Reinhard Wilhelm

*Universität des Saarlandes*

**Development of Safety-Critical Embedded Systems**

Static Program Analysis

**Winter Semester 2012/2013**

Slides based on:


- R. Wilhelm, B. Wachter: Abstract Interpretation with Applications to Timing Validation. CAV 2008: 22-36

- Helmut Seidl’s slides
A Short History of Static Program Analysis

• Early high-level programming languages were implemented on very small and very slow machines.

• Compilers needed to generate executables that were extremely efficient in space and time.

• Compiler writers invented efficiency-increasing program transformations, wrongly called optimizing transformations.

• Transformations must not change the semantics of programs.

• Enabling conditions guaranteed semantics preservation.

• Enabling conditions were checked by static analysis of programs.
Theoretical Foundations of Static Program Analysis

- Theoretical foundations for the solution of **recursive equations**: Kleene (30s), Tarski (1955)

- Gary Kildall (1972) clarified the lattice-theoretic foundation of **data-flow analysis**.

- Patrick Cousot (1974) established the relation to the programming-language semantics.
Static Program Analysis as a Verification Method

- Automatic method to derive invariants about program behavior, answers questions about program behavior:
  - will index always be within bounds at program point $p$?
  - will memory access at $p$ always hit the cache?
- answers of sound static analysis are correct, but approximate: don’t know is a valid answer!
- analyses proved correct wrt. language semantics,
Proposed Lectures Content:

1. Introductory example: rules-of-sign analysis
2. theoretical foundations: lattices
3. an operational semantics of the language
4. another example: constant propagation
5. relating the semantics to the analysis—correctness proofs
6. Further static analyses in compilers: Elimination of superfluous computations
   $\rightarrow$ available expressions
   $\rightarrow$ live variables
   $\rightarrow$ array-bounds checks
7. timing (WCET) analysis
8. analysis for runtime errors
1 Introduction

... in this course and in the Seidl/Wilhelm/Hack book:

a simple imperative programming language with:

- variables // registers
- $R = e;$ // assignments
- $R = M[e];$ // loads
- $M[e_1] = e_2;$ // stores
- if $(e)$ $s_1$ else $s_2$ // conditional branching
- goto $L;$ // no loops

An intermediate language into which (almost) everything can be translated.
In particular, no procedures. So, only intra-procedural analyses!
2 Example — Rules-of-Sign Analysis

Problem: Determine at each program point the sign of the values of all variables of numeric type.

Example program:

1: x = 0;
2: y = 1;
3: while (y > 0) do
   4: y = y + x;
   5: x = x + (-1);
Program representation as *control-flow graphs*

1. $x = 0$
2. $y = 1$
3. $x = x + (-1)$
4. true($y > 0$)
5. false($y > 0$)
6. $y = y + x$
7. $x = x + (-1)$
What are the ingredients that we need?
More ingredients?
All the ingredients:

- a set of information elements, each a set of possible signs,
- a partial order, “⊆”, on these elements, specifying the "relative strength" of two information elements,
- these together form the abstract domain, a lattice,
- functions describing how signs of variables change by the execution of a statement, abstract edge effects,
- these need an abstract arithmetic, an arithmetic on signs.
We construct the abstract domain for single variables starting with the lattice \( Signs = 2\{-,0,+\} \) with the relation \( \sqsubseteq \) =“\( \subseteq \).
The analysis should ”bind” program variables to elements in \textit{Signs}.

So, the abstract domain is $\mathbb{D} = (\textit{Vars} \rightarrow \textit{Signs}) \bot$, a \textit{Sign-environment}. $\bot \in \mathbb{D}$ is the function mapping all arguments to \{\}. The partial order on $\mathbb{D}$ is $D_1 \sqsubseteq D_2$ iff

$D_1 = \bot$ \quad or

$D_1 x \subseteq D_2 x \quad (x \in \textit{Vars})$

Intuition?
The analysis should ”bind” program variables to elements in $Signs$.

So, the abstract domain is $\mathbb{D} = (\text{Vars} \rightarrow Signs)_{\bot}$. a $\text{Sign-environment}$.

$\bot \in \mathbb{D}$ is the function mapping all arguments to $\{\}$. 

The partial order on $\mathbb{D}$ is $D_1 \sqsubseteq D_2$ iff 

$D_1 = \bot$ or 

$D_1 x \subseteq D_2 x \quad (x \in \text{Vars})$

Intuition?

$D_1$ is at least as precise as $D_2$ since $D_2$ admits at least as many signs as $D_1$
How did we analyze the program?

In particular, how did we walk the lattice for $y$ at program point 5?
How is a solution found?
Iterating until a fixed-point is reached

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>x</td>
<td>y</td>
<td>x</td>
<td>y</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Idea:

- We want to determine the sign of the values of expressions.
Idea:

- We want to determine the sign of the values of expressions.
- For some sub-expressions, the analysis may yield \{+, −, 0\}, which means, it couldn’t find out.
Idea:

- We want to determine the signs of the values of expressions.
- For some sub-expressions, the analysis may yield \{+,-,0\}, which means, it couldn’t find out.
- We replace the concrete operators $\square$ working on values by abstract operators $\square\#$ working on signs:
Idea:

- We want to determine the signs of the values of expressions.
- For some sub-expressions, the analysis may yield \{+,-,0\}, which means, it couldn’t find out.
- We replace the concrete operators $\Box$ working on values by abstract operators $\Box^\#$ working on signs:
- The abstract operators allow to define an abstract evaluation of expressions:

\[ [e]^\# : (Vars \rightarrow \text{Signs}) \rightarrow \text{Signs} \]
Determining the sign of expressions in a Sign-environment works as follows:

\[
[c] \# D = \begin{cases} 
{+} & \text{if } c > 0 \\
{-} & \text{if } c < 0 \\
{0} & \text{if } c = 0
\end{cases}
\]

\[
[v] \# = D(v)
\]

\[
[e_1 \square e_2] \# D = [e_1] \# D \square [e_2] \# D
\]

\[
[\square e] \# D = \square[e] \# D
\]
Abstract operators working on signs (Addition)

<table>
<thead>
<tr>
<th>+#</th>
<th>{0}</th>
<th>{+}</th>
<th>{-}</th>
<th>{-, 0}</th>
<th>{-, +}</th>
<th>{0, +}</th>
<th>{-, 0, +}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0}</td>
<td>{0}</td>
<td>{+}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{+}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{-}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{-, 0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{-, +}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{0, +}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{-, 0, +}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Abstract operators working on signs (Multiplication)

<table>
<thead>
<tr>
<th>×#</th>
<th>{0}</th>
<th>{+}</th>
<th>{-}</th>
<th>{−, 0}</th>
<th>{−, +}</th>
<th>{0, +}</th>
<th>{−, 0, +}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0}</td>
<td>{0}</td>
<td>{0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{+}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{-}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{−, 0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{−, +}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{0, +}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{−, 0, +}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Abstract operators working on signs (unary minus)

<table>
<thead>
<tr>
<th>−#</th>
<th>{0}</th>
<th>{+}</th>
<th>{-}</th>
<th>{−, 0}</th>
<th>{−, +}</th>
<th>{0, +}</th>
<th>{−, 0, +}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0}</td>
<td>{0}</td>
<td>{-}</td>
<td>{+}</td>
<td>{+, 0}</td>
<td>{−, +}</td>
<td>{0, -}</td>
<td>{−, 0, +}</td>
</tr>
<tr>
<td>{-}</td>
<td></td>
<td>{+}</td>
<td>{−, 0}</td>
<td>{−, +}</td>
<td>{0, -}</td>
<td>{−, 0, +}</td>
<td></td>
</tr>
</tbody>
</table>
Working an example:

\[ D = \{x \mapsto \{+\}, \, y \mapsto \{+\}\} \]

\[
\begin{align*}
& = \{+\} +^{\#} \{+\} \\
& = \{+\} \\

[x + (-y)]^{\#} D & = \{+\} +^{\#} (-[y]^{\#} D) \\
& = \{+\} +^{\#} (-\{+\}) \\
& = \{+\} +^{\#} \{-\} \\
& = \{+, -, 0\}
\end{align*}
\]
Thus, we obtain the following effects of edges $[lab]^\#$:

\[
\begin{array}{ll}
[;]^\# D &= D \\
[\text{true} \ (e)]^\# D &= D \\
[\text{false} \ (e)]^\# D &= D \\
[x = e;]^\# D &= D \oplus \{x \mapsto [e]^\# D\} \\
[x = M[e];]^\# D &= D \oplus \{x \mapsto \{+,-,0\}\} \\
[M[e_1] = e_2;]^\# D &= D \\
\end{array}
\]

\ldots \text{whenever } D \neq \bot