A Formal Design of a Tool for Static Analysis of Upper Bounds on Object Calls in Java

FMICS 2012

Konrad Siek and Paweł T. Wojciechowski

Poznań University of Technology

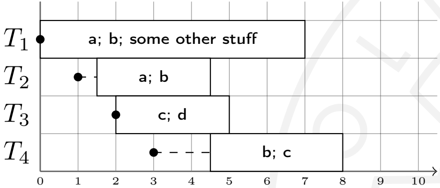
{konrad.siek,pawel.t.wojciechowski}@cs.put.edu.pl

27 August 2012

Motivation

Atomic RMI:

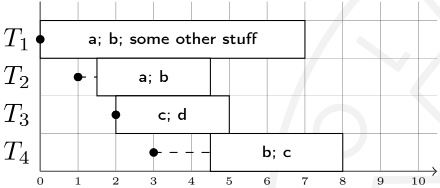
- distributed software transactional memory based on RMI,
- *Supremum Versioning Algorithm*—pessimistic concurrency control:



Motivation

Atomic RMI:

- distributed software transactional memory based on RMI,
- *Supremum Versioning Algorithm*—pessimistic concurrency control:

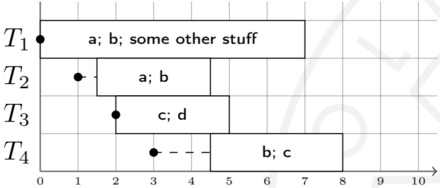


- requires resource access information *a priori* (which resources, how many times).

Motivation

Atomic RMI:

- distributed software transactional memory based on RMI,
- *Supremum Versioning Algorithm*—pessimistic concurrency control:



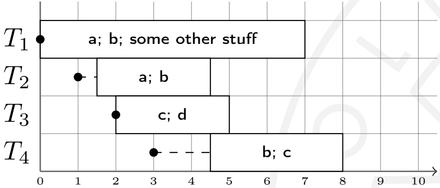
- requires resource access information *a priori* (which resources, how many times).

Defining accesses manually is a **nuisance**.

Motivation

Atomic RMI:

- distributed software transactional memory based on RMI,
- *Supremum Versioning Algorithm*—pessimistic concurrency control:



- requires resource access information *a priori* (which resources, how many times).

Defining accesses manually is a **nuisance**.
Goal: automatic **inferrence of upper bounds** and source code instrumentation.

Problem definition

Upper bound analysis

For each object o in a given program p:

Find the (smallest) upper bound on the number of times o will be used during any execution of p.

Applications

Other applications:

- analysis of interactions among threads,
- compile-time resource optimization,
- automatic refactoring and code re-writing,
- scheduling and synchronization algorithms.

Upper bound analysis overview

Components of the upper bound analysis:

- prepare an intermediate representation (*Control Flow Graph*),
- **value analysis**,
  - predict values of variables at every node of the CFG,
  - while at it, predict control flow,
- region finding—transform Jimple CFG to regions,
- **call count analysis**—predict upper bounds on method calls.

Intermediate language

Intermediate language

Jimple:

- intermediate language for Java,
- three address code ($r \leftarrow op\ a_1 ... a_n$),
- 17 instructions (although no loops),
- typesafe,
- Soot framework.

Jimple example

Java code:

```
int sum = 0;
for (int i = 0; i < resources.length; i++) {
   sum += resources[i].getBalance();
}
return sum;
```

Equivalent Jimple code:

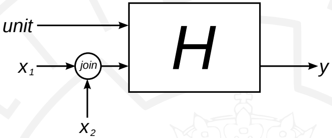
```
  sum = 0;
  i = 0;
label0:
  temp$1 = lengthof resources;
  if i < temp$1 goto label1;
  goto label2;
label1:
  temp$2 = resources[i];
  temp$3 = interfaceinvoke
         temp$2.<Resource: int getBalance()>();
  sum = sum + temp$3;
  i = i + 1;
  goto label0;
label2:
  return sum;
```

Value analysis

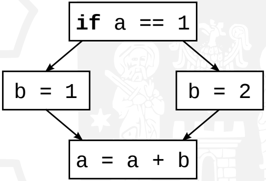
Structure of value analysis

Components of a data flow analysis:

- transfer function H,
- input data x and output data y,
- a join operator.



Apply H to every unit in the CFG.



Value analysis

- State is a quadruple $\mathbb{S} = (\mathbb{S}_V, \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I)$,
  - value map $\mathbb{S}_V$,
  - value prediction map $\mathbb{S}_P$,
  - unused edge set $\mathbb{S}_D$,
  - iteration count $\mathbb{S}_I$.

- Join operator join$(\mathbb{S}^1, ..., \mathbb{S}^n)$:
  - variables defined in any state are included in the result,
  - if a variable is defined in several states becomes ambiguous: $\{x \mapsto \{1\}\}$ joined with $\{x \mapsto \{2\}\}$ becomes $\{x \mapsto \{1, 2\}\}$.

Value analysis: selected cases

- Assignment $j = r$:

  $\mathbb{S}_V[j \mapsto \{\mathsf{val}(r, \mathbb{S}_V)\}]$.

- Conditional statement $\texttt{if}\ p\ \texttt{goto}\ l_1\ \texttt{else}\ l_2$:
  - if p evaluates to `true`:

    $\mathbb{S}_D \cup \{(s, l_2)\}$ and $\mathbb{S}_P\left[l_1 \mapsto \mathbb{S}_P[p \mapsto \texttt{true}]\right]$,
  - if p evaluates to `false`:

    $\mathbb{S}_D \cup \{(s, l_1)\}$ and $\mathbb{S}_P\left[l_2 \mapsto \mathbb{S}_P[p \mapsto \texttt{false}]\right]$,
  - otherwise:

    $\mathbb{S}_P\left[l_1 \mapsto \mathbb{S}_P[p \mapsto \texttt{true}], l_2 \mapsto \mathbb{S}_P[p \mapsto \texttt{false}]\right]$.

Value analysis: loop unfolding

Unfolding—evalloop$(h, \mathbb{G}, \mathbb{U}, i, L)$:

- Analyze all loop the statements u giving $\mathbb{G}'(u)$.
- Find all back edges except for unused ones $\mathbb{Z}$.
- If all members of $\mathbb{Z}$ are unused then finish:

  $\mathbb{S}_I[h \mapsto i]$.
- Otherwise analyze again:

  evalloop$(h, \mathbb{G}', \mathbb{U}, i + 1, L)$.

Value analysis: loop unfolding

Unfolding—evalloop$(h, \mathbb{G}, \mathbb{U}, i, L)$:

- Analyze all loop the statements u giving $\mathbb{G}'(u)$.
- Find all back edges except for unused ones $\mathbb{Z}$.
- If all members of $\mathbb{Z}$ are unused then finish:

  $\mathbb{S}_I[h \mapsto i]$.
- Otherwise analyze again:

  evalloop$(h, \mathbb{G}', \mathbb{U}, i + 1, L)$.
- However, if the limit of iterations L was exceeded:

  $\mathbb{S}_V[k \mapsto \omega, k \in \text{defs}(\mathbb{U})]$ and $\mathbb{S}_I[h \mapsto \omega]$.

Value analysis: method invocation

Invocation `invoke` $i.[j(j_1, ..., j_n)](i_1, ..., i_n)\{b_1, ..., b_m\}$:

- If depth limit is not exceeded:

  $\mathsf{join}(\mathsf{eval}(\mathbb{S}', b_1), ..., \mathsf{eval}(\mathbb{S}', b_m))$.

- Otherwise:

  $\mathbb{S}_V[k \mapsto \omega, \text{ where } k \in \mathsf{defs}(b_1) \cup ... \cup \mathsf{defs}(b_m)]$

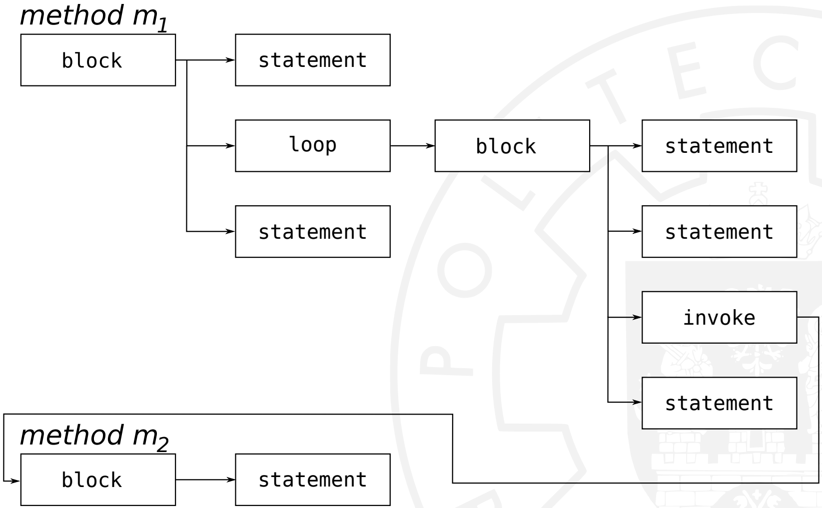
Regions

Regions

A region-based intermediate representation:

$$
\begin{array}{llcll}
\text{Unit regions} & U & \in & \mathit{Units} & ::= & \texttt{unit} \\
\text{Statement regions} & S & \in & \mathit{Statements} & ::= & \texttt{statement}\ s \\
\text{Invocation regions} & I & \in & \mathit{Invocations} & ::= & \texttt{invoke}\ j, R_1, ..., R_m, s \\
\text{Block regions} & B & \in & \mathit{Blocks} & ::= & \texttt{block}\ [R_1, ..., R_n] \\
\text{Condition regions} & C & \in & \mathit{Conditions} & ::= & \texttt{condition}\ p, R_1, R_2 \\
\text{Loop regions} & L & \in & \mathit{Loops} & ::= & \texttt{loop}\ h, R \\
\text{Regions} & R & \in & \mathit{Regions} & ::= & U \mid S \mid I \mid B \mid C \mid L
\end{array}
$$

Regions: region tree example



Call count analysis

Call count analysis

State is a map of objects to their upper bounds $\{\text{obj} \mapsto \mathbb{N} \cup \omega\}$.

Transfer function ccount(n, R).

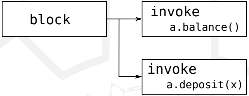
Two join operators:

- for sequencial regions,
- for alternative regions.

Call count analysis: blocks

Block `block` $[R_1, ..., R_n]$:

$$\text{addjoin}(\text{ccount}(n, R_1), ..., \text{ccount}(n, R_n)).$$



Sequential join addjoin$(\mathbb{M}_1, ..., \mathbb{M}_n)$:

$$\{k \mapsto \mathbb{M}_1(k) + ... + \mathbb{M}_n(k) \mid k \in \text{dom } \mathbb{M}_1 \cup ... \cup \text{dom } \mathbb{M}_n\}.$$

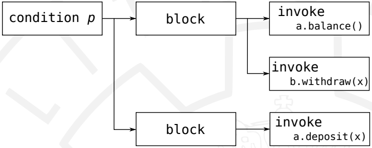
Call count analysis: conditions

Condition `condition` p, R_1, R_2, s:

- if p evaluates to `true`,

  ccount(n, R_1),
- if p evaluates to `false`,

  ccount(n, R_2),



- otherwise,

  maxjoin(ccount(n, R_1), ccount(n, R_2)).

Join for alternative evaluations maxjoin$(\mathbb{M}_1, ..., \mathbb{M}_n)$:

$$\{k \mapsto \max(\mathbb{M}_1(k), ..., \mathbb{M}_n(k)) \mid k \in \mathrm{dom}\ \mathbb{M}_1 \cup ... \cup \mathrm{dom}\ \mathbb{M}_n\}$$

Call count analysis: transfer function

- Loop `loop` h, R:

  ccount$(n * \mathbb{S}_I(h), R)$

- Invocation `invoke` $j, R_1, ..., R_m, s$:
  - mark the invocation of a method on object j:

    $\{\mathbb{S}_V(j) \mapsto n\}$,
  - analyze each possible method body:

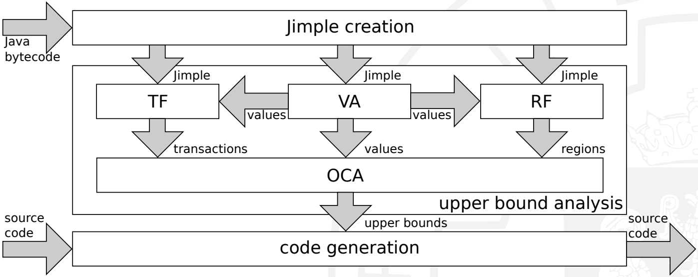
    maxjoin(ccount$(n, R_1), ...,$ ccount$(n, R_m)))$.

Guarantees

- All objects found (completeness).
- Upper bound never lower than actual number of accesses (safety).
- Static analysis finishes.

Implementation

Architecture



Implementation

Implementation details:

- identification of transactional elements (TF),
- code instrumentation (generator),
- method invocation stack,
- expression evaluation,
- code transformations and graph pruning,
- use of Soot tools where possible:
  - graphs,
  - flowsets,
  - domination and loop finders.

Code generation

Transactional code:

```
t = new Transaction(registry);


t.start();

balance = accountA.getBalance();
if (balance < 200) {
   t.rollback();
} else {
   accountA.withdraw(200);
   accountB.deposit(200);
   t.commit();
}
```

Code generation

Transactional code:

```
t = new Transaction(registry);


t.start();

balance = accountA.getBalance();
if (balance < 200) {
   t.rollback();
} else {
   accountA.withdraw(200);
   accountB.deposit(200);
   t.commit();
}
```

Processed transactional code:

```
t = new Transaction(registry);
t.accesses(accountA, 2);
t.accesses(accountB, 1);
t.start();

balance = accountA.getBalance();
if (balance < 200) {
   t.rollback();
} else {
   accountA.withdraw(200);
   accountB.deposit(200);
   t.commit();
}
```

Conclusion

Conclusion

Completed aspects:

- Static analysis:
  - value analysis,
  - regions,
  - object call count analysis.
- Implementation with code generator.

Further work:

- Performance and memory usage.

Contact information

Konrad Siek <konrad.siek@cs.put.edu.pl>
Paweł T. Wojciechowski <pawel.t.wojciechowski@cs.put.edu.pl>

INNOWACYJNA GOSPODARKA
NARODOWA STRATEGIA SPÓJNOŚCI

http://www.it-soa.eu/atomicrmi