Object inlining

Peeter Laud
Universität des Saarlandes

supported by the Esprit LTR project #28198
JAVA and CoSy Technology for Embedded Systems (JOSES)

What is object inlining?

- Uniform representation of data:
  $\Rightarrow$ Pointer to the heap, pointing to actual data.

- Aggregated representation gives
  - less pointer dereferences;
  - less objects;
  - keeps (possibly) related objects together in memory.

  But has different semantics!

We want: automatically detect aggregation possibilities.
$\Rightarrow$ Must be semantics-preserving!!!

Structure of the talk

- (Informal) semantics for the heap.
- Analysis, based on this semantics.
- Representing the objects in memory.
- Further improvements (if I have time).
  - Details of interprocedural analysis.
  - Constant objects.
- Conclusions.

Definition: semantics of the memory

- Local and global variables

  ⇒ contain references to objects.

- An object is just a set of fields.

- A field may be

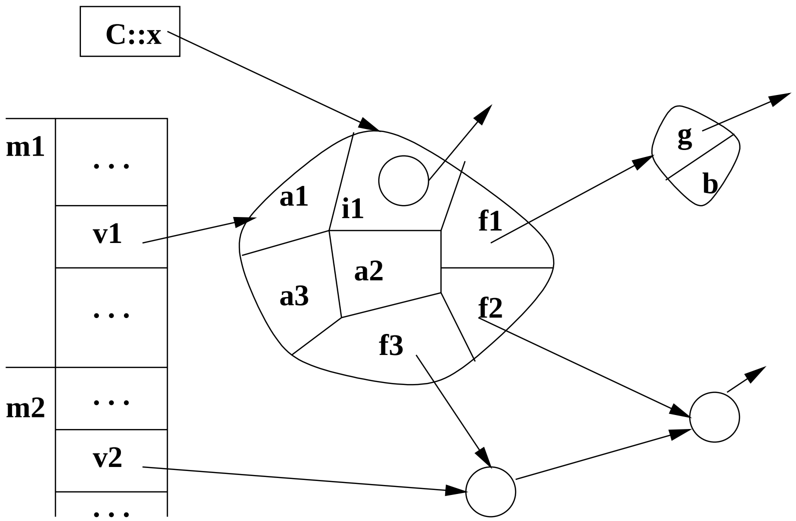
  - an atomic value (`a1`, `a2`, `a3`)
  - a reference (`f1`, `f2`, `f3`)

    ⇒ contains a reference to an object.

  - inlined (`i1`)

    ⇒ contains another object.

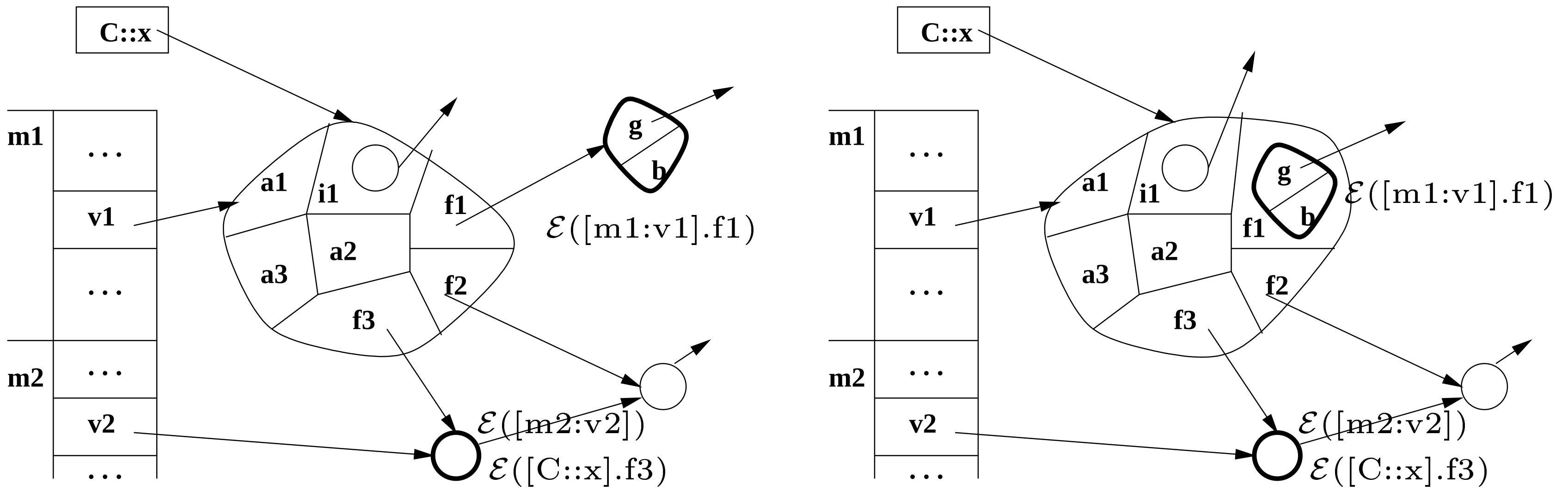
    Inlined objects may not be referenced from reference fields.



The semantics of statements is defined obviously.

Definition: semantics-preserving

- A pointer chain is either
  - a reference variable; or
  - a pointer chain, followed by a reference or inlined field.
- The heap defines a function $\mathcal{E}$ from pointer chains to objects

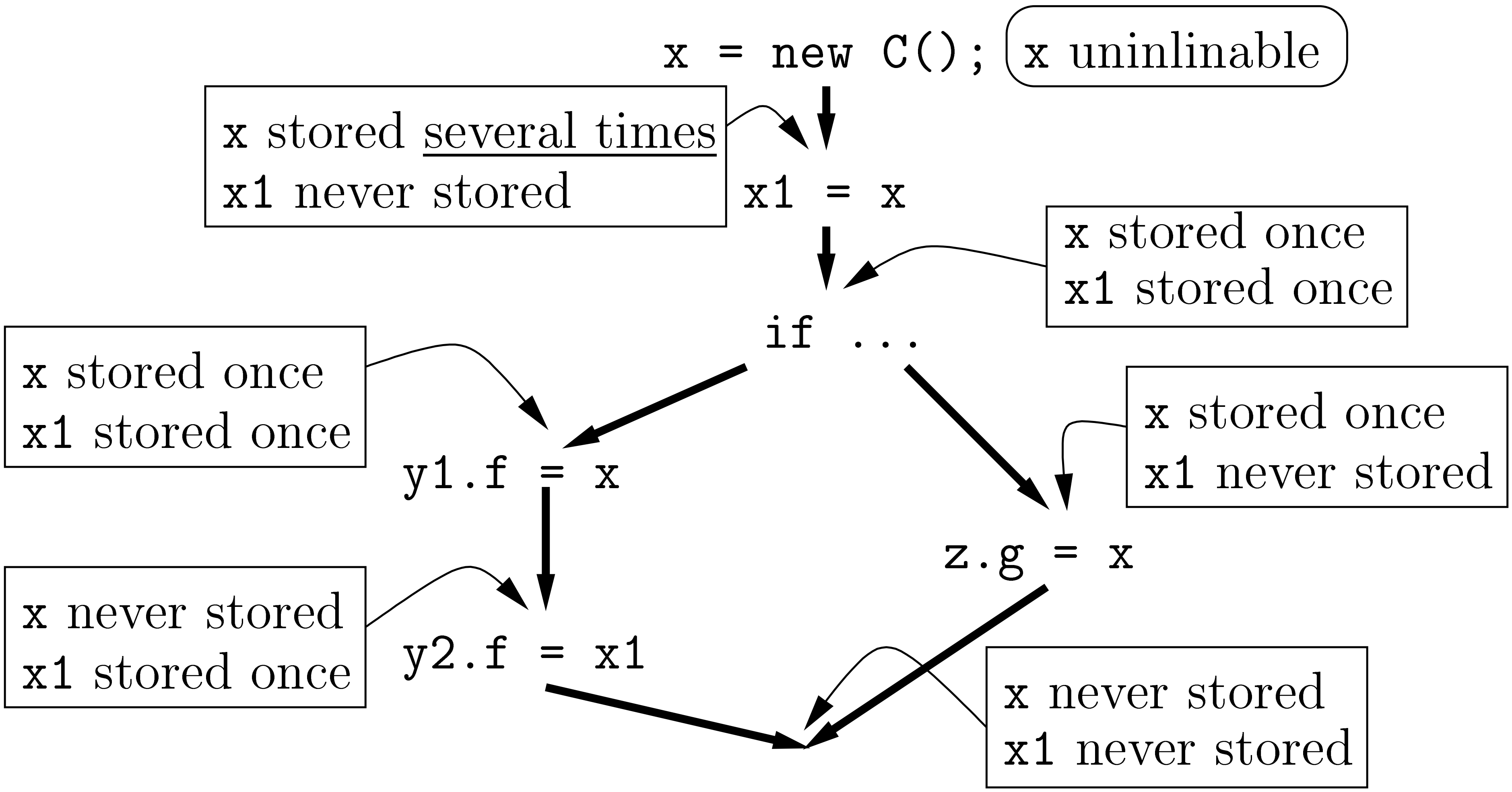


A transformation is semantics preserving, if it does not change Ker $\mathcal{E}$.
$\Rightarrow$ "semantics-preserving" = "does not change sharing patterns".

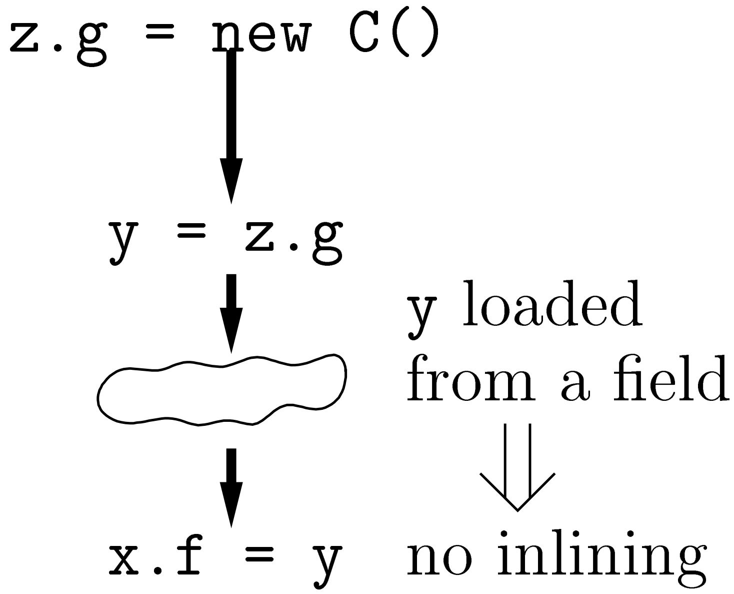
The analysis should tell us:

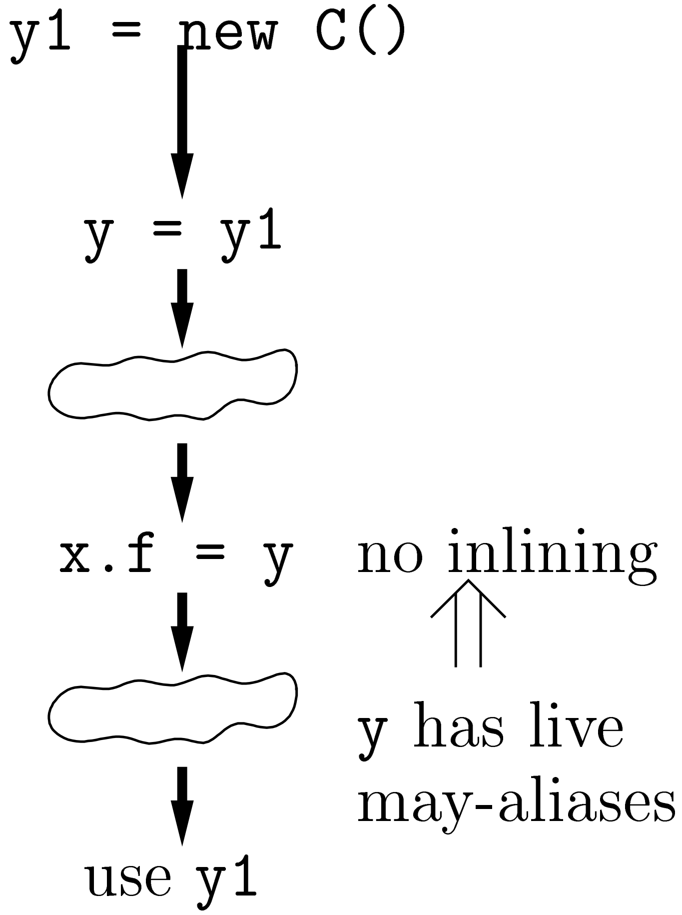
- For the reference fields:
  ⇒ Couldn't it be an inlined field instead?
  result — a set of pairs $\langle$creation point, reference field$\rangle$
- For the creation points of objects (statements `x = new C(...)`):
  ⇒ is the created object going to be inlined?
- For field stores (assignments `x.f = y`):
  ⇒ can inlining occur at this place?
  Assignment can be either reference assignment or deep copy.

An inlined object may be stored only once



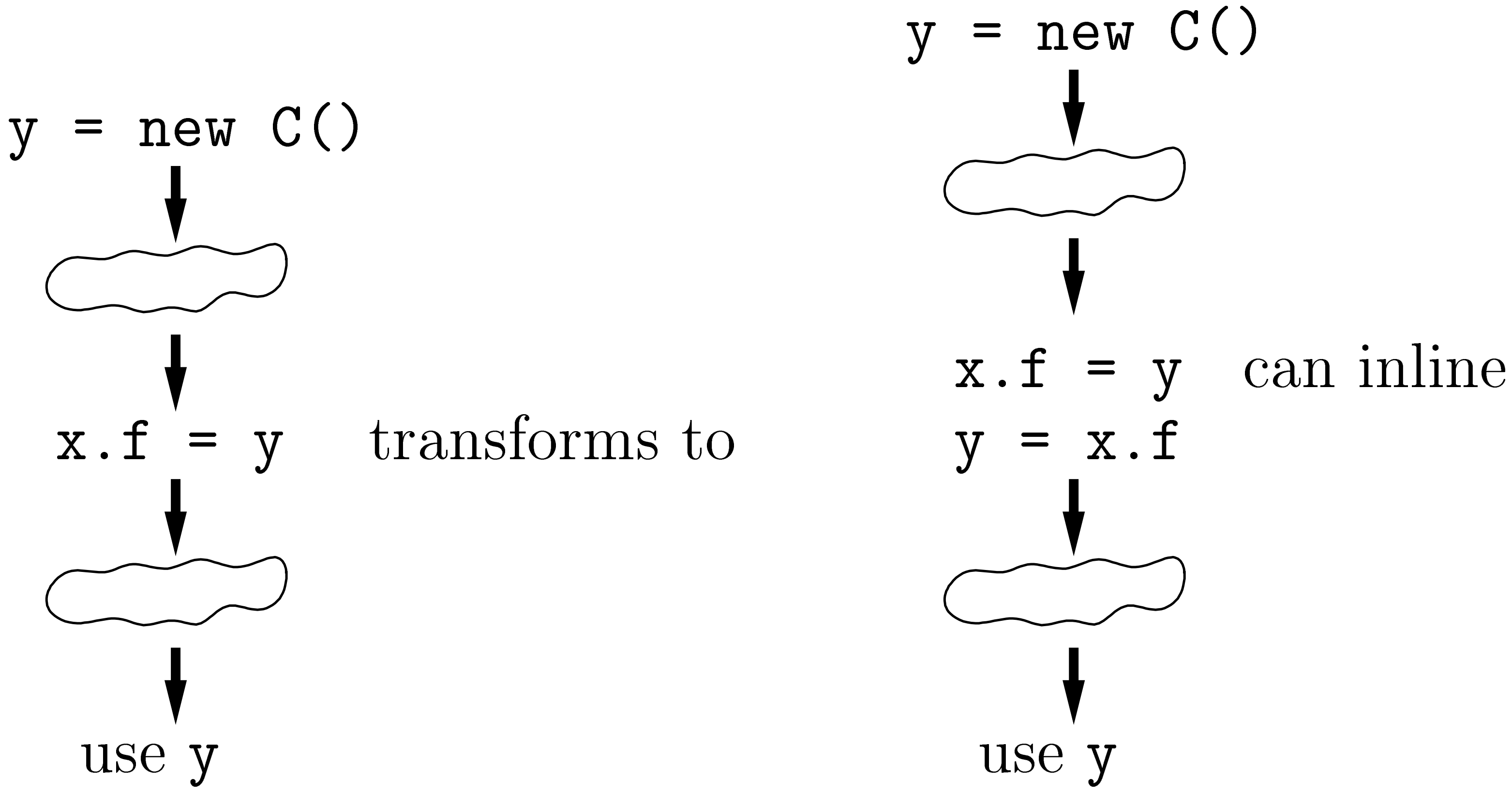
An object cannot exist in several copies (1/2)





may-alias analysis for
<u>object variables</u> (only)

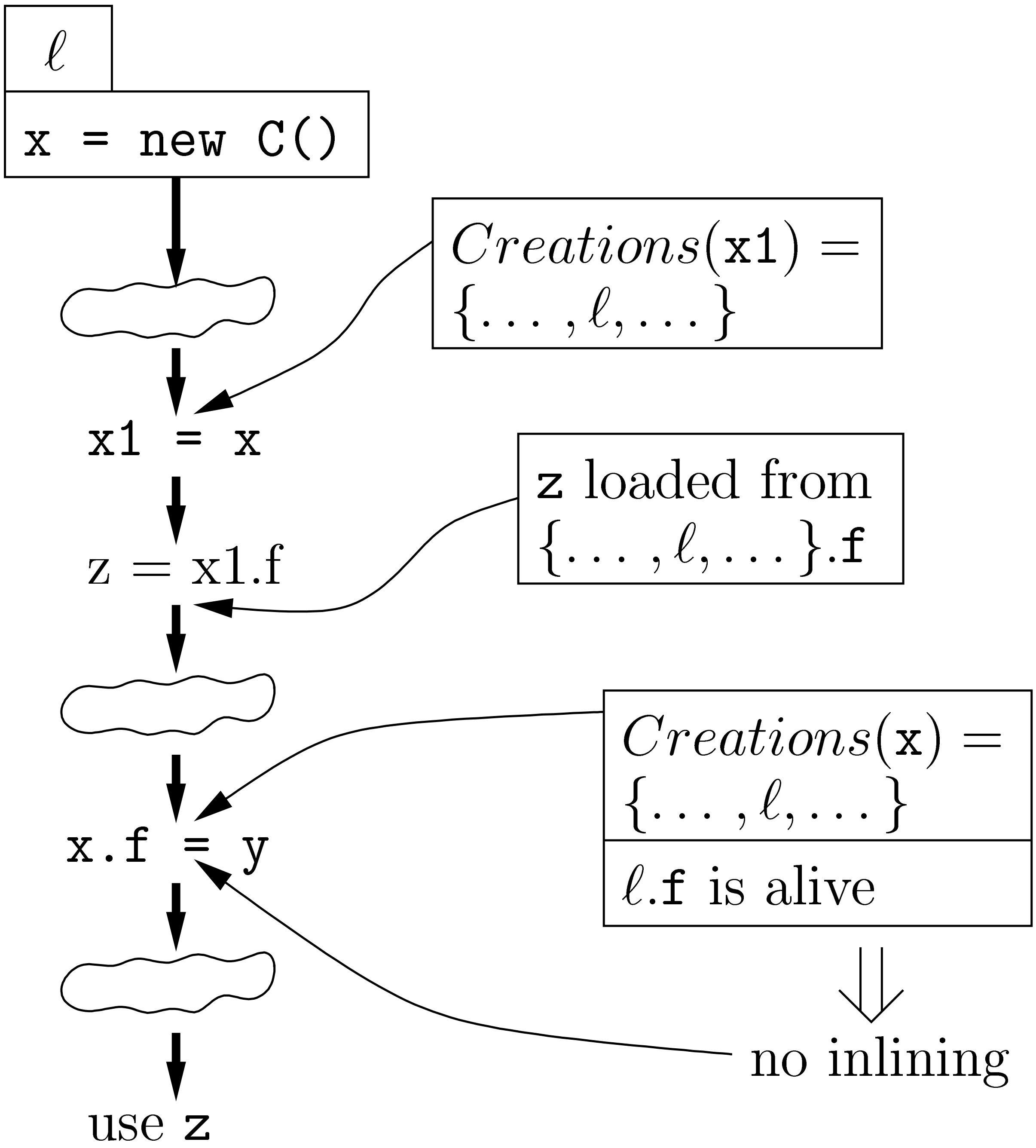
An object cannot exist in several copies (2/2)



Differences from previous slide:

- may-alias vs. must-alias (of `y` and `y1`)
- visibility of `y1`

An inlined field cannot contain several objects



Objects are not just sets of fields ...

- Object is a sequence of fields.
- Each field has a size.
- Type inference tells for each field:
  - Where are those objects created, that it points to?

$$size(\mathtt{f}) = \begin{cases} \text{predefined}, & \mathtt{f} \text{ is atomic or reference} \\ \max\limits_{\mathtt{o} \in Creations(\mathtt{f})} size(\mathtt{o}), & \mathtt{f} \text{ is inlined} \end{cases}$$

$$size(\mathtt{o}) = \sum_{\mathtt{f} \in Fields(\mathtt{o})} size(\mathtt{f}) + \langle\text{overhead}\rangle$$

Thus it is not allowed to inline an object into its own field.

ℓ_1 :   `x = new C();`

   $\dots$

ℓ_2 :   `x.f = new D1();`

   $\dots$

ℓ_3 :   `x.f = new D2();`

$\ell_2, \ell_3 \in Creations(\ell_1.\mathtt{f})$

$\mathsf{inlined}(\ell_1, \mathtt{f}) \Rightarrow \mathsf{compatible}_{(\ell_1, \mathtt{f})}(\ell_2, \ell_3)$

$\mathsf{compatible}$ may mean

- true
- do not need dynamic dispatch to distinguish
- same class

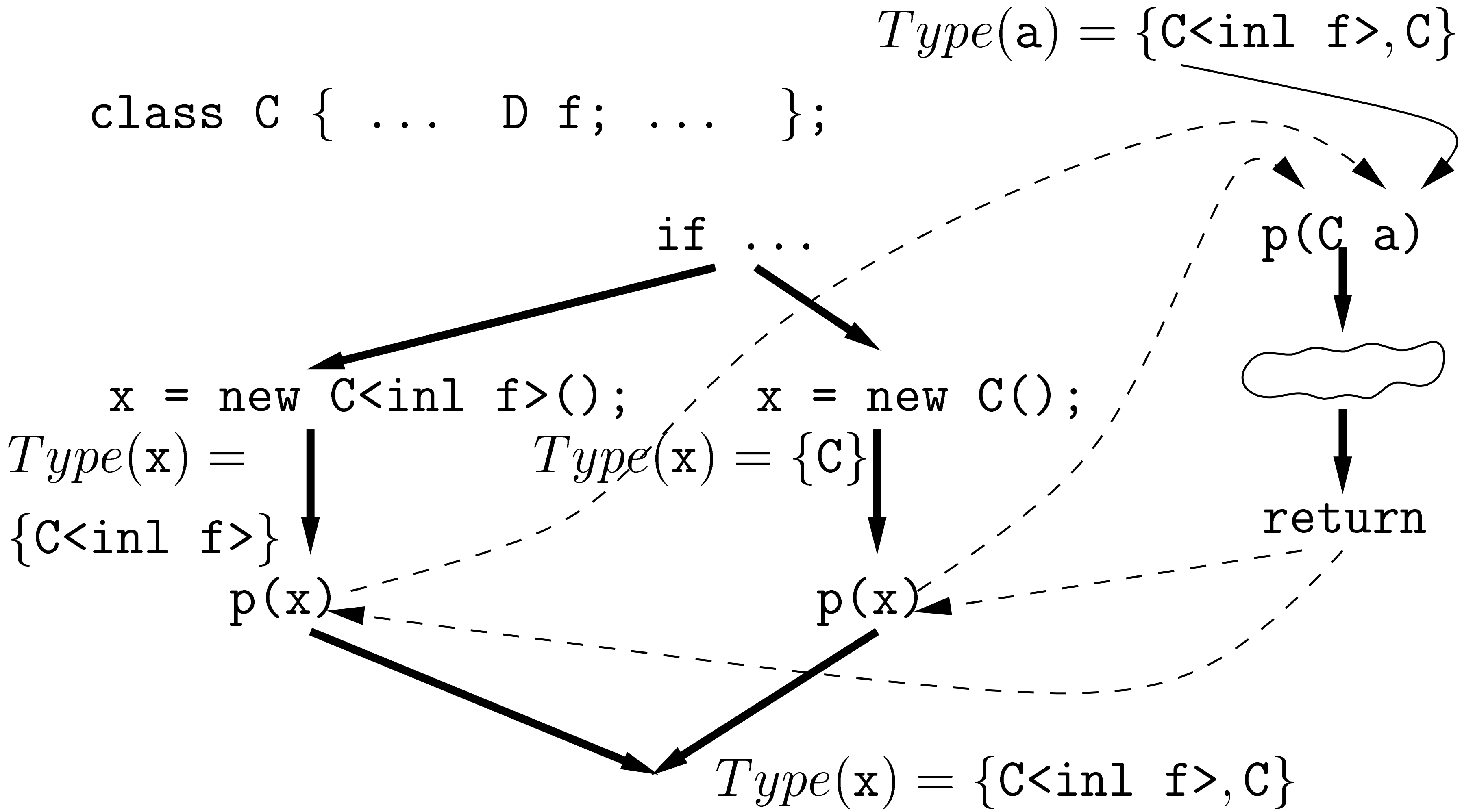
Accessing objects in memory

- At which offset is the field `f` in expression `x.f`?
- Is a field store `x.f = y` a reference assignment or a deep copy?

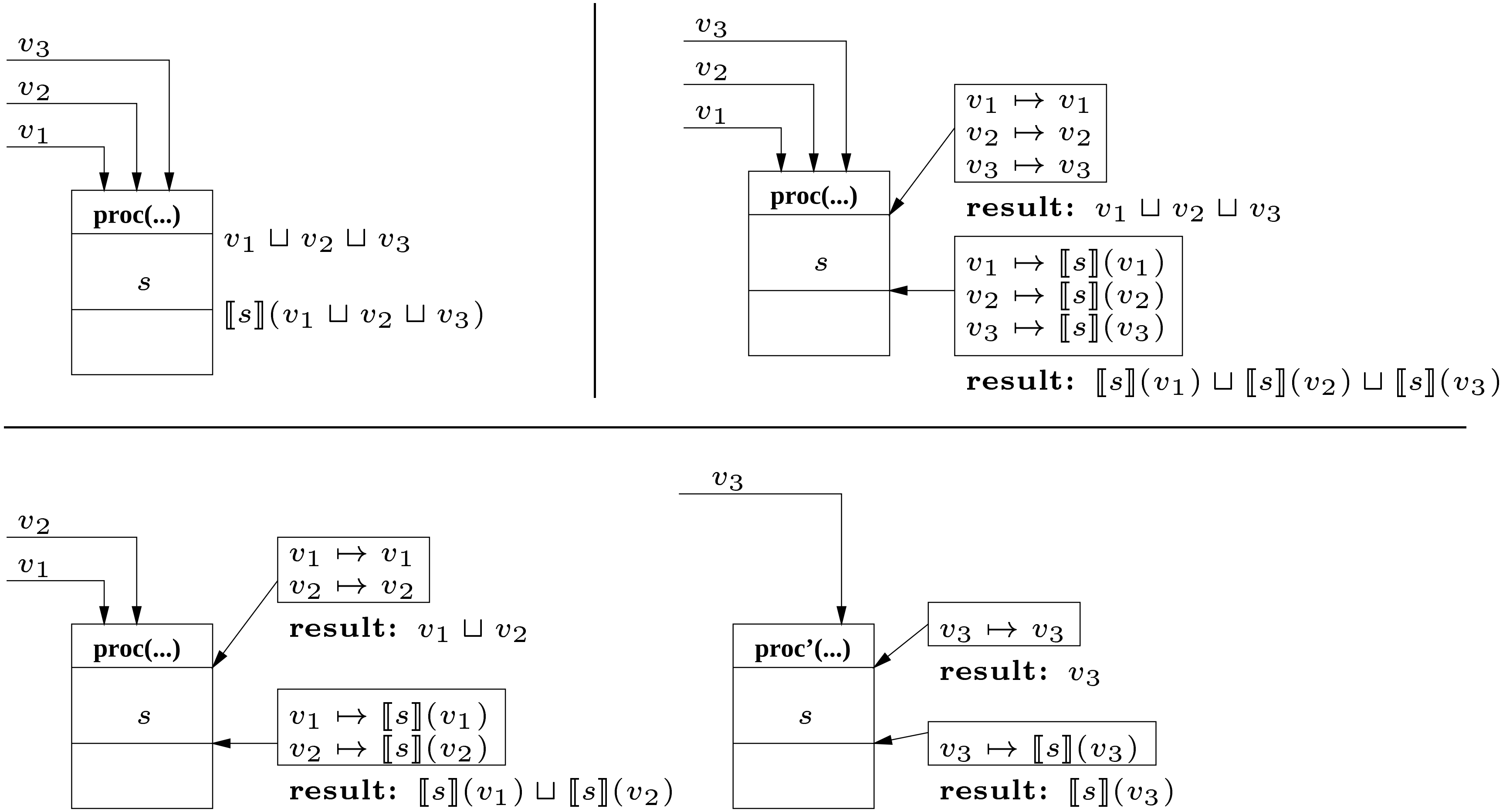
May be decided at runtime, but this is inefficient.

We are going to use existing techniques (J. Dolby & A. Chien, PLDI'97, OOPSLA'98, PLDI'00).

- A data flow analysis records
  - for objects their creation points;
  - for object variables, from which fields or object creations they have been flown.
- The results of this DFA are examined for incompatibilities.
- Incompatibilities are resolved by outlining some fields or by cloning.



$Type$(a) = {C<inl f>, C}
class C { ... D f; ... };
if ...
p(C a)
x = new C<inl f>();
x = new C();
$Type$(x) =
{C<inl f>}
$Type$(x) = {C}
return
p(x)
p(x)
$Type$(x) = {C<inl f>, C}



v_3
v_2
v_1
proc(...)
s
$v_1 \sqcup v_2 \sqcup v_3$
$[\![s]\!](v_1 \sqcup v_2 \sqcup v_3)$
v_3
v_2
v_1
proc(...)
s
$v_1 \mapsto v_1$
$v_2 \mapsto v_2$
$v_3 \mapsto v_3$
result: $v_1 \sqcup v_2 \sqcup v_3$
$v_1 \mapsto [\![s]\!](v_1)$
$v_2 \mapsto [\![s]\!](v_2)$
$v_3 \mapsto [\![s]\!](v_3)$
result: $[\![s]\!](v_1) \sqcup [\![s]\!](v_2) \sqcup [\![s]\!](v_3)$
v_2
v_1
proc(...)
s
$v_1 \mapsto v_1$
$v_2 \mapsto v_2$
result: $v_1 \sqcup v_2$
$v_1 \mapsto [\![s]\!](v_1)$
$v_2 \mapsto [\![s]\!](v_2)$
result: $[\![s]\!](v_1) \sqcup [\![s]\!](v_2)$
v_3
proc'(...)
s
$v_3 \mapsto v_3$
result: v_3
$v_3 \mapsto [\![s]\!](v_3)$
result: $[\![s]\!](v_3)$

Interprocedural analysis — effect calculation

a.k.a. Functional approach *a la* Sharir & Pnueli.

Interproc. control flow graph does not reflect that each method returns to the same point that it was called from.

- Let the analysis assign an element of the (upper) semilattice $\mathcal{L}$ to each program point.
- Effect calculation analyses the procedure once for each call context.
  $\Rightarrow$ Corresponds to replacing $\mathcal{L}$ with $\mathcal{L} \longrightarrow \mathcal{L}$.
- For bit-vector analyses:
  - $\mathcal{L}$ has a (quite) small set of generators $\mathcal{B}$.
  - The result of effect calculation $A : \mathcal{L} \longrightarrow \mathcal{L}$ is an upper semilattice homomorphism.

Cloning

- Polymorphism / code reuse makes program analysis and transformation harder.
- Solution — reduce polymorphism.

Let a transformation of a method `m` be valid for calling context generators $\mathcal{B}_{\text{good}}$ and invalid for $\mathcal{B}_{\text{bad}}$. For cloning

- create a copy of `m` called `m_good`;
- if $\mathit{Contexts}(\texttt{call m}) \subseteq \langle \mathcal{B}_{\text{good}} \rangle$ then replace `call m` by `call m_good`;
- if $\mathit{Contexts}(\texttt{call m})$ contain both good and bad contexts:
  $\Rightarrow$ clone the method containing `call m`.

Constant objects

- A constant object is not changed after it has been stored.
- Constant objects may be inlined, even if this is not semantics-preserving.
  $\Rightarrow$ As they are only read, it does not matter how many copies of them exist.

ℓ:
```
z.g = new C();
x = z.g;
 ...
x.f = y;
```

Object created at ℓ is modified after being loaded
$\Rightarrow$ non-constant

Conclusions and open problems

I wanted to convince you, that

- A language with uniform object model is no less efficient than one with explicit aggregation.
- The cost of object inlining analysis is not prohibitive.

Open questions:

- Arrays.
- Multithreading.

Implementation status

Bit-vector analyses	yes
Type inference	yes
Cloning	yes
Constant objects	no
Integration of analyses based on set-based semantics	no
Representing objects in memory	no
Program transformation	no