Precise Set Sharing Analysis for Java Programs

Mario Méndez-Lojo\textsuperscript{1}  Manuel Hermenegildo\textsuperscript{1,2}

\textsuperscript{1}University of New Mexico (USA)
\textsuperscript{2}Technical University of Madrid and IMDEA-Software (Spain)

VMCAI’08, January 8
Motivation

- Computing, at compile time, precise approximations of memory state at any program point — classic static analysis problem.
- Sharing representations proved very useful in several applications (e.g., parallelization in logic programming [JL89, MH91]), but not used to date in OO programming.
- An exception is [Spoto05], which uses pair sharing as abstraction.

Goals

- Design an analysis based on a set sharing heap representation.
- Study its precision (specially in comparison to pair sharing).
- We develop an interprocedural, context-sensitive, multivariant analysis for Java bytecode based on Abstract Interp. [CC77].
The set of distinct identifiers defined in the program is $\mathcal{V}$; any element in $\mathcal{V}$ has a type included in $K$, the program-defined classes.

Concrete states $\delta : \Sigma$ are pairs (frame,memory) that represent a valid heap configuration.

A frame maps elements of $\mathcal{V}$ to null or a location, which is linked to an object by the memory function.

Objects consist of a type and a frame, the latter representing the fields.

The set of locations reachable from $v : \mathcal{V}$ in state $\delta : \Sigma$ is obtained by applying the reachability function $R(\delta, v)$. 
Sharing in the Concrete Domain

Variables $V = \{v_1, \ldots, v_n\}$ share in state $\delta$ if there is at least one common location in their reachability sets:

$$share(\delta, V) \iff \bigcap_{i=1}^{n} R(\delta, v_i) \neq \emptyset$$

Example:

$$v = \text{new Vector}();$$
$$w = v;$$
$$y = v.\text{first};$$
$$z = \text{new Vector}();$$

It holds that $share(\delta, \{v, w, y\}) \land share(\delta, \{z\}) \land share(\delta, \{v\}) \land \ldots$
Abstract Domain

An abstract state $\sigma$ is an element of $D = DS \times DN \times DT$.

- $DS = P(P(V))$ is the sharing component.
- $DN = P(V \mapsto \{null, nnull, unk\})$ tells whether a given variable is definitely null, non null, or of unknown nullness.
- $DT = P(V \mapsto P(K))$ contains a list with all the possible types of the variable.

Concretization function $\gamma : D \mapsto P(\Sigma)$

$$\gamma(sh, nl, \tau) = \{ \delta \in \Sigma \mid \forall V \subseteq V, \text{share}(\delta, V) \text{ and } \not\exists W, V \subset W \subseteq V \text{ s.t. } \text{share}(\delta, W) \Rightarrow V \in sh \}, \text{ and } R(\delta, v) = \emptyset \text{ if } nl(v) = null, \text{ and } R(\delta, v) \neq \emptyset \text{ if } nl(v) = nnull$$
Sharing in the Abstract Domain

For the example already shown:

\[
\begin{align*}
v &= \text{new Vector();} \\
w &= v; \\
y &= v.\text{first;} \\
z &= \text{new Vector();}
\end{align*}
\]

Given an abstract state

\[
(\{\{v, w, y\}, \{z\}\}, \\
\{v/[\text{nnull}], w/[\text{nnull}], y/[\text{unk}], z/[\text{nnull}]\}, \\
\{v/[\text{Vector}], w/[\text{Vector}], y/[\text{Element}], z/[\text{Vector}]\})
\]

, \(\delta\) is an element of its concretization.
Concretization Function $\gamma$ — Examples

(We omitted the type information for simplicity.)

\[
\begin{align*}
\text{abstract state } \sigma & \quad \text{concrete state } \delta & \quad \delta \in \gamma(\sigma) \\
\{\{v, w\}, \{v, y\}, \{w, y\}\} & \quad \{v[/null], w[/null], y[/null]\} & \quad ?
\end{align*}
\]
Concretization Function $\gamma$ — Examples

(We omitted the type information for simplicity.)

abstract state $\sigma$

$$\{\{v, w\}, \{v, y\}, \{w, y\}\}$$
$$\{v/\text{null}, w/\text{null}, y/\text{null}\}$$

concrete state $\delta$

$\delta \in \gamma(\sigma)$?

no

V
W
loc_1
Y
Concretization Function $\gamma$ — Examples

(We omitted the type information for simplicity.)

abstract state $\sigma$

concrete state $\delta$

$\delta \in \gamma(\sigma)$?

{\{v, w\}, \{v, y\}, \{w, y\}}

{v/[null], w/[null], y/[null]}

{\{v, w\}, \{v, w, y\}}

{v/[null], w/[null], y/[null]}

$V$ $W$

loc_1

...
Concretization Function $\gamma$ — Examples

(We omitted the type information for simplicity.)

Abstract state $\sigma$

$\{\{v, w\}, \{v, y\}, \{w, y\}\}$
$\{v/[nnull], w/[nnull], y/[nnull]\}$

Concrete state $\delta$

$\delta \in \gamma(\sigma)$?

No

Yes

Y

V

W

loc_1

...
Concretization Function $\gamma$ — Examples

abstract state $\sigma$  

$\{\{v\}, \{w\}, \{y\}, \{v, w\}, \{v, y\}, \{w, y\}, \{v, w, y\}\}$

$\{v/[null], w/[null], y/[null]\}$

crle concrete state $\delta$  

$\delta \in \gamma(\sigma)$?

\[ \begin{align*}
&V \rightarrow \text{loc}_1 \rightarrow \cdots \\
&Y \rightarrow \text{loc}_2 \rightarrow \cdots \\
&W \rightarrow \text{loc}_3 \rightarrow \cdots 
\end{align*} \]
Concretization Function $\gamma$ — Examples

abstract state $\sigma$ | concrete state $\delta$ | $\delta \in \gamma(\sigma)?$ \\
--- | --- | --- \\
$\{\{v\}, \{w\}, \{y\}, \{v, w\},$ \\
$\{v, y\}, \{w, y\}, \{v, w, y\}\}$ | $\{v/[null], w/[null], y/[null]\}$ | yes
Concretization Function $\gamma$ — Examples

abstract state $\sigma$  

concrete state $\delta$  

$\delta \in \gamma(\sigma)$?

1. $\{\{v\}, \{w\}, \{y\}, \{v, w\}\}$  
   $\{v, y\}, \{w, y\}, \{v, w, y\}\}$  
   $\{v/\text{null}, w/\text{null}, y/\text{null}\}$  
   
   $\text{V} \rightarrow \text{loc}_1 \rightarrow \ldots$  
   $\text{Y} \rightarrow \text{loc}_2 \rightarrow \ldots$  
   $\text{W} \rightarrow \text{loc}_3 \rightarrow \ldots$  

   Yes

2. $\{\{v\}, \{w\}, \{y\}, \{v, w\}\}$  
   $\{v, y\}, \{w, y\}, \{v, w, y\}\}$  
   $\{v/\text{null}, w/\text{null}, y/\text{null}\}$  

   $\text{V} \rightarrow \text{loc}_1 \rightarrow \ldots$  
   $\text{Y} \rightarrow \text{loc}_2 \rightarrow \ldots$  
   $\text{W} \rightarrow \text{loc}_3 \rightarrow \ldots$  

   ?
Concretization Function $\gamma$ — Examples

abstract state $\sigma$

- $\{\{v\}, \{w\}, \{y\}, \{v, w\},$
- $\{v, y\}, \{w, y\}, \{v, w, y\}\}
- $\{v/[null], w/[null], y/[null]\}$

concrete state $\delta$

- $\delta \in \gamma(\sigma)?$

- $\delta \in \gamma(\sigma)$? yes
- $\delta \in \gamma(\sigma)$? invalid $\sigma$
Set Sharing Analysis

Set Sharing vs. Pair Sharing

Set sharing allows the representation of more complex abstractions than pair sharing.

- **Example**: assume an initial abstract state whose sharing component is $sh_0 = \{\{v, w\}\}$, which has the same representation in pair and set sharing. What is the resulting set sharing state after evaluating $z=v$?

  - In **pair sharing** [SS05], $SC^{PS}[z=v]sh_0 = \{\{v, w\}, \{v, z\}, \{w, z\}\}$, representable in set sharing as $\{\{v, w\}, \{v, z\}, \{w, z\}, \{v, w, z\}\} = sh_1$.

  - In **set sharing**, $SC^{SS}[z=v]sh_0 = \{\{v, w, z\}\} = sh_2$, where $sh_2 \subset sh_1$. 

Mendez-Lojo, Hermenegildo (UNM, UPM) Precise Set Sharing Analysis for Java Programs VMCAI’08, January 8 9 / 15
Set Sharing vs. Pair Sharing: Results

The experiments compare pair and set sharing domains with no nullity or type information.

\[%sh = 100 \times \left(1 - \frac{|sh|}{2^{|v|} - 1}\right)\]

<table>
<thead>
<tr>
<th></th>
<th>PS</th>
<th>SS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#sh</td>
<td>%sh</td>
</tr>
<tr>
<td>dyndisp*</td>
<td>640</td>
<td>60.37</td>
</tr>
<tr>
<td>clone</td>
<td>174</td>
<td>53.10</td>
</tr>
<tr>
<td>dfs</td>
<td>1573</td>
<td>96.46</td>
</tr>
<tr>
<td>passau*</td>
<td>5828</td>
<td>94.56</td>
</tr>
<tr>
<td>qsort</td>
<td>1481</td>
<td>67.41</td>
</tr>
<tr>
<td>intqsort</td>
<td>2413</td>
<td>66.47</td>
</tr>
<tr>
<td>pollet01*</td>
<td>793</td>
<td>89.81</td>
</tr>
<tr>
<td>zipvector*</td>
<td>6161</td>
<td>68.71</td>
</tr>
<tr>
<td>cleanness*</td>
<td>1300</td>
<td>63.63</td>
</tr>
<tr>
<td>overall</td>
<td>20363</td>
<td><strong>73.39</strong></td>
</tr>
</tbody>
</table>
Domain Combination, Multivariance

- An abstract state contains not only sharing, but also nullity and type information.
- The analysis is context sensitive and multivariant.
- The combination results in increased precision.

```java
public void append(Vector v) {
    σ₁₀ = ({{this}, {this, v}, {v}}, {this/ {nonnull}, v/ {unk}})
    σ₂₀ = ({{this}, {v}}, {this/ {nonnull}, v/ {unk}})
    if (this != v) { //append v to the end of this
        //append v to the end of this
    } else {
    }
}
```
Domain Combination, Multivariance

- An abstract state contains not only sharing, but also nullity and type information.
- The analysis is context sensitive and multivariant.
- The combination results in increased precision.

```java
public void append(Vector v) {
    σ₁₀ = ({{this}, {this, v}, {v}}, {this/ {nnull}, v/ {unk}})
    σ₂₀ = ({{this}, {v}}, {this/ {nnull}, v/ {unk}})
    if (this != v) { //append v to the end of this
        σ₁₁ = σ₂₁ = ({{this, v}}, {this/ {nnull}, v/ {nnull}})
    } else {
    }
}
```
Domain Combination, Multivariance

- An abstract state contains not only sharing, but also nullity and type information.
- The analysis is context sensitive and multivariant.
- The combination results in increased precision.

```java
public void append(Vector v) {
    \sigma_{10} = (\{\text{this}\}, \{\text{this}, v\}, \{v\}, \{\text{this}/\{\text{nonnull}\}, v/\{\text{unk}\}\})
    \sigma_{20} = (\{\text{this}\}, \{v\}, \{\text{this}/\{\text{nonnull}\}, v/\{\text{unk}\}\})
    if (this != v) { //append v to the end of this
        \sigma_{1i} = \sigma_{2i} = (\{\text{this}, v\}, \{\text{this}/\{\text{nonnull}\}, v/\{\text{nonnull}\}\})
    } else {
        \sigma_{1e} = \sigma_{10} \text{ and } \sigma_{2e} = \bot
    }
}
```
Domain Combination, Multivariance

- An abstract state contains not only sharing, but also nullity and type information.
- The analysis is context sensitive and multivariant.
- The combination results in increased precision.

```java
public void append(Vector v) {
    σ₁₀ = ({{this}, {this, v}, {v}}, {this/ {nonnull}, v/ {unk}})
    σ₂₀ = ({{this}, {v}}, {this/ {nonnull}, v/ {unk}})
    if (this != v) {
        //append v to the end of this
        σ₁ᵢ = σ₂ᵢ = ({{this, v}}, {this/ {nonnull}, v/ {nonnull}})
    } else {
        σ₁ₑ = σ₁₀ and σ₂ₑ = ⊥
    }
    σ₁ᶠ = σ₁ᵢ ⊔ σ₁ₑ = ({{this}, {this, v}, {v}}, {this/ {nonnull}, v/ {unk}})
    σ₂ᶠ = σ₂ᵢ ⊔ ⊥ = ({{this, v}}, {this/ {nonnull}, v/ {nonnull}})
}
```
Precision Improvements due to Combining Domains

The table measures the percentage of program points that are deemed to be unreachable by analysis. Since the results are correct, a larger number for $\%up$ indicates better precision.

<table>
<thead>
<tr>
<th></th>
<th>PS</th>
<th>SS</th>
<th>SSNI(\tau)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dyndisp*</td>
<td>4.22</td>
<td>4.22</td>
<td>14.08</td>
</tr>
<tr>
<td>clone</td>
<td>7.31</td>
<td>7.31</td>
<td>24.39</td>
</tr>
<tr>
<td>dfs</td>
<td>3.92</td>
<td>3.92</td>
<td>10.78</td>
</tr>
<tr>
<td>passau*</td>
<td>1.19</td>
<td>1.19</td>
<td>5.98</td>
</tr>
<tr>
<td>qsort</td>
<td>23.24</td>
<td>23.24</td>
<td>23.24</td>
</tr>
<tr>
<td>intqsort</td>
<td>22.51</td>
<td>22.51</td>
<td>22.51</td>
</tr>
<tr>
<td>pollet01*</td>
<td>18.18</td>
<td>18.18</td>
<td>36.36</td>
</tr>
<tr>
<td>zipvector*</td>
<td>1.10</td>
<td>1.10</td>
<td>9.92</td>
</tr>
<tr>
<td>cleanness*</td>
<td>11.78</td>
<td>11.78</td>
<td>15.28</td>
</tr>
<tr>
<td><strong>overall</strong></td>
<td>10.38</td>
<td>10.38</td>
<td>18.06</td>
</tr>
</tbody>
</table>
Set Sharing Analysis

Framework Pipeline

Transformation

Java Source

Java parser (in Ciao)

javac

Java bytecode

soot + Ciao

transform.

IR

Analysis

Domains

Fixpoint algorithm (AI-based)

Pre/Post pairs

Prog. Point Info ...

Mendez-Lojo, Hermenegildo (UNM, UPM) Precise Set Sharing Analysis for Java Programs VMCAI'08, January 8
Analysis Framework

- Java (bytecode) programs are compiled to a generic intermediate representation [LOPSTR’07].
- The framework derives the semantics of a program given a particular abstract domain.
- A context-sensitive, multivariant, efficient fixpoint algorithm [FTfJP’07] is at the core of the system.
- Abstract domains, like set sharing, are plugins that the analysis designer adds to the framework.
- The definition of a domain contains:
  - The abstract domain operations (partial order, least upper bound, project, extend, etc.).
  - Abstract transfer functions for the primitive operations of the language (builtins).
Q & A

Check
http://www.cliplab.org/~mario/
for related publications
P. Cousot and R. Cousot.
Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.

D. Jacobs and A. Langen.
Accurate and Efficient Approximation of Variable Aliasing in Logic Programs.

K. Muthukumar and M. Hermenegildo.
Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation.
S. Secci and F. Spoto.
Pair-sharing analysis of object-oriented programs.