Exploiting linearity in sharing analysis of object-oriented programs

Gianluca Amato

Università di Chieti–Pescara
Pescara, Italy

16th Italian Conference on Theoretical Computer Science
ICTCS 2015, 9-11 September 2015, Firenze

(joint work with M. C. Meo and F. Scozzari)

Overview

Context

Data-flow analysis
Abstract interpretation
Pointer analysis

Plan of the talk

1. Sharing analysis
2. Adding linearity
3. Adding information for fields
4. The domain of ALPs-graphs
5. Conclusion

Overview

Context

Data-flow analysis
Abstract interpretation
Pointer analysis

Plan of the talk

1. Sharing analysis
2. Adding linearity
3. Adding information for fields
4. The domain of ALPs-graphs
5. Conclusion

Sharing analysis

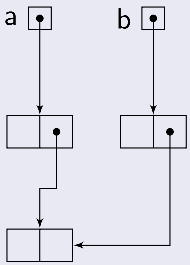
Sharing analysis aims to determine variables which are be bound to overlapping data structures at execution time.

Example

```
class Tree {
  Tree left;
  Tree right
}

T a = new Tree();
T b = new Tree();
a.right = new Tree();
b.right = a.right;
```

Heap



At the end of this program, variables *a* and *b* share.

Possible pair-sharing analysis

We represent sharing information with a set of *unordered* pairs of variables in scope. A pair (a,b) means that a and b *may* share during execution.

Formalized by Spoto & Secci, SAS 2005.

Example

$\{\}$

```
Tree a = new Tree();

Tree b = new Tree();

a.right = new Tree();

b.right = a.right;
```

Possible pair-sharing analysis

We represent sharing information with a set of *unordered* pairs of variables in scope. A pair (a,b) means that a and b *may* share during execution.

Formalized by Spoto & Secci, SAS 2005.

Example

```
{}
Tree a = new Tree();
{(a,a)}  a may be not null
Tree b = new Tree();

a.right = new Tree();

b.right = a.right;
```

Possible pair-sharing analysis

We represent sharing information with a set of *unordered* pairs of variables in scope. A pair (a,b) means that a and b *may* share during execution.

Formalized by Spoto & Secci, SAS 2005.

Example

```
{}
Tree a = new Tree();
{(a,a)}  a may be not null
Tree b = new Tree();
{(a,a),(b,b)}  a and b may be not null
a.right = new Tree();

b.right = a.right;
```

Possible pair-sharing analysis

We represent sharing information with a set of *unordered* pairs of variables in scope. A pair (a,b) means that a and b *may* share during execution.

Formalized by Spoto & Secci, SAS 2005.

Example

```
{}
Tree a = new Tree();
{(a,a)}  a may be not null
Tree b = new Tree();
{(a,a),(b,b)}  a and b may be not null
a.right = new Tree();
{(a,a),(b,b)}  a and b may be not null
b.right = a.right;
```

Possible pair-sharing analysis

We represent sharing information with a set of *unordered* pairs of variables in scope. A pair (a,b) means that a and b *may* share during execution.

Formalized by Spoto & Secci, SAS 2005.

Example

```
{}
Tree a = new Tree();
{(a,a)}  a may be not null
Tree b = new Tree();
{(a,a),(b,b)}  a and b may be not null
a.right = new Tree();
{(a,a),(b,b)}  a and b may be not null
b.right = a.right;
{(a,a),(b,b),(a,b)}  a and b may be not null, a and b may share
```

Other kind of pointer analysis

Points-to analysis

Relates a variable with the possible locations it may points. Locations are generally identified by occurrences of a `new` instruction.

If two variables may point to the same location they may share.

Alias analysis

Determines whether two variable points to the same location.

If two variables are aliases they share.

Reachability anaysis

Determines whether from a variable `a` it is possible to reach the location pointed to by variable `b`.

If `a` $\rightarrow$ `b`, then and `b` share

Overview

Context

Data-flow analysis
Abstract interpretation
Pointer analysis

Plan of the talk

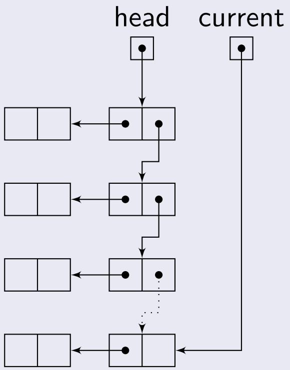
1. Sharing analysis
2. Adding linearity
3. Adding information for fields
4. The domain of ALPs-graphs
5. Conclusion

The need for linearity

Example (Creating List)

```
// create a list of length n>0
Tree create_list(int n) {
  Tree head = new Tree();
  Tree current = head;
  while (n>0) {
    current.left = new Tree();
    current.right = new Tree();
    current = current.right;
    n = n-1;
  }
  return head;
}
```

Heap



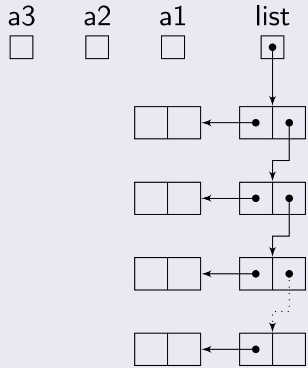
The need for linearity (2)

We would like to prove that a1, a2 and a3 do not share.
It is obvious reasoning on the concrete heap.

Example (Using List)

```
// extract first 3 elements
Tree list = create_list(5);
Tree a1 = list.left
list = list.right
Tree a2 = list.left
list = list.right
Tree a3 = list.left
```

Heap



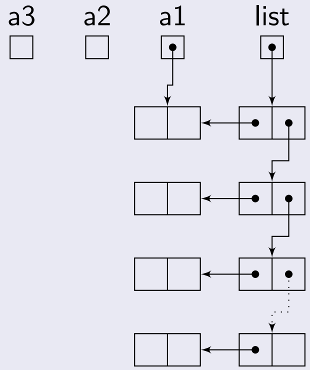
The need for linearity (2)

We would like to prove that a1, a2 and a3 do not share.
It is obvious reasoning on the concrete heap.

Example (Using List)

```
// extract first 3 elements
Tree list = create_list(5);
Tree a1 = list.left
list = list.right
Tree a2 = list.left
list = list.right
Tree a3 = list.left
```

Heap



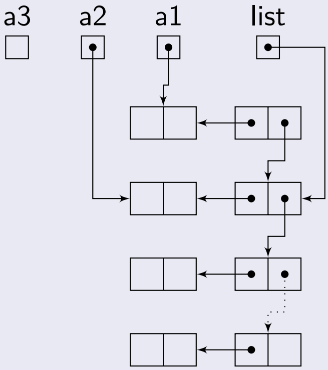
The need for linearity (2)

We would like to prove that a1, a2 and a3 do not share.
It is obvious reasoning on the concrete heap.

Example (Using List)

```
// extract first 3 elements
Tree list = create_list(5);
Tree a1 = list.left
list = list.right
Tree a2 = list.left
list = list.right
Tree a3 = list.left
```

Heap



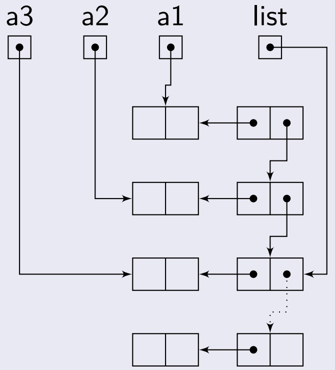
The need for linearity (2)

We would like to prove that a1, a2 and a3 do not share.
It is obvious reasoning on the concrete heap.

Example (Using List)

```
// extract first 3 elements
Tree list = create_list(5);
Tree a1 = list.left
list = list.right
Tree a2 = list.left
list = list.right
Tree a3 = list.left
```

Heap



The need for linearity (3)

Only sharing information at the level of variables.

Example (Analysis of the main program)

```
{}
Tree list = create_list(5);
```

Possile concretization

Possile (bad) concretization

The need for linearity (3)

Only sharing information at the level of variables.

Example (Analysis of the main program)

```
{}
Tree list = create_list(5);
{(list, list)}
```

Possile concretization

Possile (bad) concretization

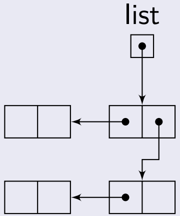
The need for linearity (3)

Only sharing information at the level of variables.

Example (Analysis of the main program)

```
{}
Tree list = create_list(5);
{(list, list)}
```

Possile concretization



Possile (bad) concretization

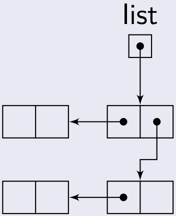
The need for linearity (3)

Only sharing information at the level of variables.

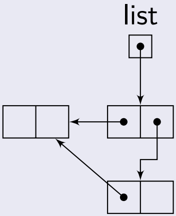
Example (Analysis of the main program)

```
{}
Tree list = create_list(5);
{(list, list)}
```

Possile concretization



Possile (bad) concretization

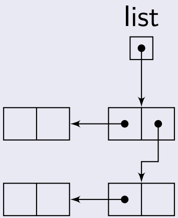


Linearity

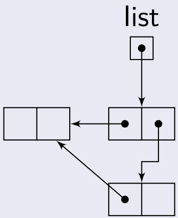
Definition (Linearity)

A variable v is non-linear if there is a location which is reachable from v following two *different* chains of field.

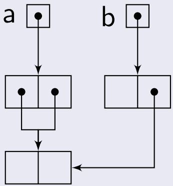
Linear heap



Non-linear heap



Complex heap



Analysis with linearity

$sh \star lin$: *sh* is the sharing information and *lin* a set of linear variables.

Example (Analysis of the main program)

```
{}
Tree list = create_list(5);
```

Possile concretization

Possile (bad) concretization

Analysis with linearity

sh ⋆ *lin*: *sh* is the sharing information and *lin* a set of linear variables.

Example (Analysis of the main program)

```
{}
Tree list = create_list(5);
{(list, list)} ⋆ {list}
```

Possile concretization

Possile (bad) concretization

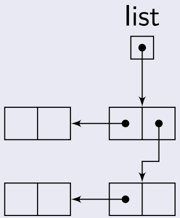
Analysis with linearity

sh ⋆ *lin*: *sh* is the sharing information and *lin* a set of linear variables.

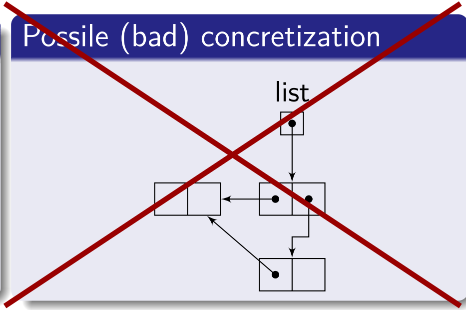
Example (Analysis of the main program)

```
{}
Tree list = create_list(5);
{(list, list)} ⋆ {list}
```

Possile concretization



Possile (bad) concretization



Overview

Context

Data-flow analysis
Abstract interpretation
Pointer analysis

Plan of the talk

1. Sharing analysis
2. Adding linearity
3. Adding information for fields
4. The domain of ALPs-graphs
5. Conclusion

The need for fields (1)

Example (Analysis with fields)

```
{}
Tree list = create_list(5);
{(list, list)} ⋆ {list}
Tree a1 = list.left
```

Good concretization

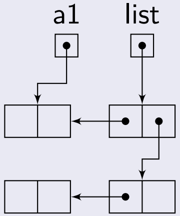
Bad concretization

The need for fields (1)

Example (Analysis with fields)

```
{}
Tree list = create_list(5);
{(list, list)} ⋆ {list}
Tree a1 = list.left
{(a1, a1), (list, list), (a1, list)} ⋆ {list, a1}
```

Good concretization



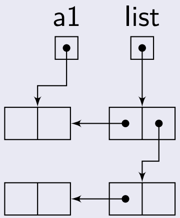
Bad concretization

The need for fields (1)

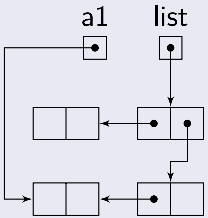
Example (Analysis with fields)

```
{}
Tree list = create_list(5);
{(list, list)} ⋆ {list}
Tree a1 = list.left
{(a1, a1), (list, list), (a1, list)} ⋆ {list, a1}
```

Good concretization



Bad concretization



The need for fields (2)

Example

```
{}
Tree list = create_list(5);

Tree a1 = list.left
```

Good concretization

Bad concretization

The need for fields (2)

Example

```
{}
Tree list = create_list(5);
{(list, list), (list.left, list.left), (list.right, list.right)} ⋆ {list}
Tree a1 = list.left
```

Good concretization

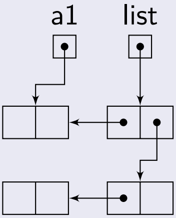
Bad concretization

The need for fields (2)

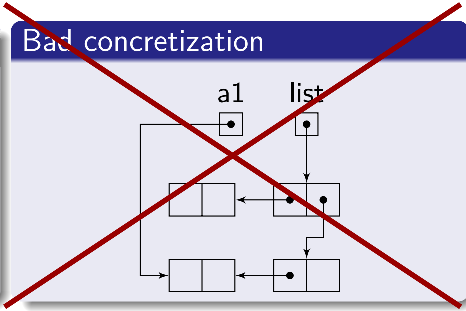
Example

```
{}
Tree list = create_list(5);
{(list, list), (list.left, list.left), (list.right, list.right)} ⋆ {list}
Tree a1 = list.left
{(a1, a1), (list.left, a1), (list, a1), (list, list), (list.left, list.left),
(list.right, list.right)} ⋆ {list, a1}
```

Good concretization



Bad concretization



The need for fields (3)

Example

{}

```
Tree list = create_list(5);
```

{(list, list), (list.left, list.left), (list.right, list.right)} ⋆ {list}

```
Tree a1 = list.left
```

{(a1,a1), (list.left, a1), (list, a1), (list, list), (list.left, list.left), (list.right, list.right)} ⋆ {list, a1}

```
list = list.right
```

We have done it!

- list.left and list.right do not share because list is linear.
- same abstract information we had after create_list...
- we can iterate without losing information

The need for fields (3)

Example

{}
```
Tree list = create_list(5);
```
{(list, list), (list.left, list.left), (list.right, list.right)} ⋆ {list}
```
Tree a1 = list.left
```
{(a1,a1), (list.left, a1), (list, a1), (list, list), (list.left, list.left), (list.right, list.right)} ⋆ {list, a1}
```
list = list.right
```
{(a1, a1), (list, list), (list.left, list.left), (list.right, list.right)} ⋆ {list, a1}

We have done it!

- `list.left` and `list.right` do not share because `list` is linear.
- same abstract information we had after `create_list`...
- we can iterate without losing information

Overview

Context

Data-flow analysis
Abstract interpretation
Pointer analysis

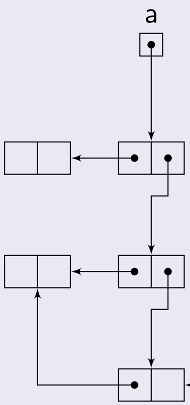
Plan of the talk

1. Sharing analysis
2. Adding linearity
3. Adding information for fields
4. The domain of ALPs-graphs
5. Conclusion

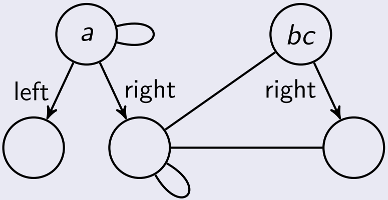
The domain of ALPs-graphs

Instead of keeping aliasing, sharing and linearity information as separate entities, we encode them in an ALPs-graph.

Heap



Abstraction



Operators

An extract from the definition of abstract operators:

$$\mathcal{SC}^l_\tau[\![v{:=}exp]\!](\mathbb{G}) = \mathrm{prune}((N' \star E' \star \ell'[v \mapsto \ell'(\mathtt{res}), \mathtt{res} \mapsto \bot]) \star sh' \star nl')$$

$$\mathcal{SC}^l_\tau[\![v.\mathtt{f}{:=}exp]\!](\mathbb{G}) = \begin{cases} \bot & \text{if } \ell'(v) = \bot \\ \mathrm{cl}(\mathrm{prune}((N' \cup N_{new} \star E' \setminus E_{del} \cup E_{new} \star \ell'[\mathtt{res} \mapsto \bot]) \star \\ \quad sh' \cup sh_{new} \star nl' \cup \{n_{\ell'(x)} \mid n_{\ell'(x)} \in N_{new}, \ell'(x.\mathtt{f}) \in nl'\})) \\ & \text{if } \ell'(v) \neq \bot \text{ and } \ell'(\mathtt{res}) = \bot \\ \mathrm{cl}(\mathrm{prune}((N' \cup N_{new} \star E' \setminus E_{del} \cup E'_{new} \star \ell'[\mathtt{res} \mapsto \bot]) \star \\ \quad sh' \cup sh'_{new} \star nl' \cup nl'_{new} \cup \{n_{\ell'(x)} \mid n_{\ell'(x)} \in N_{new}, \ell'(x.\mathtt{f}) \in nl'\})) \\ & \text{otherwise} \end{cases}$$

$$\mathcal{SC}^l_\tau\left[\begin{array}{l} \mathtt{if}\ v = \mathtt{null} \\ \mathtt{then}\ com_1\ \mathtt{else}\ com_2 \end{array}\right](\mathbb{G}) = \begin{cases} \mathcal{SC}^l_\tau[\![com_1]\!](\mathbb{G}) & \text{if } \ell(v) = \bot \\ \mathcal{SC}^l_\tau[\![com_1]\!](\mathbb{G}_{|v=\mathtt{null}}) \curlyvee \mathcal{SC}^l_\tau[\![com_2]\!](\mathbb{G}) & \text{otherwise} \end{cases}$$

$$\mathcal{SC}^l_\tau\left[\begin{array}{l} \mathtt{if}\ v = w \\ \mathtt{then}\ com_1\ \mathtt{else}\ com_2 \end{array}\right](\mathbb{G}) = \begin{cases} \mathcal{SC}^l_\tau[\![com_1]\!](\mathbb{G}) & \text{if } \ell(v) = \ell(w) \\ \mathcal{SC}^l_\tau[\![com_1]\!](\mathbb{G}_{|v=w}) \curlyvee \mathcal{SC}^l_\tau[\![com_2]\!](\mathbb{G}) & \text{otherwise} \end{cases}$$

$$\mathcal{SC}^l_\tau[\![\{com_1; \ldots; com_p\}]\!] = (\lambda s \in ALPs_\tau.s) \circ \mathcal{SC}^l_\tau[\![com_p]\!] \circ \cdots \circ \mathcal{SC}^l_\tau[\![com_1]\!]$$

Restriction to null

Program code

```
//  G1
if (v == null) {
 //  G2
 cmd
```

Question: How do we obtain G_2 from G_1?

Answer: We delete the node labeled by v and all its descendants.

Example (G_1)

Example (G_2)

Restriction to null

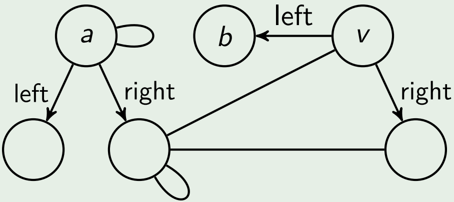
Program code

```
// G1
if (v == null) {
 // G2
 cmd
```

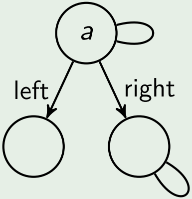
Question: How do we obtain G_2 from G_1?

Answer: We delete the node labeled by v and all its descendants.

Example (G_1)



Example (G_2)



Field Assignment

Program code

```
//  G1
v.right = null
//  G2
```

Question: How do we obtain G_2 from G_1?

Answer: Delete arrow from v labeled by *right*... but consider possible aliases of v

Example (G_1)

Example (G_2 incorrect)

Field Assignment

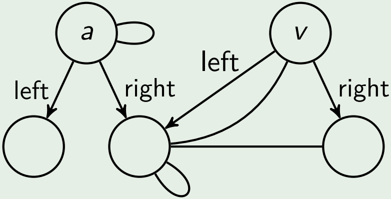
Program code

```
// G1
v.right = null
// G2
```

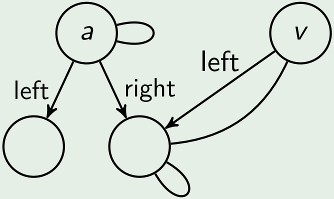
Question: How do we obtain G_2 from G_1?

Answer: Delete arrow from v labeled by *right*... but consider possible aliases of v

Example (G_1)



Example (G_2 incorrect)



Field Assignment

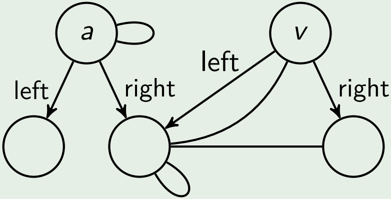
Program code

```
//  G1
v.right = null
//  G2
```

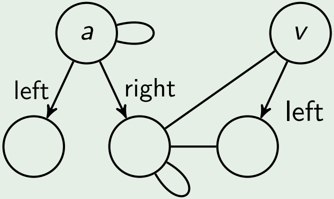
Question: How do we obtain G_2 from G_1?

Answer: Delete arrow from v labeled by *right*... but consider possible aliases of v

Example (G_1)



Example (G_2 correct)



Overview

Context

Data-flow analysis
Abstract interpretation
Pointer analysis

Plan of the talk

1. Sharing analysis
2. Adding linearity
3. Adding information for fields
4. The domain of ALPs-graphs
5. Conclusion

What we have done

- Formally define ALPs graphs
  - a Galois connection with the powerset of concrete heaps
  - using concrete heap as formalized in [Secci & Spoto 05]
- Define abstract operators needed to analyze Java code
  - on the concrete semantics defined in [Secci & Spoto 05]
- Prove correctness of these operators

What we have done

- Formally define ALPs graphs
  - a Galois connection with the powerset of concrete heaps
  - using concrete heap as formalized in [Secci & Spoto 05]
- Define abstract operators needed to analyze Java code
  - on the concrete semantics defined in [Secci & Spoto 05]
- Prove correctness of these operators

What we have done

- Formally define ALPs graphs
  - a Galois connection with the powerset of concrete heaps
  - using concrete heap as formalized in [Secci & Spoto 05]
- Define abstract operators needed to analyze Java code
  - on the concrete semantics defined in [Secci & Spoto 05]
- Prove correctness of these operators

Todo

- Experimental evaluation
  - developing an implementation in our static analyzer `Jandom`
  - `https://github.com/jandom-devel/Jandom`
- Determine computational complexity of operators
  - easy
  - all operators in PTIME
- Optimality of the semantic operators
  - are the abstract operators as precise as possible?
  - hard and not very rewarding
- Many possible tricks and variations
  - possible aliasing (helps assignment)
  - variable depths of ALPs graphs

Todo

- Experimental evaluation
  - developing an implementation in our static analyzer `Jandom`
  - `https://github.com/jandom-devel/Jandom`
- Determine computational complexity of operators
  - easy
  - all operators in PTIME
- Optimality of the semantic operators
  - are the abstract operators as precise as possible?
  - hard and not very rewarding
- Many possible tricks and variations
  - possible aliasing (helps assignment)
  - variable depths of ALPs graphs

Todo

- Experimental evaluation
  - developing an implementation in our static analyzer `Jandom`
  - `https://github.com/jandom-devel/Jandom`
- Determine computational complexity of operators
  - easy
  - all operators in PTIME
- Optimality of the semantic operators
  - are the abstract operators as precise as possible?
  - hard and not very rewarding
- Many possible tricks and variations
  - possible aliasing (helps assignment)
  - variable depths of ALPs graphs

Todo

- Experimental evaluation
  - developing an implementation in our static analyzer `Jandom`
  - `https://github.com/jandom-devel/Jandom`
- Determine computational complexity of operators
  - easy
  - all operators in PTIME
- Optimality of the semantic operators
  - are the abstract operators as precise as possible?
  - hard and not very rewarding
- Many possible tricks and variations
  - possible aliasing (helps assignment)
  - variable depths of ALPs graphs

Thanks

Variants of sharing analysis

Pair sharing and set sharing

Pair sharing Only pair of variables are considered.

Set sharing Sets of variables are considered.
{`a`, `b`, `c`} means that there is an object which is reachable from `a`, `b` and `c`. This is different from (`a`, `b`), (`b`, `c`), (`a`, `c`).

May/must sharing

May sharing (`a`, `b`) means that variables `a` and `b` *might* share. Also called *possible sharing* and *definite non-sharing*.

Must sharing (`a`, `b`) means that variables `a` and `b` *must* share. Also called *definite sharing* and *possible non-sharing*.

History

- Possible sharing has been thoroughly investigated for logic programs.
- Pair sharing analysis for Java:

  S. Secci and F. Spoto
  "Pair-Sharing. Analysis of Object-Oriented Programs"
  SAS 2005

- Set sharing analysis for Java:

  M. Méndez-Lozo, M. V. Hermenegildo
  "Precise set-sharing analysis for Java-style programs"
  VMCAI 2008