Improving Static Analyses of C Programs with Conditional Predicates

Sandrine Blazy David Bühler Boris Yakobowski

IRISA - University of Rennes

CEA, LIST, Software Safety Lab

October 20, 2014

Context of our Work

- Static analysis of C programs to prove their safety.
- Flow-sensitive analysis.
- Well-known loss of precision when two control-flow paths meet.
- But path-sensitive analyses are often too costly.

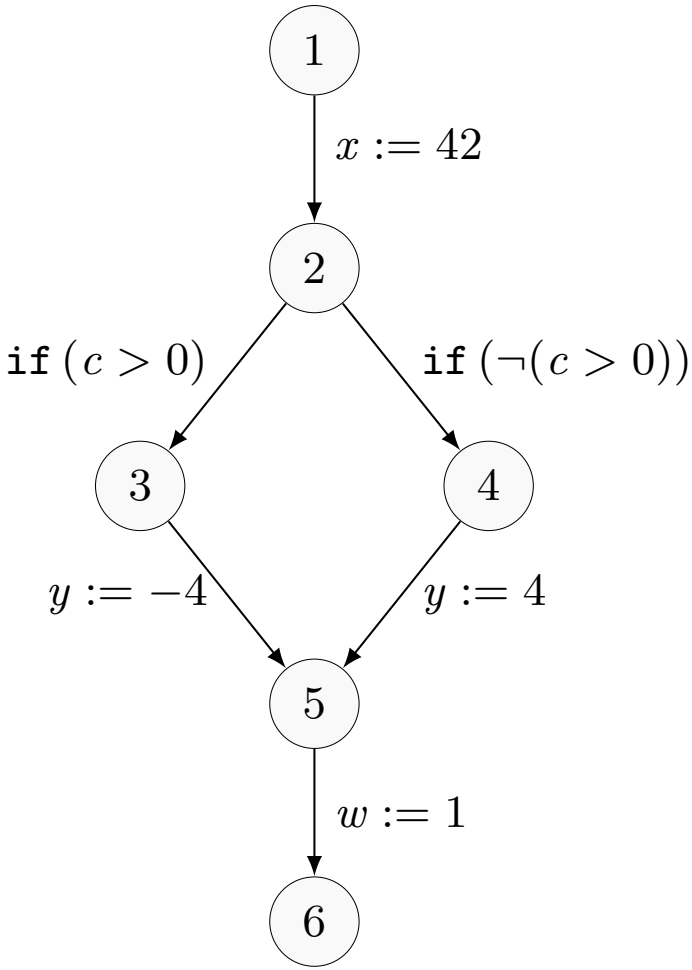
Abstract Interpretation

- Approximates a concrete not computable semantics through abstract domains.
- Abstract domains usually represent sets of concrete states.
- Sound analyses: those abstractions must capture all possible behaviors of the program.
- Continuing trade-off between precision and efficiency:
  - abstract domains must be sufficiently precise to exclude error cases and simple enough to be scalable.

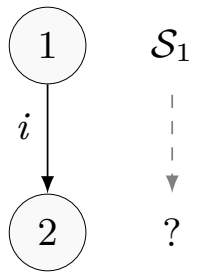
Simple Idealized Language

- Programs are represented as a Control Flow Graph.
- Edges are labelled by a statement:
  - assignment $v := e$
  - assume guards `if` (e)

```
1   x = 42;
2   if (c > 0)
3       y = -4;
4   else
5       y = 4;
6   w = 1;
```

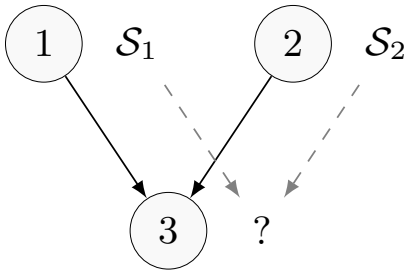


Dataflow Analysis



- For each statement, an abstract transfer function over-approximates concrete semantics on abstract domain.

$$\mathcal{S}_2 = [\![i]\!]^\sharp (\mathcal{S}_1)$$



- When two control-flow paths meet, a join operation $\sqcup$ over-approximates the union of the concrete states on each incoming edges.

$$\mathcal{S}_3 = \mathcal{S}_1 \sqcup \mathcal{S}_2$$

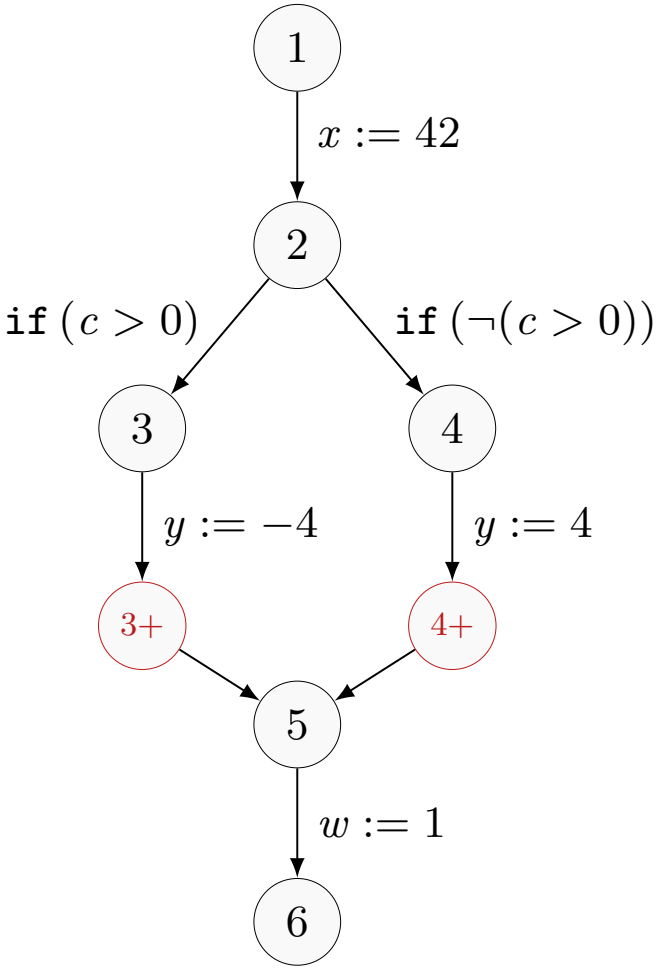
- Computation of a fixpoint from an initial state.

Interval Domain

- At each program point, the possible value of each variable is represented by an interval.
- The analysis starts at the entry point with the special value $\top$ (the values of all variables are unknown).
- Abstract transfer functions follow interval arithmetics.
- Join operator:

$$\begin{aligned} \mathcal{S} &= \lambda x.\, [x_1, x_2] \\ \mathcal{S}' &= \lambda x.\, [x_1', x_2'] \\ \mathcal{S} \sqcup \mathcal{S}' &= \lambda x.\, [\min(x_1, x_1'), \max(x_2, x_2')] \end{aligned}$$

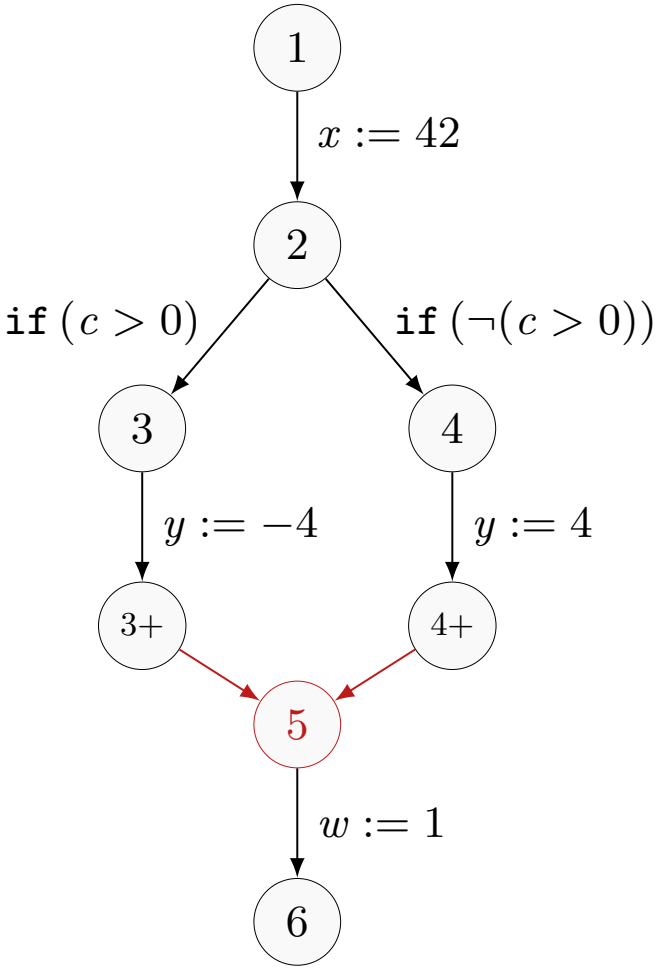
Disjunction



- The join operation often leads to loss of precision.

3+	$x \in [42]$	$c \in [1; +\infty]$	$y \in [-4]$
4+	$x \in [42]$	$c \in [-\infty; 0]$	$y \in [4]$

Disjunction



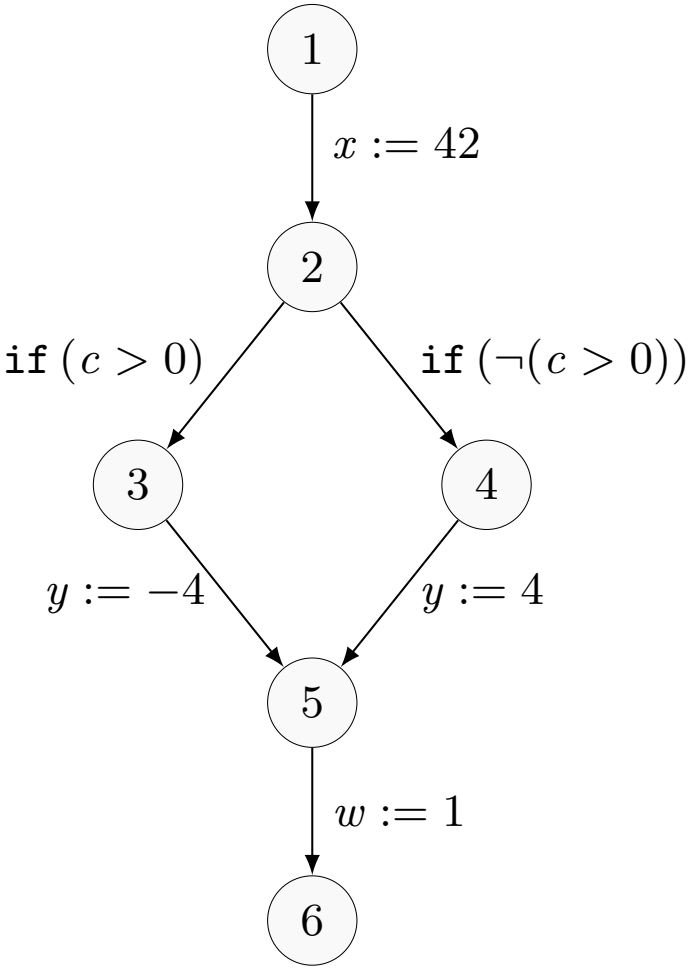
- The join operation often leads to loss of precision.

3+	$x \in [42]$	$c \in [1; +\infty]$	$y \in [-4]$
4+	$x \in [42]$	$c \in [-\infty; 0]$	$y \in [4]$
5	$x \in [42]$	$c \in \top$	$y \in [-4; 4]$

- Here, y cannot be equal to 0.

How to minimize the loss of precision at join points ?

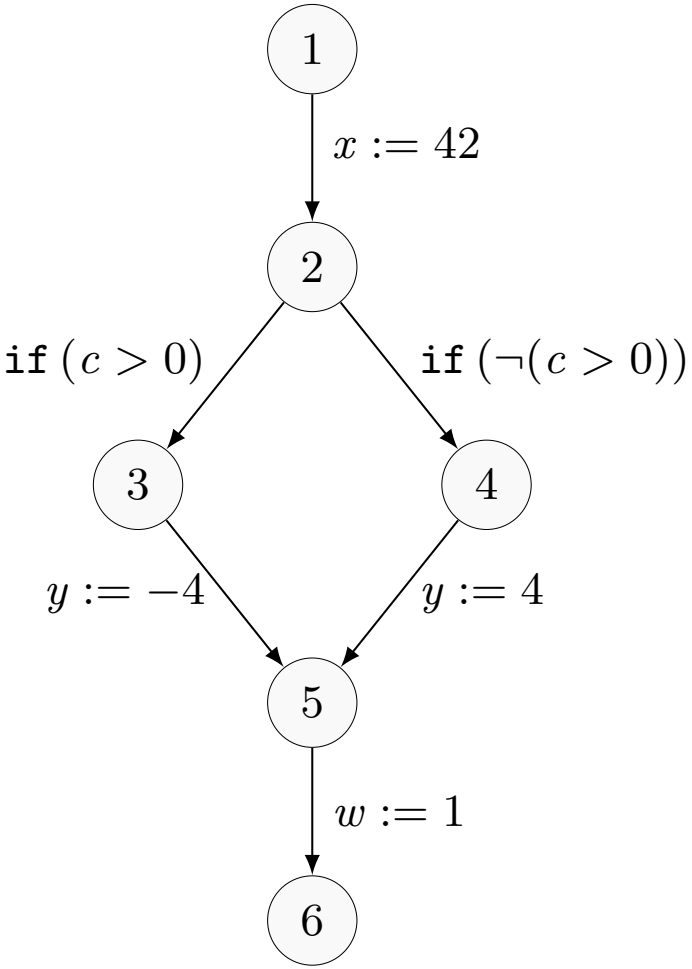
Trace Partitioning



- A well-known technic: trace partitioning.
- Main idea: keep separate abstract states for different paths in the CFG.

5	$\texttt{if}_\texttt{T}$:	$x \in [42]$	$c \in [1; +\infty]$	$y \in [-4]$
	$\texttt{if}_\texttt{F}$:	$x \in [42]$	$c \in [-\infty; 0]$	$y \in [4]$

Trace Partitioning



- Here, trace partitioning allows to exclude the potential error case where $y = 0$.
- Drawback: the analysis continues with multiple states in parallel: more costly.

6	$\texttt{if}_\texttt{T}: \quad x \in [42] \quad c \in [1; +\infty] \quad y \in [-4] \quad w \in [1]$
	$\texttt{if}_\texttt{F}: \quad x \in [42] \quad c \in [-\infty; 0] \quad y \in [4] \quad w \in [1]$

Predicated Analysis

- Our proposal: one abstract state with further information under predicates.

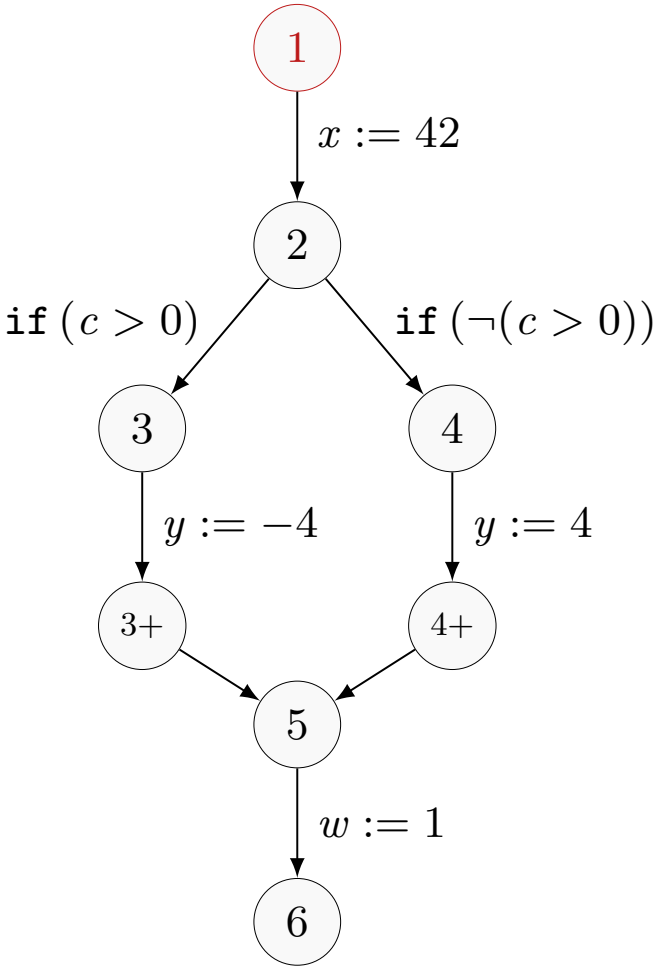
5	$\texttt{true}$	$\mapsto$	$\begin{cases} x \in [42] \\ y \in [-4; 4] \end{cases}$
	$c > 0$	$\mapsto$	$y \in [-4]$
	$\neg\,(c > 0)$	$\mapsto$	$y \in [4]$

Predicated Domain

The predicated domain is two-fold:

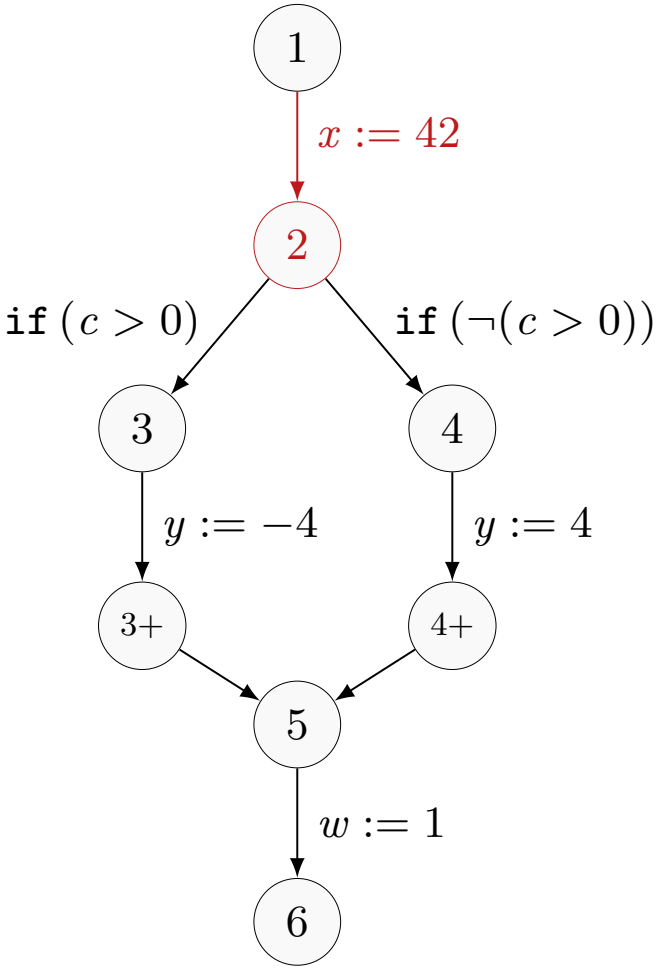
- a set of **implications** from predicates (deriving from conditionals) to values of the interval domain.
  - The value under the predicate `true` is always the broadest one.
  - The values under non-`true` guards bring extra-information coming from merged branches.
- a **context**, namely a boolean predicate that holds at the considered point, used to create new implications at join points.

Example



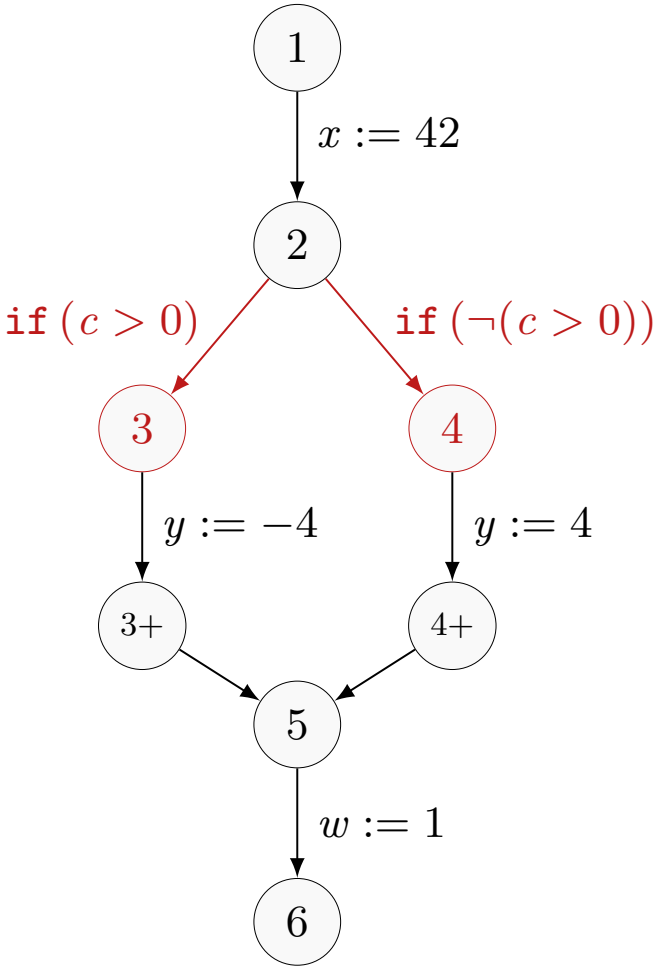
	context	*implications*
1	`true`	$\texttt{true} \mapsto \top$

Example



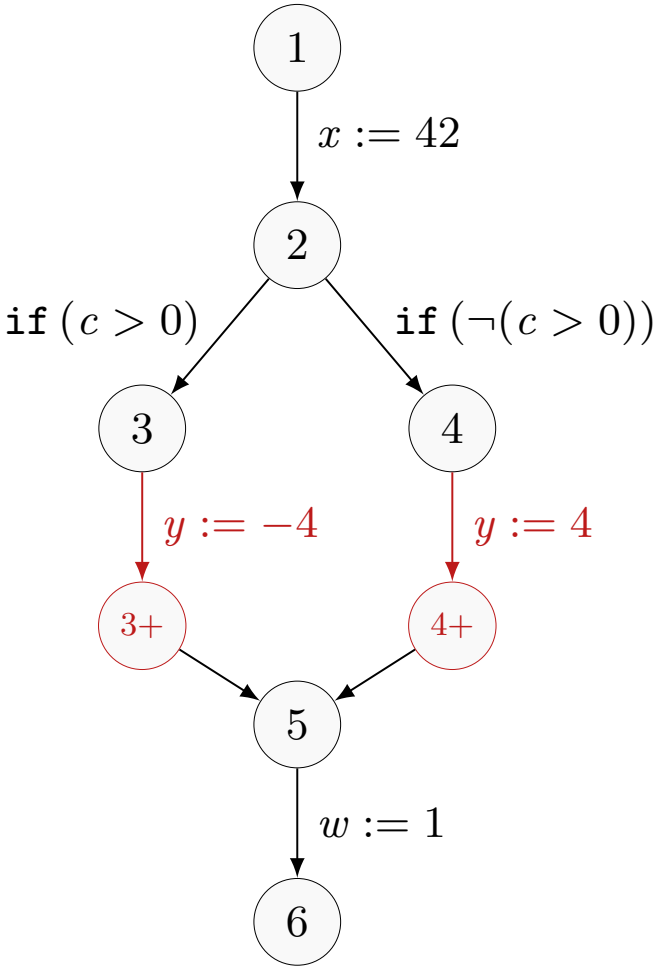
	context	*implications*
1	`true`	`true` $\mapsto \top$
2	`true`	`true` $\mapsto x \in [42]$

Example



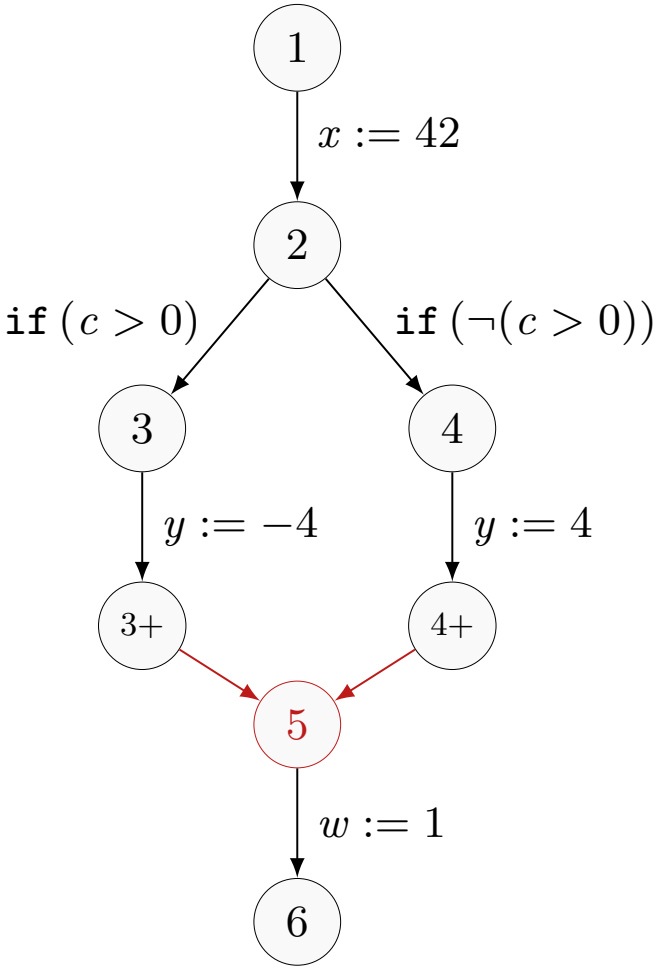
	context	*implications*
1	`true`	`true` $\mapsto \top$
2	`true`	`true` $\mapsto x \in [42]$
3	$c > 0$	`true` $\mapsto x \in [42]$
4	$\neg\,(c > 0)$	`true` $\mapsto x \in [42]$

Example



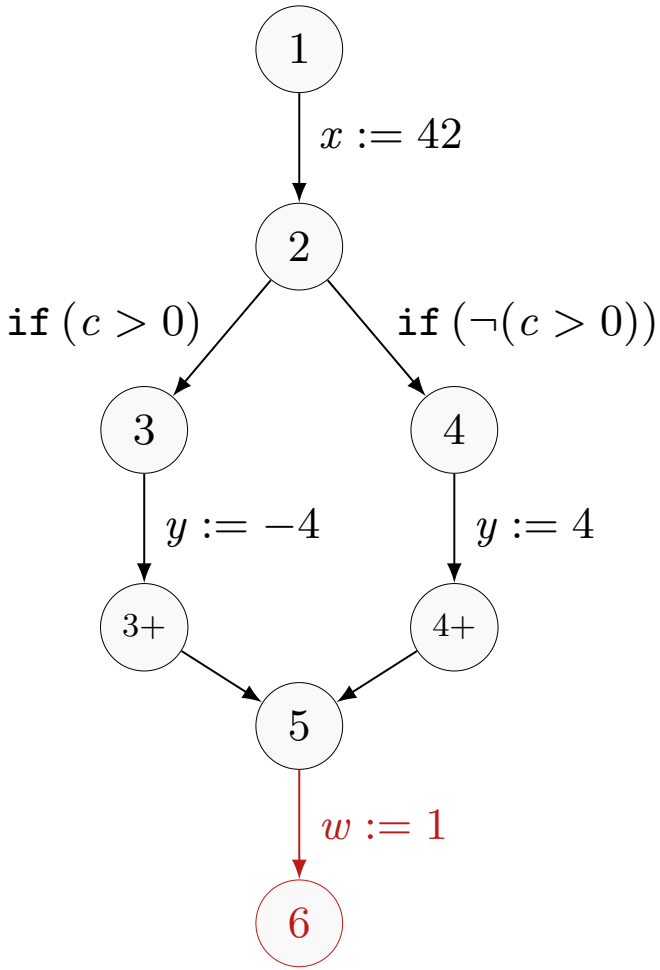
	context	*implications*
1	`true`	`true` $\mapsto \top$
2	`true`	`true` $\mapsto x \in [42]$
3	$c > 0$	`true` $\mapsto x \in [42]$
3+	$c > 0$	`true` $\mapsto x \in [42] \quad y \in [-4]$
4	$\neg(c > 0)$	`true` $\mapsto x \in [42]$
4+	$\neg(c > 0)$	`true` $\mapsto x \in [42] \quad y \in [4]$

Example



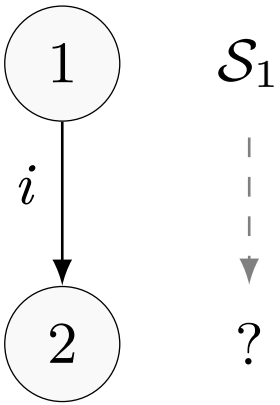
	context	*implications*
3+	$c > 0$	$\texttt{true} \mapsto x \in [42] \quad y \in [-4]$
4+	$\neg\,(c > 0)$	$\texttt{true} \mapsto x \in [42] \quad y \in [4]$
5	`true`	$\texttt{true} \mapsto \begin{cases} x \in [42] \\ y \in [-4; 4] \end{cases}$ $c > 0 \mapsto y \in [-4]$ $\neg\,(c > 0) \mapsto y \in [4]$

Example



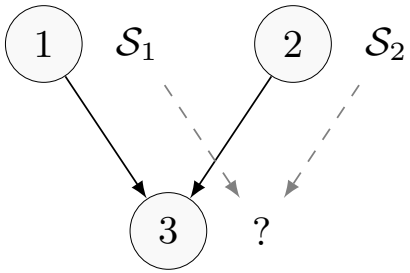
	context	*implications*
5	`true`	$\texttt{true} \mapsto \begin{cases} x \in [42] \\ y \in [-4; 4] \end{cases}$
		$c > 0 \mapsto y \in [-4]$
		$\neg(c > 0) \mapsto y \in [4]$
6	`true`	$\texttt{true} \mapsto \begin{cases} x \in [42] \\ y \in [-4; 4] \\ w \in [1] \end{cases}$
		$c > 0 \mapsto y \in [-4]$
		$\neg(c > 0) \mapsto y \in [4]$

Transfer Functions



- apply the transfer function on intervals to each value on the right side of the implications;
- on an assignment $x := e$, remove any predicate whose truth value depends on x;
- on an assume guard `if` (e), add e to the current context.

Join



$$\mathcal{S}_1 : \begin{array}{c} C_1 \\ \{p_i \to v_i\} \end{array} \qquad \mathcal{S}_2 : \begin{array}{c} C_2 \\ \{q_j \to w_j\} \end{array} \qquad \begin{array}{l} : \textit{context} \\ : \textit{implications} \end{array}$$

- Disjunction of contexts: $C_1 \vee C_2$
- Implications are:
  - the implications valid in both previous states

$$p_i \wedge q_j \to v_i \sqcup w_j$$

  - the implications valid in one state + negation of other context

$$\begin{array}{rcl} \neg C_2 \wedge p_i & \to & v_i \\ \neg C_1 \wedge q_j & \to & w_j \end{array}$$

Avoiding redundancy

- Redundant values in implications impair the performance of the analysis.
- To avoid redundancy, the underlying abstract domain must provide:
  - a lighter transfer function for values under non-`true` guards, that avoids relearning information already modeled by the value under `true`;
  - a difference operation able to extract the specific information of each of two abstract values, not contained in their join.

Genericity

- Dealing with predicates is challenging...
- ... but at a join point, implications can precisely model the specific information of each branch.

- Predicated analysis is a **generic** framework, independent of the underlying domain.
- In particular, we instanciated such analyses on simple domains in the *Frama-C* platform.

Frama-C

http://frama-c.com

- A modular platform dedicated to the analysis of C code through several plugins.
- Among them, the *Value Analysis*:
  - Abstract interpretation based;
  - Emits alarms at potentially unsafe program points;
  - Domain: small sets of discrete values or intervals with congruences + alias analysis for pointers;
  - Trace partitioning: propagates separately multiple abstract states, whose number is limited by a parameter called *slevel*.

SCADE code

Predicated analyses are efficient to resolve usual pattern codes in SCADE-generated programs.

```
1    if (c) {
2        ...
3        v = expr;
4    }
5    ...
6    if (c) {
7        /* assert Value: initialisation: \initialized(v); */
8        x = v;
9    }
```

The trace partitioning of the *Value Analysis* handles such patterns, but is too costly on huge nested conditionals.

More SCADE code

```
  if (OUTbSame_2_1_1_2_1) v_15076_1_2_1 = (unsigned char)1;
  else {
    if (clk_14887_2_1_1_2_1) clk_14971_1_1_2_1 = _L24_1_2_1_1_2_1;
    else
      if (clk_14889_2_1_1_2_1) clk_14971_1_1_2_1 = (unsigned char)1;
      else clk_14971_1_1_2_1 = v_14912_2_1_1_2_1;
    if (clk_14971_1_1_2_1) v_15076_1_2_1 = (unsigned char)1;
    else {
      if (clk_14887_2_1_1_2_1) clk_14973_1_1_2_1 = (unsigned char)(
                               ! _L24_1_2_1_1_2_1 & ! _L25_1_2_1_1_2_1);
      else
        if (clk_14889_2_1_1_2_1) clk_14973_1_1_2_1 = (unsigned char)0;
        else
          if (clk_14891_2_1_1_2_1) clk_14973_1_1_2_1 = (unsigned char)1;
          else clk_14973_1_1_2_1 = (unsigned char)0;
      if (clk_14973_1_1_2_1) v_15076_1_2_1 = (unsigned char)1;
      else v_15076_1_2_1 = (unsigned char)0;
    }
  }
  clk_15035_1_2_1 = (unsigned char)(! v_15076_1_2_1);
}
else clk_15035_1_2_1 = (unsigned char)0;
if (clk_15035_1_2_1) {
  if (_L9_1_1_1_1_1_2_1) {
    if (_135__L9_1_1_1_1_1_2_1) tmp3 = _133__L5_1_1_1_1_1_2_1;
    else tmp3 = (unsigned char)0;
    if (tmp3) tmp8 = LOCsCursBlockHF_1_2_1.iBlockDirByInc;
    else {
      if (_135__L9_1_1_1_1_1_2_1) tmp2 = (unsigned char)(! _133__L5_1_1_1_1_1
      else tmp2 = (unsigned char)0;
      if (tmp2)
        if (_L5_1_1_1_1_1_2_1) tmp8 = 1; else tmp8 = 2;
      else tmp8 = 0;
    }
  }
  else tmp8 = 0;
  _139__L9_1_1_1_1_1_1_1_2_1 = (unsigned char)((tmp8 == 2) | (tmp8 == 1));
  if (_139__L9_1_1_1_1_1_1_1_2_1)
    if (tmp8 == 2) tmp6 = LOCsCursBlockHF_1_2_1.iNextBlockInUpDir;
    else tmp6 = LOCsCursBlockHF_1_2_1.iNextBlockInDnDir;
  else tmp6 = -1;
  _140__L5_1_1_1_1_1_1_1_2_1 = (unsigned char)(tmp6 < g_iBlockNumST);
  if (tmp8 == 2) _14_tmp_1_1 = LOCsCursBlockHF_1_2_1.iPntFltStateUp;
```

Domain of Initialized Variables

- Set of variables that are garanteed to have been initialized before.
- Join operation:

$$\mathcal{V}_1 \sqcup \mathcal{V}_2 \triangleq \mathcal{V}_1 \cap \mathcal{V}_2$$

- Transfer functions:

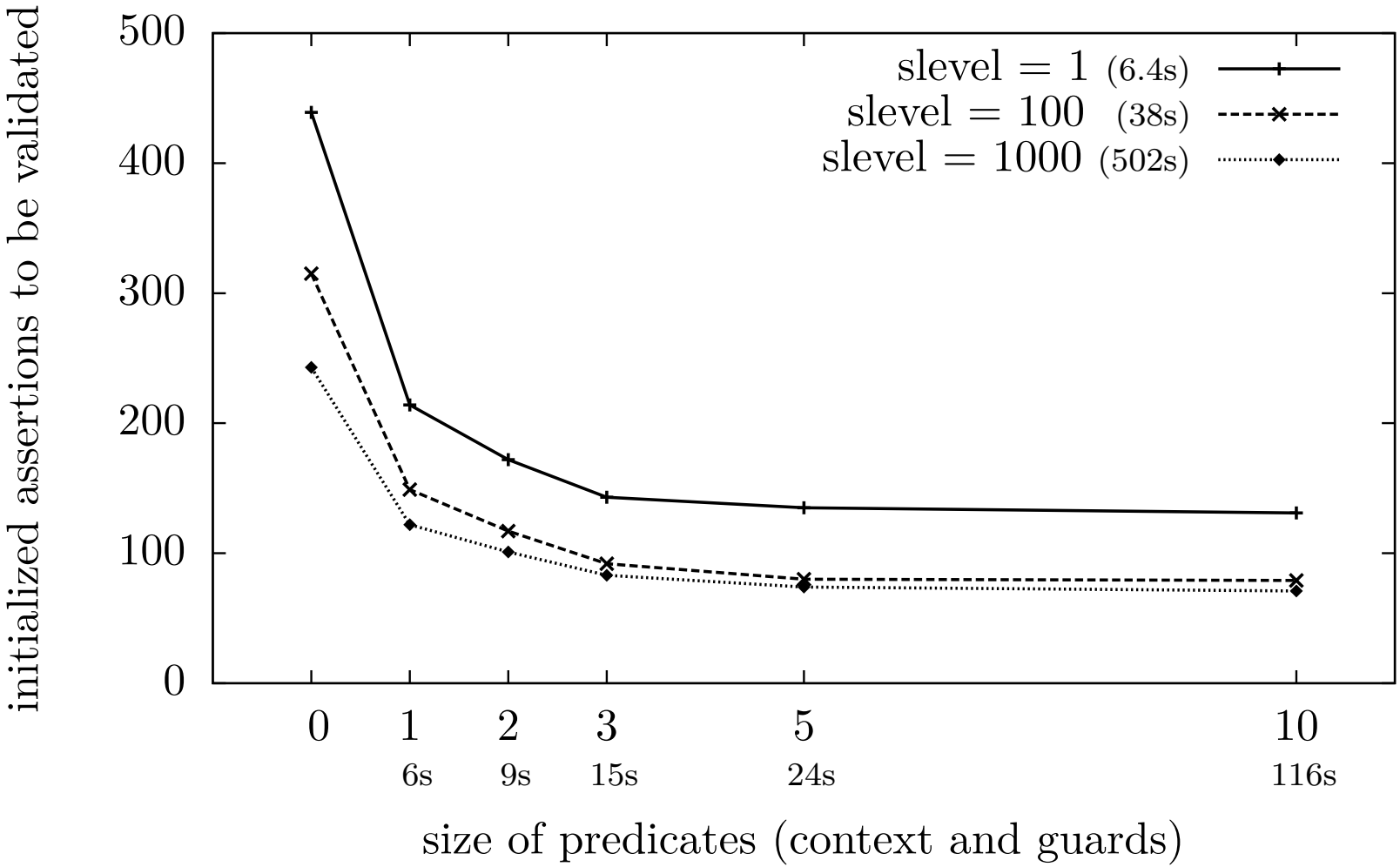
$$\begin{aligned} [\![\texttt{if}\,(e)]\!]^\sharp(\mathcal{V}) &\triangleq \mathcal{V} \\ [\![x := e]\!]^\sharp(\mathcal{V}) &\triangleq \begin{cases} \mathcal{V} \cup \{x\} & \text{if } \texttt{var}\,(e) \subseteq \mathcal{V} \\ \mathcal{V} \setminus \{x\} & \text{otherwise} \end{cases} \end{aligned}$$

Implementation

- Plugin above the *Value Analysis*: uses its alias information to remove predicates whose truth value is modified.
- Predicates restricted to negation, conjunction and disjunction of uninterpreted C expressions (stored in DNF form).
- Limitation over the number of litterals in predicates: *clevel*.

Experimental Results

Tests on a SCADE generated program of ~5000 lines of code.



Future Works

- Improve the interpretation of guards:
  - For now, C expressions used in predicates are not interpreted;
  - Goal: handle arithmetic entailments between guards like $x > 0$ and $x > 1$.
- Select relevant predicates (and remove the others) by heuristics at join points or by a lighter pre-analysis.
  - Goal: speed up the analysis.
- Apply predicated analysis over more complex domains.