CS 565, Jens Palsberg Oct 24, 1998

# Lecture Note 6: Flow Analysis

## Contents


**1 Motivation** **2**

**2 Flow Constraints** **4**

**3 The Constraint Solver** **11**

**4 Quality of Information** **14**

## 1 Motivation

Motivation: optimizing compilers and programming environments.

Flow information is a finite set of classes.

- Suppose the flow set for the expression E is {A,B,C}.
- Invariant: E can only evaluate to `null` or an instance of a class mentioned in {A,B,C}.
- We can inline a message send if the flow set for the receiver is a singleton set.
- We can use a switch statement to dispatch messages if the flow set for the receiver contains a reasonably small number of elements.
- We can discard "dead code", e.g., implementations of m in classes other than A, B, and C in the example below.
- To compute flow sets we will do flow analysis.

$$\begin{array}{ll} & \mathrm{E} \quad . \ \mathrm{m}(\ \ldots\ ) \\ & \uparrow \\ \{\mathrm{A,B,C}\} & \end{array}$$

# Flow Analysis Algorithms

- Flow analysis by
  - *solving* constraints, with a
  - fast iterative algorithm.
- Improvements by:
  - trace graph and
  - code copying.
- Useful for both statically-typed and dynamically-typed languages.

# World-Assumptions

- The *closed-world* assumption:

  All parts of the program are known at compile-time. A compile-time analysis is for the whole program. Example: flow analysis like the ones to be presented in this note.

- The *open-world* assumption:

  Some parts of the program are unknown at compile-time. A compile-time analysis is local for each fragment of the program. Example: inference of principal types.

## 2 Flow Constraints

- Assume that all program variable and argument names are distinct (rename if necessary).
- For a program, there will be a family of flow variables.
- Use the notation: $[\![\texttt{x}]\!]$, $[\![\texttt{f}]\!]$, $[\![\texttt{E}]\!]$ for the flow variables for the instance variable `x`, the formal argument `f`, and the expression `E`.
- Use the notation $[\![\texttt{this-C}]\!]$ for the flow variable for "this" in the class `C`.
- Generate constraints from the syntax of expressions.
- Any flow analysis must be approximative.
- Too large sets are sound and conservative, but very large sets are useless.
- How precise information can we get? How fast can we compute it?
- We will work with flow constraints of three forms:

$$\begin{array}{rcll} c & \in & X & \textit{start constraints} \\ X & \subseteq & Y & \textit{propagation constraints} \\ (c \in X) \Rightarrow (Y & \subseteq & Z) & \textit{conditional constraints} \end{array}$$

  There is always a unique minimal solution.

# Flow Constraints

12) "ID = EXP". For the assignment we have:

$$[\![\text{EXP}]\!] \subseteq [\![\text{ID}]\!]$$

16) "`this`". If `this` occurs in a method in class C, then we have

$$\texttt{C} \in [\![\texttt{this-C}]\!]$$

17) "`new C()`". For object creation we have

$$\texttt{C} \in [\![\texttt{new C()}]\!]$$

21) "EXP . ID ( $\text{EXP}_1, \ldots, \text{EXP}_n$ )". For a message send we have:

$$\texttt{C} \in [\![\text{EXP}]\!] \Rightarrow \begin{cases} [\![\text{EXP}_1]\!] \subseteq [\![\text{ID}_1]\!] \\ \vdots \\ [\![\text{EXP}_n]\!] \subseteq [\![\text{ID}_n]\!] \\ [\![\text{EXP}_0]\!] \subseteq [\![\text{EXP}\,\texttt{.}\,\text{ID}\,\texttt{(}\,\text{EXP}_1\texttt{,}\ldots\texttt{,}\,\text{EXP}_n\texttt{)}]\!] \end{cases}$$

where `C` implements a method for the message:

type ID( $\text{type}_1$ $\text{ID}_1, \ldots, \text{type}_n$ $\text{ID}_n$) {
  $\text{EXP}_0$
}

# Example

Suppose the classes `A`, `B`, `C`, `D` implement the interface `I`.

```
class A implements I {
  I x = new D();
  public I m(I f) {  // flow info for f = { B }
    return f.m(x);
  }
}

class B implements I {
  public I m(I g) {  // flow info for g = { C,D }
    return this;
  }
}

... new A().m(new B()).m(new C()) ...

// flow info for  new A().m(new B())  = { B }
```

# Constraints for the Example

$$\texttt{D} \in [\![\texttt{x}]\!]$$

$$\texttt{B} \in [\![\texttt{this-B}]\!]$$

$$\texttt{A} \in [\![\texttt{new A()}]\!] \qquad \texttt{B} \in [\![\texttt{new B()}]\!]$$

$$\texttt{C} \in [\![\texttt{new C()}]\!] \qquad \texttt{D} \in [\![\texttt{new D()}]\!]$$

$$\texttt{A} \in [\![\texttt{f}]\!] \Rightarrow \begin{cases} [\![\texttt{x}]\!] \subseteq [\![\texttt{f}]\!] \\ [\![\texttt{f.m(x)}]\!] \subseteq [\![\texttt{f.m(x)}]\!] \end{cases}$$

$$\texttt{B} \in [\![\texttt{f}]\!] \Rightarrow \begin{cases} [\![\texttt{x}]\!] \subseteq [\![\texttt{g}]\!] \\ [\![\texttt{this-B}]\!] \subseteq [\![\texttt{f.m(x)}]\!] \end{cases}$$

$$\texttt{A} \in [\![\texttt{new A()}]\!] \Rightarrow \begin{cases} [\![\texttt{new B()}]\!] \subseteq [\![\texttt{f}]\!] \\ [\![\texttt{f.m(x)}]\!] \subseteq [\![\texttt{new A().m(new B())}]\!] \end{cases}$$

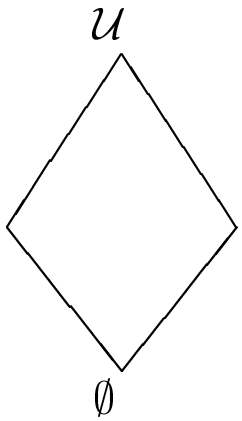
$$\texttt{B} \in [\![\texttt{new A()}]\!] \Rightarrow \begin{cases} [\![\texttt{new B()}]\!] \subseteq [\![\texttt{g}]\!] \\ [\![\texttt{this-B}]\!] \subseteq [\![\texttt{new A().m(new B())}]\!] \end{cases}$$

$$\texttt{A} \in [\![\texttt{new A().m(new B())}]\!] \Rightarrow \begin{cases} [\![\texttt{new C()}]\!] \subseteq [\![\texttt{f}]\!] \\ [\![\texttt{f.m(x)}]\!] \subseteq [\![\texttt{new A().m(new B()).m(new C())}]\!] \end{cases}$$

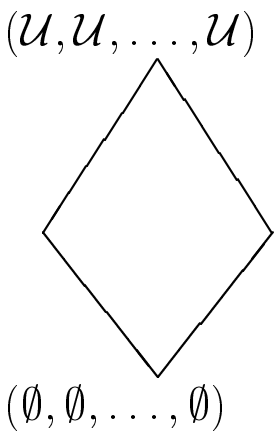
$$\texttt{B} \in [\![\texttt{new A().m(new B())}]\!] \Rightarrow \begin{cases} [\![\texttt{new C()}]\!] \subseteq [\![\texttt{g}]\!] \\ [\![\texttt{this-B}]\!] \subseteq [\![\texttt{new A().m(new B()).m(new C())}]\!] \end{cases}$$

# The Space of Flow Sets

- Each flow variable ranges over sets of classes, i.e., we have the lattice:

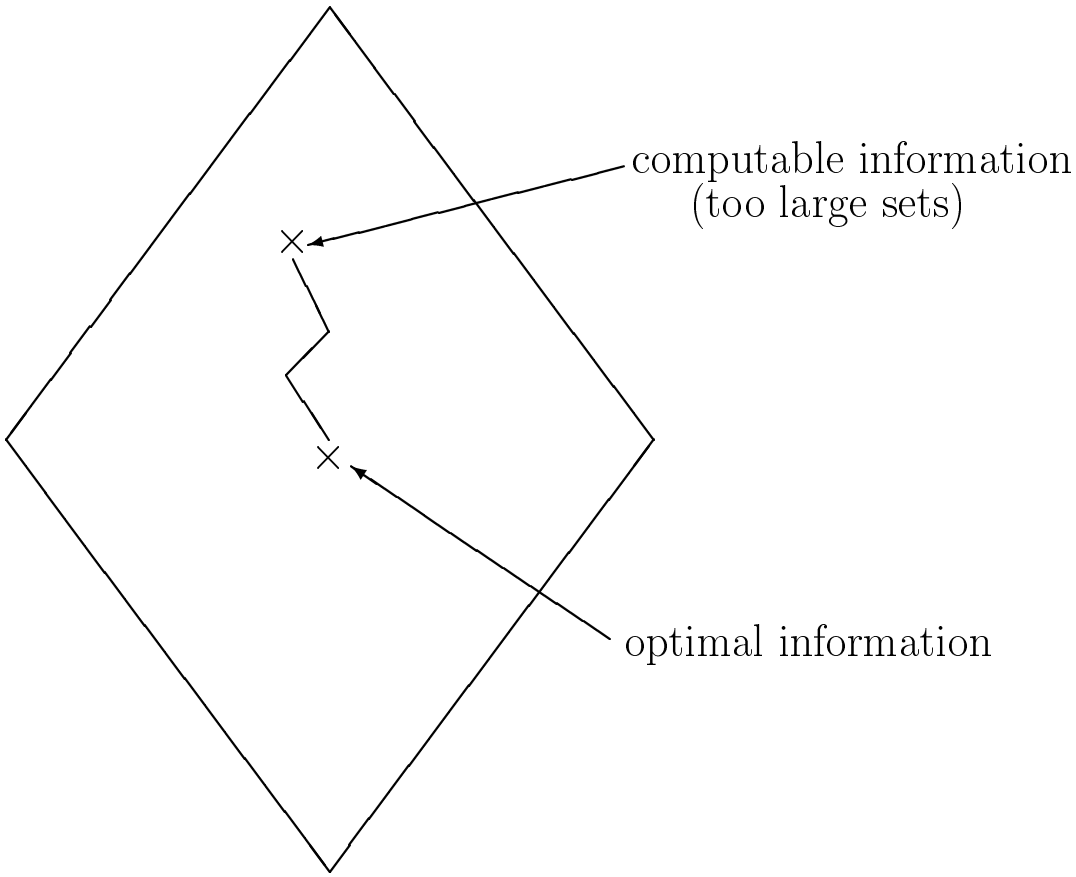



- The tuple of *all* flow variables ranges over the lattice:




- Nice mathematical properties.

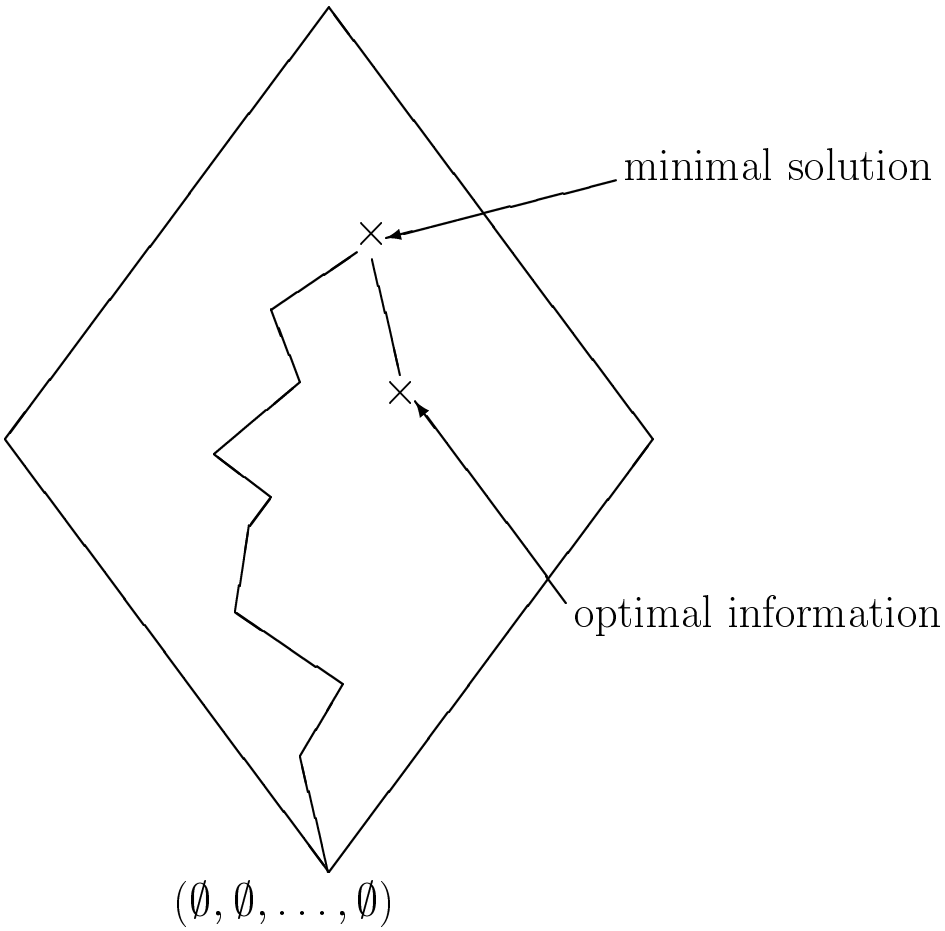
# Optimal and Computable Information




- Computable information is above optimal information.
- The top of the lattice corresponds to trivial information.

# Algorithm

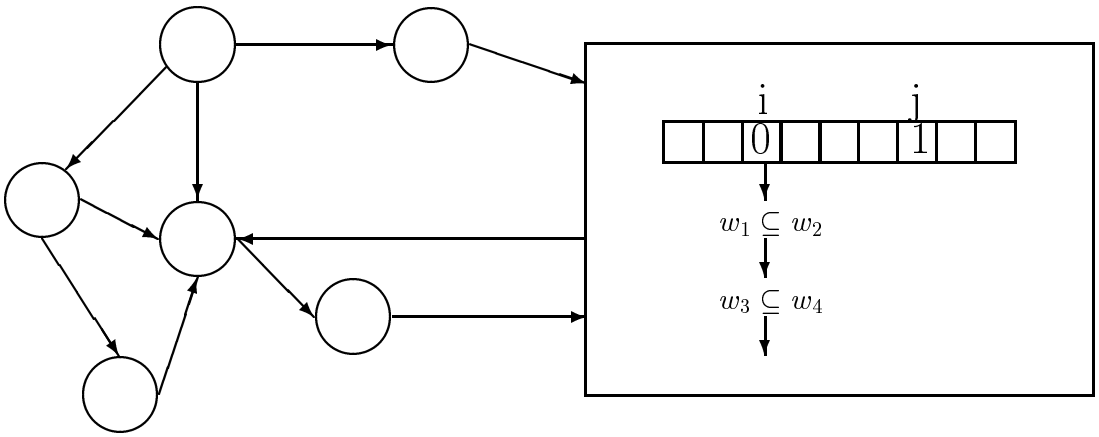
- The minimal solution can be computed in worst-case cubic time.
- Iterate towards the minimal solution:




- Use clever data structures.

## 3 The Constraint Solver

- Accepts constraints one at the time.
- Maintains the minimal solution.
- Its internal data structure is a graph.
- The graph has one node for each flow variable.
- An edge $v \rightarrow w$ implies $v \subseteq w$.
- The value of each flow variable is recorded in a bit vector.
- Conditional constraints are tied to the relevant bit.




- $B(X) =$ *bitvector in node* $X$
- $K(X, i) =$ *list of pending constraints for bit* $i$ *in node* $X$

# Solver Operations

INSERT$(C \subseteq X) =$
  **for** $i \in C$ **do** PROPAGATE($X$,$i$) **end**

INSERT$(X \subseteq Y) =$
  create an edge $X \rightarrow Y$;
  **for** $i \in B(X)$ **do** PROPAGATE($Y$,$i$) **end**

INSERT$(c \in X \Rightarrow Y \subseteq Z) =$
  **if** $B(X, c)$ **then**
    INSERT$(Y \subseteq Z)$
  **else**
    $K(X, c) = K(X, c)$++$(Y \subseteq Z)$
  **end**

PROPAGATE($v$,$i$) $=$
  **if** $\neg B(v, i)$ **then**
    $B(v, i) =$ true;
    **for** $v \rightarrow w$ **do** PROPAGATE($w$,$i$) **end**;
    **for** $k \in K(v, i)$ **do** INSERT$(k)$ **end**;
    $K(v, i) = ()$
  **end**

# An Amortized Analysis

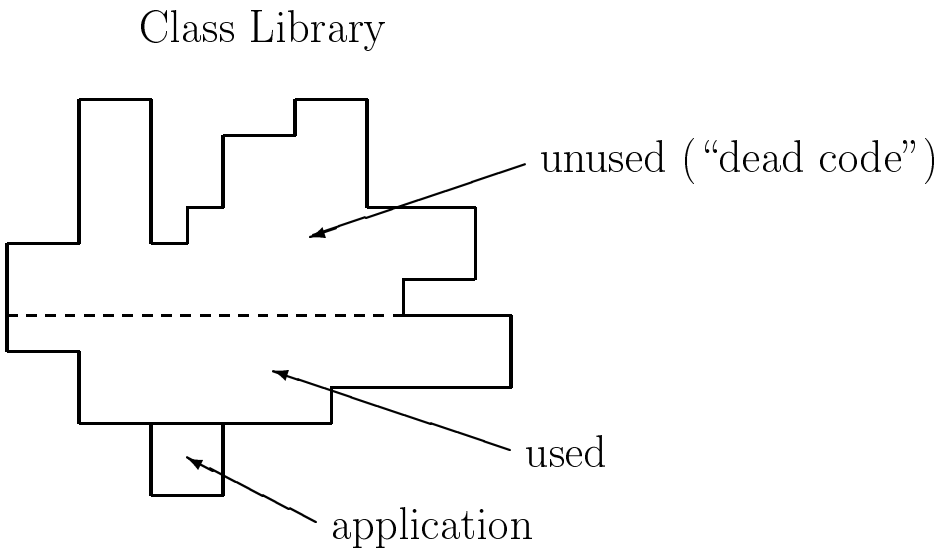
- Each activity uses up some part of the graph.
- Each bit is propagated along a specific edge at most once.
- Since there are $O(n^2)$ edges this totals $O(n^3)$.
- Each of the $O(n^2)$ constraints may:
  - be inserted into and deleted from a list once; and
  - cause the creation of a single edge,
- Which is $O(1)$ for each constraint.
- The total cost is thus $O(n^3)$.
- In practice, locality in programs gives more like linear behavior.

## 4 Quality of Information

- The basic algorithm is not very precise.
- It has several weak spots:
  - huge class libraries;
  - polymorphic methods; and
  - polymorphic container classes.
- They can be improved by:
  - dead code detection; and
  - code copying.
- A series of papers and PhD theses have given ever better techniques to handle these problems.
- Some techniques are general.
- Others require insight into the specific language in question.
- This note explains the idea of how to copy methods and classes in a rather static way which may square the size of the program.
- Only *adaptive* code-copying strategies seem to work in practice.

# The Trace Graph

- For a given application, most of the class library counts as "dead code".
- A trace graph gives a conservative estimate of code that can be ignored.
- This gives both more precise information and a considerable speed-up of the analysis.
- The key idea: reachability.



# The Extended Solver

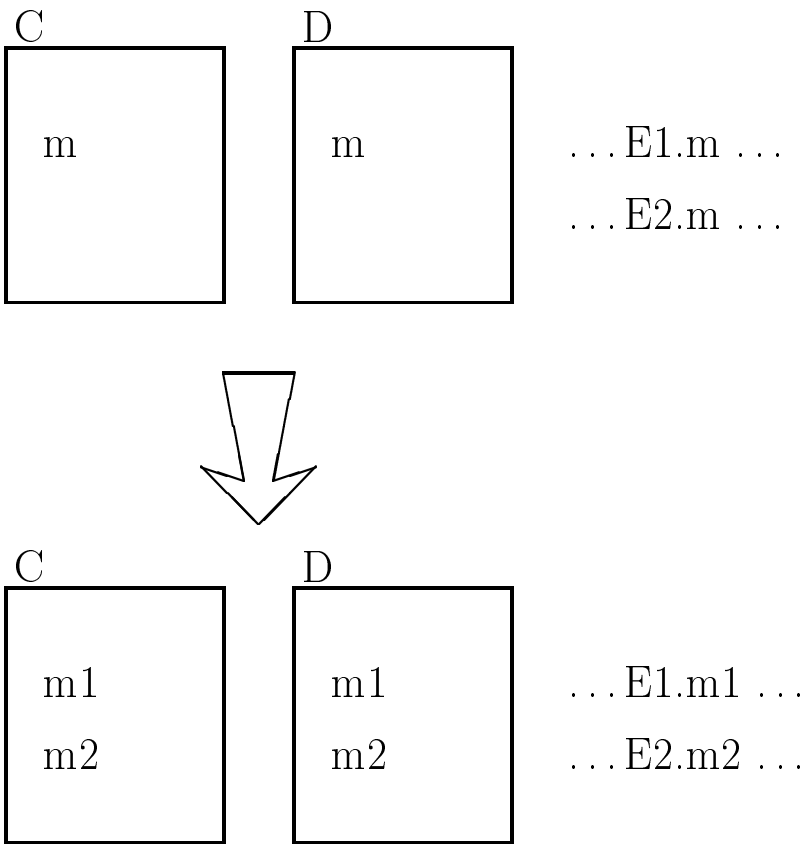
- Maintain a bit vector of *live* methods.
- Only insert constraints from the reachable parts.

```
L = ∅;
for k ∈ CONSTRAINTS(main) do
  INSERT(k)
end;
while L≠LIVE do
  choose m from LIVE\L;
  for k ∈ CONSTRAINTS(m) do
    INSERT(k);
    L = L∪{m}
  end
end;
remove all methods not in L
```

- The algorithm is still $O(n^3)$.
- The flow sets of live methods may be improved in this way.

# Copying Methods

- A program transformation to handle polymorphism.
- Creates a (nominal) copy of each method implementation for each syntactic invocation.
- Can be iterated to yield ever better information.



# Example

Suppose the classes `A` and `B` implement the interface `I`.

A polymorphic use of the identity function:

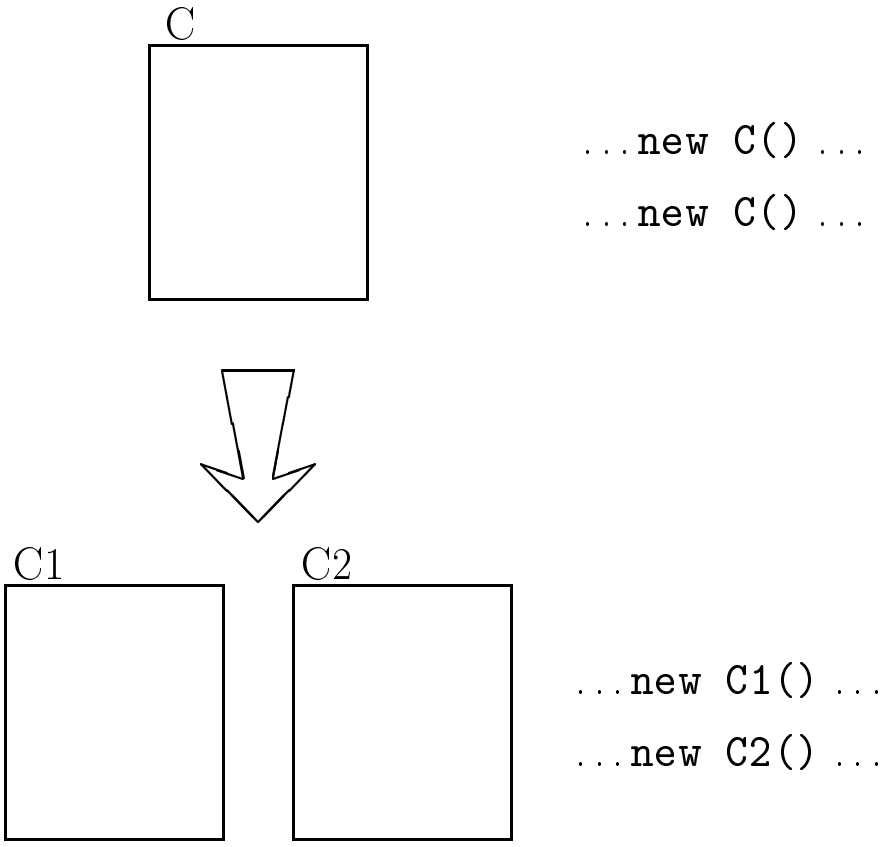
```
class C {
  I id(I x) {
    return x;  // flow set for x = { A,B }
  }
}
new C().id(new A()).m(5);
new C().id(new B()).m(5);
```

and a version where we make two copies of `id`:

```
class Cprime {
  I id1(I x) {
    return x;  // flow set for x = { A }
  }
  I id2(I x) {
    return x;  // flow set for x = { B }
  }
}
new Cprime().id1(new A()).m(5);
new Cprime().id2(new B()).m(5);
```

# Copying Classes

- A program transformation for container classes that gives polymorphic instance variables.
- Creates a (nominal) copy of each class for each syntactic "`new C()`".
- Can be iterated to yield ever better information.



# Example

A polymorphic use where `A` and `B` implement `I`:

```
class Container {
  I x;  // flow set for x = { A,B }
  Container put(I val) { x = val; return this;}
  I get() { return x; }
}
new Container().put(new A()).get().m(5);
new Container().put(new B()).get().m(5);
```

and a version where we make two copies of `Container`:

```
class Container1 {
  I x;  // flow set for x = { A }
  Container1 put(I val) { x = val; return this;}
  I get() { return x; }
}
class Container2 {
  I x;  // flow set for x = { B }
  Container2 put(I val) { x = val; return this;}
  I get() { return x; }
}
new Container1().put(new A()).get().m(5);
new Container2().put(new B()).get().m(5);
```