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

Example

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

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

At the end of this program, variables \textit{a} and \textit{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.


Example

```java
{ }
Tree a = new Tree();

Tree b = new Tree();

a.right = new Tree();

b.right = a.right;
```
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.


Example

\[
\begin{align*}
\{\} \\
\text{Tree } a &= \text{ new Tree();} \\
\{a,a\} &\quad \text{a may be not null} \\
\text{Tree } b &= \text{ new Tree();} \\
\text{a.right } &= \text{ new Tree();} \\
\text{b.right } &= \text{ a.right;}
\end{align*}
\]
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.


**Example**

```java
{}  // Empty set
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.


Example

```java
{}  
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.


**Example**

```java
{}  
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
```
### Points-to analysis

Relates a variable with the possible locations it may point. 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 analysis

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

If $a \rightarrow b$, then $a$ 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)

```c
// 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

```
head

```

```
current

```

---

Gianluca Amato
Sharing and linearity
ICTCS 2015
The need for linearity (2)

We would like to prove that $a_1$, $a_2$ and $a_3$ do not share. It is obvious reasoning on the concrete heap.

Example (Using List)

```c
// 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
```
The need for linearity (2)

We would like to prove that $a_1$, $a_2$ and $a_3$ 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
```
We would like to prove that $a_1$, $a_2$ and $a_3$ do not share. It is obvious reasoning on the concrete heap.

Example (Using List)

```java
// 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
```
The need for linearity (2)

We would like to prove that \(a_1\), \(a_2\) and \(a_3\) do not share. It is obvious reasoning on the concrete heap.

Example (Using List)

```cpp
// 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 diagram showing the relationship between the elements and the list structure.
The need for linearity (3)

Only sharing information at the level of variables.

Example (Analysis of the main program)

```
{}

Tree list = create_list(5);
```

Possible concretization

Possible (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))}
```
The need for linearity (3)

Only sharing information at the level of variables.

Example (Analysis of the main program)

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

Possible concretization

Possible (bad) concretization
The need for linearity (3)

Only sharing information at the level of variables.

Example (Analysis of the main program)

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

Possile concretization

Possile (bad) concretization
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)

\[
\{ \}
\text{Tree list} = \text{create\textunderscore list}(5);
\]
Analysis with linearity

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

**Example (Analysis of the main program)**

\{
\}

\{list, list\} \star \{list\}

**Possile concretization**

**Possile (bad) concretization**
**Analysis with linearity**

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

**Example (Analysis of the main program)**

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

**Possible concretization**

**Possible (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)

```{language=java}
{}
Tree list = create_list(5);
{(list, list)} ∗ {list}
Tree a1 = list.left
```

Good concretization

![Good concretization diagram]

Bad concretization

![Bad concretization diagram]
The need for fields (1)

Example (Analysis with fields)

```plaintext
{} 
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

```cpp
{}
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)} \star \{list\}
Tree a1 = list.left
{(a1, a1), (list.left, a1), (list, a1), (list, list), (list.left, list.left),
(list.right, list.right)} \star \{list, a1\}

Good concretization

![Good concretization diagram]

Bad concretization

![Bad concretization diagram]
The need for fields (3)

Example

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

```java
{}
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.
An extract from the definition of abstract operators:

\[
SC^I_\tau[v:=\exp](G) = \text{prune}((N' \ast E' \ast \ell'[v \mapsto \ell'(\text{res}), \text{res} \mapsto \bot]) \ast sh' \ast nl')
\]

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

\[
SC^I_\tau[ \begin{array}{l}
\text{if } v = \text{null} \\
\text{then } \text{com}_1 \text{ else } \text{com}_2
\end{array} ](G) = \begin{cases} 
SC^I_\tau[\text{com}_1](G) & \text{if } \ell(v) = \bot \\
SC^I_\tau[\text{com}_1](G|v=\text{null}) \uplus SC^I_\tau[\text{com}_2](G) & \text{otherwise}
\end{cases}
\]

\[
SC^I_\tau[ \begin{array}{l}
\text{if } v = w \\
\text{then } \text{com}_1 \text{ else } \text{com}_2
\end{array} ](G) = \begin{cases} 
SC^I_\tau[\text{com}_1](G) & \text{if } \ell(v) = \ell(w) \\
SC^I_\tau[\text{com}_1](G|v=w) \uplus SC^I_\tau[\text{com}_2](G) & \text{otherwise}
\end{cases}
\]

\[
SC^I_\tau[\{\text{com}_1; \ldots; \text{com}_p\}] = (\lambda s \in \text{ALPs}_\tau.s) \circ SC^I_\tau[\text{com}_p] \circ \cdots \circ SC^I_\tau[\text{com}_1]
\]
Restriction to null

Program code

```c
// 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 ($G_1$)**

```
  a
  ↓ left
  ↓ right
  b → v

  left
  right
```

**Example ($G_2$)**

```
  a
  ↓ left
  ↓ right
```

Gianluca Amato
Sharing and linearity
ICTCS 2015 20 / 27
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 ($G_2$)
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 ($G_1$)**

![Diagram of $G_1$](image1)

**Example ($G_2$ incorrect)**

![Diagram of $G_2$ incorrect](image2)
Program code

// G_1
v.right = null
// G_2

Question: How do we obtain $G_2$ from $G_1$?
Answer: Delete arrow from $v$ labeled by $right$... but consider possible aliases of $v$
Question: How do we obtain $G_2$ from $G_1$?

Answer: Delete arrow from $v$ labeled by $right$... but consider possible aliases of $v$
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](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
- **Experimental evaluation**
  - developing an implementation in our static analyzer Jandom
  - [https://github.com/jandom-devel/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](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
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.
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