Abstract interpretation

Application to stack allocation and synchronization elimination in Java™

Bruno Blanchet
INRIA Rocquencourt
Bruno.Blanchet@inria.fr

February 28th, 2001
Projet MOSCOVA
MObilité, Sécurité, COncurrence, Vérification et Analyse

- Join-calculus: a new model of distributed programming
- Security
- Validation and debugging of concurrent software (Caml garbage collection, Ariane 5)
Part I

Abstract interpretation
Abstract interpretation

- Static analysis: determine runtime properties of programs without executing them.

- However, exact static analysis undecidable for most properties.

- $\Rightarrow$ perform approximations, but safe approximations.
An example: signs - first idea

Given a program, determine the sign of each variable at each program point.

<table>
<thead>
<tr>
<th>Integer</th>
<th>Sign</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>&gt; 0</td>
</tr>
<tr>
<td>-3</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

B. Blanchet
An example: signs - abstraction

Sets of integers \xrightarrow{\text{concretization}} \text{Sign} \xrightarrow{\text{abstraction}}

\begin{align*}
\leq 0 & \quad \neq 0 & \quad \geq 0 \\
< 0 & \quad \neq 0 & \quad > 0 \\
\bot & \quad \neq 0 & \quad \top
\end{align*}
An example: signs - abstraction

Sets of integers → Concretization → Sign

Lattice

B. Blanchet
An example: signs - analysis of a program

\[
x = -5;
1:
while (x < 0) \{
  2:
  x++;  
  3:

1: \quad 2: \quad 3: \quad 4:
\begin{array}{|c|c|c|c|}
\hline
< 0 & \bot & \bot & \bot \\
\hline
\end{array}
\]

4:
An example: signs - analysis of a program

\[
x = -5;
1:
while (x < 0) \{
  2:
  x++; 
  3:
\}
4:
\]

\[
\begin{array}{|c|c|c|c|}
\hline
& 1: & 2: & 3: & 4: \\
\hline
< 0 & \bot & \bot & \bot \\
< 0 & < 0 & \leq 0 & \bot \\
\hline
\end{array}
\]

B. Blanchet
An example: signs - analysis of a program

x = -5;
1:
while (x < 0) {
    2: x++;
    3: < 0
    4: < 0
\begin{tabular}{|c|c|c|c|}
\hline
1: & 2: & 3: & 4: \\
\hline
< 0 & ⊥ & ⊥ & ⊥ \\
< 0 & < 0 & ≤ 0 & ⊥ \\
< 0 & < 0 & ≤ 0 & 0 \\
\hline
\end{tabular}
4:
Historical background

- Invented by Patrick and Radhia Cousot.

- The first papers: POPL’77 and POPL’79.

- Two important papers: Journal of Logic and Computation, 1992 and PLILP’92.
Achievements and future perspectives

- Verification of Ariane 5 software after the crash.

- Project of verification of plane softwares (Airbus A380).

- Projects of verification of Java programs, cryptographic protocols, ... Semantic watermarking...
Part II

Escape analysis
Introduction: what is escape analysis?

- Consider an object $o$ allocated in a method $m$. Does $o$ escape from $m$?
  $\iff$ Is $o$ still reachable after the return from $m$?

- Abstract interpretation interprocedural analysis.
Applications

- Stack allocation: object $o$ does not escape from $m$
  \[\Rightarrow o \text{ can be allocated on the stack in } m.\]
  13 to 95% of data stack allocated

- Synchronization elimination: object $o$ does not escape
  \[\Rightarrow o \text{ is local to the current thread}\]
  \[\Rightarrow \text{no need to synchronize calls on } o.\]
  more than 20% of synchronizations eliminated on most programs, 94 and 99% on two examples

Speedup up to a 1.75 factor (geometric mean 1.27).

B. Blanchet
class LimVect {
    int count = 0;
    Object[] el;

    LimVect(int n) { el = new Object[n]; }
    void put(Object o) { el[count++] = o; }
    Object get(int n) { return el[n]; }

    static Object run() {
        LimVect local = new Lim Vect(4);
        local.put(new Integer(1));
        return local.get(0);
    }
}
Example

class LimVect {
    int count = 0;
    Object[] el;

    LimVect(int n) { el = new Object[n]; }
    void put(Object o) { el[count++] = o; }
    Object get(int n) { return el[n]; }

    static Object run() {
        LimVect local = new LimVect(4);
        local.put(new Integer(1));
        return local.get(0);
    }
}

Heap

LimVect local

Object[] el

int count

1

result of run

B. Blanchet
The same example with stack allocation

```java
static Object run() {
    LimVect local = alloca_new LimVect();
    local.el = alloca_new Object[n];
    local.put(new Integer(1));
    return local.get(0);
}
```
The same example with stack allocation

static Object run() {
    LimVect local = alloca_new LimVect();
    local.el = alloca_new Object[n];
    local.put(new Integer(1));
    return local.get(0);
}
The same example with stack allocation

```java
static Object run() {
    LimVect local =alloca_new LimVect();
    local.el =alloca_new Object[n];
    local.put(new Integer(1));
    return local.get(0);
}
```
Synchronization elimination example

```java
{  
    java.util.Random r = new java.util.Random(RunTests.seed);
    for(int i = 0; i < length; i++)
    {
        array[i] = r.nextInt(); // synchronization
    }
}
```

r does not escape, so is local to the current thread, so the call to `r.nextInt()` does not need to be synchronized.

B. Blanchet
Abstraction

exact information  approximate information
What part of each value escapes?  escape context

concrete domain  abstract domain

integer

B. Blanchet
Definition of type heights

Height of a type $\tau = \top[\tau] \in \mathbb{N}$.
Main property: if $\tau$ contains $\tau'$ (as a field), $\top[\tau] \geq \top[\tau']$
If that does not contradict the preceding property, and $\tau$ contains $\tau'$, $\top[\tau] \geq \top[\tau'] + 1$

```java
class LimVect {
    int count;
    Object[] el;
}
```

B. Blanchet

16
Definition of type heights: recursive types

Height of a type $\tau = \top[\tau] \in \mathbb{N}$.
Main property: if $\tau$ contains $\tau'$ (as a field), $\top[\tau] \geq \top[\tau']$
If that does not contradict the preceding property, and $\tau$ contains $\tau'$, $\top[\tau] \geq \top[\tau'] + 1$

```java
class Tree {
    Object element;
    Tree[] sons;
}
```

B. Blanchet
Definition of contexts

The escaping part of an object is represented by the escape context of the object. The escape context is the height $T[\tau] \in \mathbb{N}$ of the type $\tau$ of the escaping part.

$$\text{Ctx} \equiv \mathbb{N}$$

![Diagram showing the escape context of an object]
Why integers?

- Integers provide a fast analysis.
- They also provide enough precision in practice.
  
  In particular, precise information for the top of the data structure.

Historically, Park and Goldberg PLDI’92, extended and improved by Deutsch POPL’97 and Blanchet POPL’98, OOPSLA’99.

B. Blanchet
Structure of the compiler

Bytecode

Escape analyzer

TurboJ compiler

C code

C compiler

Native code

B. Blanchet
## Our benchmarks

Pentium MMX 233 MHz, 128 Mb RAM, Jdk 1.1.5.

<table>
<thead>
<tr>
<th>Benchmark programs</th>
<th>Size (kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dhry</td>
<td>6</td>
</tr>
<tr>
<td>Symantec</td>
<td>19</td>
</tr>
<tr>
<td>javac</td>
<td>600</td>
</tr>
<tr>
<td>turboJ</td>
<td>788</td>
</tr>
<tr>
<td>JLex</td>
<td>89</td>
</tr>
<tr>
<td>jess</td>
<td>402</td>
</tr>
<tr>
<td>javacc</td>
<td>497</td>
</tr>
</tbody>
</table>

B. Blanchet
Stack allocation and synchronization elimination: a precise analysis

B. Blanchet
Understanding the origin of speedups

Speedup coming from stack allocation have 3 causes.

- Main cause: decrease of the GC workload;
- Stack allocation is faster than heap allocation with a mark and sweep GC;
- Better data locality.
Speedup

- dhry
- Symantec
- javac
- turboJ
- JLex
- jess
- javacc

- Total
- Inlining
- GC
- Cache miss
- Synchro

B. Blanchet
Analysis time: A fast analysis

% of compilation time

Analysis
C generation overhead
C compilation overhead

dhry  Symantec  javac  turboJ  JLex  jess  javacc

B. Blanchet
Conclusion

- Very reasonable cost.

- High speedups: factor 1.27 on average (geometric mean).
  
  - Speedup coming from a decrease of the GC workload, and to a less important extent, from a better data locality.
  
  - Stack allocation gives larger speedups with a mark and sweep GC.
  
  - Synchronization elimination gives important speedups.