Modularity in Lattices:
A Case Study on the Correspondence between Top-Down and Bottom-Up Analyses

Ghila Castelnuovo
Mayur Naik
Noam Rinetzky
Mooly Sagiv
Hongseok Yang

Tel Aviv University
Georgia Institute of Technology
Tel Aviv University
Tel Aviv University
University of Oxford

SAS’15 10-September-2015
Research problem

- A precise compositional (heap) analysis
Research problem

- A precise compositional (heap) analysis

- Compositional?
  - **Bottom-Up:** Context-independent
  - **Top-Down:** Context-dependent
Research problem

- A precise compositional (heap) analysis

- Compositional?
  - **Bottom-Up:** Context-independent
  - **Top-Down:** Context-dependent
Research problem

- A precise compositional (heap) analysis

✓ Compositional
  - Bottom-Up: Context-independent
  - Top-Down: Context-dependent
Research problem

- A precise compositional (heap) analysis
- Precise?
Research problem

- A precise compositional (heap) analysis

- Precise?
  - Precise enough for a particular client?
Challenges

- Accounting for **all** calling contexts
  - Soundness
  - Precision
  - Scalability
    - Size of procedure summaries
    - Cost of summary instantiation
Contributions

- **Modular connection analysis** [Ghiya & Hendren, ’96]*
  - Lightweight heap analysis
  - Used for parallelization

- Provably as **precise** as the top-down version
  - Top-down analysis sound (by abstract interpretation)
  - Implies **soundness**

- Experimental evaluation
  - Bottom-up **scales** much better than the top-down
  - Little loss of **precision** compared to original analysis

*Slightly modified version of the original analysis
This paper is a mere glimpse ...

- **Ghila Castelnuovo**’s Master Thesis:
  Modular lattices for compositional Interprocedural Analysis

- **Framework of compositional analysis**

- **Guaranteed precision relative to top-down analysis**

"... Mission: To explore strange new worlds, to seek out new life and new civilizations, to boldly go where no one has gone before."

(Starting in baby steps...)
\[ \lbrack \text{st} \rbrack (d) = d \sqcup C_{\text{st}} \]

- Transformers defined using \( \sqcup \)
  - \( C_{\text{st}} \) is an element in the domain
Composition by adaptation

\[ [st1; st2](d) = (d \sqcup C_1) \sqcup C_2 \]

- Transformers defined using \( \sqcup \)
  - \( C_i \) is an element in the domain
  - Recall: \( \sqcup \) is commutative, associative, idempotent

Adapt the result of analyzing \( d \) instead of analyzing \( d \sqcup d' \)!

Compositional analysis:

\[ \llbracket p()\rrbracket^\#(d \sqcup d') = \llbracket p()\rrbracket^\#(d) \sqcup d' \]
static main(){
    l1 z = new h1
    l2 w = new h2
    l3 u = new h3
    l4 v = new h4
    l5 u.f = v
    l6 if(...) z.f = w
}

Are Z and W connected?

Maybe

Are Z and U connected?

No!
Interprocedural CA can be expensive...

# of calling contexts: 1

```java
main()
{
    X = new h₁
    Y = new h₂
    p₁()
}
```
Compositional CA *(simplified)*

- Partition abstract domain
- \(\square\)-based transformers

⇒ Compositionality by adaptation
CA: Partition abstract domain

- $D = (\text{Partition}(V), \sqsubseteq) \simeq (\text{Equiv}(V), \sqsubseteq)$
  - $\text{Partition}(V)$ Set of partitioning of $V$
  - $\text{Equiv}(V)$ Set of equivalence relations over $V$
  - $\sqsubseteq$ Refinement
CA: Abstract transformers (simplified)

\[
\llbracket st \rrbracket(d) = d \sqcup C_{st}
\]

- Transformers defined using \(\sqcup\)
  - \(C_{st}\) is a constant partition, e.g., \(C_{w.f=u} = U_{w,u} = \{\{w,u\}, \{z\}, \{v\}\}\)

- \(\llbracket x = \text{null} \rrbracket\#(d) = d\)
- \(\llbracket x = \text{new} \rrbracket\#(d) = d\)
- \(\llbracket x.f = y \rrbracket\#(d) = d \sqcup U_{xy}\)
- \(\llbracket x = y \rrbracket\#(d) = d \sqcup U_{xy}\)
- \(\llbracket x = y.f \rrbracket\#(d) = d \sqcup U_{xy}\)
CA: Abstract transformers (simplified)

\[
\left[ \text{st} \right](d) = d \sqcup C
\]

- Transformers defined using \(\sqcup\)
  - \(C_{\text{st}}\) is a constant partition, e.g., \(C_{w=u} = U_{w,u} = \{\{w,u\}, \{z\}, \{v\}\}\)

- \(\left[ x = \text{null} \right](d) = d\)
- \(\left[ x = \text{new} \right](d) = d\)
- \(\left[ x.f\ y \right](d) = d \sqcup U_{xy}\)
- \(\left[ x = y \right](d) = d \sqcup U_{xy}\)
- \(\left[ x = y.f \right](d) = d \sqcup U_{xy}\)
CA: Abstract transformers

(Moving on towards the real thing ...)
CA: Abstract transformers

- Transformers defined using \( \sqcup \) and \( \sqcap \)
  - \( U_{x,y} = \{ \{x,y\}, \{z\}, \{w\} \} \)
  - \( S_x = \{ \{x\}, \{y,z,w\} \} \)

\[
\begin{align*}
[x = \text{null}]^\#(d) &= d \sqcap S_x \\
[x = \text{new}]^\#(d) &= d \sqcap S_x \\
[x.f = y]^\#(d) &= d \sqcup U_{xy} \\
[x = y]^\#(d) &= (d \sqcap S_x) \sqcup U_{xy} \\
[x = y.f]^\#(d) &= (d \sqcap S_x) \sqcup U_{xy}
\end{align*}
\]
CA: Abstract transformers

- Transformers defined using $\square$ and $\sqcap$
  - $U_{x,y} = \{ \{x,y\}, \{z\}, \{w\}\}$
  - $S_x = \{ \{x\}, \{y,z,w\}\}$

- $[x = \text{null}](d) = d \sqcap S_x$
- $[x = \text{new}](d) = d \sqcap S_x$
- $[x.f = y](d) = d \cup U_{xy}$
- $[x = y](d) = (d \sqcap S_x) \sqcup U_{xy}$
- $[x = y.f](d) = (d \sqcap S_x) \sqcup U_{xy}$
Can we use adaptation?

- Transformers defined using $\Box$ and $\sqcap$
  - $U_{x,y} = \{\{x,y\}, \{z\}, \{w\}\}$
  - $S_x = \{\{x\}, \{y,z,w\}\}$

Goal: composition by adaptation

$p() \# (d) \sqcup p() \# (d')$
Modularity in Lattices

- For adaptation: $\left( d \sqcup d' \right) \cap d_p = (d \cap d_p) \sqcup d'$
Modularity in Lattices

- For adaptation: \((d \sqcup d') \cap d_p = (d \cap d_p) \sqcup d'\)

- An element \(d_p\) in a lattice \(D\) is **right modular** iff
  \[\forall d, d' \in D. \quad \text{if } d' \sqsubseteq d_p \quad \text{then } (d \sqcup d') \cap d_p = (d \cap d_p) \sqcup d'\]

- \(D\) is **modular** if all its elements are right modular

- The **partition domain** is **NOT** a modular lattice
- But it is modular enough ...
Conditionally adaptable transformers

- CA Transformers: \[
\llbracket \text{st} \rrbracket \#(d) = (d \cap S_x) \sqcup U_{x,y}
\]

- \(U_{x,y}\) and \(S_x\) are right-modular

\[\Rightarrow\] Conditionally adaptable transformers

- \(\forall d, d' \in D.\) if \(d' \subseteq U_{x,y} \ldots\) and \(d' \subseteq S_x \ldots\)
  then \(\llbracket \text{st} \rrbracket \#(d \sqcup d') = \llbracket \text{st} \rrbracket \#(d) \sqcup d'\)
Compositional connection analysis

- Intra-procedural analysis is **conditionally adaptable**
  - Delay the operation of a join \((d \sqcup d')\)
  - Adapt the result

- Inter-procedural analysis is **unconditionally adaptable**!
  \[\Rightarrow\] Hence, compositional
Compositional connection analysis

(We are now at warp 7)
Compositional connection analysis

Procedure call are **conditionally adaptable**
- Represent any procedure inputs as \( d = t \sqcup d' \)
- \( \forall st. st = ( \ldots \sqcap d_p \ldots ) \Rightarrow d' \subseteq d_p \)
- \( t \) is a particular element in the **Triad Domain**

**Phase I** Analyze every procedure **once** on \( t \)

**Phase II** Instantiate \( p(t) \) with information from call context
Who is \( \iota \) ?

\[ D[w, \overline{w}, x, \overline{x}, y, \overline{y}, z, \overline{z}] \]

\[ \text{Id} \]

Diagram: Network of nodes labeled with variables and their negations.
Triad domain

- Partition domain comprised of
  - $G$ current values: $x, y, x$
  - $\bar{x}, \bar{y}, \bar{z}$
  - $G$ auxiliary (temporary) values: $\hat{x}, \hat{y}, \hat{z}$
    - To compute effect of procedure calls using "Relational join"

Globals

Locals

- $L$ Current local variables
Entering a procedure (top-down)

\[ \lbrack \text{entry} \rbrack(d) = (d \cap R_G) \cup l \]

\[ R_G = \{ G, \{ \bar{x} \}, \{ \dot{x} \} | x \in G \} \]

\( p() \{ \\
\ldots \\
q() ; \\
} \)

\[ \prod_{R_G} \]

\[ \sqcup \]

\[ l \]
“Entering a procedure” (bottom-up)

\[
[\text{entry}](d) = 1
\]
Returning from a procedure (TD & BU)

$$\llbracket \text{return} \rrbracket(d_{\text{exit}}, d_{\text{call}}) = (f_{\text{call}}(d_{\text{call}}) \sqcup f_{\text{exit}}(d_{\text{exit}})) \cap R_{\bar{G} \cup G}$$

- Rename current to •
- Rename input to •
Returning from a procedure (TD & BU)

\[
\llbracket \text{return} \rrbracket (d_{\text{exit}}, d_{\text{call}}) = (f_{\text{call}}(d_{\text{call}}) \cup f_{\text{exit}}(d_{\text{exit}})) \cap R\tilde{G} \cup \tilde{G},
\]

\begin{align*}
p() \{ \\
\quad \ldots \\
\quad q(); \\
\}
\end{align*}

\begin{align*}
q() \{ \\
\quad z = \text{new}(); \\
\quad z.f = v \\
\}
\end{align*}
Returning from a procedure (TD & BU)

\[
\llbracket \text{return} \rrbracket (d_{\text{exit}}, d_{\text{call}}) =
\left( f_{\text{call}}(d_{\text{call}}) \uplus f_{\text{exit}}(d_{\text{exit}}) \right) \cap R_{\overline{G} \cup G},
\]

```plaintext
p() {
    ...
    q();
}
```

```plaintext
q() {
    z = new()
    z.f = v
}
```
Returning from a procedure (TD & BU)

\[
\llbracket \text{return} \rrbracket(d_{\text{exit}}, d_{\text{call}}) = (f_{\text{call}}(d_{\text{call}}) \sqcup f_{\text{exit}}(d_{\text{exit}})) \cap R_{\bar{G} \cup G},
\]

\begin{align*}
p() & \{ \\
& \ldots \\
& q(); \\
\} \\
q() & \{ \\
& \text{z = new()} \\
& \text{z.f = v} \\
& \}
\end{align*}
Coincidence Theorem

\[
\llbracket \text{return} \rrbracket (\llbracket C_{\text{body}} \rrbracket \circ \llbracket \text{entry} \rrbracket (d), d) = \llbracket \text{return} \rrbracket (\llbracket C_{\text{body}} \rrbracket (t), d)
\]

\[
\llbracket p() \rrbracket \text{ bottom-up}
\]

\[
\llbracket p() \rrbracket \text{ top-down}
\]
Where is the magic?

- The magic is in the proof!
The magic is in the proof!

- Proof shows that effect of calling context can be delayed
- Non-trivial
  - But rewarding
- Key observations
  - Uniform entry states
  - Counterpart representation
  - ....
Uniform entry states

\[
\llbracket \text{entry} \rrbracket (d) = (d \cap R_G) \sqcup l \quad \text{vs} \quad \llbracket \text{entry} \rrbracket (d) = l
\]

\( R_G = \{ G, \{ x \}, \{ x \} \mid x \not\in G \} \)

\( p() \{ \\
\quad \ldots \\
\quad q(); \\
\}

\[ \prod_{R_G} \]

\[ l = \]
Uniform entry states

\[
\lceil \text{entry} \rceil (d) = (d \cap R_G) \cup l \quad \text{vs} \quad \lceil \text{entry} \rceil (d) = l
\]

\( R_G = \{G, \{x\}, \{x\} \mid x \in G\} \)

\begin{align*}
p() \{ & \quad \ldots \quad \\
& \quad q() ; \\
\} &
\end{align*}
Uniform entry states

\[
\llbracket\text{entry}\rrbracket(d) = (d \cap R_G) \sqcup \perp \quad \text{vs} \quad \llbracket\text{entry}\rrbracket(d) = \perp
\]

\(R_G = \{G, \{x\}, \{x\} \mid x \in G\}\)

Conditionally adaptable

\(U_{w,x} \quad U_{y,z} \quad \sqsubseteq d_p\)
Counterpart representation

\[
[entry](d) = (d \cap R_G) \cup \bot \quad \text{vs} \quad [entry](d) = \bot
\]

\[\text{R}_G = \{G, \{x\}, \{x\} \mid x \in G\}\]
$[[\text{entry}]](d) = (d \cap R_G) \sqcup l$ vs $[[\text{entry}]](d) = l$

$R_G = \{G, \{x\}, \{x\} | x \in G\}$
$$\left\lfloor \text{entry} \right\rfloor (d) = (d \cap R_G) \sqcup l \quad \text{vs} \quad \left\lfloor \text{entry} \right\rfloor (d) = l$$

$$R_G = \{G, \{x\}, \{x\} | x \in G\}$$
Counterpart representation

\[
\llbracket \text{entry} \rrbracket(d) = (d \cap R_G) \sqcup I \quad \text{vs} \quad \llbracket \text{entry} \rrbracket(d) = I
\]

\[R_G = \{G, \{x\}, \{x\} \mid x \in G\}\]

\[p() \{
    \ldots
    q() ;
\}

\[U_{w,x} \sqsubseteq R_G\]
Experimental results
Experimental results

- Compared 3 versions of connection analysis
  - Original top-down
  - Triad top-down
  - Triad bottom-up (compositional)

\[
\text{Original} \\
\left\{ \begin{array}{ll}
\text{Merge} & x \neq \text{null} \land y \neq \text{null} \\
\text{Skip} & \text{otherwise}
\end{array} \right.
\]

\[
\text{Ours} \\
\left\{ \begin{array}{ll}
\text{Merge}
\end{array} \right.
\]
## Experimental setup (DaCapo)

<table>
<thead>
<tr>
<th>Description</th>
<th>Method(s)</th>
<th>Bytecodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grande2 Java Grande Kernels</td>
<td>237</td>
<td>13,724</td>
</tr>
<tr>
<td>Grande3 Java Grande Large apps</td>
<td>1,162</td>
<td>75,139</td>
</tr>
<tr>
<td>Antlr Parser generator</td>
<td>2,400</td>
<td>128,684</td>
</tr>
<tr>
<td>Weka Machine Learning Library</td>
<td>3,391</td>
<td>223,291</td>
</tr>
<tr>
<td>Bloat Optimizations and Analysis tool</td>
<td>4,699</td>
<td>311,727</td>
</tr>
</tbody>
</table>

- JRE 1.6; Linux; Intel Xeon 2.13GHz; 123GB RAM
- Using Chord program analysis framework
Experimental evaluation

Precision
- Near perfect overlap
- Only 2-5% is lost

- Original top-down
- bottom-up (= modified top-down)
# Experimental evaluation

## Scalability

<table>
<thead>
<tr>
<th></th>
<th>Bottom-up</th>
<th>Original Top-down</th>
<th>Triad Top-down</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Summaries computation</td>
<td>instantiation</td>
<td></td>
</tr>
<tr>
<td>Grande2</td>
<td>0.6 sec</td>
<td>0.9 sec</td>
<td>1 sec</td>
</tr>
<tr>
<td></td>
<td>0.9 sec</td>
<td></td>
<td>0.9 sec</td>
</tr>
<tr>
<td>Grande3</td>
<td>43 sec</td>
<td>1:21 min</td>
<td>1:11 min</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>51 sec</td>
</tr>
<tr>
<td>Antlr</td>
<td>16 sec</td>
<td>30 sec</td>
<td>1:23 sec</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25 sec</td>
</tr>
<tr>
<td>Weka</td>
<td>46 sec</td>
<td>2:48 min</td>
<td>Timeout!</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Timeout!</td>
</tr>
<tr>
<td>Bloat</td>
<td>3:03 min</td>
<td>30 min</td>
<td>Timeout!</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Timeout!</td>
</tr>
</tbody>
</table>
Experimental evaluation

Scalability

- Top down blows-up

### Graphs

**Scalability Graphs**

- **Weka**
  - X-axis: # of incoming abstract states
  - Y-axis: % statements

- **Grande2**
  - X-axis: # of incoming abstract states
  - Y-axis: % statements

- **Bloat**
  - X-axis: # of incoming abstract states
  - Y-axis: % statements

Legend:

- • Modified top-down
- • Original top-down
- ▲ Bottom-up
Related work

- **General theory** [Cousot & Cousot, CC’02]

- **Modular analysis for logical programs**
  [Codish et al. POPL’03] [Giacoabazi, JLP’98]

- **Abstract domain for modular analyses**
  [Giacoabazi et al., TCS’99]

- **Condensation** and modular analyses
  [Giacoabazi et al. TOCL’05, TOPLAS’98]
  - Condensing abstract domains allow to derive bottom-up analyses with the same precision as top-down ones
  - Lattice-theoretic characterization:
    \[ F(a \otimes b) = a \otimes F(b) \]
Limitations and future work

- Transfer functions are input-independent
  - Limited expressivity

- Generalize to other instances
  - Copy constant propagation
  - Taint analysis

- Castelnuovo’s thesis has a general framework
  - But it is still rather restricted
Summary

- A precise scalable compositional heap analysis

```
Var g1, g2, ...
dah()
  foo()
...
```

```
main()
  bar()
...
```

```
dah()
  foo()
...
```

```
bar()
  dah()
  foo()
...
```

```
foo()
  ...
```

Top Down

Bottom up
Thank you!