Aliasing control in an imperative pure calculus

Marco Servetto[1], Elena Zucca[2]

(1) Victoria University of Wellington – (2) University of Genova

CINA final meeting, Civitanova Marche, January 2016
Presented at APLAS'15

Contribution

- imperative object calculus with two modifiers for aliasing control

Contribution

- imperative object calculus with two modifiers for aliasing control
- a `lent` reference cannot introduce aliasing
  between the denoted portion of store [reachable object graph] and others
  cannot be used in assignment and as constructor arguments

Contribution

- imperative object calculus with two modifiers for aliasing control
- a `lent` reference cannot introduce aliasing
  between the denoted portion of store [reachable object graph] and others
  cannot be used in assignment and as constructor arguments
- a `capsule` reference denote an isolated portion of store [reachable object graph]
  unique entry point is the `capsule` reference itself

P. Almeida. Balloon types: Controlling sharing of state in data types. ECOOP'97

J. Boyland. Semantics of fractional permissions with nesting. TOPLAS 32(6), 2010.

D. Clarke, T. Wrigstad. External uniqueness is unique enough. ECOOP'03

C.S. Gordon, M.J. Parkinson, J. Parsons, A. Bromeld, J. Duy. Uniqueness and reference immutability for safe parallelism. OOPSLA'12

J. Hogg. Islands: Aliasing protection in object-oriented languages. OOPSLA'91

K. Naden, R. Bocchino, J. Aldrich, K. Bierho. A type system for borrowing permissions. POPL'12.

Novelties

1. expressivity enhanced by promotion and swapping rules

Novelties

1. expressivity enhanced by promotion and swapping rules
   - promotion: an expression which uses external references only as `lent` is a `capsule`

Novelties

1. expressivity enhanced by promotion and swapping rules
   - promotion: an expression which uses external references only as `lent` is a `capsule`
   - swapping: a `lent` reference can be freely used if all other references are regarded as `lent`

Novelties

1. expressivity enhanced by promotion and swapping rules
   - promotion: an expression which uses external references only as `lent` is a `capsule`
   - swapping: a `lent` reference can be freely used if all other references are regarded as `lent`
2. execution model as pure calculus
   - no memory, just rewriting source code

Novelties

1. expressivity enhanced by promotion and swapping rules
   - promotion: an expression which uses external references only as `lent` is a `capsule`
   - swapping: a `lent` reference can be freely used if all other references are regarded as `lent`
2. execution model as pure calculus
   - no memory, just rewriting source code
   - object graphs are represented at the syntactic level

1. Execution model
2. Type system
3. Examples
4. Conclusion

Syntax

convention: *ds* is a sequence of *d*

Java-like flavour is matter of taste

cd	$::=$	$\texttt{class}\ C\ \{fds\ mds\}$	class declaration
fd	$::=$	$C\ f$	field declaration
md	$::=$	$T\ m\ \mu\ (T_1\,x_1,\ldots,T_n\,x_n)\ \{\texttt{return}\ e\}$	method declaration
e	$::=$	$x \mid e.f \mid e.m(es) \mid e.f = e' \mid \texttt{new}\ C(es) \mid (ds\ e)$	expression
d	$::=$	$T\ x = e$	variable declaration
T	$::=$	$\mu\ C$	type
μ	$::=$	$\texttt{capsule} \mid \texttt{lent} \mid \epsilon$	optional type modifier

Traditional imperative model

- $\langle e, \mu \rangle \longrightarrow \langle e', \mu' \rangle$
- μ memory or store
- map from locations into storable values
- locations are a runtime notion and memory is flat
- variables are a language notion and obey scoping rules (shadowing) and α-conversion

Our model

- $e \longrightarrow e'$
- key idea: the block construct, introducing local variable declarations, plays the role of store when such declarations have been evaluated

An example

```
class D { int f; ...}
class C { D f1; D f2; ...}
```

```
D z=new D(0)
C x=new C(z,z)
C y=x
D w=new D(y.f1.f+1)
x.f2=w
x
```

An example

```
class D { int f; ...}
class C { D f1; D f2; ...}
```

```
D z=new D(0)
C x=new C(z,z)
C y=x
D w=new D(y.f1.f+1)
x.f2=w
x
```

$\longrightarrow$

```
D z=new D(0)
C x=new C(z,z)
D w=new D(x.f1.f+1)
x.f2=w
x
```

```
D z=new D(0)
C x=new C(z,z)
D w=new D(x.f1.f+1)
x.f2=w
x
```

$\longrightarrow$

```
D z=new D(0)
C x=new C(z,z)
D w=new D(z.f+1)
x.f2=w
x
```

```
D z=new D(0)
C x=new C(z,z)
D w=new D( z.f +1)
x.f2=w
x
```

$\longrightarrow$

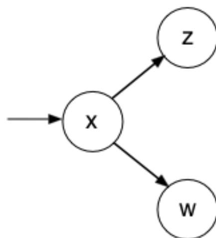
```
D z=new D(0)
C x=new C(z,z)
D w=new D(0+1)
x.f2=w
x
```

```
D z=new D(0)            D z=new D(0)
C x=new C(z,z)          C x=new C(z,z)
D w=new D( 0+1 )  ⟶    D w=new D(1)
x.f2=w                  x.f2=w
x                       x
```

```
D z=new D(0)
C x=new C(z,z)
D w=new D(1)
x.f2=w
x
```

```
D z=new D(0)
C x=new C(z,w)
D w=new D(1)
x
```

Final result

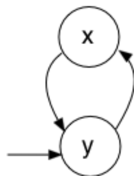



```
D z=new D(0)
C x=new C(z,w)
D w=new D(1)
x
```

Store can be recursive

```
class B { B f; ...}
```

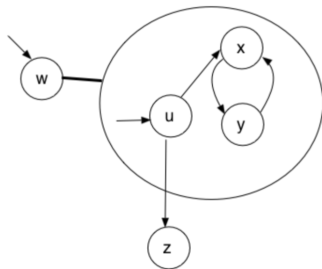
```
B x= new B(y)
B y= new B(x)
y
```



Store can be hierarchical

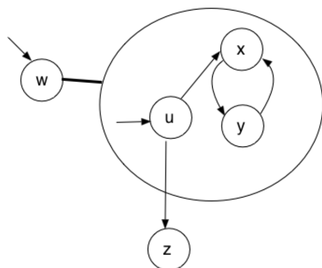
```
D z= new D(0)
A w= (
   B x= new B(y)
   B y= new B(x)
   A u= new A(x,z)
   u )
w
```

Store can be hierarchical




```
D z= new D(0)
A w= (
    B x= new B(y)
    B y= new B(x)
    A u= new A(x,z)
    u )
w
```

Store can be hierarchical




```
D z= new D(0)
A w= (
    B x= new B(y)
    B y= new B(x)
    A u= new A(x,z)
    u )
w
```

Advantage: constraints about aliasing are modeled at the syntax level, notably

- the object denoted by `y` can only be reached through `w`: the block construct
- the object denoted by `w` is not isolated: the block has free variables

Types and type contexts

cd	$::=$	$\texttt{class}\ C\ \{fds\ mds\}$	class declaration
fd	$::=$	$C\ f$	field declaration
md	$::=$	$T\ m\ \mu\ (T_1\,x_1, \ldots, T_n\,x_n)\ \{\texttt{return}\ e\}$	method declaration
e	$::=$	$x \mid e.f \mid e.m(es) \mid e.f = e' \mid \texttt{new}\ C(es) \mid (ds\ e)$	expression
d	$::=$	$T\ x = e$	variable declaration
T	$::=$	$\mu\ C$	type
μ	$::=$	$\texttt{capsule} \mid \texttt{lent} \mid \epsilon$	optional type modifier
Δ	$::=$	$\Gamma; xss$	type context
Γ	$::=$	$T_1\,x_1 \ldots T_n\,x_n$	type assignment

C with no modifier = standard type

Subtyping relation

$$\texttt{capsule}\, C \leq\ C \leq \texttt{lent}\, C$$

Subtyping relation

$$\texttt{capsule}\ C \leq\ C \leq \texttt{lent}\ C$$

$$\texttt{capsule}\ C \quad \overset{\textit{promotion}}{\Longleftarrow} \quad C$$

promotion can be used against the subtype hierarchy

Typing judgment

$$\Gamma; xs_1 \ldots xs_n \vdash e : T$$

expression e has type T under type assignment Γ

Typing judgment

$$\Gamma; xs_1 \ldots xs_n \vdash e : T$$

expression e has type T under type assignment Γ
$\mathtt{dom}^{\mathtt{std}}(\Gamma)$ is partitioned in groups $xs_0, xs_1, \ldots, xs_n$

Typing judgment

$$\Gamma; xs_1 \ldots xs_n \vdash e : T$$

expression e has type T under type assignment Γ
$\mathtt{dom}^{\mathtt{std}}(\Gamma)$ is partitioned in groups $xs_0, xs_1, \ldots, xs_n$
no aliasing is introduced among (portions of store reachable from) $xs_0, xs_1, \ldots, xs_n$

Typing judgment

$$\Gamma; xs_1 \ldots xs_n \vdash e : T$$

expression e has type T under type assignment Γ
$\mathtt{dom}^{\mathtt{std}}(\Gamma)$ is partitioned in groups $xs_0, xs_1, \ldots, xs_n$
no aliasing is introduced among (portions of store reachable from) $xs_0, xs_1, \ldots, xs_n$

variables in xs_0 can be freely used
variables in $xs_1, \ldots, xs_n$ are lent-locked = can only be used as `lent`

Typing judgment

$$\Gamma; xs_1 \ldots xs_n \vdash e : T$$

expression e has type T under type assignment Γ
$\mathtt{dom}^{\mathtt{std}}(\Gamma)$ is partitioned in groups $xs_0, xs_1, \ldots, xs_n$
no aliasing is introduced among (portions of store reachable from) $xs_0, xs_1, \ldots, xs_n$

variables in xs_0 can be freely used
variables in $xs_1, \ldots, xs_n$ are lent-locked = can only be used as `lent`

a group of lent-locked variables is introduced by promotion
a group can be unlocked by swapping

Typing rules (1)

$$(\text{T-PROM})\quad \frac{\Gamma; \mathit{xss}\ \mathit{xs} \vdash e :\ C}{\Gamma; \mathit{xss} \vdash e : \texttt{capsule}\ C} \quad \mathit{xs} = \texttt{dom}^{\texttt{std}}(\Gamma) \backslash \mathit{xss}$$

$$(\text{T-SWAP})\quad \frac{\Gamma; \mathit{xss}\ \mathit{xs}' \vdash e : \mu\ C}{\Gamma; \mathit{xss}\ \mathit{xs} \vdash e : \mu'\ C} \quad \begin{array}{l} \mathit{xs}' = \texttt{dom}^{\texttt{std}}(\Gamma) \backslash (\mathit{xss}\ \mathit{xs}) \\ \mu' = \begin{cases} \texttt{lent} & \text{if } \mu = \epsilon \\ \mu & \text{otherwise} \end{cases} \end{array}$$

Typing rules (2)

$$(\text{T-SUB})\quad \frac{\Delta \vdash e : T}{\Delta \vdash e : T'} \quad T \leq T'$$

$$(\text{T-VAR})\quad \frac{}{\Gamma; xss \vdash x : \mu'\, C} \quad \begin{array}{l} \Gamma(x) = \mu\, C \\ \mu' = \begin{cases} \texttt{lent} & \text{if } x \in xss \\ \mu & \text{otherwise} \end{cases} \end{array}$$

Typing rules (3)

$$(\textsc{t-field-access})\quad \frac{\Delta \vdash e : \mu\, C}{\Delta \vdash e.f : \mu\, C_i} \quad \begin{array}{l}\texttt{fields}(C) = C_1\ f_1 \ldots C_n\ f_n\\ f = f_i\end{array}$$

$$(\textsc{t-meth-call})\quad \frac{\Delta \vdash e_i : T_i \quad \forall i \in 0..n}{\Delta \vdash e_0.m(e_1, \ldots, e_n) : T} \quad \begin{array}{l}T_0 = \mu\, C\\ \texttt{mtype}(C, m) = \langle T, \mu, T_1 \ldots T_n\rangle\end{array}$$

$$(\textsc{t-field-assign})\quad \frac{\Delta \vdash e : C \quad \Delta \vdash e' : C_i}{\Delta \vdash e.f = e' : C_i} \quad \begin{array}{l}\texttt{fields}(C) = C_1\ f_1 \ldots C_n\ f_n\\ f = f_i\end{array}$$

$$(\textsc{t-new})\quad \frac{\Delta \vdash e_i : C_i \quad \forall i \in 1..n}{\Delta \vdash \texttt{new}\ C(e_1, \ldots, e_n) : C} \quad \texttt{fields}(C) = C_1\ f_1 \ldots C_n\ f_n$$

$$(\textsc{t-block})\quad \frac{\Gamma[\Gamma']; xss \vdash e_i : T_i\ \forall i \in 1..n \quad \Gamma[\Gamma']; xss \vdash e : T}{\Gamma; xss \vdash (T_1\ x_1 = e_1 \ldots T_n\ x_n = e_n\ e) : T} \quad \Gamma' = T_1\, x_1 \ldots T_n\, x_n$$

Promotion

How to introduce a `capsule`?

Promotion

How to introduce a `capsule`?

```
D z= new D(0)
capsule C x= (
  D y= new D(z.f+1)
  new C(y,y) )
x
```

Promotion

How to introduce a `capsule`?

```
D z= new D(0)                D z= new D(0)
capsule C x= (               capsule C x= (
  D y= new D(z.f+1)   -->*     D y= new D(1)
  new C(y,y) )                 new C(y,y) )
x                            x
```

Counterexample

```
D z= new D(0)
capsule C x= (   //ill-typed
  D y= z
  new C(y,y) )
x
```

Counterexample

```
D z= new D(0)
capsule C x= (   //ill-typed
  D y= z
  new C(y,y) )
x
```

$\longrightarrow$

```
D z= new D(0)
C x= new C(z,z)
x
```

Swapping

How to modify (the object denoted by) a `lent` reference?

```
lent D z= new D(0)
z.f=z.f+1
```

the singleton group `z` is swapped with the empty set

Swapping to achieve promotion

```
D z= new D(0)
capsule C x= (
  D y= new D(z.f=z.f+1)
  new C(y,y) )
x
```

Swapping to achieve promotion

```
D z= new D(0)
capsule C x= (
  D y= new D(z.f=z.f+1)
  new C(y,y) )
x
```

$\longrightarrow^{\star}$

```
D z= new D(1)
capsule C x= (
  D y= new D(1)
  new C(y,y) )
x
```

A programming example

Example input file:

```
Bob
1 500 2 1300
Mark
42 8 99 100
```

Can we use a conventional Java Scanner to read the file and obtain new data as `capsule`?

```
class Reader {
  static capsule Customer readCustomer(lent Scanner s){
    Customer c=new Customer(s.nextLine())
    while(s.hasNextNum() ){
      c.addShopHistory(s.nextNum())
      }
    return c //ok, capsule promotion here
    }
  }
```

A programming example

```
class Reader {...//as before
  static capsule Customer updateCustomer(
                      capsule Customer old,lent Scanner s){
    Customer c=old//capsule 'old' is opened
    while(s.hasNextNum() ){
      c.addShopHistory(s.nextNum())
      }
    return c //ok, capsule promotion here
    }
  }
```

Results

- Soundness
  If $\vdash e$, and $e \longrightarrow^{\star} e'$, then either e' is a value, or $e' \longrightarrow$

Results

- Soundness
  If $\vdash e$, and $e \longrightarrow^{\star} e'$, then either e' is a value, or $e' \longrightarrow$
- Capsule has the expected behaviour
  If $\vdash \mathcal{E}[e]$, $\Gamma = \texttt{typectx}(\mathcal{E})$,
  $\Gamma; \emptyset \vdash e : \texttt{capsule}\ C$, and $\mathcal{E}[e] \longrightarrow^{\star} \mathcal{E}'[v]$,
  then v is closed

Conclusion

- we believe that the novel pure setting has a great potential of really achieving a better understanding of aliasing

Conclusion

- we believe that the novel pure setting has a great potential of really achieving a better understanding of aliasing
- further work: simplify calculus and proof technique
- express and formally verify other properties of object graphs studied in literature on ownership, e.g., immutability

Conclusion

- we believe that the novel pure setting has a great potential of really achieving a better understanding of aliasing
- further work: simplify calculus and proof technique
- express and formally verify other properties of object graphs studied in literature on ownership, e.g., immutability
- long term goal: Hoare-like logic for local reasoning, as in separation logic

Thanks