1);}
}
```

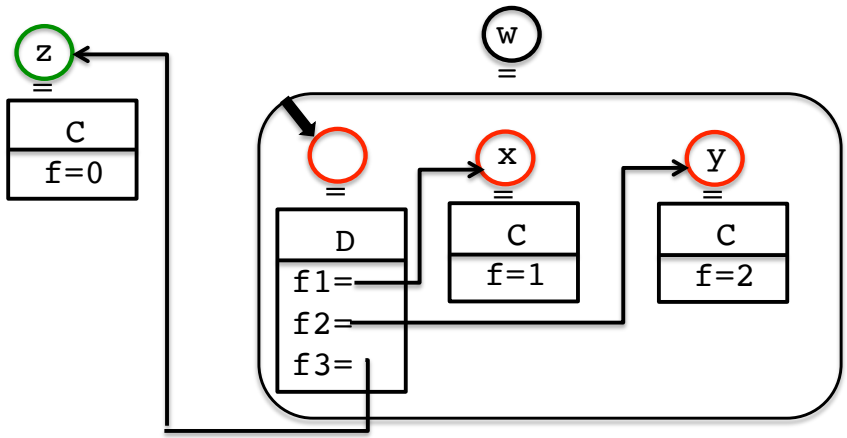
```
C z = new C(0);
D w ={C x=new C(1);C y=new C(2); new D(x,y,z)}
```



Immutability

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x = new C(1); C y = new C(2);  new D(x,y,z1);}
}
```

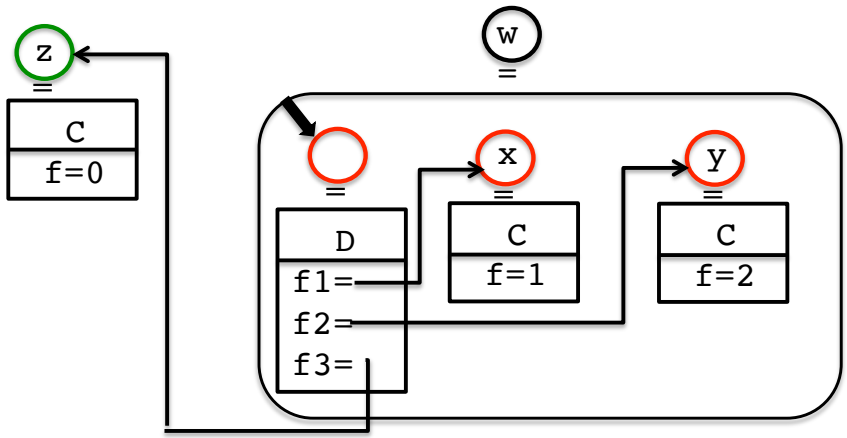
```
C z = new C(0);
D w ={C x=new C(1);C y=new C(2); new D(x,y,z)}
```



Immutability

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x = new C(1); C y = new C(2);  new D(x,y,z1);}
}
```

```
C z = new C(0);
D w ={C x=new C(1);C y=new C(2); new D(x,y,z)}
```

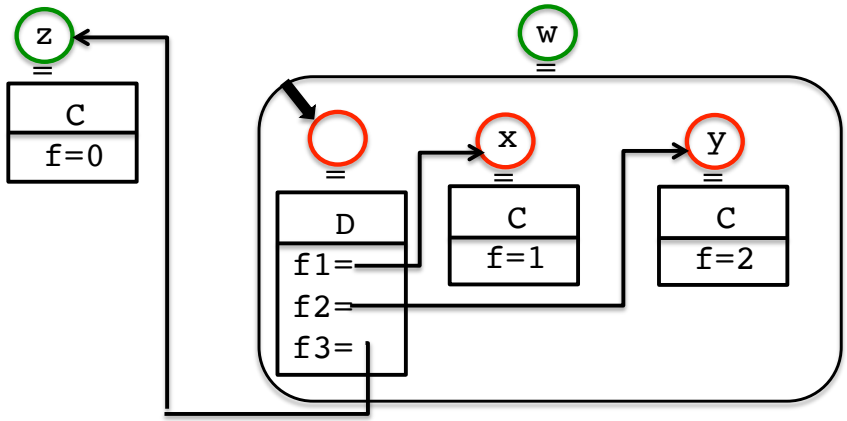


Since `z` is immutable if there are no `w.f=...` also `w` is immutable

Immutability

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x = new C(1); C y = new C(2);  new D(x,y,z1);}
}
```

```
C z = new C(0);
D w = m(z);
```



Since `z` is immutable if there are no `w.f=...` also `w` is immutable

Uniqueness (capsule)

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,y);}
}
```

Uniqueness (capsule)

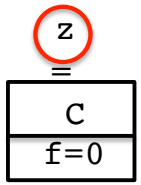
```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,y);}
}
```

```
C z = new C(0);
D w = m(z);
```

Uniqueness (capsule)

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,y);}
}
```

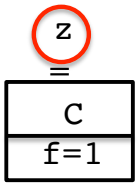
```
C z = new C(0);
D w = {C x=new C(z.f=z.f+1);C y=new C(z.f); new D(x,x,y)}
```



Uniqueness (capsule)

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,y);}
}
```

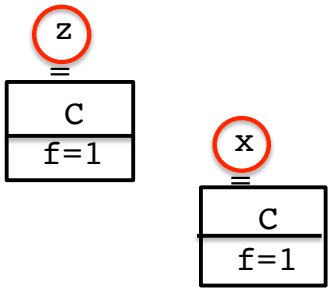
```
C z = new C(0);
D w = {C x=new C(z.f=z.f+1);C y=new C(z.f); new D(x,x,y)}
```



Uniqueness (capsule)

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,y);}
}
```

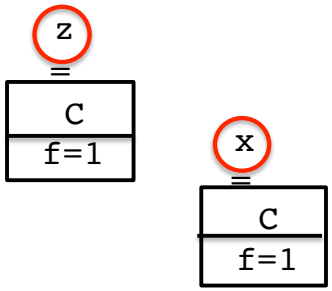
```
C z = new C(0);
D w = {C x=new C(1);C y=new C(z1.f); new D(x,x,y)}
```



Uniqueness (capsule)

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,y);}
}
```

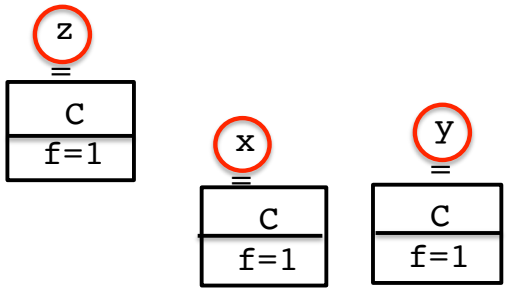
```
C z = new C(0);
D w = {C x=new C(1);C y=new C(z1.f); new D(x,x,y)}
```



Uniqueness (capsule)

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,y);}
}
```

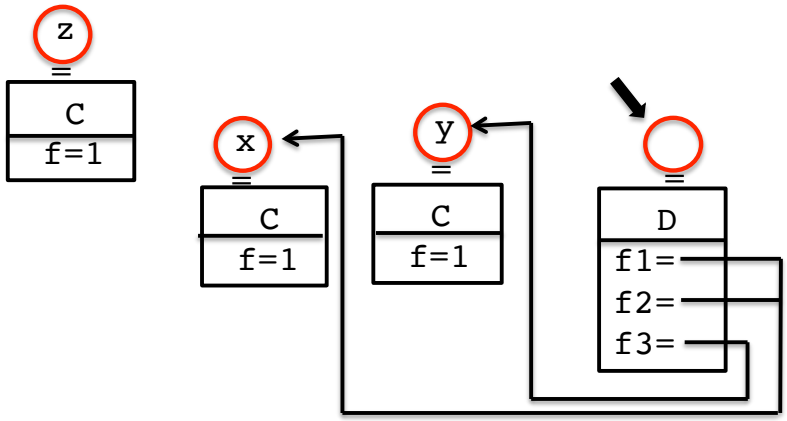
```
C z = new C(0);
D w = {C x=new C(1);C y=new C(1); new D(x,x,y)}
```



Uniqueness (capsule)

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,y);}
}
```

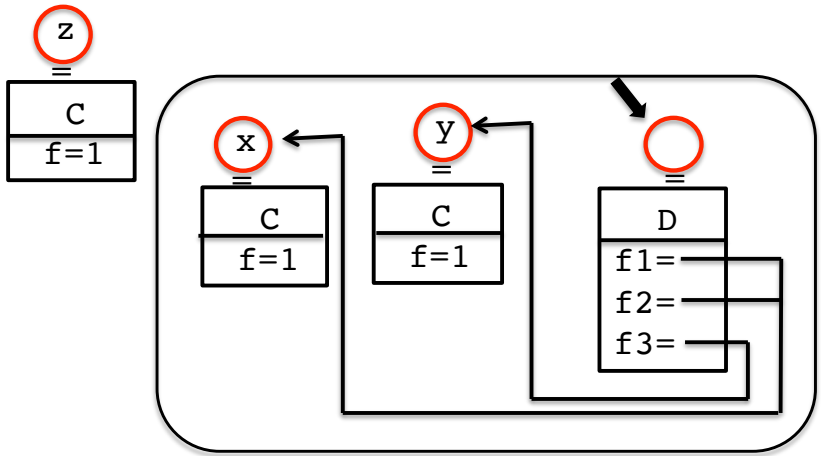
```
C z = new C(0);
D w = {C x=new C(1);C y=new C(1); new D(x,x,y)}
```



Uniqueness (capsule)

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,y);}
}
```

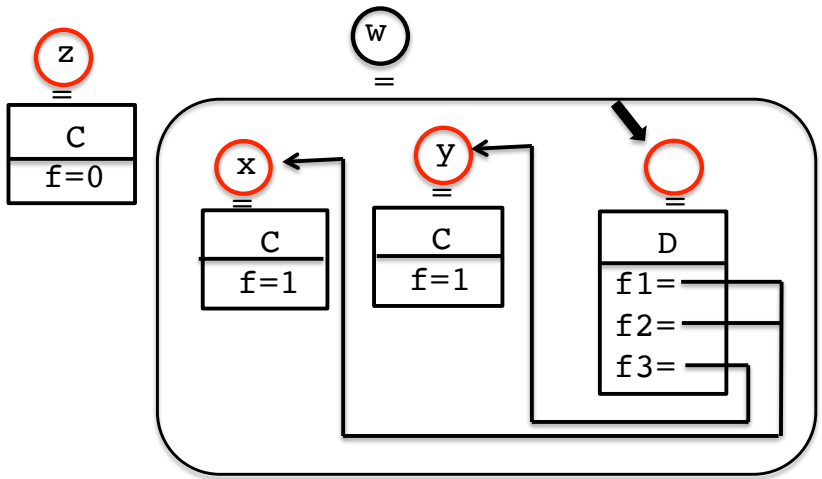
```
C z = new C(0);
D w = {C x=new C(1);C y=new C(1); new D(x,x,y)}
```



Uniqueness (capsule)

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,y);}
}
```

```
C z = new C(0);
D w = {C x=new C(1);C y=new C(1); new D(x,x,y)}
```

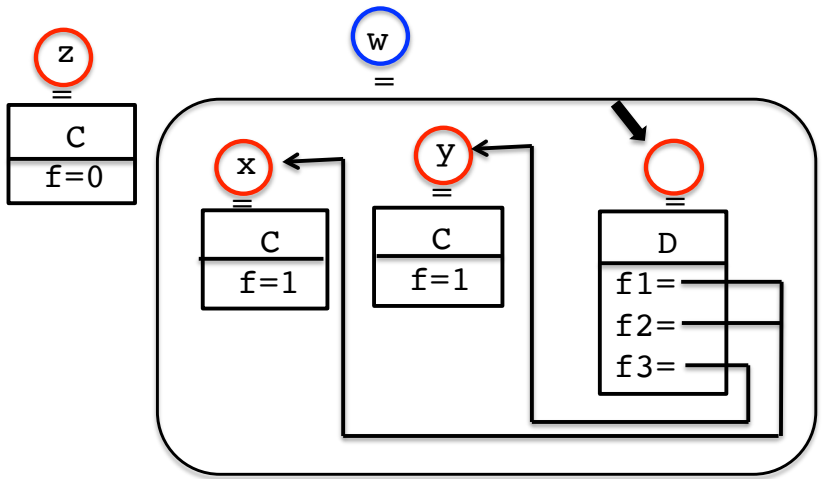


Since `w` is not connected to any mutable reference, then `w` is a capsule

Uniqueness (capsule)

```
class D {
 C f1; C f2; C f3;
 D m(C z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,y);}
}
```

```
C z = new C(0);
D w = {C x=new C(1);C y=new C(1); new D(x,x,y)}
```



Since `w` is not connected to any mutable reference, then `w` is a capsule

Not a capsule

Not a capsule

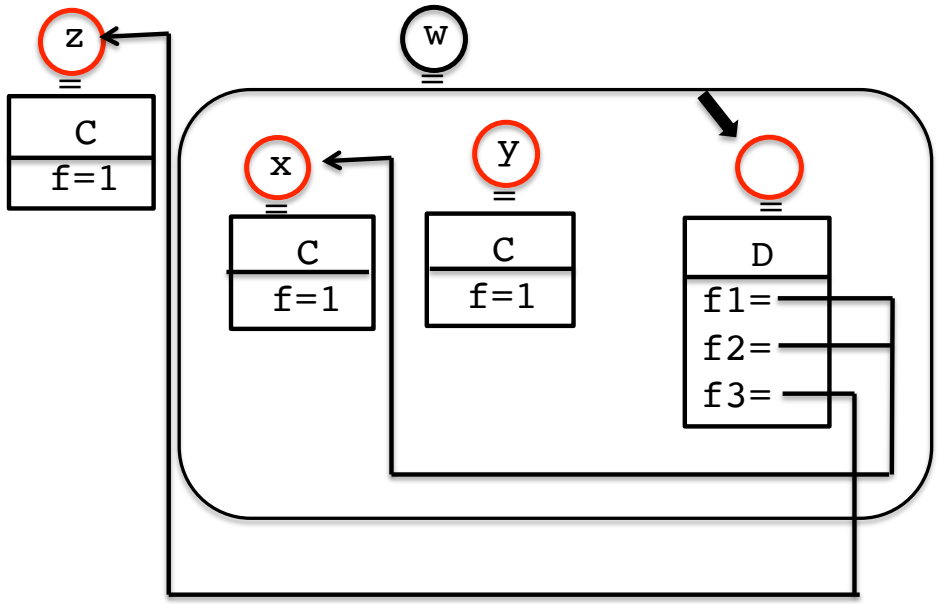
```
D m(D z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,z1);}
```

Not a capsule

```
D m(D z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,z1);}

C z = new C(0);
D w = {C x=new C(z.f=z.f+1); C y=new C(z.f);  new D(x,x,z)};
```

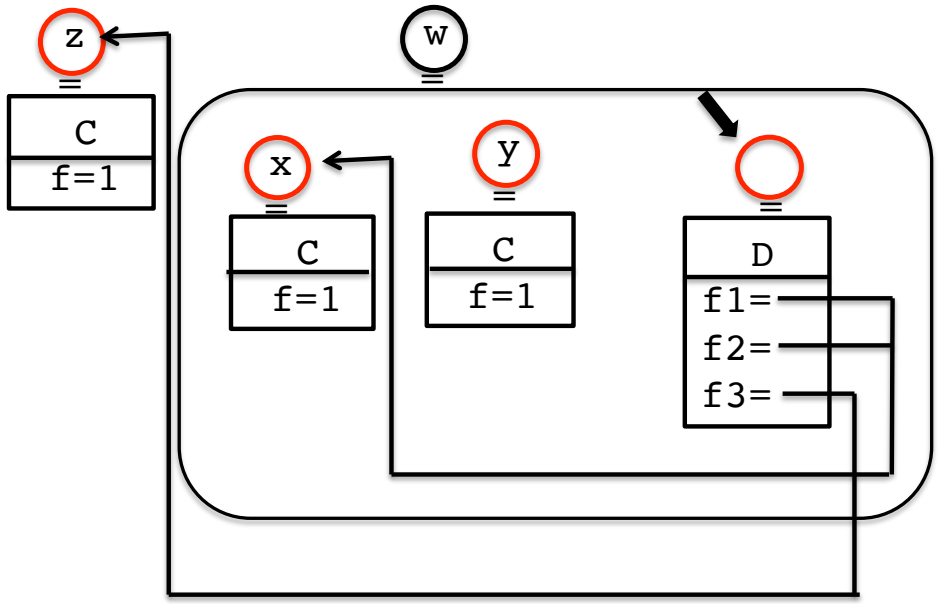
Not a capsule

```
D m(D z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,z1);}

C z = new C(0);
D w = {C x=new C(z.f=z.f+1); C y=new C(z.f);  new D(x,x,z)};
```



Not a capsule

```
D m(D z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,z1);}

C z = new C(0);
D w = {C x=new C(z.f=z.f+1); C y=new C(z.f);  new D(x,x,z)};
```

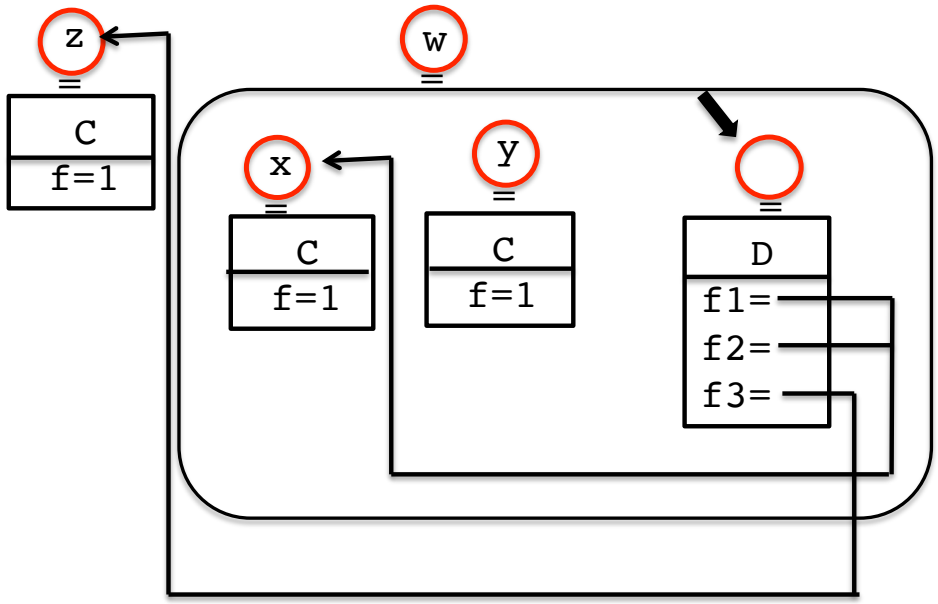


The reference `w` is connected to a mutable reference!

The reference `w` is just a mutable reference

The reference w is just a mutable reference

```
D m(D z1){C x=new C(z1.f=z1.f+1); C y=new C(z1.f);  new D(x,x,z1);}

C z = new C(0);
D w = m(z);
z.f = 7;    // modifies w.f3
```



Outline

Our proposal: static checks via a type and effect system

- enrich types with type modifiers
- define syntax directed rules that
  1. infer sharing (possibly) introduced by the evaluation of an expression
  2. enforce the restriction that only objects referred to by mutable references can be mutated
  3. check that unique references refers to capsules

Modifiers

Types are either primitive types, like `int` (and then immutable), or references

$$\mu\, C$$

Modifiers

Types are either primitive types, like `int` (and then immutable), or references

$$\mu\, C$$

depending on the modifier of μ there are restrictions and assumptions

Modifiers

Types are either primitive types, like `int` (and then immutable), or references

$$\mu\, C$$

depending on the modifier of μ there are restrictions and assumptions

μ ::= `mut` no restrictions, no assumptions

Modifiers

Types are either primitive types, like **`int`** (and then immutable), or references

$$\mu\, C$$

depending on the modifier of μ there are restrictions and assumptions

μ ::= **`mut`** no restrictions, no assumptions

| **`read`** readonly: $x.f{=}e$ is not legal

Modifiers

Types are either primitive types, like **`int`** (and then immutable), or references

$$\mu\, C$$

depending on the modifier of μ there are restrictions and assumptions

μ ::= **`mut`** no restrictions, no assumptions

| **`read`** readonly: $x.f{=}e$ is not legal

| **`imm`** readonly +
the reachable subgraph will not be modified
through any other reference

Modifiers

Types are either primitive types, like **`int`** (and then immutable), or references

$$\mu\, C$$

depending on the modifier of μ there are restrictions and assumptions

μ ::= **`mut`** no restrictions, no assumptions

| **`read`** readonly: $x.f{=}e$ is not legal

| **`imm`** readonly +
the reachable subgraph will not be modified
through any other reference

| **`caps`** the reachable subgraph is a capsule
can be used at most once

Modifiers

Types are either primitive types, like **int** (and then immutable), or references

$$\mu\, C$$

depending on the modifier of μ there are restrictions and assumptions

μ ::= **mut** no restrictions, no assumptions

| **read** readonly: $x.f{=}e$ is not legal

| **imm** readonly +
the reachable subgraph will not be modified
through any other reference

| **caps** the reachable subgraph is a capsule
can be used at most once

$$\textbf{caps} \leq \textbf{mut} \leq \textbf{read} \qquad \textbf{caps} \leq \textbf{imm} \leq \textbf{read}$$

Computing sharing effects

- define a type system to infer sharing (possibly) introduced by the evaluation of an expression:

Computing sharing effects

- define a type system to infer sharing (possibly) introduced by the evaluation of an expression:
- $\Gamma \vdash e : C \mid \mathcal{S}$

  Γ type assignment $x_1 : \mu_1\, C_1, \ldots, x_n : \mu_n\, C_n$
  C the type (class) of the result of the expression
  $\mathcal{S}$ sharing relation
  = equivalence relation on free variables of e plus `res`

Computing sharing effects

- define a type system to infer sharing (possibly) introduced by the evaluation of an expression:
- $\Gamma \vdash e : C \mid \mathcal{S}$

  Γ type assignment $x_1 : \mu_1\, C_1, \ldots, x_n : \mu_n\, C_n$
  C the type (class) of the result of the expression
  $\mathcal{S}$ sharing relation
  = equivalence relation on free variables of e plus `res`
- x and y in the same equivalence class means
  evaluation of e can introduce sharing between x and y

Computing sharing effects

- define a type system to infer sharing (possibly) introduced by the evaluation of an expression:
- $\Gamma \vdash e : C \mid \mathcal{S}$

  Γ type assignment $x_1 : \mu_1\, C_1, \ldots, x_n : \mu_n\, C_n$
  C the type (class) of the result of the expression
  $\mathcal{S}$ sharing relation
  = equivalence relation on free variables of e plus `res`
- x and y in the same equivalence class means
  evaluation of e can introduce sharing between x and y
- if x is in the equivalence class of `res`
  evaluation of e returns a reference in sharing with x

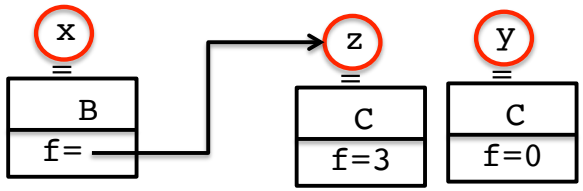
Operations introducing sharing between variables

- field assignment

Operations introducing sharing between variables

- field assignment

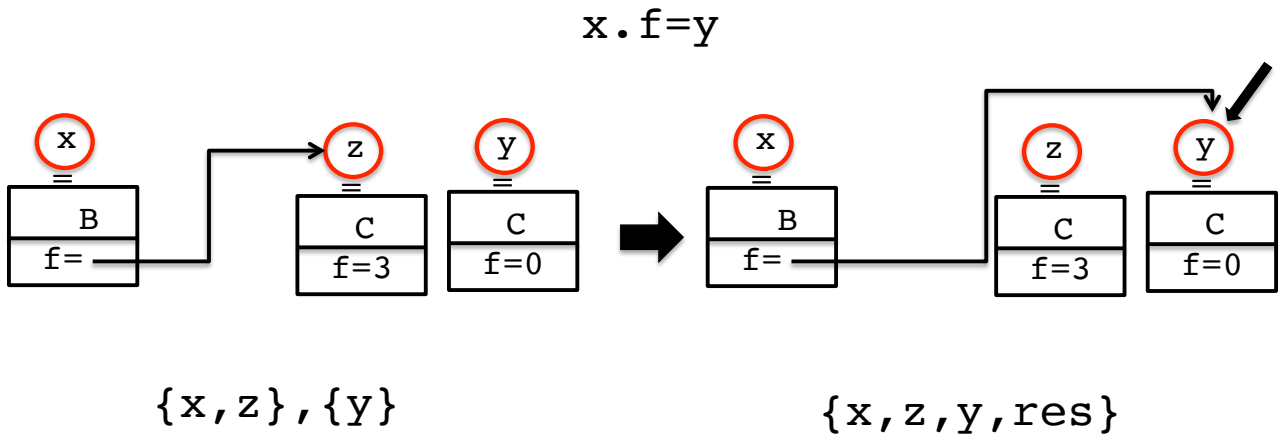
`x.f=y`



`{x,z},{y}`

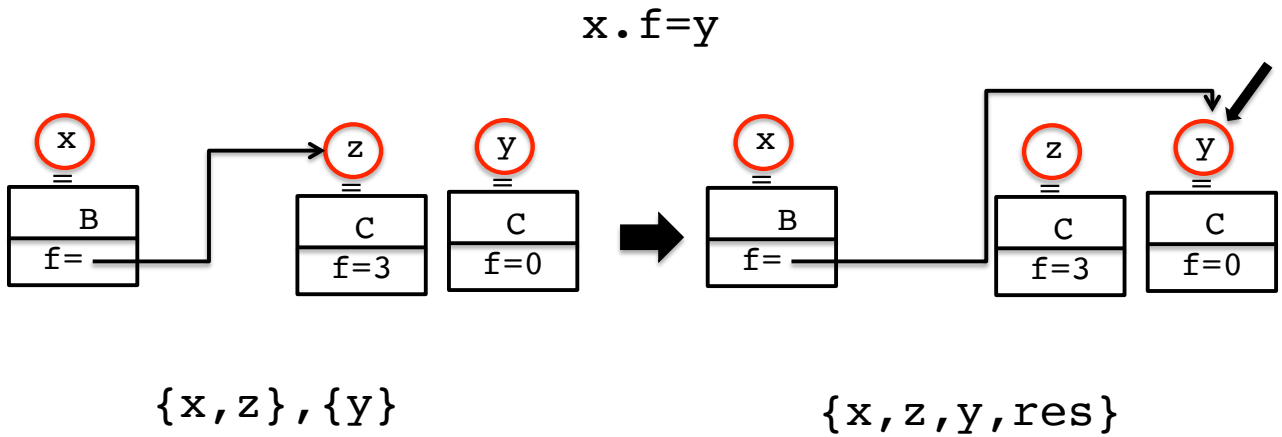
Operations introducing sharing between variables

- field assignment



Operations introducing sharing between variables

- field assignment

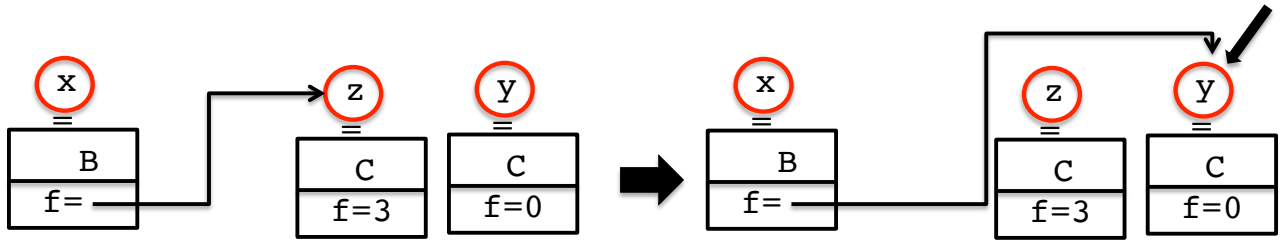


- object creation

Operations introducing sharing between variables

- field assignment

```
x.f=y
```

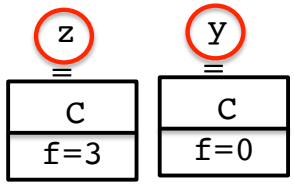



```
{x,z},{y}
```

```
{x,z,y,res}
```

- object creation

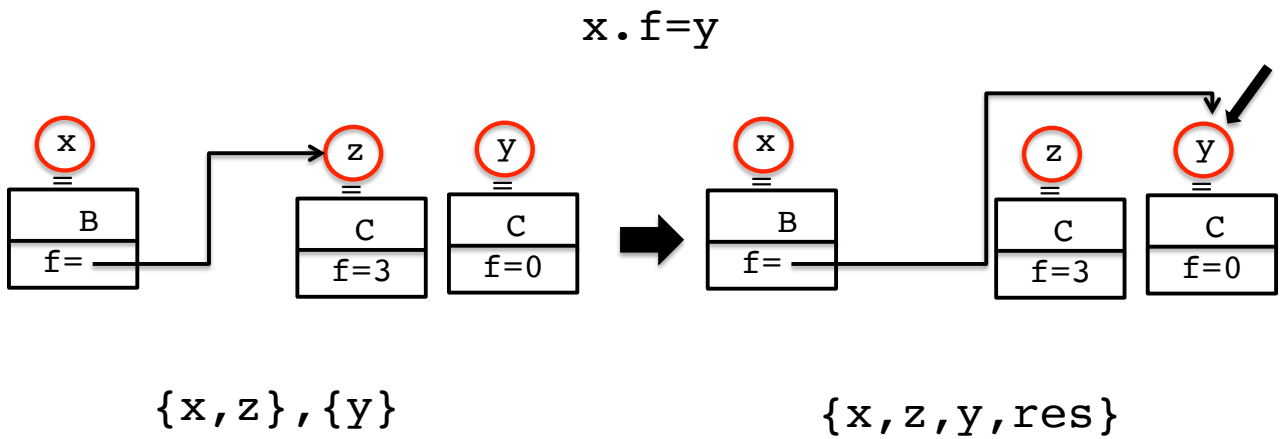
```
new D(z,y)
```



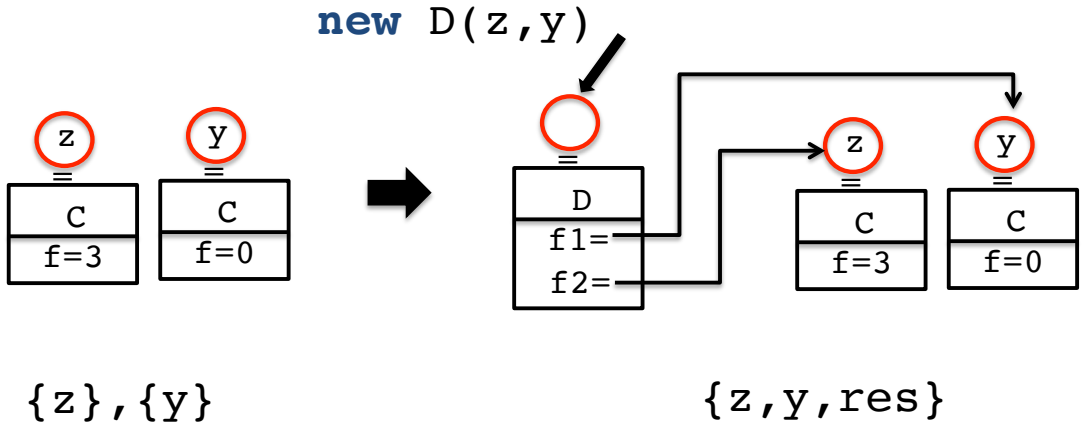

```
{z},{y}
```

Operations introducing sharing between variables

- field assignment



- object creation



Detecting uniqueness

- uniqueness is detected when the result of an expression is disjoint from any of its free mutable variable

Detecting uniqueness

- uniqueness is detected when the result of an expression is disjoint from any of its free mutable variable
- formally, the equivalence class of `res` does not contain mutable variables in $\mathcal{S}$

Detecting uniqueness

- uniqueness is detected when the result of an expression is disjoint from any of its free mutable variable
- formally, the equivalence class of `res` does not contain mutable variables in $\mathcal{S}$
- if `class C{D f;}` and `class D{C f;}` then

Detecting uniqueness

- uniqueness is detected when the result of an expression is disjoint from any of its free mutable variable
- formally, the equivalence class of `res` does not contain mutable variables in $\mathcal{S}$
- if **`class`** `C{D f;}` and **`class`** `D{C f;}` then

$$\{\texttt{C w=(x.f=y); C u=(z.f); u}\}$$

Detecting uniqueness

- uniqueness is detected when the result of an expression is disjoint from any of its free mutable variable
- formally, the equivalence class of `res` does not contain mutable variables in $\mathcal{S}$
- if **`class`** `C{D f;}` and **`class`** `D{C f;}` then

$$\{\texttt{C w=(x.f=y); C u=(z.f); u}\}$$

where `x` : `D`, `y` : `C` and `z` : `D`, is not a capsule

Detecting uniqueness

- uniqueness is detected when the result of an expression is disjoint from any of its free mutable variable
- formally, the equivalence class of `res` does not contain mutable variables in $\mathcal{S}$
- if **`class`** `C{D f;}` and **`class`** `D{C f;}` then

$$\{\texttt{C w=(x.f=y); C u=(z.f); u}\}$$

where `x` : `D`, `y` : `C` and `z` : `D`, is not a capsule since

$$\Gamma \vdash \{\texttt{C w=(x.f=y); C u=(z.f); u}\} : \texttt{C} \mid \{\texttt{x},\texttt{y}\},\{\texttt{z},\texttt{res}\}$$

Detecting uniqueness

- uniqueness is detected when the result of an expression is disjoint from any of its free mutable variable
- formally, the equivalence class of `res` does not contain mutable variables in $\mathcal{S}$
- if **`class`** `C{D f;}` and **`class`** `D{C f;}` then

$$\{\texttt{C w=(x.f=y); C u=(z.f); u}\}$$

where `x` : `D`, `y` : `C` and `z` : `D`, is not a capsule since

$$\Gamma \vdash \{\texttt{C w=(x.f=y); C u=(z.f); u}\} : \texttt{C} \mid \{\texttt{x},\texttt{y}\},\{\texttt{z},\texttt{res}\}$$

unless `z` is immutable

Detecting uniqueness

- uniqueness is detected when the result of an expression is disjoint from any of its free mutable variable
- formally, the equivalence class of `res` does not contain mutable variables in $\mathcal{S}$
- if **`class`** `C{D f;}` and **`class`** `D{C f;}` then

$$\{\texttt{C w=(x.f=y); C u=(z.f); u}\}$$

where `x` : `D`, `y` : `C` and `z` : `D`, is not a capsule since

$$\Gamma \vdash \{\texttt{C w=(x.f=y); C u=(z.f); u}\} : \texttt{C} \mid \{\texttt{x},\texttt{y}\},\{\texttt{z},\texttt{res}\}$$

unless `z` is immutable

- whereas

$$\{\texttt{C w=(x.f=y); C u=}\textbf{\texttt{new}}\ \texttt{D(}\textbf{\texttt{new}}\ \texttt{C()).f; u}\}$$

is a capsule

Detecting uniqueness

- uniqueness is detected when the result of an expression is disjoint from any of its free mutable variable
- formally, the equivalence class of `res` does not contain mutable variables in $\mathcal{S}$
- if **`class`** `C{D f;}` and **`class`** `D{C f;}` then

$$\{\texttt{C w=(x.f=y); C u=(z.f); u}\}$$

where `x : D`, `y : C` and `z : D`, is not a capsule since

$$\Gamma \vdash \{\texttt{C w=(x.f=y); C u=(z.f); u}\} : \texttt{C} \mid \{\texttt{x},\texttt{y}\},\{\texttt{z},\texttt{res}\}$$

unless `z` is immutable

- whereas

$$\{\texttt{C w=(x.f=y); C u=}\textbf{\texttt{new}}\ \texttt{D(}\textbf{\texttt{new}}\ \texttt{C()).f; u}\}$$

is a capsule since

$$\Gamma \vdash \{\texttt{C w=(x.f=y); C u=}\textbf{\texttt{new}}\ \texttt{D(}\textbf{\texttt{new}}\ \texttt{C()).f; u}\} : \texttt{C} \mid \{\texttt{x},\texttt{y}\},\{\texttt{z}\},\{\texttt{res}\}$$

Detecting uniqueness

- uniqueness is detected when the result of an expression is disjoint from any of its free mutable variable
- formally, the equivalence class of `res` does not contain mutable variables in $\mathcal{S}$
- if **`class`** `C{D f;}` and **`class`** `D{C f;}` then

$$\{\texttt{C w=(x.f=y); C u=(z.f); u}\}$$

where `x : D`, `y : C` and `z : D`, is not a capsule since

$$\Gamma \vdash \{\texttt{C w=(x.f=y); C u=(z.f); u}\} : \texttt{C} \mid \{\texttt{x},\texttt{y}\},\{\texttt{z},\texttt{res}\}$$

unless `z` is immutable

- whereas

$$\{\texttt{C w=(x.f=y); C u=}\textbf{\texttt{new}}\ \texttt{D(}\textbf{\texttt{new}}\ \texttt{C()).f; u}\}$$

is a capsule since

$$\Gamma \vdash \{\texttt{C w=(x.f=y); C u=}\textbf{\texttt{new}}\ \texttt{D(}\textbf{\texttt{new}}\ \texttt{C()).f; u}\} : \texttt{C} \mid \{\texttt{x},\texttt{y}\},\{\texttt{z}\},\{\texttt{res}\}$$

Outline

Programming examples

modifier of `this` in violet

Programming examples

modifier of **this** in violet

```
class List { ...

 caps List deepcopy(read)/*{this},{res}*/{ ... }

 mut List concat (mut, read List other)
    /*{this,res,other}*/{ ... }

 caps List concatcopy (read, read List other)
  /*{this},{res},{other}*/{ ... }
 }
```

Programming examples

```
class IntListReader {
  static caps IntList readIntList (mut Scanner s) /*{s},{res}*/{
    mut IntList list=new IntList()
    while(s.hasNextNum()){
      list.addInt(s.nextNum())
    }
    return list// capsule recovery
  }
}

class Scanner { ...
  boolean hasNextNum (read)
  int nextNum (mut)
}
```

Programming examples

```
class IntListReader {...//as before
  static  caps IntList update(caps IntList old, mut Scanner s)
  /*{s},{res}*/ {
    mut IntList list=old//we open the capsule 'old'
    while(s.hasNextNum()){
      list.addInt(s.nextNum())
    }
    return list
  }
}
```

Programming examples

```
class Person{
  private mut PersonList friends;
  read PersonList readFriends (read)/*{this,res}*/{
    return this.friends;
  }
  mut PersonList getFriends (mut)/*{this,res}*/{
    return this.friends;
  }
}
```

Programming examples

```
class Person{
  private mut PersonList friends;
  read PersonList readFriends (read)/*{this,res}*/{
    return this.friends;
  }
  mut PersonList getFriends (mut)/*{this,res}*/{
    return this.friends;
  }
}
```

two getter methods with different type annotations:

Programming examples

```
class Person{
  private mut PersonList friends;
  read PersonList readFriends (read)/*{this,res}*/{
    return this.friends;
  }
  mut PersonList getFriends (mut)/*{this,res}*/{
    return this.friends;
  }
}
```

two getter methods with different type annotations:
`p.readFriends()` can be invoked on any `p`, **`imm`** if `p` is **`imm`**

Programming examples

```
class Person{
  private mut PersonList friends;
  read PersonList readFriends (read)/*{this,res}*/{
    return this.friends;
  }
  mut PersonList getFriends (mut)/*{this,res}*/{
    return this.friends;
  }
}
```

two getter methods with different type annotations:

`p.readFriends()` can be invoked on any `p`, **`imm`** if `p` is **`imm`**

`p.getFriends()` p cannot be **`read`** or **`imm`**

Outline

(Some) related work

- (variants of) capsule property:
  isolated [Gordon et al. OOPSLA'12]
  external uniqueness [Clarke&Wrigstadt ECOOP'03]
  balloon [Almeida ECOOP'97, Servetto et al. WODET'14]
  island [Dietl et al. ECOOP'07]
- ownership: *x* is "owned" by *y*, always true, capsule notion more dynamic
- types as compositions of capabilities
  [Haller&Odersky ECOOP'10, Clebsch et al. AGERE'15,Castegren&Wrigstad ECOOP'16]
- the Rust language `rust-lang.org`
- the Pony language `ponylang.org`

Conclusions

- type and effect system which infers sharing possibly introduced by the evaluation of an expression

Conclusions

- type and effect system which infers sharing possibly introduced by the evaluation of an expression
- very expressive

Conclusions

- type and effect system which infers sharing possibly introduced by the evaluation of an expression
- very expressive
- we have a formalisation in Coq of the typing and the evaluation (modelling sharing relations is the most challenging part and is still not completely satisfying)

Conclusions

- type and effect system which infers sharing possibly introduced by the evaluation of an expression
- very expressive
- we have a formalisation in Coq of the typing and the evaluation (modelling sharing relations is the most challenging part and is still not completely satisfying)
- we have proved the correctness of the dynamic semantics of our syntactic model for an imperative OO language w.r.t. the standard semantics of imperative calculi relying on a global memory

Future work

Future work

- (short term) complete the soundness proof in Coq

Future work

- (short term) complete the soundness proof in Coq
- (short term) handle lent (borrowed) references
  = the reachable graph can be manipulated, but not shared, by a client

Future work

- (short term) complete the soundness proof in Coq
- (short term) handle lent (borrowed) references
  = the reachable graph can be manipulated, but not shared, by a client
- (long term) investigate (a form of) Hoare logic on top of our model

References


Paola Giannini, Marco Servetto, and Elena Zucca.
Types for immutability and aliasing control.
In *ICTCS 16*, volume 1720 of *CEUR Workshop Proceedings*, pages 62–74. CEUR-WS.org, 2016.


Paola Giannini, Marco Servetto, and Elena Zucca.
Tracing sharing in an imperative pure calculus: extended abstract.
In *FTfJP'17 - Formal Techniques for Java-like Programs*, pages 6:1–6:6, 2017.


Paola Giannini, Marco Servetto, and Elena Zucca.
A type and effect system for sharing.
In *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 1513–1515, 2017.


Paola Giannini, Marco Servetto, and Elena Zucca.
A type and effect system for uniqueness and immutability.
In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 1038–1045, 2018.


Paola Giannini, Tim Richter, Marco Servetto, and Elena Zucca.
Tracing sharing in an imperative pure calculus.
CoRR, abs/1803.05838, 2018.


Paola Giannini, Marco Servetto, Elena Zucca, and James Cone.
Flexible recovery of uniqueness and immutability (extended version).
CoRR, abs/1807.00137, 2018.

Thanks