Numerical static analysis with Soot

Gianluca Amato

Università “G. d’Annunzio” di Chieti–Pescara

ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis SOAP 2013

(joint work with Francesca Scozzari and Simone Di Nardo Di Maio)
Jandom

JVM-based Analyzer for Numerical DOMains

- forward intra-procedural analyses
- numerical properties
- different target languages
  - a simple C-style imperative language
  - linear transition systems
  - Baf, Jimple (sort of . . .)
- written in Scala (JVM-based comes from here)

NEW features

- inter-procedural summary-based analysis
- pair sharing analyses
Jandom

JVM-based Analyzer for Numerical DOMains

- forward intra-procedural analyses
- numerical properties
- different target languages
  - a simple C-style imperative language
  - linear transition systems
  - Baf, Jimple (sort of...)
- written in Scala (JVM-based comes from here)

NEW features

- inter-procedural summary-based analysis
- pair sharing analyses
Jandom

JVM-based Analyzer for Numerical DOMains

- forward intra-procedural analyses
- numerical properties
- different target languages
  - a simple C-style imperative language
  - linear transition systems
  - Baf, Jimple (sort of...)
- written in Scala (JVM-based comes from here)

NEW features

- inter-procedural summary-based analysis
- pair sharing analyses

HELP! looking for new acronym
Jandom architecture

Interprocedural analyzer

Flow graph analyzer

Basic block analyzer

ASM
Jimple
Baf

Abstract environment

Basic domains

native
PPL
APRON

Interpretation
Jandom architecture

Interprocedural analyzer

Flow graph analyzer

Basic block analyzer
- ASM
- Jimple
- Baf

Abstract environment

Basic domains
- native
- PPL
- APRON

Interpretation
Basic domains describe general properties of program executions and are not tied to a specific target language.

- several families of basic domains
  - numerical domains
  - sharing domains
- each family has its own API
- all basic domains support:
  - lattice operations
  - widening (upper bound which guarantees termination)
- similar to a FlowSet in Soot but
  - immutable
  - type safe
  - no collection-style methods such as add, iterator, etc...
Basic domains
describe general properties of program executions and are not tied to a specific target language.

- several families of basic domains
  - numerical domains
  - sharing domains

- each family has its own API
- all basic domains support:
  - lattice operations
  - widening (upper bound which guarantees termination)

- similar to a FlowSet in Soot but
  - immutable
  - type safe
  - no collection-style methods such as add, iterator, etc...
Basic domains

Basic domains describe general properties of program executions and are not tied to a specific target language.

- several families of basic domains
  - numerical domains
  - sharing domains

- each family has its own API

- all basic domains support:
  - lattice operations
  - widening (upper bound which guarantees termination)

- similar to a FlowSet in Soot but
  - immutable
  - type safe
  - no collection-style methods such as add, iterator, etc...
Basic domains describe general properties of program executions and are not tied to a specific target language.

- several families of basic domains
  - numerical domains
  - sharing domains
- each family has its own API
- all basic domains support:
  - lattice operations
  - widening (upper bound which guarantees termination)
- similar to a FlowSet in Soot but
  - immutable
  - type safe
  - no collection-style methods such as add, iterator, etc...
Basic domains

Basic domains describe general properties of program executions and are not tied to a specific target language.

- several families of basic domains
  - numerical domains
  - sharing domains

- each family has its own API

- all basic domains support:
  - lattice operations
  - widening (upper bound which guarantees termination)

- similar to a FlowSet in Soot but
  - immutable
  - type safe
  - no collection-style methods such as add, iterator, etc...
Numerical domains

Represent the values of numerical variables.

Example (Nested loop)

```plaintext
for (x = 0; x < 10; x++)
    for (y = x; y < 10; y++)
        // do something here
```

Example (Invariant inside the nested loop)

\[
\begin{align*}
0 \leq x & \leq 9 \\
y & \geq 9 \\
y - x & \leq 0
\end{align*}
\]
Numerical domains

Represent the values of numerical variables.

Example (Nested loop)

```c
for (x = 0; x < 10; x++)
    for (y = x; y < 10; y++)
        // do something here
```

Example (Invariant inside the nested loop)

\[
\begin{align*}
0 & \leq x \leq 9 \\
y & \geq 9 \\
y - x & \leq 0
\end{align*}
\]
Represent the values of numerical variables.

Example (Nested loop)

```java
for (x = 0; x < 10; x++)
    for (y = x; y < 10; y++)
        // do something here
```

Example (Invariant inside the nested loop)

\[
\begin{cases}
0 \leq x \leq 9 \\
y \geq 9 \\
y - x \leq 0
\end{cases}
\]
The API for numerical domains is well understood:

- linear assignment
  - \( x = 3x + 2y \)
- non-deterministic assignment
  - \( x = ? \)
- intersection with half-planes
  - if \( (x \leq y - z) \) then
- projection over a lower dimensional space
  - istore 3
- embedding onto a higher dimension space
  - iload 3
- and other...
Three different sources for numerical domains:

1. **Jandom native implementations**
   - interval and paralleloptope domains
   - JVM not well suited to the purpose, see *W. Kahan and Joseph D. Darcy*  
     *How Java’s Floating-Point Hurts Everyone Everywhere*

2. **Parma Polyehdra Library (PPL) based domains**
   - many domains: polyehdra, octagons, congruences, etc.
   - need wrappers to expose a common interface

3. **in the future...** add support for the APRON library
Implementations of numerical domains

Three different sources for numerical domains:

1. **Jandom** native implementations
   - interval and parallelootope domains
   - JVM not well suited to the purpose, see
     
     *W. Kahan and Joseph D. Darcy*
     
     *How Java’s Floating-Point Hurts Everyone Everywhere*

2. **Parma Polyhedra Library (PPL)** based domains
   - many domains: polyhedra, octagons, congruences, etc...
   - need wrappers to expose a common interface

3. in the future... add support for the APRON library
Three different sources for numerical domains:

1. **Jandom** native implementations
   - interval and parallelotope domains
   - JVM not well suited to the purpose, see
     *W. Kahan and Joseph D. Darcy*
     *How Java’s Floating-Point Hurts Everyone Everywhere*

2. **Parma Polyehdra Library (PPL)** based domains
   - many domains: polyehdra, octagons, congruences, etc.
   - need wrappers to expose a common interface

3. in the future... add support for the APRON library
Implementations of numerical domains

Three different sources for numerical domains:

1. **Jandom** native implementations
   - interval and parallelotope domains
   - JVM not well suited to the purpose, see
     
     *W. Kahan and Joseph D. Darcy*
     
     *How Java’s Floating-Point Hurts Everyone Everywhere*

2. **Parma Polyehdra Library (PPL)** based domains
   - many domains: polyehdra, octagons, congruences, etc.
   - need wrappers to expose a common interface

3. in the future... add support for the APRON library
Jandom architecture

Interprocedural analyzer

Flow graph analyzer

Basic block analyzer
- ASM
- Jimple
- Baf

Abstract environment

Basic domains
- native
- PPL
- APRON

Interpretation
Abstract environments

An abstract environment

- is the glue between the basic domains and the language we want to analyze
- maps operations in the language into operations on the domains

\[
\begin{align*}
  i_0 &= 10 \\
  s_1 &= 20 \\
  x_0 &\leq 100 \\
  x_3 + x_4 &\leq 1 \\
  x_4 &\leq 2
\end{align*}
\]
Abstract environments

Bytecode:

- iadd

Translation:

- $x_3 = x_3 + x_4$
- remove variable $x_4$

JVM numerical domain

This

$i0 = 10$

$s1 = 20$

Locals

Stack

0

1

Abstract environment

$x_0$

$x_1$

$x_2$

$x_3$

$x_4$

$x_1 \leq 100$

$x_3 + x_4 \leq 1$

$x_4 \leq 2$
Abstract environments

Bytecode:
- iadd

Translation:
- $x_3 = x_3 + x_4$
- remove variable $x_4$

JVM

Numerical domain

$1 \leq x_0$

$1 \leq x_1 

x_3 + x_4 \leq 1$

$x_4 \leq 2$
Abstract environments

Bytecode:
- iadd

Translation:
- $x_3 = x_3 + x_4$
- remove variable $x_4$

JVM numerical domain

**locals**
- this
  - $i0 = 10$
  - $s1 = 20$

**stack**
- 1

abstract environment

Numerical static analysis with Soot

Gianluca Amato (Chieti–Pescara)

SOAP 2013
Abstract environments

**Bytecode:**
- `iadd`

**Translation:**
- \( x_3 = x_3 + x_4 \)
- remove variable \( x_4 \)

```
JVM
locals
<table>
<thead>
<tr>
<th>this</th>
</tr>
</thead>
<tbody>
<tr>
<td>i0 = 10</td>
</tr>
<tr>
<td>s1 = 20</td>
</tr>
</tbody>
</table>

stack
| 1 |

abstract environment
- \( x_0 \)
- \( x_1 \)
- \( x_2 \)
- \( x_3 \)
- \( x_4 \)

numerical domain
- \( x_1 \leq 100 \)
- \( x_3 \leq 1 \)
- \( x_4 \leq 2 \)```
Abstract environments

Bytecode:
- iadd

Translation:
- $x_3 = x_3 + x_4$
- remove variable $x_4$

JVM numerical domain

<table>
<thead>
<tr>
<th>locals</th>
<th>abstract environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>this</td>
<td></td>
</tr>
<tr>
<td>$i0 = 10$</td>
<td></td>
</tr>
<tr>
<td>$s1 = 20$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stack</th>
<th>abstract environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Numerical static analysis with Soot
Jandom architecture

Interprocedural analyzer

Flow graph analyzer

Basic block analyzer

Abstract environment

Basic domains

Interpretation

ASM

Jimple

Baf

native

PPL

APRON

Numerical static analysis with Soot
A somewhat different definition of basic block:

**Definition (Basic block)**

A basic block is a sequence of instructions such that only the first one may be the target of a jump.

Consequences:
- encompass the standard definition of basic block
- fewer basic blocks are needed
- basic block may have many outgoing edges

Moreover
- we want a return statement to begin a basic block
Basic blocks

A somewhat different definition of basic block:

**Definition (Basic block)**

A basic block is a sequence of instructions such that only the first one may be the target of a jump.

Consequences:

- encompass the standard definition of basic block
- fewer basic blocks are needed
- basic block may have many outgoing edges

Moreover

- we want a return statement to begin a basic block
A somewhat different definition of basic block:

**Definition (Basic block)**

A basic block is a sequence of instructions such that only the first one may be the target of a jump.

Consequences:

- encompass the standard definition of basic block
- fewer basic blocks are needed
- basic block may have many outgoing edges

Moreover

- we want a return statement to begin a basic block
Basic blocks and *Soot*

We use the standard Block class and two new BlockGraph classes:

**BigBlockGraph** builds a BlockGraph according to our requirements

**UnitBlockGraph** build a BlockGraph where each block is composed of a single unit (useful for debugging).

These are written in Java and could be integrated into *Soot*.

**Implementation notes**

In the case of BigBlockGraph, overriding computeLeader was not enough, since buildBlocks method assumes that every jump instruction is the tail of a block.

This could be changed in *Soot* itself.
Basic blocks and Soot

We use the standard Block class and two new BlockGraph classes:

**BigBlockGraph** builds a BlockGraph according to our requirements

**UnitBlockGraph** build a BlockGraph where each block is composed of a single unit (useful for debugging).

These are written in Java and could be integrated into *Soot*.

**Implementation notes**

In the case of BigBlockGraph, overriding computeLeader was not enough, since buildBlocks method assumes that every jump instruction is the tail of a block.

This could be changed in *Soot* itself.
Basic block analyzer

A basic block analyzer:
- takes a block
- takes an input property
- returns a set of target blocks and the corresponding property

Example (Analysis of a basic block)

Input:
\[
0 \leq x_0 \\
x_0 + x_1 = 0
\]

Block:

- if \(x_0 < 10\) goto \(l_a\)
- if \(x_0 < 0\) goto \(l_b\)
- \(x_0 = x_0 + 1\)

Output:

- \(\langle l_a, 0 \leq x_0 \leq 9 \\
x_0 + x_1 = 0 \rangle\)
- \(\langle l_b, \text{false} \rangle\)
- \(\langle \text{fallthrough, } 1 \leq x_0 \\
x_0 + x_1 = 1 \rangle\)
Generally *Jimple* is considered simpler to analyze than *Baf*. I am not entirely sure this holds in our case:

- *Jimple* has less instructions, but we need to interpret expressions
- *Jimple* has no stack, but the easiest way to analyze expressions is to evaluate them recursively, hence re-introducing a stack
- if analyzing an entire assignment in one step, analysis may be faster and more precise

Then, why do not move to *Grimple*?

- numerical domains have API to analyze a result of complex linear assignments and comparison
- in *Grimple* these expression are almost ready to be fed to the abstract domain
Generally, *Jimple* is considered simpler to analyze than *Baf*. I am not entirely sure this holds in our case:

- *Jimple* has less instructions, but we need to interpret expressions.
- *Jimple* has no stack, but the easiest way to analyze expressions is to evaluate them recursively, hence re-introducing a stack.
- If analyzing an entire assignment in one step, analysis may be faster and more precise.

Then, why do not move to *Grimple*?

- Numerical domains have API to analyze a result of complex linear assignments and comparison.
- In *Grimple* these expressions are almost ready to be fed to the abstract domain.
Example of possible benefits of *Grimple*

Consider

- the **octagon domain**, which represents all conditions of the kind 
  \[ \pm x_1 \pm x_2 \leq c \]
- the assignment \( z = z + x + y \)
- the precondition \( z = w \land x + y = 0 \)
- after the assignment, \( z = w \land x + y = 0 \) still holds
- if we break the assignment in \( z = z + x \) and \( z = z + y \) we loose the property after the first assignment.
Consider

- the octagon domain, which represents all conditions of the kind $\pm x_1 \pm x_2 \leq c$
- the assignment $z = z + x + y$
- the precondition $z = w \land x + y = 0$
- after the assignment, $z = w \land x + y = 0$ still holds
- if we break the assignment in $z = z + x$ and $z = z + y$ we loose the property after the first assignment.
Example of possible benefits of *Grimple*

Consider

- the **octagon domain**, which represents all conditions of the kind
  \[ \pm x_1 \pm x_2 \leq c \]
- the assignment \( z = z + x + y \)
  
  - the precondition \( z = w \land x + y = 0 \)
  
  - after the assignment, \( z = w \land x + y = 0 \) still holds
  
  - if we break the assignment in \( z = z + x \) and \( z = z + y \) we loose the property after the first assignment.
Example of possible benefits of \textit{Grimple}

Consider

- the \textbf{octagon domain}, which represents all conditions of the kind \( \pm x_1 \pm x_2 \leq c \)
- the assignment \( z = z + x + y \)
- the precondition \( z = w \land x + y = 0 \)
- after the assignment, \( z = w \land x + y = 0 \) still holds
- if we break the assignment in \( z = z + x \) and \( z = z + y \) we lose the property after the first assignment.
Example of possible benefits of *Grimple*

Consider

- the **octagon domain**, which represents all conditions of the kind
  \[ \pm x_1 \pm x_2 \leq c \]
- the assignment \( z = z + x + y \)
- the precondition \( z = w \land x + y = 0 \)
- after the assignment, \( z = w \land x + y = 0 \) still holds
- if we break the assignment in \( z = z + x \) and \( z = z + y \) we loose the property after the first assignment.
Example of possible benefits of *Grimple*

Consider

- the **octagon domain**, which represents all conditions of the kind
  \[ \pm x_1 \pm x_2 \leq c \]
- the assignment \( z = z + x + y \)
- the precondition \( z = w \land x + y = 0 \)
- after the assignment, \( z = w \land x + y = 0 \) still holds
- if we break the assignment in \( z = z + x \) and \( z = z + y \) we loose the property after the first assignment.
Jandom architecture

Interprocedural analyzer

Flow graph analyzer

Basic block analyzer
- ASM
- Jimple
- Baf

Abstract environment

Basic domains
- native
- PPL
- APRON

Interpretation
Flow graph analyzer

Similar to BranchedFlowAnalysis but

- over Blocks instead of Units;
- directly supports use of widening to ensure convergence of analysis;
- directly support ascending and descending phases;
- it will support many iterations strategies.
Flow graph analyzer

Similar to BranchedFlowAnalysis but:

- over Blocks instead of Units;
- directly supports use of widening to ensure convergence of analysis;
- directly support ascending and descending phases;
- it will support many iterations strategies.
Flow graph analyzer

Similar to BranchedFlowAnalysis but

- over Blocks instead of Units;
- directly supports use of widening to ensure convergence of analysis;
- directly support ascending and descending phases;
- it will support many iterations strategies.

Widening

- Widening should replace union on loops for domains with infinite ascending chains.
- Possible in BranchedFlowAnalysis but not as much as flexible.
Flow graph analyzer

Similar to BranchedFlowAnalysis but

- over Blocks instead of Units;
- directly supports use of widening to ensure convergence of analysis;
- directly support ascending and descending phases;
- it will support many iterations strategies.

Ascending and descending phases

- Widening causes loss of precision. It is possible to partially recover precision with descending chains.
- Again, something is possible with BranchedFlowAnalysis but not as much as flexible
Flow graph analyzer

Similar to BranchedFlowAnalysis but

- over Blocks instead of Units;
- directly supports use of widening to ensure convergence of analysis;
- directly support ascending and descending phases;
- it will support many iterations strategies.

Iteration strategies

Worklist algorithms are not always the best choice:

- recursive vs iterative strategies;
- guided abstract interpretation.
An example of intraprocedural analysis

```java
static void nested() {
    int z = 0;
    // z=0
    for (int i = 0; i < 10; i++)
        // 0 ≤ z ∧ 0 ≤ i ≤ 10 ∧ i ≤ z + 1
        for (int j = 0; j < i; j++)
            // i ≤ 10 ∧ 0 ≤ j ∧ j ≤ i ∧ j ≤ z ∧ i − z − 2j − 1 ≤ 0
            z = z + 1;
}
```

Actually, the result of the analyzer is much less nicer, since properties are reported on the intermediate representation, not the Java code.
Using *Soot* vs extending *Soot*

At the moment, Jandom uses *Soot* to implement a completely different framework.

Another choice would be to extend *Soot* to support the kind of analysis we are interested in.

Integration would obviously be beneficial, but there are some stopovers:
- implementation language: Scala vs Java
- Jandom supports different target languages

Thinking about this...
What we like in *Soot*

Multiple intermediate representations

Facilities for intra-procedural analyses such as
- automatic generation of control flow graphs

Facilities for inter-procedural analyses such as:
- ability to browse the classes and methods in a Scene
- automatic computation of call-graphs
What we do not like in *Soot*

Documentation
- not well organized
- not always complete

Not enough type safety at the IR level
- for example, an *AndExpr* may have numeric operands
- makes it difficult to check whether I have considered all possible cases when analyzing instructions
- but I am biased... I use Scala, after all.

Some annoying missing minor functionalities
- how do I get the maximum stack size in a *Baf* body?
Future work

- Make Jandom **definitely** work instead of barely work
- Polishing interfaces
- Polishing user interface
- Speed optimization
  - Evaluate trade-off between mutable and immutable domains
  - Evaluate trade-off between functional and imperative style
- Using *Dava* to analyze directly over the AST?
Sharing analysis

Two variables $x$ and $y$ share if it is possible to reach from them a common object.

**Example (Variables $x$ and $y$ share)**

Jandom implements an inter-procedural analysis for possible pair sharing, as defined by *Spoto and Secci* (SAS ’05).
Possible pair sharing analysis

Given variables $x$, $y$ and $z$ the set \{(x, x), (y, y), (x, y)\} means:
- $x$ may share with itself (i.e., it is possibly not null);
- $y$ may share with itself (i.e., it is possibly not null);
- $x$ and $y$ may share;
- $z$ is definitively null.

Other variants:
- set sharing + class analysis
  M. Mendéz-Lojo and M. Hermenegildo (VMCAI ’08)
- pair sharing + linearity + aliasing
  a future work of ours
Given variables \( x, y \) and \( z \) the set \{ \( (x, x), (y, y), (x, y) \) \} means:

- \( x \) may share with itself (i.e., it is possibly not null);

- \( y \) may share with itself (i.e., it is possibly not null);

- \( x \) and \( y \) may share;

- \( z \) is definitively null.

Other variants:

- set sharing + class analysis
  M. Mendéz-Lojo and M. Hermenegildo (VMCAI ’08)

- pair sharing + linearity + aliasing
  a future work of ours
Possible pair sharing analysis

Given variables $x$, $y$ and $z$ the set \{$(x, x), (y, y), (x, y)$\} means:

- $x$ may share with itself (i.e., it is possibly not null);
- $y$ may share with itself (i.e., it is possibly not null);
- $x$ and $y$ may share;
- $z$ is definitively null.

Other variants:

- set sharing + class analysis
  - M. Mendéz-Lojo and M. Hermenegildo (VMCAI ’08)
- pair sharing + linearity + aliasing
  - a future work of ours
Possible pair sharing analysis

Given variables \( x, y \) and \( z \) the set \( \{(x, x), (y, y), (x, y)\} \) means:

- \( x \) may share with itself (i.e., it is possibly not null);
- \( y \) may share with itself (i.e., it is possibly not null);
- \( x \) and \( y \) may share;
- \( z \) is definitively null.

Other variants:

- set sharing + class analysis
  M. Mendez-Lojo and M. Hermenegildo (VMCAI ’08)
- pair sharing + linearity + aliasing
  a future work of ours
Possible pair sharing analysis

Given variables $x$, $y$ and $z$ the set $\{(x, x), (y, y), (x, y)\}$ means:

- $x$ may share with itself (i.e., it is possibly not null);
- $y$ may share with itself (i.e., it is possibly not null);
- $x$ and $y$ may share;
- $z$ is definitively null.

Other variants:

- set sharing + class analysis
  
  M. Mendez-Lojo and M. Hermenegildo (VMCAI ’08)

- pair sharing + linearity + aliasing
  
  a future work of ours
Possible pair sharing analysis

Given variables $x$, $y$ and $z$ the set $\{(x, x), (y, y), (x, y)\}$ means:
- $x$ may share with itself (i.e., it is possibly not null);
- $y$ may share with itself (i.e., it is possibly not null);
- $x$ and $y$ may share;
- $z$ is definitively null.

Other variants:
- *set sharing + class analysis*
  
  M. Mendéz-Lojo and M. Hermenegildo (VMCAI ’08)

- *pair sharing + linearity + aliasing*
  
  a future work of ours
An API for sharing analysis?

Looking for a standard API for sharing analysis:

- language independent;
- suitable for other memory based analysis such as class or aliasing analysis.

At the moment, modeled over standard *Baf*/*Jimple* operations:

- assignment of variables/fields to variables/fields
- test for nullness
- test for runtime class
Looking for a standard API for sharing analysis:

- language independent;
- suitable for other memory based analysis such as class or aliasing analysis.

At the moment, modeled over standard *Baf*/*Jimple* operations:

- assignment of variables/fields to variables/fields
- test for nullness
- test for runtime class
An example of interprocedural analysis

```java
static int recursb(int x) {
    return recursa(x + 1);
}

static int recursa(int x) {
    if (x < 0)
        return recursb(x);
    else
        return x;
}
```

Inteprocedural analysis proves on call to recursa:

\[
ret \geq x \land ret \geq 0
\]

on call to recursb:

\[
ret \geq x + 1 \land ret \geq 0
\]