Compositional Pointer and Escape Analysis for Java Programs

John Whaley and Martin Rinard

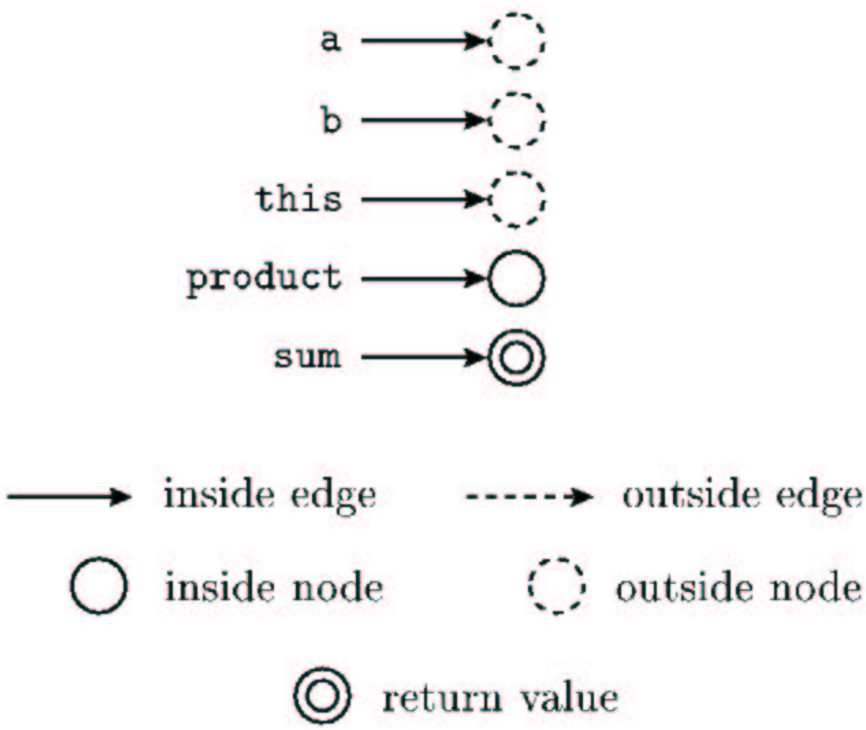
08/16/01

Élodie-Jane Sims. sims@ens.fr

Points-to escape graphs

- Points-to graphs : Characterize how local variables and fields in objects refer to other objects.
- escape information : Characterize how objects allocated in one region of the program can escape to be accessed by an other region.

```
class complex {
  double x,y;
  complex(double a, double b) { x = a; y = b; }
  complex multiply(complex a) {
    complex product =
      new complex(x*a.x - y*a.y,x*a.y + y*a.x);
    return(product);
  }
  complex add(complex a) {
    complex sum = new complex(x+a.x,y+a.y);
    return sum;
  }
  complex multiplyAdd(complex a, complex b) {
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return(sum);
  }
}
```



```
class multisetElement {
  Object element;
  int count;
  multisetElement next;

  multisetElement(Object e, multisetElement n) {
    count = 1;
    element = e;
    next = n;
  }
  synchronized boolean check(Object e) {
    if (element.equals(e)) {
      count++;
      return(true);
    } else return false;
  }
  synchronized multisetElement insert(Object e) {
    multisetElement m = this;
    while (m != null) {
      if (m.check(e)) return this;
      m = m.next;
    }
    return new multisetElement(e, this);
  }
}

class multiset {
  multisetElement elements;
  multiset() {
    elements = null;
  }
  synchronized void addElement(Object e) {
    if (elements == null)
      elements = new multisetElement(e,null);
    else elements = elements.insert(e);
  }
}
```

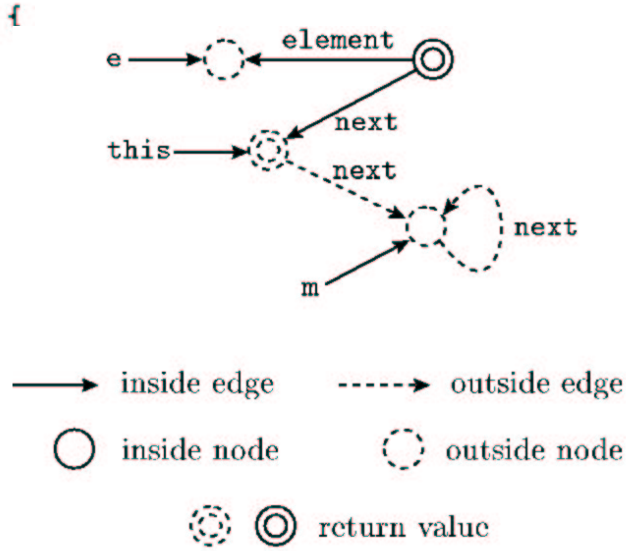


Figure 5: Analysis Result for **insert**

Properties of the algorithm

- analyze arbitrary part of the program
- more precise as more of the program is analyzed
- can distinguish where it does and does not have complete information

Ô analyze each method independently of its callers

Ô capable of analyzing a method without analyzing all of the method that it invokes

Applications

- eleminate synchronization for objects that are accessed by only one thread
- allocate objects on the stack instead of in the heap

Escape information

An object can **escape** if it is:

- reachable from a parameter or is the result value of the currently analyzed method
- reachable from a parameter or the result value of an invoked method, and we don't know what the invoked method do with it
- reachable from a static class variable or a runnable object

An object is **captured** if it does not *escape*.

Escape information propagation constraint :

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in O \cup I}{e(n_1) \subseteq e(n_2)}$$

Program objects

- $v \in V$: variables
  - $l \in L$ local variables
  - $p \in P$ formal parameter variable
  - $cl \in CL$ class names

- $f \in F$: field object (ex: $v.f$ or $cl.f$)

- $op \in OP$: methods

  Ý cl receiver class
  Ý $p_0, ..., p_k$: $p_0.op(p_1, ..., p_k)$

- $st \in ST$: nodes of control flow graph, $enter_{op}, exit_{op} \in ST$

- $m \in M$: method invocation site

- $n \in N$: nodes

Points-to escaped graph is an abstraction

- nodes represent objects
- edges represent references between objects

Properties:

a a single outside object may be represented by multiple outside nodes

a each object is represented by at most one inside node

a all outside/inside references have a corresponding outside/inside edge in the points-to escape graph

a if an object is represented by a captured node, it is represented by only that node

- captured objects are reachable only via paths that start with the local variables

- if a node is captured at the end of a method, the objects that it represents become inacessible as soon as the method return

The nodes

- N_I : inside nodes
  - represent objects created inside the currently analyzed region and accessed via inside edges

- N_O : outside nodes
  - represent objects created outside the currently analyzed region or accessed via outside edges
  - N_L : load nodes, if $l_1 = l_2$.f then the nodes pointed by l_1 are load nodes
  - N_P : parameter nodes
  - N_{CL} : class nodes
  - N_R : return nodes, if we have a statement $l_1 = l_2$.op(...) and the analysis skip the call it create a return node to which l_1 will point to.

The edges

- $O \subseteq (N \times F) \times N_L$: outside edges, represent references created outside the currently analyzed method
- $I \subseteq (V \cup (N \times F)) \times N$: inside edges, represent references created inside the currently analyzed method

A points-to escape graph

$$< O, I, e, r >$$

- $O \subseteq (N \times F) \times N_L$: outside edges
- $I \subseteq (V \cup (N \times F)) \times N$: inside edges
- $e : N \rightarrow 2^{P \cup CL \cup T \cup M}$: escape function
- $r \subseteq N$: result nodes *($\neq$ return nodes)*

Points-to escape graph's comments

- the escape function is just here to avoid to have to compute it when we need escape information, it just follow the definition of *a node can escape*.

- result nodes $\neq$ return nodes

  result nodes = node the analyzed method return *(i.e nodes pointed by l if we have a statement $return\ l$ in the program)*

  return nodes = the "wrong" nodes created by skipping call site when analysing this method

Statements

- l = v
- $l_1 = l_2$.f
- l_1.f $= l_2$
- return l
- l = new cl
- l = l_0.op(l_1,..., l_k) : *method invocation site*

Global algorithm

ä precompile the program to obtain a control-flow graph of each method with the kind of statement we want

ä do a topological sorting of the methods in the order of calling

ä analyze the methods in the reverse topological sort order and use a fixed-point algorithm within each strongly connected component

ä apply a **dataflow algorithm** on each method to obtain a points-to graph at each program point

- note that the control-flow graph is build without using any information from the test of an `if` or a `while`

Dataflow algorithm for a method

ä initialize the points-to information at the entry point of the method
- the parameters point to the corresponding parameter nodes
- the class names point to the corresponding class nodes

ä **processes the statements** in the method until reaching a fixed point.

Process a statement

- ä joins the points-to escape graphs flowing into the statement from all of its immediate predecessors in the control-flow graph

- ä apply the **statement's transfer function** to the joined points-to escape graph

Statement's transfer function

$$I' = (I \text{ - } \mathsf{Kill}_I) \cup \mathsf{Gen}_I$$
$$O' = O \cup \mathsf{Gen}_O$$

Statement	Kill_I	Gen_I	Kill_O
l = v	edgesFrom(I, l)	$\{\mathsf{l}\} \times I(\mathsf{v})$	$\emptyset$
$\mathsf{l}_1 = \mathsf{l}_2.\mathsf{f}$, $S_E = \emptyset$	edgesFrom(I, l)	$\{\mathsf{l}_1\} \times S_I$	$\emptyset$
$\mathsf{l}_1 = \mathsf{l}_2.\mathsf{f}$, $S_E \neq \emptyset$	edgesFrom(I, l)	$\{\mathsf{l}_1\} \times (S_I \cup \{n\})$	$(S_E \times \{\mathsf{f}\}) \times \{n$
$\mathsf{l}_1.\mathsf{f} = \mathsf{l}_2$	$\emptyset$	$(I(\mathsf{l}_1) \times \{\mathsf{f}\}) \times I(\mathsf{l}_2)$	$\emptyset$
l = new cl	edgesFrom(I, l)	$\{\langle\ \mathsf{l},\ n\rangle\}$	$\emptyset$

- $\mathsf{l}_1 = \mathsf{l}_2.\mathsf{f}$: n the load node
  $S_E = \{n_2 \in I(\mathsf{l}_2).escaped(\langle O, I, e, r\rangle, n_2)\}$: set of escaped nodes to which l_2 points
  $S_I = \cup\{I(n_2,\mathsf{f}).n_2 \in I(\mathsf{l}_2)\}$: set of nodes accessible via inside edges from l_2.f
  S

- return l : $r' = I(\mathsf{l})$

- $\mathsf{l} = \mathsf{l}_0.\mathsf{op}(\mathsf{l}_1, \ldots, \mathsf{l}_k)$: *Call rules*

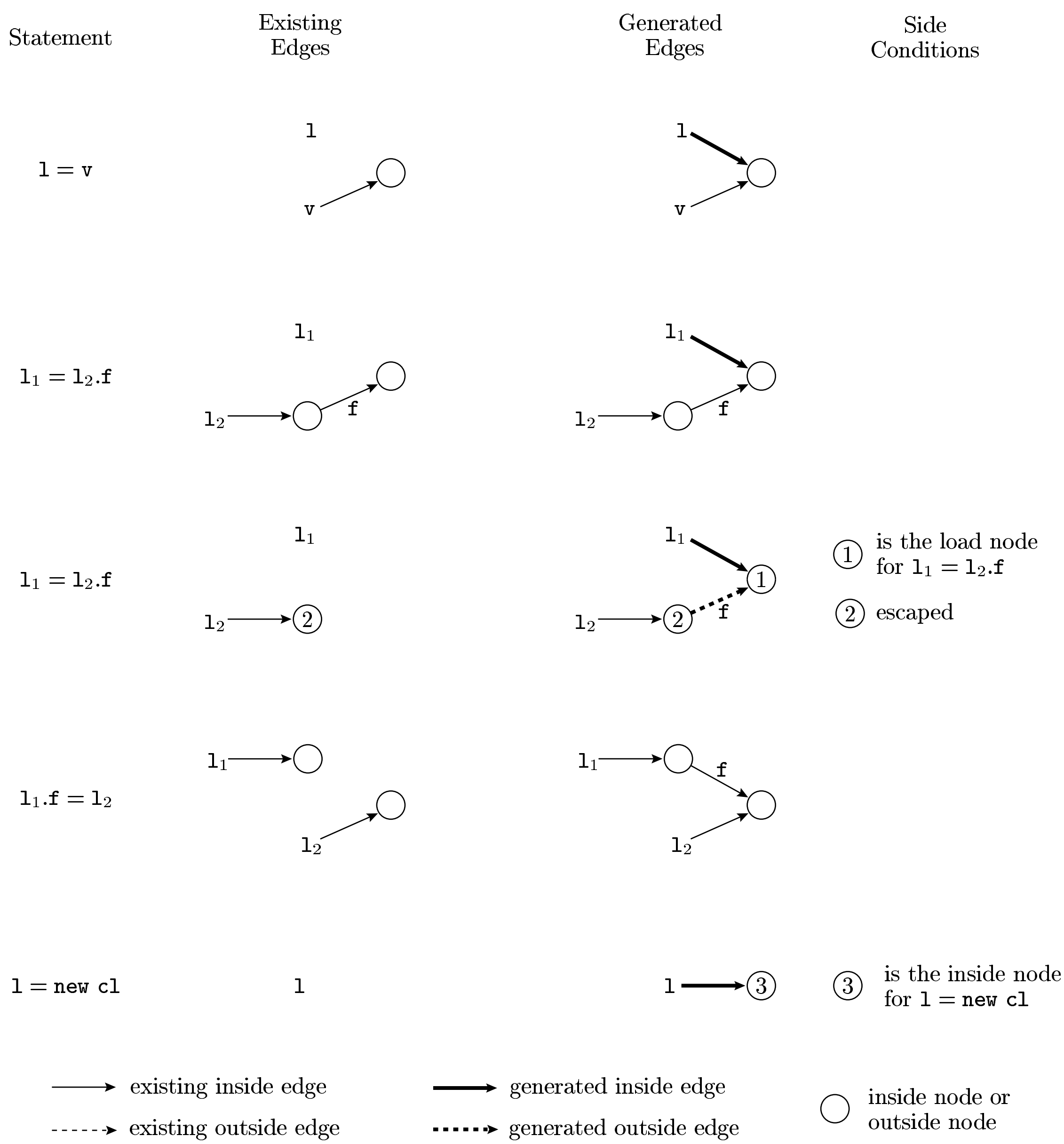


Figure 7: Generated Edges for Basic Statements

Call statement
l = l_0.op(l_1,..., l_k)

Two choices :

- skipping the call :
  - ä remove inside edges from l
  - ä create a new return node for this *method invocation site*
  - ä add an inside edge from the l to this new return node
  - ä update the escape function (the parameter escapes by the call site)
- analyzing the call : **mapping process**
  - ä map some nodes and edges of the called method's points-to graph to corresponding nodes and edges in the current points-to escapes graph
  - ä using this mapping add nodes and edges to the current points-to escapes graph

Mapping rules

The mapping algorithm take :

' the graph of the caller method with the escape function and with the call skipped : $< O, I, e, r >$, O are the outside edges, I are the inside edges, e is the escape function, r the set of nodes that represent objects that may be returned by the method

' the graph of the callee with the escape function : $< O_R, I_R, e_R, r_R >$

à and it build a mapping from the nodes of the callee graph to the nodes of the caller, and build a new graph $< O_M, I_M, e_M, r_M >$.

- Initialization : [`rule 4`]
- Parameters : [`rule 1`]
- Return values + assign l : [`rule 8 + 11`]
- Add edges using the mapping : [`rule 5, 7`]
- Complete the mapping : [`rule 3, 6, 7`]

$$\frac{0 \le i \le k, n \in I(\mathtt{l}_i)}{n \in \mu(n_{\mathtt{p}_i})} \quad (1)$$

$$\frac{\mathtt{cl} \in \mathbb{CL}}{n_{\mathtt{cl}} \in \mu(n_{\mathtt{cl}})} \quad (2)$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in O_R, \langle\langle n_3, \mathtt{f}\rangle, n_4\rangle \in I, \quad n_3 \in \mu(n_1), n_1 \notin N_I}{n_4 \in \mu(n_2)} \quad (3)$$

$$\begin{array}{ll} O \subseteq O_M & I - \mathrm{edgesFrom}(I, \mathtt{l}) \subseteq I_M \\ e(n) \subseteq e_M(n) & r \subseteq r_M \end{array} \quad (4)$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I_R}{(\mu(n_1) \times \{\mathtt{f}\}) \times \mu(n_2) \subseteq I_M} \quad (5)$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I_R, n \in \mu(n_1), n_2 \in N_I \cup N_R}{n_2 \in \mu(n_2)} \quad (6)$$

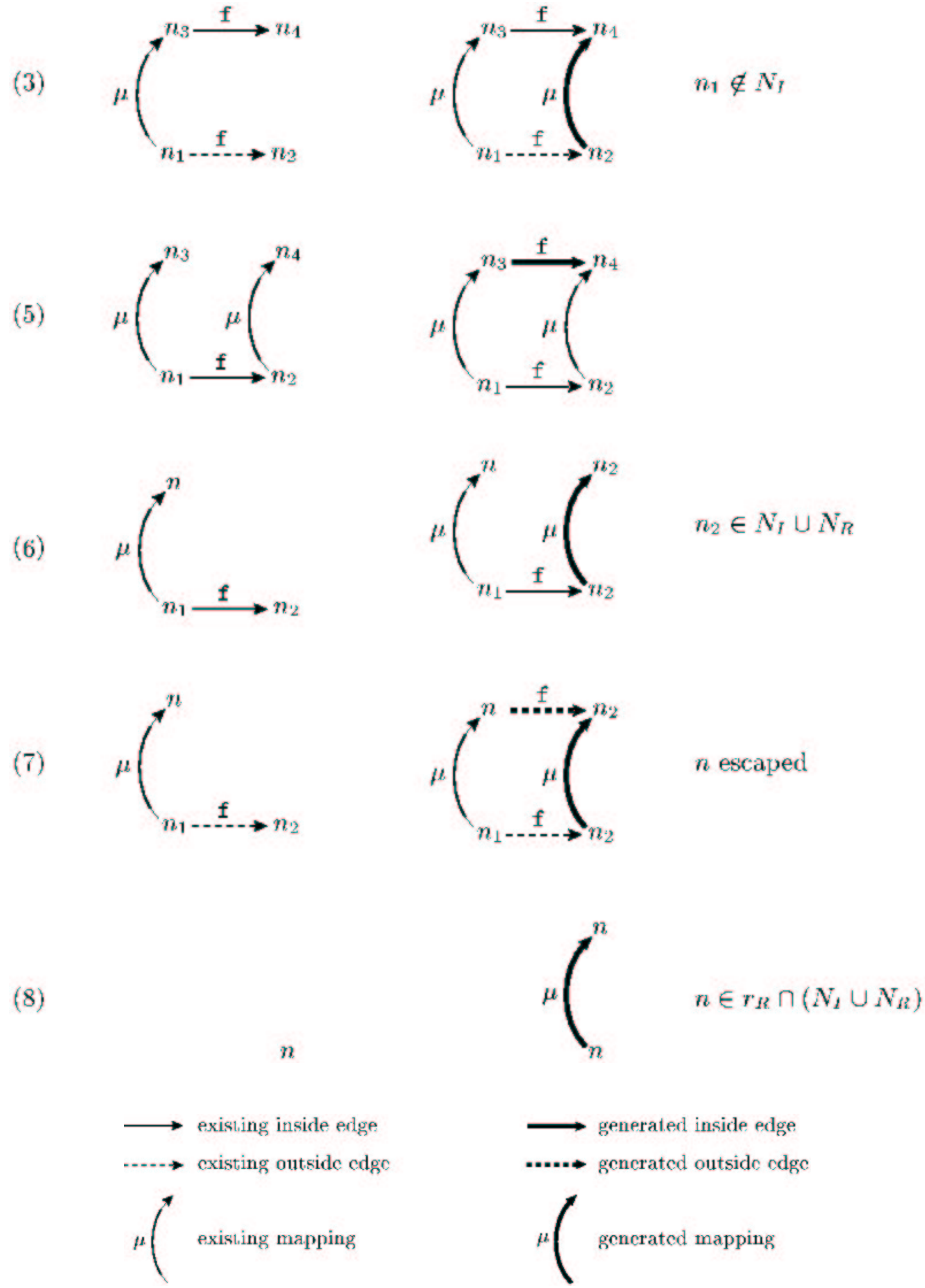
$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in O_R, n \in \mu(n_1), \quad \mathrm{escaped}(\langle O_M, I_M, e_M, r_M\rangle, n)}{\langle\langle n, \mathtt{f}\rangle, n_2\rangle \in O_M, n_2 \in \mu(n_2)} \quad (7)$$

$$\frac{n \in r_R \cap (N_I \cup N_R)}{n \in \mu(n)} \quad (8)$$

$$\frac{n' \in \mu(n)}{e_R(n) - \mathtt{P} \subseteq e_M(n')} \quad (9)$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I_M \cup O_M}{e_M(n_1) \subseteq e_M(n_2)} \quad (10)$$

$$\cup\{\mu(n).n \in r_R\} \subseteq I_M(\mathtt{l}) \quad (11)$$


(3)
$n_1 \notin N_I$
(5)
(6)
$n_2 \in N_I \cup N_R$
(7)
n escaped
(8)
$n \in r_R \cap (N_I \cup N_R)$
existing inside edge
existing outside edge
existing mapping
generated inside edge
generated outside edge
generated mapping

Explanation of the rules

- *Initialization :* To build the new graph the algorithm first copy the caller points-to graph but remove from it the insides edges from l (because l will be assigned) [`rule 4`].

- *Parameters :* Then it links the actual parameters with the formal parameter of the callee [`rule 1`].

- *Return values + assign l :* We said that the mapping is a kind of equivalence between the nodes of the callee and the nodes of the caller. By the [`rule 8`], the algorithm map the result nodes of the callee to themselves if they are inside nodes of the callee or if they are return nodes of the callee *(i.e. nodes created by a skipped call when computing the points-to graph of the callee)*. Mapping a node to itself "add this node into the new graph", if we look at the [`rule 11`], which is used at the end of the algorithm, it add an inside edge from l to all the nodes “equivalents” of a result node of the callee, so the [`rule 8`] is a way to make l point to the result nodes which are created by the callee.

- *Add edges using the mapping :*

  - [`rule 5`] add an inside field edge to the new graph using the mapping equivalence

  - [`rule 7`] add an outside edge to the new graph (after having complete the mapping)

- *Complete the mapping :*

  - [`rule 3`] complete the mapping to add an equivalence between an outside edge of the callee and an inside edge of the caller

  - [`rule 6`] complete the mapping to add an equivalence between an inside edge of the callee and an inside edge of the caller *(with the help of the [`rule 5`])*

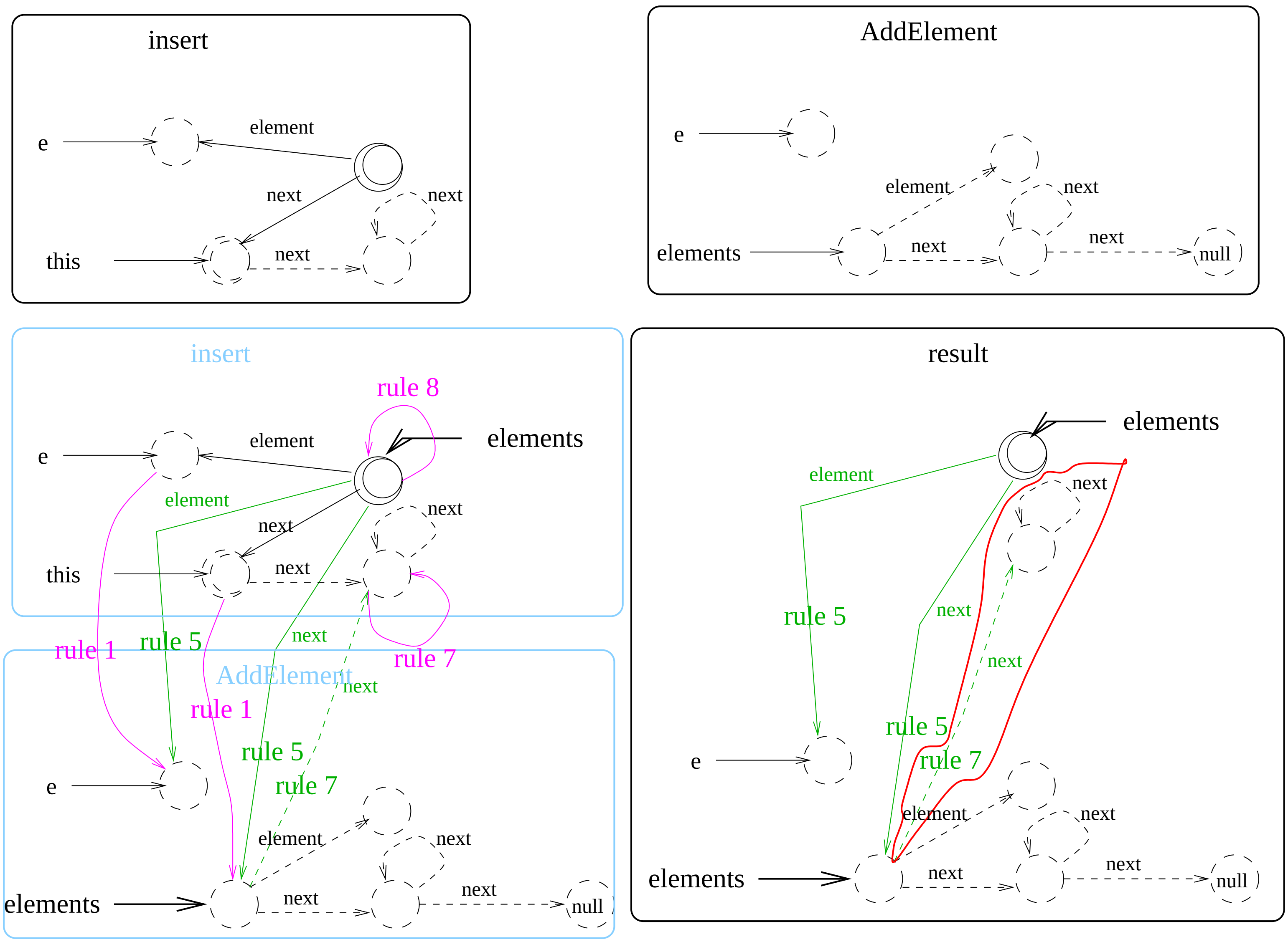
  - [`rule 7`] complete the mapping to add an equivalence between an outside edge of the callee and an outside edge of the caller

Example of application of mapping rules

`elements = elements.insert(e)`

caller : AddElement : l0 = elements, l1 = e

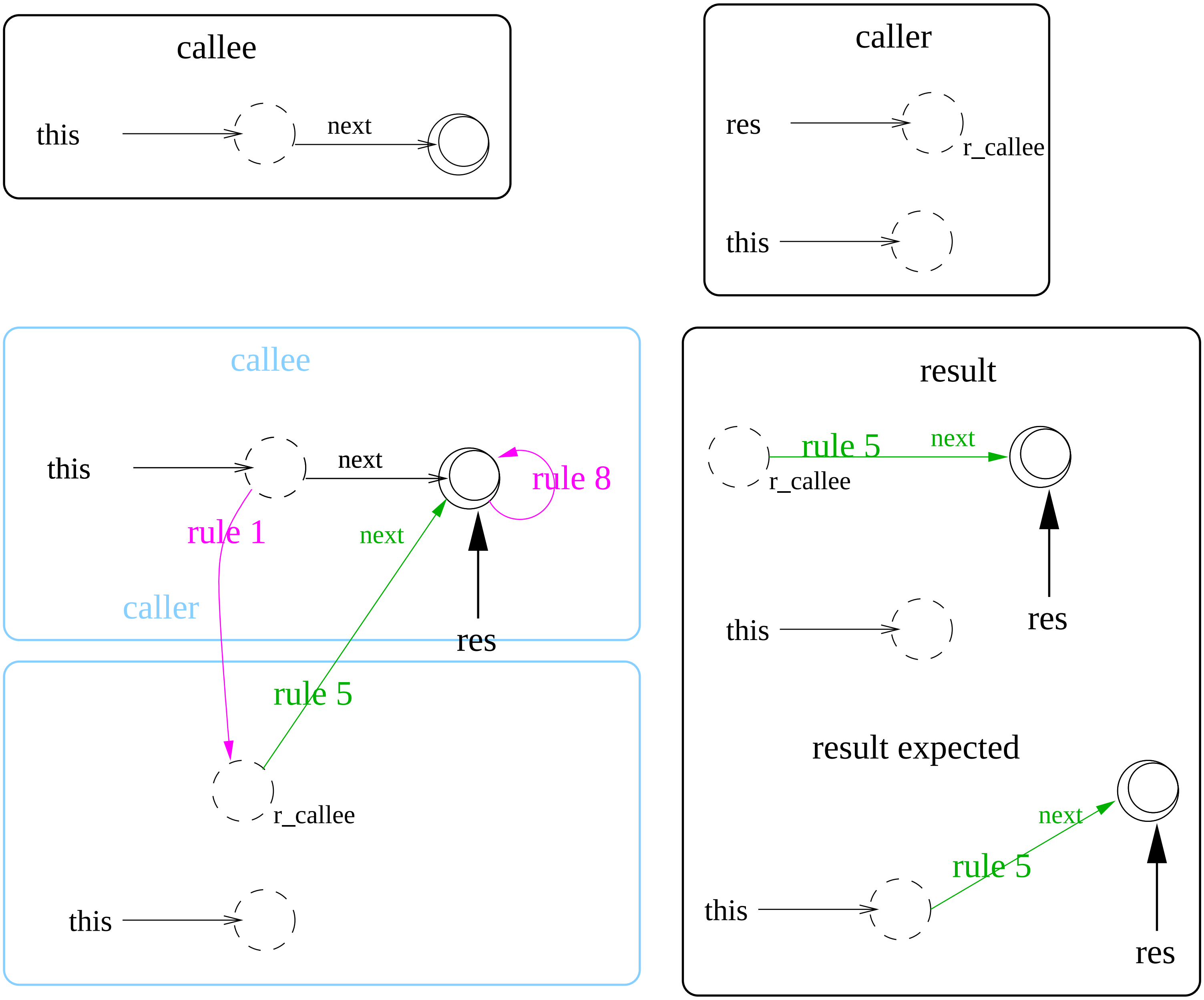
callee : insert : p0 = this, p1=e



Pb in case $l = l_0.op(l_1, ..., l_i)$ where there is a k such that $l = l_k$

```
class list{
list next;
list(){}

synchronized list callee (){
next = new list();
return(next);
}
synchronized void caller (){
res = this;
res = res.callee();
}
}
```


callee
this
next
caller
res
r_callee
this
callee
this
next
rule 8
rule 1
next
caller
res
rule 5
r_callee
this
result
rule 5
next
r_callee
this
res
result expected
next
rule 5
this
res