Ownership Types Coming of Age

Sophia Drossopoulou,
SFM 2015, Bertinoro

J. Noble, J, Vitek, J. M. Potter,: Flexible Alias Protection, ECOOP 98

D. Clarke, J. M. Potter, J. Noble: Ownership Types for Flexible Alias Protection, OOPSLA'98,

D. Clarke, S. Drossopoulou: Ownership, Encapsulation and Disjointness of Types and Effects, OOPSLA'02

K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. ECOOP 2004.

Wrigstad and Clarke. External Uniqueness is Unique Enough. ECOOP 2004

R. L. Bocchino, V.S. Adve, D.Dig, S. V. Adve, S. Heumann, A Type and Effect System for Deterministic Parallel Java, OOPSLA 2009

E. Cohen, M. Moskal, W. Schulte, S. Tobies. Local Verification of Global Invariants in Concurrent Programs. CAV 2010

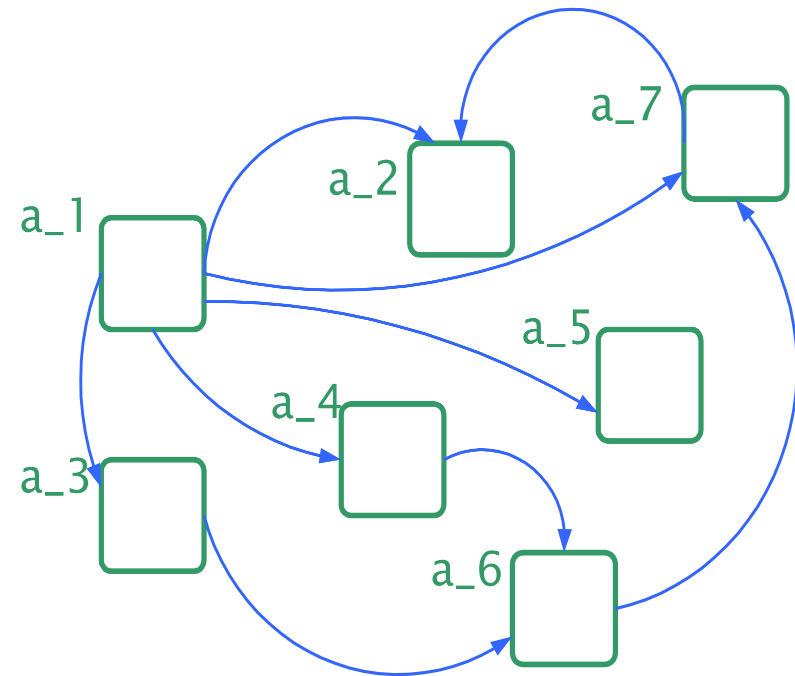
C. S. Gordon, M. J. Parkinson, J.Parsons, A. Bromfield, and J.Duffy, Uniqueness and Reference Immutability for Safe Parallelism, OOPSLA 2012

…

also C.A.R. Hoare, He Jifeng, B. Liskov, M. Rinard, J. Vitek, J. Aldrich, M. Schwarzbach, J Palsberg, A. Milanova, A. Routlev, P. S. Almeida, B. Bokowski, J. Boyland, M. Felleisen, W. DietlA. Summers, … etc

The problem: Pointer Spaghetti

Consider following heap (boxes indicate objects at address a_1, … a_6; arrows indicate fields pointing to objects):



Assume that we establish some property, e.g. PROP(a_2). Then a_4 executes a method with body code_0.

Is PROP(a_2) preserved? Not necessarily, because .

Does it matter?

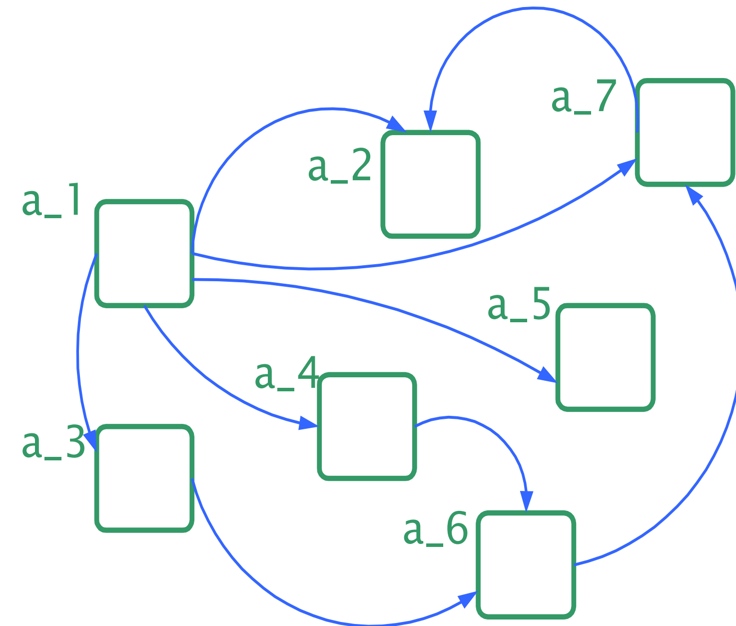
Pointers introduce aliasing. Aliasing makes reasoning about programs difficult; evolution of aliasing makes it even more so.

However, if we were sure that

,

then we would also know that

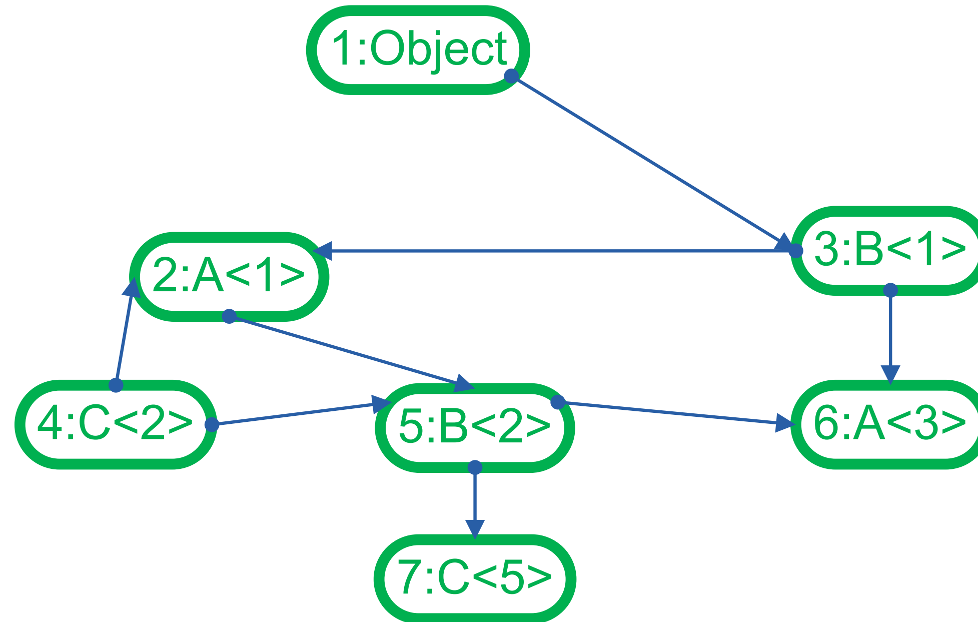
PROP(a_2) will be preserved.



A Solution:
Restrict/Characterize Pointer Spaghetti /Aliasing through Ownership Types

- Every object is owned by one other object - usually
- Ownership is acyclic
- Ownership therefore effectively organizes objects into a tree - usually

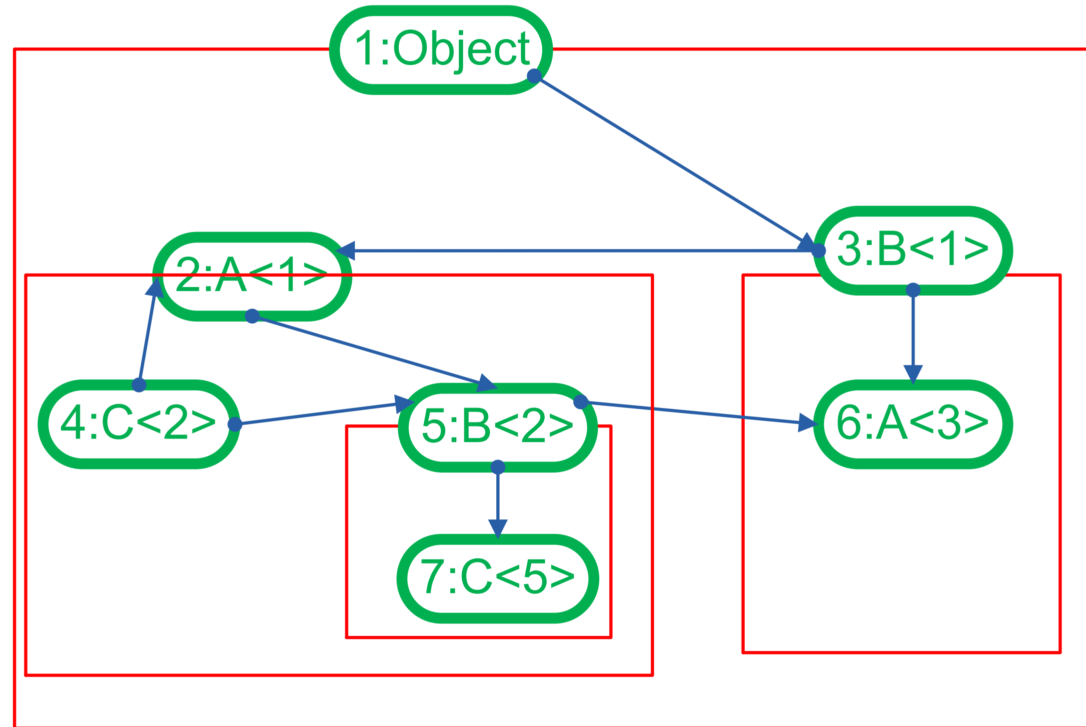
Ownership - diagrammatically



The green, rounded boxes are objects. Their contents describe their address, their class, and their owner. Eg, 4:C<2> is an object at address 4, of class C, owned by 2.

The blue arrows indicate field references.

Ownership – diagrammatically - boxes

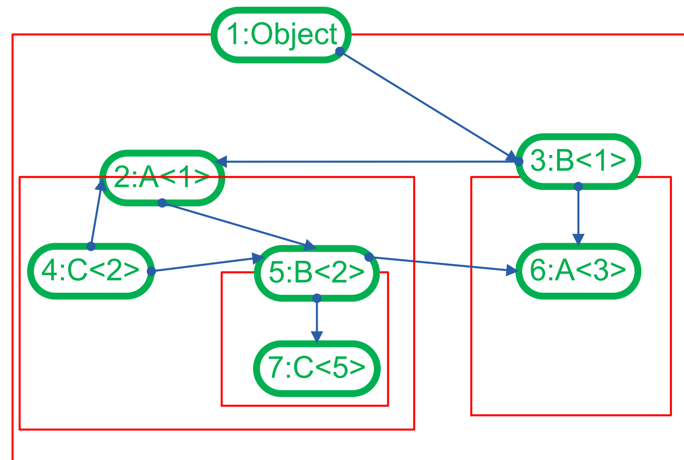


The green, rounded boxes are objects. Their contents describe their address, their class, and their owner. Eg, 4:C<2> is an object at address 4, of class C, owned by 2.

The blue arrows indicate field references.

We indicate ownership through the red boxes.

How can we use ownersip?



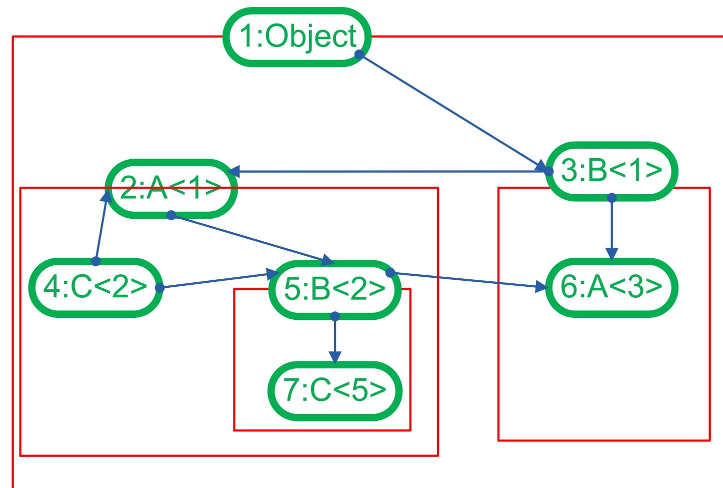
We can use the boxes to describe a part of the heap, without even knowing exactly which objects belong in it.
For example, in the particular heap, we have

2.owned={ 4, 5 }   2.owned*={ 4, 5, 7 } 3.owned = { 6 }

In general, we know that if x and y are not aliases, and neither owns the other, then

$$x.owned^* \cap y.owned^* = \emptyset$$

How can we use ownersip? - 2



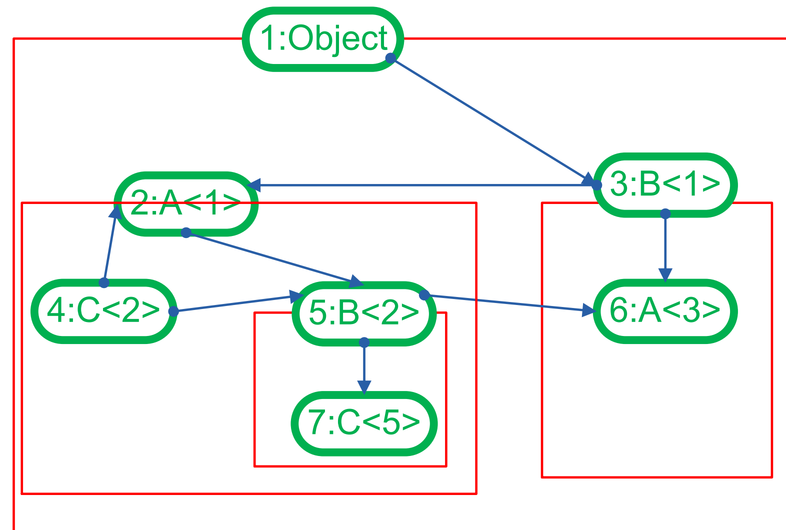
We can use ownership to describe the effect of a computation, and then use this knowledge for, among other things,

- Concurrency without races
- Parallelism without locking
- Program Transformations
- Object Initialization
- Object Cloning
- Garbage Collection
- Sharing without Copying
- Reasoning about Programs

Object owning another Object

We assume a mapping owner: addr → addr,

For example, given the objects from below, we have



that owner(2)=1, owner(7)=5.

How can we express ownership?

Class declarations come with ownership parameters (as generics), which characterize the owner of the object, and the owner of objects pointed at by fields.

```
class C<p1,p2,…p_m>{    …    }
```

where m>= 1

Types have ownership arguments (as in generics)

```
C<q1,p2,…q_m>
```

where
- m is the number of ownership parameters for class C,
- q_i ∈ { this, root, p_1, … p_n}
- p_j are the ownership parameters of the class (D) containing the type C<q1,p2,…q_m>.

Eg

```
C<this,p3,root,p3>
```

Example of Ownership Types

```
class A<p1>{
 // p1: owner of the current A object

    B<this> aB;
    // aB is owned by the current A object              }

class B<p1>{
        // p1: owner of the current B object

    B<p1> f1;
    // f1  is owned by the  owner of the current object

    B<this> f2;
    // f2  is owned by the  current object

    B<root> f3;
    // f1  is owned by root
    }
```

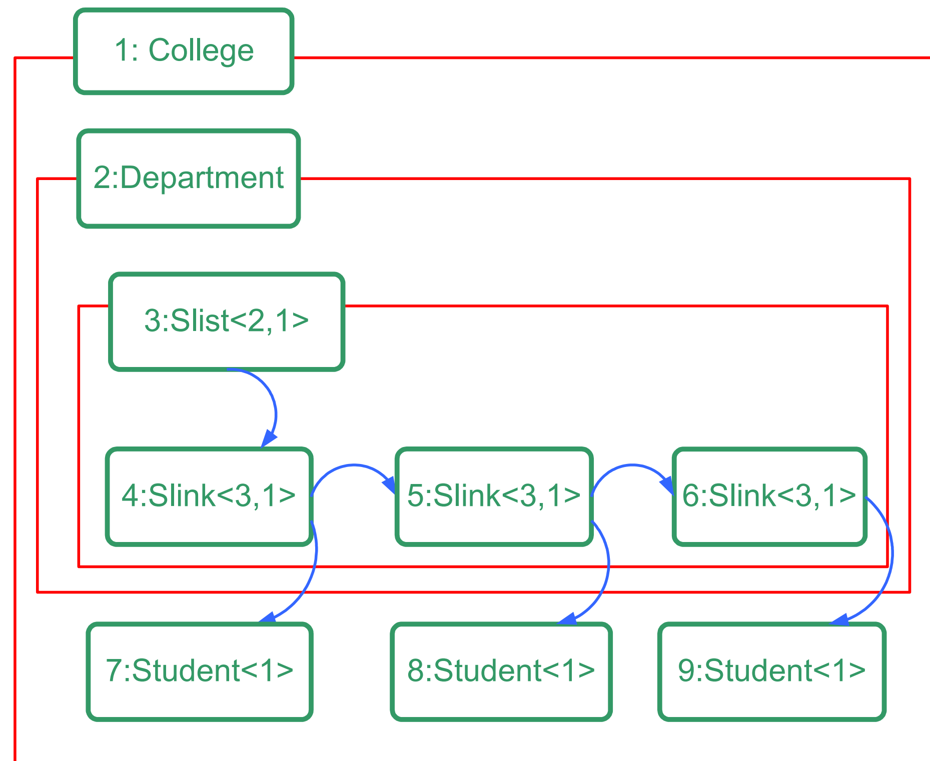
Then, declaration `A<root> anA;` allows for ownership boxes:

Now, consider lists of students

```
class SList<p1,p2>{
 // p1: owner of list, p2: owner of students
    Slink<this,p2> first;
    // the list owns the first link
}
class SLink<p1,p2>{
 // p1: owner of link, p2: owner of students
    Slink<p1,p2> next;
    // p1 owns the next link
    Student< p2> s;
    // p2 owns the student
}
```

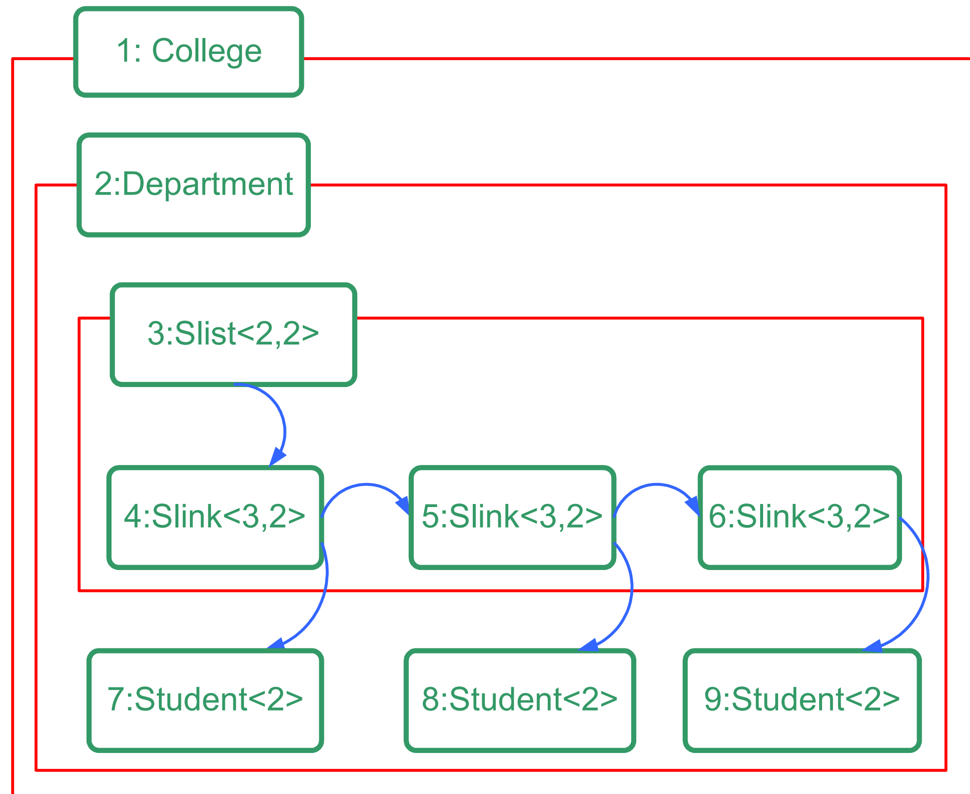
What do the types `SList<o2,o1>` and `SList<o2,o2>` describe?

Then, for objects o1 and o2, where o1 owns o2, the type `SList<o2,o1>` describes



Thus,
- 3 "owns" 4, 5 and 6,
- 2 "owns" 3,
- 1 "owns" 2, 7, 8, 9.

On the other hand, the type `SList<o2,o2>` would be



Thus,
- 3 "owns" 4, 5 and 6,
- 2 "owns" 3, 7, 8, 9.
- 1 "owns" 2.

Consider, for example, the case where departments have control over their students' data, but service departments teach students together with others, and share control over students' data:

```
class Department<p1>{
 // p1: owner of the department
    SList<this,this> ownedStList;
    // the department owns the students
}
class ServDepartm<p1>{
 // p1: owner of the service department
    SList<this,p1> sharedStList;
    // the owner of the service department owns the students
}
```

And now, a College has `doc` and `ee` departments, `dramSoc` a service department, and `visitors` an external department:

```
class College<p1>{
 // p1: owner of the college

    Department<this> doc, ee;
    // the college owns doc and ee

    ServDepartm<this> dramSoc;
    // the college owns dramSoc

    Department<p1> visitors;
    // the owner of the college owns visitors
}
```

Space for ownership diagram

Simple Type rules for Ownership Types

Example

Given variables

```
College<o>    collg1;
College<this> collg2;
```

What should be the type of

```
collg1.doc;
collg1.visitors;
collg2.doc;
collg1.visitors;
```

Well-formed classes & types, type for field read

```
class C<p1,…pn>{   … D<q1,…,qm> f … }
```

well formed iff ???

```
C<q1,q2, …qn>
```
is well-formed iff ???

```
e: C<q1, …,qn>
class C<p1,…pn> {  … D<r1, …. rm> f …. }
--------------------------------------------
e.f : D< ??? >
```

Well-formed classes & types, type for field read - 2

class C<p1,…pn>{ … D<q1,….qm> f … }

well formed iff q1,…qm ∈ { this, root, p1, … pn}

D was declared with m parameters

C<q1,…qm> well formed iff

q1,…qm ∈ { this, root, p1, … pn}

p1,…pm are in scope

C was declared with m parameters

$$\frac{e: C\langle q1, \ldots, qn\rangle \quad q1, \ldots qn \neq this \qquad \textbf{class}\ C\langle p1, \ldots pn\rangle\ \{\ \ldots\ D\langle r1, \ldots, rm\rangle\ f\ \ldots.\ \}}{e.f : D\langle r1, \ldots, rm[q1, \ldots qn/p1, \ldots, pn]\rangle}$$

Objects conforming to types

In a simple object-oriented language, we have

Heap = Address -> Object

Object = ClassId x (FieldId -> Address)

Weak conformance

$$\frac{\chi(a) = (C, \ldots)}{\chi \vdash_w a : C}$$

Strong conformance

$$\frac{\chi(a) = (C, \ldots) \qquad \text{for all } f \text{ defined in } C;\ \text{if } type(C,f)=D \text{ then } \chi \vdash_w \chi(a,f) : D}{\chi \vdash a : C}$$

extend the above to cover ownership types

Objects conforming to types

In a simple object-oriented language, we have

Heap = Address -> Object

Object = ClassId x **Address*** x (FieldId -> Address)

Weak conformance

$$\frac{\chi(a) = (C,\ a1,\ldots an,\ \ldots)}{\chi \ \vdash_w a \ : \ C\langle a1,\ldots.an\rangle}$$

Strong conformance

$$\frac{\begin{array}{l} \chi(a) = (C, a1,\ldots an,\ \ldots) \\ \text{for all } f \text{ defined in } C; \\ \quad \text{if } type(C\langle p1,\ldots pn\rangle, f) = D\langle q1,\ldots qm\rangle \\ \quad \text{then } \chi \ \vdash_w \chi(a,f) \ : \ D\langle q1,\ldots qm[a1,\ldots an/p1\ldots.pn][a/this]\rangle \end{array}}{\chi \vdash a : \ C\langle a1,\ldots an\rangle}$$

Runtime Representation of Objects

Which of the components below are necessary for program execution?

Heap = Address -> Object

Object = ClassId x Address* x (FieldId -> Address)