IA159 Formal Methods for Software Analysis

Configurable Program Analysis

Jan Strejček

Faculty of Informatics
Masaryk University

Focus and sources

focus

- data-flow analysis (a sort of abstract interpretation, again)
- software model checking (= abstract reachability)
- configurable program analysis

source

- D. Beyer, S. Gulwani, and D. A. Smith: Combining Model Checking and Data-Flow Analysis, Chapter 16 of Handbook of Model Checking, Springer, 2018.

Motivation

- similarity of data-flow analysis and abstract reachability
- generalized into configurable program analysis (CPA)
- various known algorithms can be seen as CPA instances
- CPAs are easy to compose
- used in CPAchecker

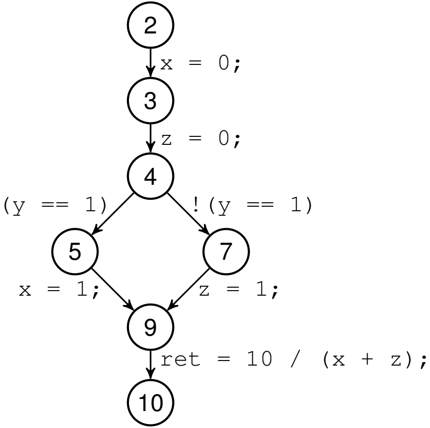
Control-flow automata

- graph representation of functions
- nodes = program locations
- edges = assumptions and assignments

Control-flow automata

- graph representation of functions
- nodes = program locations
- edges = assumptions and assignments

```
 1  int foo(int y) {
 2    int x = 0;
 3    int z = 0;
 4    if (y == 1) {
 5      x = 1;
 6    } else {
 7      z = 1;
 8    }
 9    return 10 / (x + z);
10  }
```



Control-flow automata

Definition (control-flow automaton)

Control-flow automaton (CFA) is a triple (L, l_0, G), where

- L is a finite set of program locations,
- $l_0 \in L$ is an initial program location, and
- $G \subseteq L \times Ops \times L$ are control-flow edges labeled with operations Ops.

- we assume that programs handle only integer variables and contain no function calls

Semantics of control-flow automata

- a concrete state is an assignment c that assigns values to program variables and also to program counter
- C denotes the set of all concrete states
- subsets $r \subseteq C$ are called regions
- each $g = (l, o, l') \in G$ defines the transition relation $\xrightarrow{g} \subseteq C \times \{g\} \times C$ (this is the semantics of CFA)
- we write $c \xrightarrow{g} c'$ instead of $(c, g, c') \in \xrightarrow{g}$
- we write $c \rightarrow c'$ if $c \xrightarrow{g} c'$ for some $g \in G$
- c is reachable from a region r if there is a state $c' \in r$ such that $c' \rightarrow^* c$, where $\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$

Semi-lattice

Definition (semi-lattice)

Semi-lattice is a tuple $(E, \sqsubseteq, \sqcup, \top)$, where

- $(E, \sqsubseteq)$ is a partially ordered set such that each $M \subseteq E$ has the least upper bound $sup(M)$,
- $\sqcup$ is the join operator satisfying $x \sqcup y = sup(\{x, y\})$,
- $\top$ is the top element $\top = sup(E)$.

Semi-lattice

Definition (semi-lattice)

Semi-lattice is a tuple $(E, \sqsubseteq, \sqcup, \top)$, where

- $(E, \sqsubseteq)$ is a partially ordered set such that each $M \subseteq E$ has the least upper bound $sup(M)$,
- $\sqcup$ is the join operator satisfying $x \sqcup y = sup(\{x, y\})$,
- $\top$ is the top element $\top = sup(E)$.

- elements of E represent abstract states

Abstract domain

Definition (abstract domain)

Abstract domain is a tuple $(C, \mathcal{E}, [\![\cdot]\!])$, where

- C is the set of concrete states,
- $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$ is a semi-lattice,
- $[\![\cdot]\!] : E \to 2^C$ is a concretization function.

Abstract domain

Definition (abstract domain)

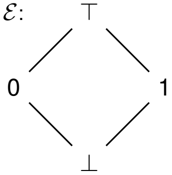
Abstract domain is a tuple $(C, \mathcal{E}, [\![\cdot]\!])$, where

- C is the set of concrete states,
- $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$ is a semi-lattice,
- $[\![\cdot]\!] : E \to 2^C$ is a concretization function.

- abstract domain determines the aspects of the program that are analyzed
- $[\![e]\!]$ returns the set of concrete states represented by e

Example

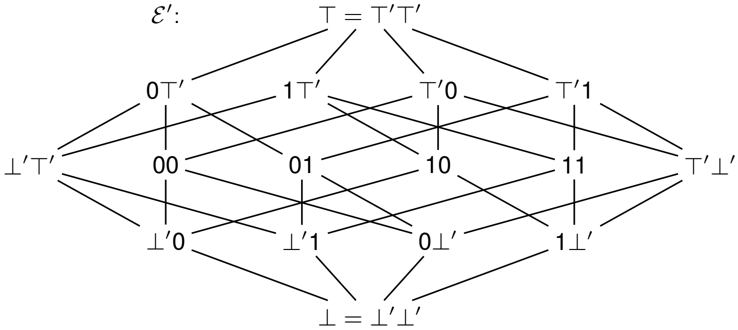
Abstract domain $(C, \mathcal{E}, [\![\cdot]\!])$ for tracking specific values $0, 1$ of a variable $\mathtt{x}$



- $[\![\top]\!] = C$
- $[\![0]\!] = \{c \in C \mid c(\mathtt{x}) = 0\}$
- $[\![1]\!] = \{c \in C \mid c(\mathtt{x}) = 1\}$
- $[\![\bot]\!] = \emptyset$

Example

Abstract domain $(C, \mathcal{E}', [\![\cdot]\!])$ for tracking specific values 0, 1 of variables $\mathtt{x}$, $\mathtt{z}$



$\mathcal{E}'$:

$\top = \top'\top'$

$0\top'$ $1\top'$ $\top'0$ $\top'1$

$\bot'\top'$ 00 01 10 11 $\top'\bot'$

$\bot'0$ $\bot'1$ $0\bot'$ $1\bot'$

$\bot = \bot'\bot'$

- $[\![01]\!] = \{c \in C \mid c(\mathtt{x}) = 0 \ \wedge \ c(\mathtt{z}) = 1\}$
- $[\![0\top']\!] = \{c \in C \mid c(\mathtt{x}) = 0\}$

Transfer relation

transfer relation

- $\rightsquigarrow \subseteq E \times G \times E$ represents for each abstract state e its abstract successor e' under edge g
- we write $e \overset{g}{\rightsquigarrow} e'$ instead of $(e, g, e') \in \rightsquigarrow$
- we write $e \rightsquigarrow e'$ if $e \overset{g}{\rightsquigarrow} e'$ for some $g \in G$
- for example, for g with assignment `x = z + 1` we have
    - $\top'0 \overset{g}{\rightsquigarrow} 10$
    - $01 \overset{g}{\rightsquigarrow} \top'1$

Data-flow analysis

Data-flow analysis (DFA)

- may forward abstract interpretation
- program locations are now handled explicitly, i.e., we work with pairs (l, e) instead of l being a part of e

Data-flow analysis (DFA)

- may forward abstract interpretation
- program locations are now handled explicitly, i.e., we work with pairs (l, e) instead of l being a part of e

inputs $(L, \mathcal{A}, \rightsquigarrow, l_0, e_0)$

- program locations L, abstract domain $\mathcal{A}$, transfer relation $\rightsquigarrow$
- initial abstract state (l_0, e_0)

Data-flow analysis (DFA)

- may forward abstract interpretation
- program locations are now handled explicitly, i.e., we work with pairs (l, e) instead of l being a part of e

inputs $(L, \mathcal{A}, \rightsquigarrow, l_0, e_0)$

- program locations L, abstract domain $\mathcal{A}$, transfer relation $\rightsquigarrow$
- initial abstract state (l_0, e_0)

output

- reached : $L \rightarrow E$ gives a reachable abstract state $(l, \text{reached}(l))$ for each l
- reached overapproximates all concrete reachable states, i.e., each concrete state c reachable from $[\![(l_0, e_0)]\!]$ is in $\bigcup_{l \in L} [\![(l, \text{reached}(l))]\!]$

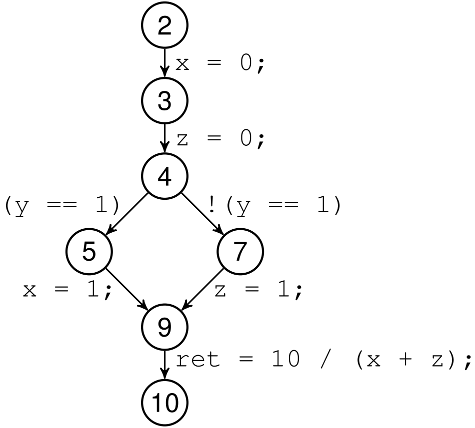
Data-flow analysis (DFA)

```
algorithm $\texttt{DFA}(L, \mathcal{A}, \rightsquigarrow, l_0, e_0)$
    waitList $\leftarrow \{l_0\}$
    reached$(l_0) \leftarrow e_0$
    while waitList $\neq \emptyset$ do
        choose $l$ from waitList
        waitList $\leftarrow$ waitList $\setminus \{l\}$
        foreach $(l', e')$ such that $(l, \text{reached}(l)) \rightsquigarrow (l', e')$ do
            if $e' \not\sqsubseteq \text{reached}(l')$ then
                reached$(l') \leftarrow \text{reached}(l') \sqcup e'$
                waitList $\leftarrow$ waitList $\cup \{l'\}$
    return reached
```

- if reached(l') has not been defined yet, then
  - $e' \not\sqsubseteq$ reached(l') is true
  - reached(l') $\sqcup\, e'$ evaluates to e'
- the algorithm finishes if the height of the semi-lattice in $\mathcal{A}$ is finite

Example

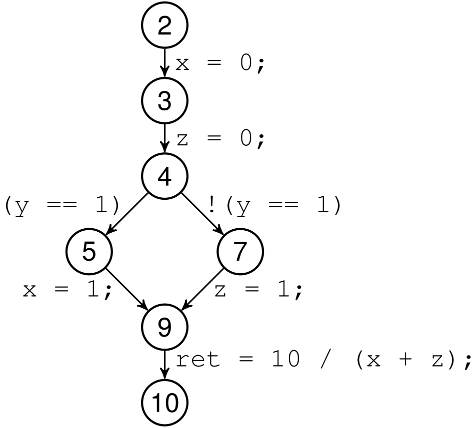
Track the values $0, 1$ of variables $\mathtt{x}, \mathtt{z}$ using the data-flow analysis with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.



Example

Track the values $0, 1$ of variables $\texttt{x}, \texttt{z}$ using the data-flow analysis with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

reached(2) $= \top'\top'$



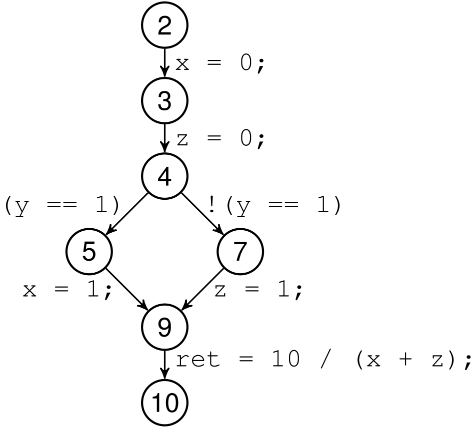
waitList $= \{2\}$

Example

Track the values 0, 1 of variables x, z using the data-flow analysis with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

reached(2) $= \top'\top'$

reached(3) $= 0\top'$



waitList $= \{3\}$

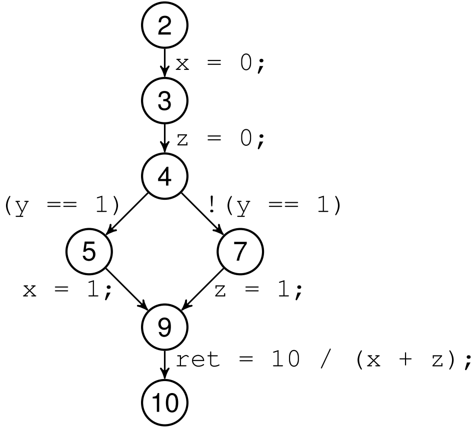
Example

Track the values $0, 1$ of variables $\mathtt{x}, \mathtt{z}$ using the data-flow analysis with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

$\text{reached}(2) = \top'\top'$

$\text{reached}(3) = 0\top'$

$\text{reached}(4) = 00$



$\text{waitList} = \{4\}$

Example

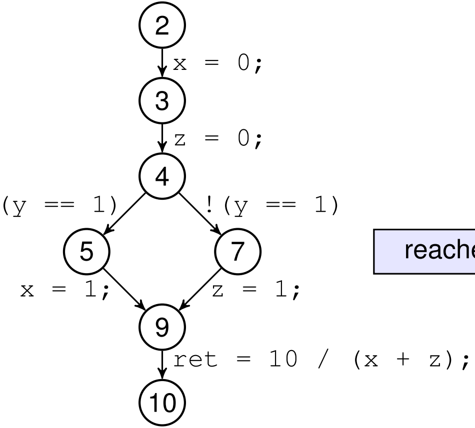
Track the values $0, 1$ of variables $\mathtt{x}, \mathtt{z}$ using the data-flow analysis with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

$\text{reached}(2) = \top'\top'$

$\text{reached}(3) = 0\top'$

$\text{reached}(4) = 00$

$\text{reached}(5) = 00$



$\text{reached}(7) = 00$

$\text{waitList} = \{5, 7\}$

Example

Track the values 0, 1 of variables x, z using the data-flow analysis with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

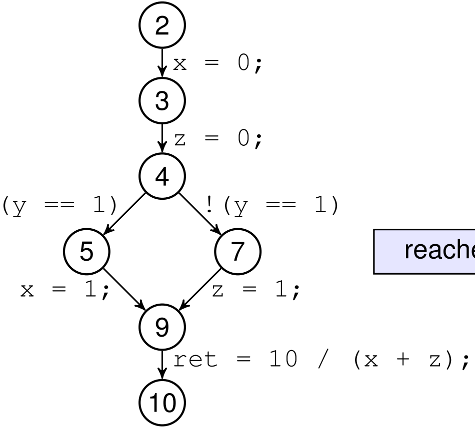
reached(2) $= \top'\top'$

reached(3) $= 0\top'$

reached(4) $= 00$

reached(5) $= 00$

reached(9) $= 10$

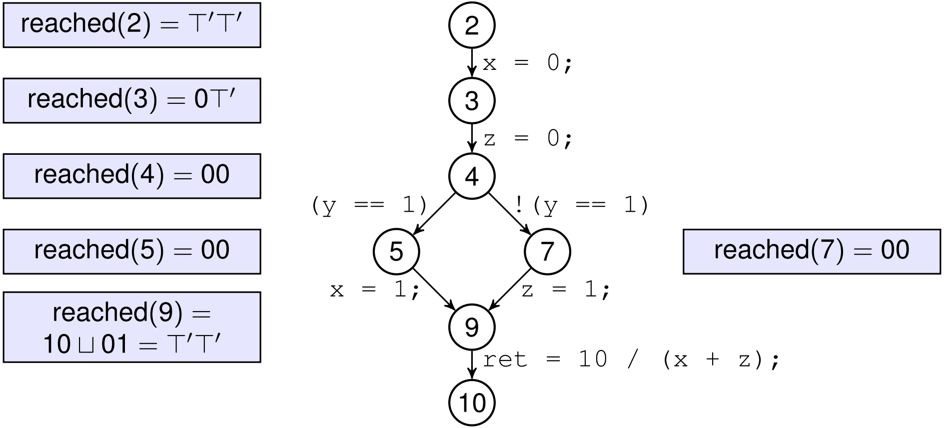


reached(7) $= 00$

waitList $= \{7, 9\}$

Example

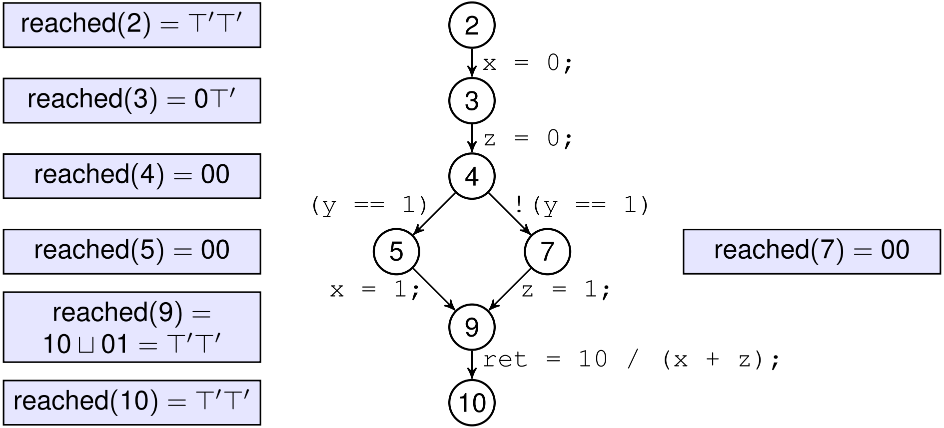
Track the values 0, 1 of variables $\mathtt{x}, \mathtt{z}$ using the data-flow analysis with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.



waitList $= \{9\}$

Example

Track the values 0, 1 of variables x, z using the data-flow analysis with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.



reached(2) $= \top'\top'$

reached(3) $= 0\top'$

reached(4) $= 00$

reached(5) $= 00$

reached(7) $= 00$

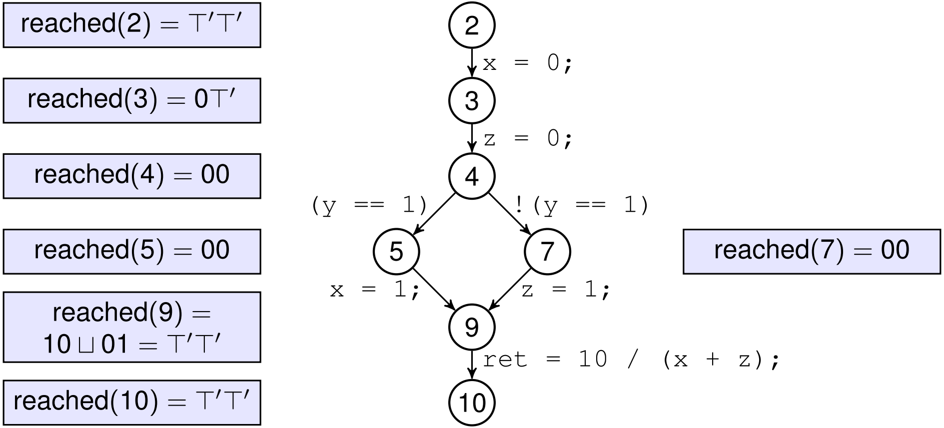
reached(9) $=$ $10 \sqcup 01 = \top'\top'$

reached(10) $= \top'\top'$

waitList $= \{10\}$

Example

Track the values $0, 1$ of variables $\mathtt{x}, \mathtt{z}$ using the data-flow analysis with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.



waitList $= \emptyset$

Software model checking

Software model checking

- computes all reachable abstract states according to the transfer relation
- join operator is never applied

Software model checking

- computes all reachable abstract states according to the transfer relation
- join operator is never applied

inputs $(L, \mathcal{A}, \rightsquigarrow, l_0, e_0)$

- program locations L, abstract domain $\mathcal{A}$, transfer relation $\rightsquigarrow$
- initial abstract state (l_0, e_0)

Software model checking

- computes all reachable abstract states according to the transfer relation
- join operator is never applied

inputs $(L, \mathcal{A}, \rightsquigarrow, l_0, e_0)$

- program locations L, abstract domain $\mathcal{A}$, transfer relation $\rightsquigarrow$
- initial abstract state (l_0, e_0)

output

- reached $\subseteq L \times E$ of reachable abstract states
- reached overapproximates all concrete reachable states, i.e., each concrete state c reachable from $[\![(l_0, e_0)]\!]$ is in $\bigcup_{(l,e)\in\text{reached}} [\![(l, e)]\!]$

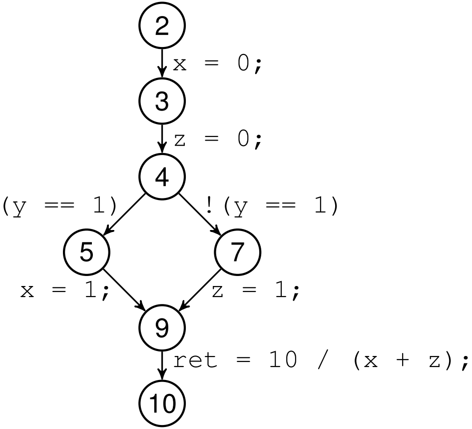
Software model checking

```
algorithm Reach($L, \mathcal{A}, \rightsquigarrow, l_0, e_0$)
    waitList ← $\{(l_0, e_0)\}$
    reached ← $\{(l_0, e_0)\}$
    while waitList $\neq \emptyset$ do
        choose $(l, e)$ from waitList
        waitList ← waitList $\setminus \{(l, e)\}$
        foreach $(l', e')$ such that $(l, e) \rightsquigarrow (l', e')$ do
            if there is no $(l', e'') \in$ reached such that $e' \sqsubseteq e''$ then
                reached ← reached $\cup \{(l', e')\}$
                waitList ← waitList $\cup \{(l', e')\}$
    return reached
```

- finishes if the semi-lattice is finite
- there are infinite semi-lattices of a finite height
- typically slower, but more precise than data-flow analysis

Example

Track the values $0, 1$ of variables `x`, `z` using software model checking with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

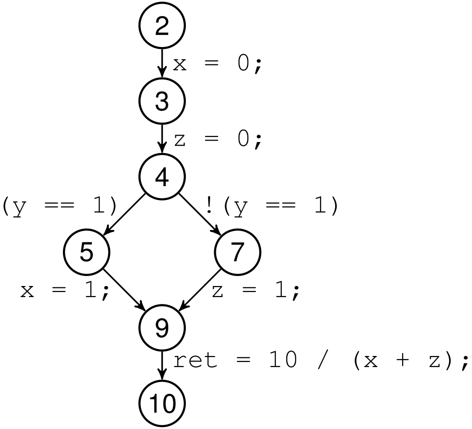


Example

Track the values 0, 1 of variables $\texttt{x}, \texttt{z}$ using software model checking with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

reached $= \{\,(2, \top'\top'),$

$\}$



waitList $= \{(2, \top'\top')\}$

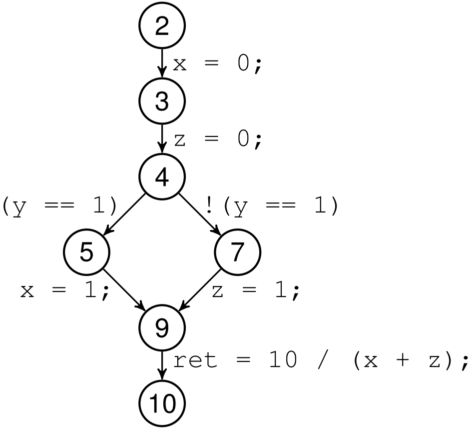
Example

Track the values $0, 1$ of variables `x`, `z` using software model checking with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

$\text{reached} = \{\, (2, \top'\top'),$

$(3, 0\top'),$

$\}$

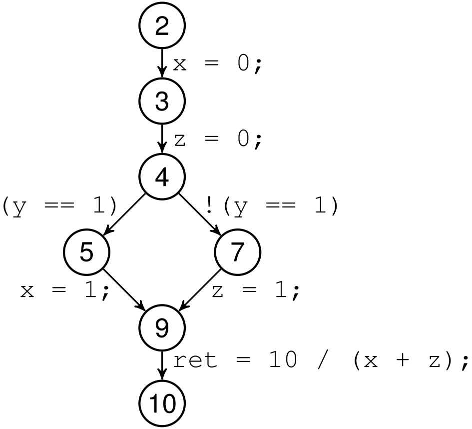


$\text{waitList} = \{(3, 0\top')\}$

Example

Track the values $0, 1$ of variables `x`, `z` using software model checking with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

$$\text{reached} = \{\, (2, \top'\top'), \quad (3, 0\top'), \quad (4, 00), \quad \}$$



$\text{waitList} = \{(4, 00)\}$

Example

Track the values 0, 1 of variables `x`, `z` using software model checking with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

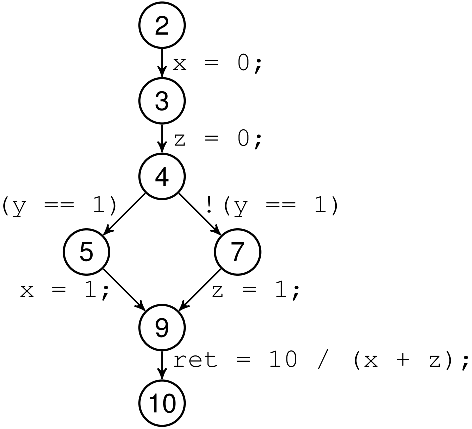
$\text{reached} = \{ (2, \top'\top'),$

$(3, 0\top'),$

$(4, 00),$

$(5, 00),$
$(7, 00),$

$\}$

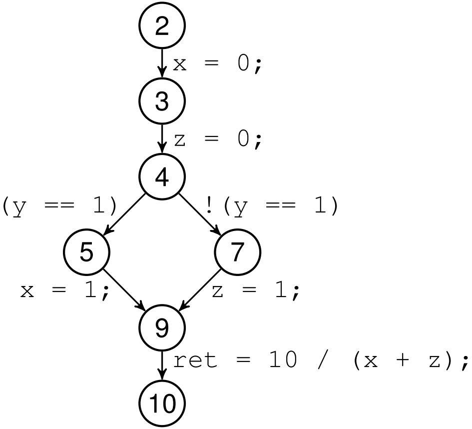


$\text{waitList} = \{(5, 00), (7, 00)\}$

Example

Track the values 0, 1 of variables `x`, `z` using software model checking with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

$$\text{reached} = \{\, (2, \top'\top'),\ (3, 0\top'),\ (4, 00),\ (5, 00),\ (7, 00),\ (9, 10),\ \}$$



$$\text{waitList} = \{(7, 00), (9, 10)\}$$

Example

Track the values 0, 1 of variables `x`, `z` using software model checking with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

reached $= \{\,(2, \top'\top'),$

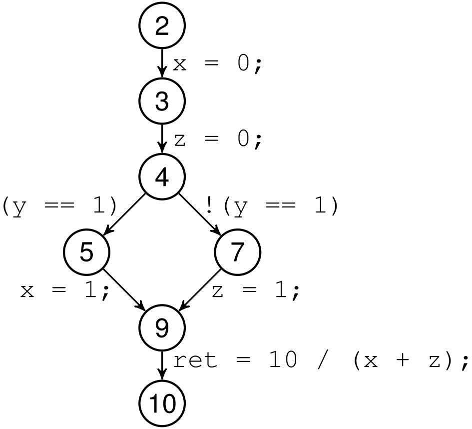
$(3, 0\top'),$

$(4, 00),$

$(5, 00),$
$(7, 00),$

$(9, 10),$
$(9, 01),$

$\}$

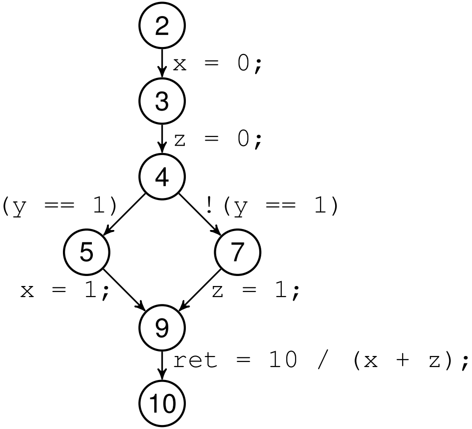


waitList $= \{(9, 10), (9, 01)\}$

Example

Track the values $0, 1$ of variables `x`, `z` using software model checking with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

$$\text{reached} = \{\,(2, \top'\top'),\quad (3, 0\top'),\quad (4, 00),\quad (5, 00),\quad (7, 00),\quad (9, 10),\quad (9, 01),\quad (10, 10),\,\}$$

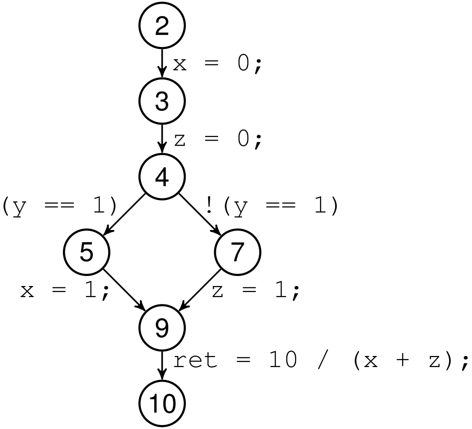


$\text{waitList} = \{(9, 01), (10, 10)\}$

Example

Track the values 0, 1 of variables `x`, `z` using software model checking with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

$$\begin{aligned} \text{reached} = \{\, & (2, \top'\top'), \\ & (3, 0\top'), \\ & (4, 00), \\ & (5, 00), \\ & (7, 00), \\ & (9, 10), \\ & (9, 01), \\ & (10, 10), \\ & (10, 01)\} \end{aligned}$$

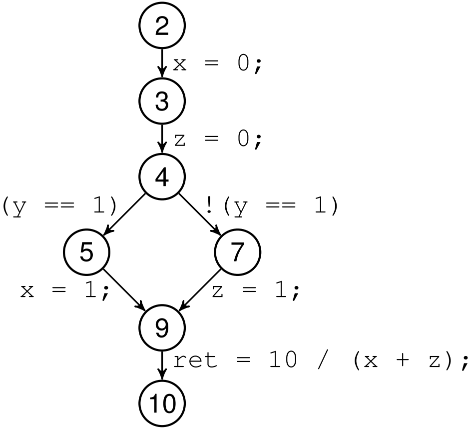


$\text{waitList} = \{(10, 10), (10, 01)\}$

Example

Track the values 0, 1 of variables `x`, `z` using software model checking with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

$$\begin{aligned}\text{reached} = \{\, & (2, \top'\top'), \\ & (3, 0\top'), \\ & (4, 00), \\ & (5, 00), \\ & (7, 00), \\ & (9, 10), \\ & (9, 01), \\ & (10, 10), \\ & (10, 01)\}\end{aligned}$$

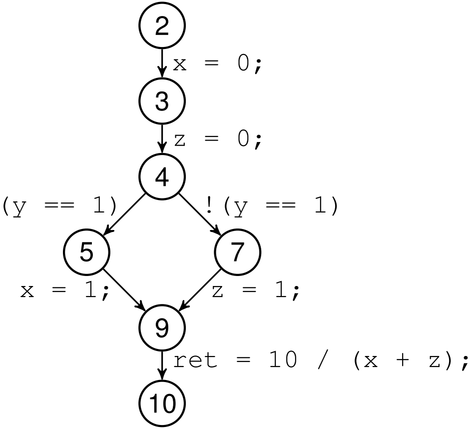


$\text{waitList} = \{(10, 01)\}$

Example

Track the values $0, 1$ of variables `x`, `z` using software model checking with the abstract domain based on the semi-lattice $\mathcal{E}'$ and the initial abstract state $(2, \top'\top')$.

$$
\begin{aligned}
\text{reached} = \{ & (2, \top'\top'), \\
& (3, 0\top'), \\
& (4, 00), \\
& (5, 00), \\
& (7, 00), \\
& (9, 10), \\
& (9, 01), \\
& (10, 10), \\
& (10, 01)\}
\end{aligned}
$$



waitList $= \emptyset$

Configurable program analysis

Similarity of the two algorithms

```
algorithm DFA($L, \mathcal{A}, \rightsquigarrow, l_0, e_0$)
    waitList ← $\{l_0\}$
    reached($l_0$) ← $e_0$
    while waitList $\neq \emptyset$ do
        choose $l$ from waitList
        waitList ← waitList $\setminus \{l\}$
        foreach $(l', e')$ such that $(l, \text{reached}(l)) \rightsquigarrow (l', e')$ do
            if $e' \not\sqsubseteq \text{reached}(l')$ then
                reached($l'$) ← reached($l'$) $\sqcup e'$
                waitList ← waitList $\cup \{l'\}$
    return reached
```

```
algorithm Reach($L, \mathcal{A}, \rightsquigarrow, l_0, e_0$)
    waitList ← $\{(l_0, e_0)\}$
    reached ← $\{(l_0, e_0)\}$
    while waitList $\neq \emptyset$ do
        choose $(l, e)$ from waitList
        waitList ← waitList $\setminus \{(l, e)\}$
        foreach $(l', e')$ such that $(l, e) \rightsquigarrow (l', e')$ do
            if there is no $(l', e'') \in$ reached such that $e' \sqsubseteq e''$ then
                reached ← reached $\cup \{(l', e')\}$
                waitList ← waitList $\cup \{(l', e')\}$
    return reached
```

Configurable program analysis

Definition (configurable program analysis)

Configurable program analysis (CPA) is a tuple $(\mathcal{A}, \rightsquigarrow, \mathrm{merge}, \mathrm{stop})$, where

- $\mathcal{A}$ is an abstract domain,
- $\rightsquigarrow$ is a transfer relation,
- merge : $E \times E \rightarrow E$ is a merge operator that combines two abstract states such that it can weaken the second abstract state based on the first one (correspond to widening), i.e., $e' \sqsubseteq \mathrm{merge}(e, e') \sqsubseteq \top$,
- stop : $E \times 2^E \rightarrow \mathbb{B}$ is a termination check such that $\mathrm{stop}(e, R)$ checks if e is covered (in some sense) by the set of abstract states R.

Configurable program analysis

- unifies the data-flow analysis and software model checking
- handles program locations implicitly

Configurable program analysis

- unifies the data-flow analysis and software model checking
- handles program locations implicitly

inputs $(\mathcal{A}, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, e_0)$

- CPA $(\mathcal{A}, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$
- initial abstract state e_0

Configurable program analysis

- unifies the data-flow analysis and software model checking
- handles program locations implicitly

inputs $(\mathcal{A}, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, e_0)$

- CPA $(\mathcal{A}, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$
- initial abstract state e_0

output

- reached $\subseteq E$ of reachable abstract states
- reached overapproximates all concrete reachable states, i.e., each concrete state c reachable from $[\![e_0]\!]$ is in $\bigcup_{e \in \mathsf{reached}} [\![e]\!]$

Configurable program analysis

```
algorithm $\texttt{CPA}(\mathcal{A}, \rightsquigarrow, \mathrm{merge}, \mathrm{stop}, e_0)$
    waitList $\leftarrow \{e_0\}$
    reached $\leftarrow \{e_0\}$
    while waitList $\neq \emptyset$ do
        choose $e$ from waitList
        waitList $\leftarrow$ waitList $\setminus \{e\}$
        foreach $e'$ such that $e \rightsquigarrow e'$ do
            foreach $e'' \in$ reached do
                $e_{new} = \mathrm{merge}(e', e'')$
                if $e_{new} \neq e''$ then
                    waitList $\leftarrow$ (waitList $\cup \{e_{new}\}) \setminus \{e''\}$
                    reached $\leftarrow$ (reached $\cup \{e_{new}\}) \setminus \{e''\}$
            if $\neg\mathrm{stop}(e', \mathrm{reached})$ then
                waitList $\leftarrow$ waitList $\cup \{e'\}$
                reached $\leftarrow$ reached $\cup \{e'\}$
    return reached
```

Notes

typical instances of merge

- $\text{merge}^{sep}(e, e') = e'$
- $\text{merge}^{join}(e, e') = e \sqcup e'$

Notes

typical instances of merge

- $\mathrm{merge}^{sep}(e, e') = e'$
- $\mathrm{merge}^{join}(e, e') = e \sqcup e'$

typical instances of stop

- $\mathrm{stop}^{sep}(e, R) = (\exists e' \in R . e \sqsubseteq e')$
- $\mathrm{stop}^{join}(e, R) = e \sqsubseteq \bigsqcup_{e' \in R} e'$

- another parameter of all the algorithms is the order in which elements of waitList are processed
- for example, it can correspond to depth-first search or breadth-first search
- different strategies are used in practice

Notes

there are CPA instances for

- reachable-code analysis
- constant propagation
- reaching definitions
- predicate analysis
- observer automata
- value analysis
- symbolic execution
- …

Notes

- a CPA can be constructed as a combination of simpler CPAs (easier to implement)
- combinations used in practice
  - constant propagation + predicate analysis + (strengthening)
  - predicate analysis + observer automata
  - …
- can be extended with the notion of precision and CEGAR
- can be used for computation of program invariants
- implemented in CPAchecker

Notes

- a CPA can be constructed as a combination of simpler CPAs (easier to implement)
- combinations used in practice
  - constant propagation + predicate analysis + (strengthening)
  - predicate analysis + observer automata
  - …
- can be extended with the notion of precision and CEGAR
- can be used for computation of program invariants
- implemented in CPAchecker

CPAchecker

- verification tool developed by the group of Dirk Beyer since 2007
- implements various techniques, supports their combinations
- available under the Apache 2.0 License
- `https://cpachecker.sosy-lab.org/`