Closed Symbolic Execution for Verifying Program Termination

Germán Vidal

Technical University of Valencia

12th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation
SCAM 2012

Sep 23-24, 2012
Riva del Garda, Italy

Motivation

Using symbolic execution for proving program termination
(and other liveness properties)

- there are already a few ad-hoc approaches: AProVE, Costa, Julia (Haskell, Prolog, Java bytecode)
- we aim at presenting a higher-level, language independent scheme using well-known notions and techniques from partial evaluation

Motivation

Using symbolic execution for proving program termination (and other liveness properties)

- there are already a few ad-hoc approaches: AProVE, Costa, Julia (Haskell, Prolog, Java bytecode)
- we aim at presenting a higher-level, language independent scheme using well-known notions and techniques from partial evaluation

Motivation

Using symbolic execution for proving program termination
(and other liveness properties)

- there are already a few ad-hoc approaches: AProVE, Costa, Julia (Haskell, Prolog, Java bytecode)
- we aim at presenting a higher-level, language independent scheme using well-known notions and techniques from partial evaluation

Symbolic execution

- Extension of standard execution for unkown input data (symbolic values)

- Usually an underapproximation of standard execution
  - requires subsumption and abstraction for efficiency

- Mainly used for testing and debugging in imperative languages

Symbolic execution

- Extension of standard execution for unkown input data (symbolic values)
- Usually an underapproximation of standard execution
  - requires subsumption and abstraction for efficiency
- Mainly used for testing and debugging in imperative languages

Symbolic execution

- Extension of standard execution for unkown input data (symbolic values)
- Usually an underapproximation of standard execution
  - requires subsumption and abstraction for efficiency
- Mainly used for testing and debugging in imperative languages

Symbolic execution: example

$$
\begin{array}{ll}
l_0: & x := \textbf{input}(); \\
l_1: & \textbf{while } x > 0 \textbf{ do} \\
l_2: & \quad\quad x := x - 1; \\
l_3: & \textbf{done}
\end{array}
$$

$$
\begin{array}{c}
\langle l_0, \{x \mapsto \bot\}\rangle \\
\downarrow \\
\langle l_1, \{x \mapsto 1\}\rangle \\
\downarrow \\
\langle l_2, \{x \mapsto 1\}\rangle \\
\downarrow \\
\langle l_1, \{x \mapsto 0\}\rangle \\
\downarrow \\
\langle l_3, \{x \mapsto 0\}\rangle
\end{array}
\quad \Rightarrow \quad
\begin{tikzcd}
 & \langle l_0, \{x \mapsto \bot\}, true\rangle \arrow[d] \\
 & \langle l_1, \{x \mapsto X\}, true\rangle \arrow[d] \arrow[dl] \\
\langle l_3, \{x \mapsto X\}, \neg(X > 0)\rangle & \langle l_2, \{x \mapsto X\}, X > 0\rangle \arrow[d] \\
 & \langle l_1, \{x \mapsto X - 1\}, X > 0\rangle \arrow[d] \arrow[dl] \\
\langle l_3, \{x \mapsto X - 1\}, \neg(X > 1)\rangle & \langle l_2, \{x \mapsto X - 1\}, X > 1\rangle \arrow[d] \\
 & \cdots
\end{tikzcd}
$$

Symbolic execution: example

$$
\begin{array}{ll}
l_0: & x := \textbf{input}(); \\
l_1: & \textbf{while } x > 0 \textbf{ do} \\
l_2: & \qquad x := x - 1; \\
l_3: & \textbf{done}
\end{array}
$$

$$
\begin{array}{c}
\langle l_0, \{x \mapsto \bot\}\rangle \\
\downarrow \\
\langle l_1, \{x \mapsto 1\}\rangle \\
\downarrow \\
\langle l_2, \{x \mapsto 1\}\rangle \\
\downarrow \\
\langle l_1, \{x \mapsto 0\}\rangle \\
\downarrow \\
\langle l_3, \{x \mapsto 0\}\rangle
\end{array}
\quad \Rightarrow \quad
\begin{array}{cc}
 & \langle l_0, \{x \mapsto \bot\}, true\rangle \\
 & \downarrow \\
 & \langle l_1, \{x \mapsto X\}, true\rangle \\
\swarrow & \downarrow \\
\langle l_3, \{x \mapsto X\}, \neg(X > 0)\rangle & \langle l_2, \{x \mapsto X\}, X > 0\rangle \\
 & \downarrow \\
 & \langle l_1, \{x \mapsto X - 1\}, X > 0\rangle \\
\swarrow & \downarrow \\
\langle l_3, \{x \mapsto X - 1\}, \neg(X > 1)\rangle & \langle l_2, \{x \mapsto X - 1\}, X > 1\rangle \\
 & \downarrow \\
 & \ldots
\end{array}
$$

Symbolic execution: example

$$
\begin{array}{ll}
l_0: & x := \textbf{input}(); \\
l_1: & \textbf{while } x > 0 \textbf{ do} \\
l_2: & \qquad x := x - 1; \\
l_3: & \textbf{done}
\end{array}
$$

$$
\begin{tikzcd}
\langle l_0, \{x \mapsto \bot\}\rangle \arrow[d] \\
\langle l_1, \{x \mapsto 1\}\rangle \arrow[d] \\
\langle l_2, \{x \mapsto 1\}\rangle \arrow[d] \\
\langle l_1, \{x \mapsto 0\}\rangle \arrow[d] \\
\langle l_3, \{x \mapsto 0\}\rangle
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
 & \langle l_0, \{x \mapsto \bot\}, \mathit{true}\rangle \arrow[d] \\
 & \langle l_1, \{x \mapsto X\}, \mathit{true}\rangle \arrow[d] \arrow[dl] \\
\langle l_3, \{x \mapsto X\}, \neg(X > 0)\rangle & \langle l_2, \{x \mapsto X\}, X > 0\rangle \arrow[d] \\
 & \langle l_1, \{x \mapsto X - 1\}, X > 0\rangle \arrow[d] \arrow[dl] \\
\langle l_3, \{x \mapsto X - 1\}, \neg(X > 1)\rangle & \langle l_2, \{x \mapsto X - 1\}, X > 1\rangle \arrow[d] \\
 & \ldots
\end{tikzcd}
$$

Partial evaluation

- Extension of standard execution for some unkown input data (symbolic values)
- Usually an overapproximation of standard execution
  - requires subsumption and abstraction for termination
- Mainly used for program specialization in declarative (functional, logic, etc) languages

Partial evaluation

- Extension of standard execution for some unkown input data (symbolic values)
- Usually an overapproximation of standard execution
  - requires subsumption and abstraction for termination
- Mainly used for program specialization in declarative (functional, logic, etc) languages

Partial evaluation

- Extension of standard execution for some unkown input data (symbolic values)
- Usually an overapproximation of standard execution
  - requires subsumption and abstraction for termination
- Mainly used for program specialization in declarative (functional, logic, etc) languages

Partial evaluation: example

$$
\begin{array}{lcl}
\text{power } x \ 0 & = & 1 \\
\text{power } x \ 1 & = & x \\
\text{power } x \ \text{n} & = & x \ * \ \text{power } x \ (\text{n} - 1)
\end{array}
$$

$$
\begin{tikzcd}
 & \text{power } x \ (n+2) \arrow[d] \\
 & x * \text{power } x \ (n+1) \arrow[dl, "\{n \mapsto 0\}"'] \arrow[d, "\{n \mapsto m+1\}"] \\
x * x & x * x * \text{power } x \ (m+1) \arrow[u, bend right]
\end{tikzcd}
$$

$$
\Rightarrow \quad
\begin{array}{lcl}
\text{power}_{+2} \ x \ n & = & x * \text{power}_{+1} \ x \ n \\
\text{power}_{+1} \ x \ 0 & = & x * x \\
\text{power}_{+1} \ x \ n & = & x * x * \text{power}_{+1} \ x \ (n-1)
\end{array}
$$

Partial evaluation: example

$$
\begin{array}{lcl}
\text{power } x \ 0 & = & 1 \\
\text{power } x \ 1 & = & x \\
\text{power } x \ \text{n} & = & x \ * \ \text{power } x \ (\text{n} - 1)
\end{array}
$$

$$
\begin{tikzcd}
 & \text{power } x \ (n+2) \arrow[d] \\
 & x * \text{power } x \ (n+1) \arrow[dl, "\{n \mapsto 0\}"'] \arrow[d, "\{n \mapsto m+1\}"] \\
x * x & x * x * \text{power } x \ (m+1) \arrow[u, bend right]
\end{tikzcd}
$$

$$
\Rightarrow \quad
\begin{array}{lcl}
\text{power}_{+2} \ x \ n & = & x * \text{power}_{+1} \ x \ n \\
\text{power}_{+1} \ x \ 0 & = & x * x \\
\text{power}_{+1} \ x \ n & = & x * x * \text{power}_{+1} \ x \ (n-1)
\end{array}
$$

Partial evaluation: example

$$
\begin{array}{lcl}
\text{power } x \text{ } 0 & = & 1 \\
\text{power } x \text{ } 1 & = & x \\
\text{power } x \text{ n} & = & x \; * \; \text{power } x \text{ (n} - 1)
\end{array}
$$

$$
\begin{tikzcd}
 & \text{power } x \ (n+2) \arrow[d] \\
 & x * \text{power } x \ (n+1) \arrow[dl, "\{n \mapsto 0\}"'] \arrow[d, "\{n \mapsto m+1\}"] \\
x * x & x * x * \text{power } x \ (m+1) \arrow[u, bend right]
\end{tikzcd}
$$

$$
\Rightarrow \quad
\boxed{
\begin{array}{lcl}
\text{power}_{+2} \ x \ n & = & x * \text{power}_{+1} \ x \ n \\
\text{power}_{+1} \ x \ 0 & = & x * x \\
\text{power}_{+1} \ x \ n & = & x * x * \text{power}_{+1} \ x \ (n-1)
\end{array}
}
$$

This work

- Use symbolic execution to overapproximate standard executions (as in partial evaluation)
- Use the symbolic execution graph for verifying program termination (or other liveness properties)

In particular,

- we produce a term rewriting system that represents the transitions of the symbolic execution graph (as in partial evaluation)
- and apply existing termination provers for term rewriting systems

This work

- Use symbolic execution to overapproximate standard executions (as in partial evaluation)
- Use the symbolic execution graph for verifying program termination (or other liveness properties)

In particular,

- we produce a term rewriting system that represents the transitions of the symbolic execution graph (as in partial evaluation)
- and apply existing termination provers for term rewriting systems

This work

- Use symbolic execution to overapproximate standard executions (as in partial evaluation)
- Use the symbolic execution graph for verifying program termination (or other liveness properties)

In particular,

- we produce a term rewriting system that represents the transitions of the symbolic execution graph (as in partial evaluation)
- and apply existing termination provers for term rewriting systems

Example

$$
\begin{array}{ll}
l_0 : & x := \textbf{input}(); \\
l_1 : & \textbf{while } x > 0 \textbf{ do} \\
l_2 : & \qquad x := x - 1; \\
l_3 : & \textbf{done}
\end{array}
$$

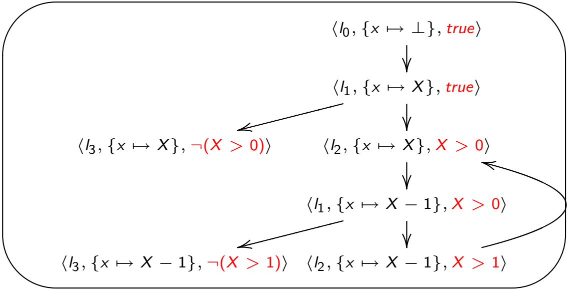
$\Rightarrow$

$$
\begin{array}{lll}
f_0\ \_ & = & f_1\ x \\
f_1\ x & = & \text{if } (x > 0) \\
 & & \text{then } f_3\ x \\
 & & \text{else } f_4\ x \\
f_4\ x & = & f_5\ (x - 1) \\
f_5\ x & = & \text{if } (x > 0) \\
 & & \text{then } f_3\ x \\
 & & \text{else } f_4\ x
\end{array}
$$

Terminating! (using AProVE)

Example

$$
\begin{array}{ll}
l_0: & x := \textbf{input}(); \\
l_1: & \textbf{while}\ x > 0\ \textbf{do} \\
l_2: & \quad x := x - 1; \\
l_3: & \textbf{done}
\end{array}
$$



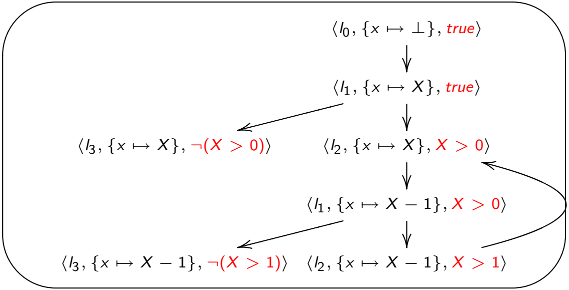
$\Rightarrow$

$$
\begin{array}{lcl}
f_0\ \_ & = & f_1\ x \\
f_1\ x & = & \text{if } (x > 0) \\
 & & \text{then } f_3\ x \\
 & & \text{else } f_4\ x \\
f_4\ x & = & f_5\ (x - 1) \\
f_5\ x & = & \text{if } (x > 0) \\
 & & \text{then } f_3\ x \\
 & & \text{else } f_4\ x
\end{array}
$$

Terminating! (using AProVE)

Example

$$
\begin{array}{lll}
l_0: & x := \mathbf{input}(); & \\
l_1: & \mathbf{while}\ x > 0\ \mathbf{do} & \\
l_2: & \qquad x := x - 1; & \\
l_3: & \mathbf{done} &
\end{array}
$$



$\Rightarrow$

$$
\begin{array}{lll}
f_0\ \_ & = & f_1\ x \\
f_1\ x & = & \text{if } (x > 0) \\
 & & \text{then } f_3\ x \\
 & & \text{else } f_4\ x \\
f_4\ x & = & f_5\ (x - 1) \\
f_5\ x & = & \text{if } (x > 0) \\
 & & \text{then } f_3\ x \\
 & & \text{else } f_4\ x
\end{array}
$$

Terminating! (using AProVE)

Example

$$
\begin{array}{ll}
l_0: & x := \textbf{input}(); \\
l_1: & \textbf{while}\ x > 0\ \textbf{do} \\
l_2: & \qquad x := x - 1; \\
l_3: & \textbf{done}
\end{array}
$$

$$
\begin{tikzcd}
 & \langle l_0, \{x \mapsto \bot\}, \mathit{true} \rangle \arrow[d] \\
 & \langle l_1, \{x \mapsto X\}, \mathit{true} \rangle \arrow[dl] \arrow[d] \\
\langle l_3, \{x \mapsto X\}, \neg(X > 0) \rangle & \langle l_2, \{x \mapsto X\}, X > 0 \rangle \arrow[d] \\
 & \langle l_1, \{x \mapsto X - 1\}, X > 0 \rangle \arrow[dl] \arrow[d] \\
\langle l_3, \{x \mapsto X - 1\}, \neg(X > 1) \rangle & \langle l_2, \{x \mapsto X - 1\}, X > 1 \rangle \arrow[uu, bend right]
\end{tikzcd}
\quad \Rightarrow \quad
\begin{array}{lcl}
f_0\ \_ & = & f_1\ x \\
f_1\ x & = & \text{if } (x > 0) \\
 & & \text{then } f_3\ x \\
 & & \text{else } f_4\ x \\
f_4\ x & = & f_5\ (x - 1) \\
f_5\ x & = & \text{if } (x > 0) \\
 & & \text{then } f_3\ x \\
 & & \text{else } f_4\ x
\end{array}
$$

Terminating! (using AProVE)

Proof-of-concept implementation

SETT (Symbolic Execution-based Termination Tool)

- simple imperative programs with integers, basic arithmetic, assignments, conditionals and jumps

web interface: http://kaz.dsic.upv.es/sett/

Conclusions

- Powerful scheme for proving program termination
- Same scheme can be used
  - for other (dynamic) programming languages (Erlang, JavaScript)
  - for other (liveness) properties