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 unknown 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 unknown 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 unknown 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

\[ \begin{align*}
  l_0 : & \quad x := \text{input}(); \\
  l_1 : & \quad \text{while } x > 0 \text{ do} \\
  l_2 : & \quad x := x - 1; \\
  l_3 : & \quad \text{done}
\end{align*} \]
Symbolic execution: example

\[ l_0 : \ x := \text{input}(); \]
\[ l_1 : \ \text{while } x > 0 \ \text{do} \]
\[ l_2 : \ x := x - 1; \]
\[ l_3 : \ \text{done} \]
Symbolic execution: example

\[ l_0 : \quad x := \text{input}(); \]
\[ l_1 : \quad \textbf{while} \ x > 0 \ \textbf{do} \]
\[ l_2 : \quad x := x - 1; \]
\[ l_3 : \quad \text{done} \]

\[ \langle l_0, \{ x \mapsto \bot \} \rangle \]
\[ \downarrow \]
\[ \langle l_1, \{ x \mapsto 1 \} \rangle \]
\[ \downarrow \]
\[ \langle l_2, \{ x \mapsto 1 \} \rangle \]
\[ \downarrow \]
\[ \langle l_1, \{ x \mapsto 0 \} \rangle \]
\[ \downarrow \]
\[ \langle l_3, \{ x \mapsto 0 \} \rangle \]

\[ \langle l_0, \{ x \mapsto \bot \}, \text{true} \rangle \]
\[ \downarrow \]
\[ \langle l_1, \{ x \mapsto X \}, \text{true} \rangle \]
\[ \downarrow \]
\[ \langle l_3, \{ x \mapsto X \}, \neg (X > 0) \rangle \]
\[ \langle l_2, \{ x \mapsto X \}, X > 0 \rangle \]
\[ \downarrow \]
\[ \langle l_1, \{ x \mapsto X - 1 \}, X > 0 \rangle \]
\[ \downarrow \]
\[ \langle l_3, \{ x \mapsto X - 1 \}, \neg (X > 1) \rangle \]
\[ \langle l_2, \{ x \mapsto X - 1 \}, X > 1 \rangle \]
\[ \downarrow \]
\[ \ldots \]
Partial evaluation

- Extension of standard execution for some unknown 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 unknown 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 unknown 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

\[
\begin{align*}
\text{power } x \ 0 & \quad = \quad 1 \\
\text{power } x \ 1 & \quad = \quad x \\
\text{power } x \ n & \quad = \quad x \times \text{power } x \ (n - 1)
\end{align*}
\]

\[
\begin{align*}
\text{power } x \ (n + 2) & \\
\downarrow & \\
x \times \text{power } x \ (n + 1) & \\
\{ n \mapsto 0 \} & \\
\downarrow & \\
x \times x & \\
\text{power } x \ (m + 1) & \\
\{ n \mapsto m + 1 \} & \\
\downarrow & \\
x \times x & \\
x \times x \times \text{power } x \ (m + 1) & \\
\end{align*}
\]

\[
\begin{align*}
\Rightarrow \\
\text{power}_{+2} \ x \ n & \quad = \quad x \times \text{power}_{+1} \ x \ n \\
\text{power}_{+1} \ x \ 0 & \quad = \quad x \times x \\
\text{power}_{+1} \ x \ n & \quad = \quad x \times x \times \text{power}_{+1} \ x \ (n - 1)
\end{align*}
\]
Partial evaluation: example

\[
\begin{align*}
power \times 0 &= 1 \\
power \times 1 &= x \\
power \times n &= x \times power \times (n - 1)
\end{align*}
\]

\[
\begin{array}{ccc}
\text{power} \times (n + 2) & \downarrow & \\
\times \times \text{power} \times (n + 1) & \downarrow & \\
\times \times \times \text{power} \times (m + 1) & \uparrow & \\
\end{array}
\]

\[
\begin{align*}
power_{+2} \times n &= x \times power_{+1} \times n \\
power_{+1} \times 0 &= x \times x \\
power_{+1} \times n &= x \times x \times power_{+1} \times (n - 1)
\end{align*}
\]

G Vidal (Valencia)
Partial evaluation: example

\[
\begin{align*}
\text{power } x 0 &= 1 \\
\text{power } x 1 &= x \\
\text{power } x n &= x \times \text{power } x (n-1)
\end{align*}
\]

\[
\begin{align*}
\text{power } x (n+2) \\
\downarrow \\
\times \times \text{power } x (n+1) \\
\{n \rightarrow 0\} \\
\downarrow \\
\times \times x \times \text{power } x (m+1) \\
\{n \rightarrow m+1\} \\
\end{align*}
\]

\[
\Rightarrow
\begin{align*}
\text{power}_{+2} x n &= \times \times \text{power}_{+1} x n \\
\text{power}_{+1} x 0 &= \times \times x \\
\text{power}_{+1} x n &= \times \times x \times \times \text{power}_{+1} x (n-1)
\end{align*}
\]
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

\[ l_0 : \ x := \text{input}(); \]
\[ l_1 : \ \text{while} \ x > 0 \ \text{do} \]
\[ l_2 : \ x := x - 1; \]
\[ l_3 : \ \text{done} \]

Terminating! (using AProVE)
Example

\[\begin{align*}
\text{l}_0 &: \quad x := \text{input}(); \\
\text{l}_1 &: \quad \text{while } x > 0 \text{ do} \\
\text{l}_2 &: \quad x := x - 1; \\
\text{l}_3 &: \quad \text{done}
\end{align*}\]

Terminating! (using AProVE)
Example

\[ l_0 : \ x \ := \ \text{input}(); \]
\[ l_1 : \ \text{while} \ x > 0 \ \text{do} \]
\[ l_2 : \ x \ := \ x - 1; \]
\[ l_3 : \ \text{done} \]

\[
\langle l_0, \{x \mapsto \bot\}, \text{true} \rangle \\
\downarrow \\
\langle l_1, \{x \mapsto X\}, \text{true} \rangle \\
\downarrow \\
\langle l_2, \{x \mapsto X\}, X > 0 \rangle \\
\downarrow \\
\langle l_1, \{x \mapsto X - 1\}, X > 0 \rangle \\
\downarrow \\
\langle l_3, \{x \mapsto X - 1\}, \neg(X > 0) \rangle \\
\downarrow \\
\langle l_3, \{x \mapsto X - 1\}, \neg(X > 1) \rangle
\]

\[
f_0 \ = \ f_1 \ x \\
f_1 \ x \ = \ \text{if} \ (x > 0) \ \text{then} \ f_3 \ x \ \text{else} \ f_4 \ x \\
f_4 \ x \ = \ f_5 \ (x - 1) \\
f_5 \ x \ = \ \text{if} \ (x > 0) \ \text{then} \ f_3 \ x \ \text{else} \ f_4 \ x
\]

Terminating! (using AProVE)
Example

\[ l_0 : \quad x := \text{input}(); \]
\[ l_1 : \quad \text{while } x > 0 \text{ do} \]
\[ l_2 : \quad x := x - 1; \]
\[ l_3 : \quad \text{done} \]

\[ \langle l_0, \{ x \mapsto \bot \}, \text{true} \rangle \]
\[ \downarrow \]
\[ \langle l_1, \{ x \mapsto X \}, \text{true} \rangle \]
\[ \downarrow \]
\[ \langle l_3, \{ x \mapsto X \}, \neg (X > 0) \rangle \quad \langle l_2, \{ x \mapsto X \}, X > 0 \rangle \]
\[ \downarrow \]
\[ \langle l_1, \{ x \mapsto X - 1 \}, X > 0 \rangle \]
\[ \downarrow \]
\[ \langle l_3, \{ x \mapsto X - 1 \}, \neg (X > 1) \rangle \quad \langle l_2, \{ x \mapsto X - 1 \}, X > 1 \rangle \]

\[ f_0 = f_1 x \]
\[ f_1 x = \text{if } (x > 0) \text{ then } f_3 x \text{ else } f_4 x \]
\[ f_4 x = f_5 (x - 1) \]
\[ f_5 x = \text{if } (x > 0) \text{ then } f_3 x \text{ else } f_4 x \]

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