Definitions as Pushdown Systems

Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

29/02/2012, 9th MC Meeting, Action IC0701, Darmstadt
1. Introduction to K

2. Definition of Boolean Programs

3. Definitions producing Push-down Systems

4. Model-checking the Push-down System

5. Conclusion
Introduction to \textit{K}

\textbf{2.} \textit{K} Definition of Boolean Programs

\textbf{3.} \textit{K} Definitions producing Push-down Systems

\textbf{4.} Model-checking the Push-down System

\textbf{5.} Conclusion
K Project

Started in 2003 by Grigore Roșu at UIUC, motivated mainly by teaching programming languages and noticing that the existing semantic frameworks have limitations.

Project thesis:
Rewriting gives an appropriate environment to formally define the semantics of real-life programming languages and to test and analyze programs written in those languages.

Joint work between Formal Systems Laboratory (FSL) from University of Illinois at Urbana-Champaign (UIUC) lead by Grigore Roșu and Formal Methods in Software Engineering (FMSE) from Al. I. Cuza University (UAIC) lead by presenter.

Main Web page: http://k-framework.org/
Motivation

- The Semantics of Programming languages is informally presented in manuals
- Each model checker, static verifier, run-time verifier of the same language L uses its own encoding of L
- Therefore programming languages must have formal semantics
- Executable specifications could help
  - Design and maintain mathematical definitions
  - Easily test/analyze language updates/extensions
  - Explore/Abstract non-deterministic executions
- one definition for L and develop the other tools w.r.t. this definition
Introduction to K

Roots are in Rewriting Semantics Project

Reduction Semantics with Evaluation Contexts

Small-Step SOS

Big-Step SOS

The Chemical Abstract Machine (CHAM)

Modular SOS

The K Semantic Framework

Rewrite Semantics Project

[J. Meseguer, G. Roşu, T. Şerbănuţă]
K at work

DEMO with online-interface

https://fmse.info.uaic.ro/tools/

Kompile imppp.k and Krun programs/div-nondet.imppp
K Ingredients

- **Computations**
  - Sequences of tasks
  - Capture the sequential fragment of programming languages
  - Syntax annotations specify order of evaluation

- **Configurations**
  - Multisets (bags) of nested cells
  - High potential for concurrency and modularity

- **K rules**
  - Specify only what needed, precisely identify what changes
  - More concise, modular, and concurrent than regular rewrite rule
Plan

1. Introduction to K
2. Definition of Boolean Programs
3. Definitions producing Push-down Systems
4. Model-checking the Push-down System
5. Conclusion
Running example: Boolean Programs (BP)

\[
Stmt ::= \text{skip}; \\
    | \text{return}; \\
    | \text{return Exp;} \ [\text{strict}] \\
    | \text{if( Decider ) then Stmts else Stmts endif} \ [\text{strict(1)}] \\
    | \text{while( Decider ) do Stmts endwhile} \ [\text{strict(1)}] \\
    | \#Id := Exp; \ [\text{strict(2)}] \\
    | \text{assert( Decider );} \\
    | \text{Exp ( Exp)s;} \ [\text{strict}] \\
    | \text{print( Exp)s;} \ [\text{strict}] \\
    | \text{goto \#Id;} \\
    | \text{decl Ids;} \\
    | \#Id ( Ids )begin Stmts end
\]

\[
Lstmt ::= Stmt
\]

\[
Exp ::= \#Bool \\
    | \#Id \\
    | Exp \mid Exp \ [\text{strict}] \\
    | Exp \& Exp \ [\text{strict}] \\
    | Exp \wedge Exp \ [\text{strict}] \\
    | Exp = Exp \ [\text{strict}] \\
    | Exp \neq Exp \ [\text{strict}] \\
    | Exp \rightarrow Exp \ [\text{strict(1)}] \\
    | ! Exp \ [\text{strict}] \\
\]

Decider ::= ? \\
    | Exp
BP Configuration

$PGM \xrightleftharpoons{} \text{execute} \quad \text{fstack} \quad \text{env} \quad \text{genv} \quad \text{store} \quad \text{nextLoc}$

$\bullet$ $\bullet$ $\bullet$ $\bullet$ $0$

Definitions as Pushdown Systems
BP Rules: assignment and nondeterministic operator

**Assignment Rule**

\[ X := V ; X \mapsto L \]

**Store Rule**

\[ L \mapsto K \mapsto V \]

**Nondeterministic Operator Rule**

\[ ? \sim K \sim C \Rightarrow \text{true} \sim K \sim C \]

\[ ? \sim K \sim C \Rightarrow \text{false} \sim K \sim C \]
Definition of Boolean Programs

BP Rules: if-then-else-endif

RULE

if(true) then $S_s$ else $S_s'$ endif

$S_s$

RULE

if(false) then $S_s$ else $S_s'$ endif

$S_s'$
Definition of Boolean Programs

BP Rules: function call

RULE

\[
(F(Xs)_S, Vs) ; \rightsquigarrow K \\
\text{bindto} (Xs, Vs) \rightsquigarrow Ss \rightsquigarrow \text{return} ;
\]

genv

GEnv

fstack

\[
(Env, K)
\]

env

GEnv
BP Rules: function return

\[ k \quad \text{return } V ; \mathcal{R} K' \quad \Rightarrow \quad V \mathcal{R} K \]

\[ \text{fstack} \quad (\text{Env}, K) \quad \Rightarrow \quad \cdot \quad \Rightarrow \quad \text{Env} \]
Plan

1. Introduction to $\mathbb{K}$

2. $\mathbb{K}$ Definition of Boolean Programs

3. $\mathbb{K}$ Definitions producing Push-down Systems

4. Model-checking the Push-down System

5. Conclusion
PDS-based Analysis

a real programming language $\mathcal{L}$ (like C)

1st abstraction

an abstraction of $\mathcal{L}$ (boolean programs)

2nd abstraction

push-down systems
Main idea

- a ground $\mathcal{K}$ configuration is abstracted into a PDS configuration.
  - PDS configurations are very easy to express in $\mathcal{K}$.

- $\mathcal{K}$ rules (a part of them) are adjusted to "produce" PDS rules when they are applied on a given program.

- The execution of the program supplies a PDS that is an "abstraction" of the program behavior.

- The produced PDS can be used to check properties of the program.
configuration ⇒ PDS Configuration

Notation: \( \text{state}(a \mapsto \ell_a \ b \mapsto \ell_b, \ell_a \mapsto v_a \ \ell_b \mapsto v_b) = a \mapsto v_a \ b \mapsto v_b \)
BP-PDS: configuration
rules $\Rightarrow$ PDS rules: assignment

\[
\frac{k}{X := V} \quad \frac{env}{X \mapsto \ell Env} \quad \frac{store}{\ell \mapsto St} \quad \frac{fstack}{(F, _, _)}
\]

\[
\langle currentLocation, F \rangle \leftrightarrow \langle X \mapsto V \ state(Env, St), F \rangle
\]

\[
X \mapsto V \ state(Env, St) \text{ becomes the new currentLocation}
\]
BP-PDS: assignment

\[ X := V ; \]
\[ X \mapsto L \ Env \]
\[ L \mapsto K \ Store \]
\[ L \mapsto V \]
\[ (F, \_ , \_ ) \]

\[ \text{state}( X \mapsto L \ Env , L \mapsto V \ Store ) \]

\[ \text{pds} < \text{Loc} , F > \rightarrow < \text{state}( X \mapsto L \ Env , L \mapsto V \ Store ) , F >; \]
K rules $\Rightarrow$ PDS rules: function call

\[
(F(Xs).Ss)(Vs) \leadsto K
\]

\[
\langle \text{currentLocation}, F' \rangle \leftrightarrow \langle \text{state}(GEnv, St), F F' \rangle
\]

\text{state}(GEnv, St) \text{ becomes the new currentLocation}
BP-PDS: function call

\[
(F(Xs) . Ss) (Vs); \sim K \\
\textbf{bindto}(Xs, Vs) \sim Ss \sim \text{return}; \\
\]

\[
(F(Xs) . Ss) (Vs); \sim K \\
\textbf{bindto}(Xs, Vs) \sim Ss \sim \text{return}; \\
\]

\[
(\textbf{fstack})(F^{'}, Env^{'}, K^{'}) \\
(F, Env, K) (F^{'}, Env^{'}, K^{'}) \\
\]

\[
\text{env} \\
\text{GEnv} \\
\text{store} \\
\text{Store} \\
\]

\[
\text{lastPdsLoc} \\
\text{Loc} \\
\text{env} \\
\text{GEnv} \\
\text{store} \\
\text{Store} \\
\]

\[
\text{pds} \\
\text{recDepth} \\
D \\
D + \text{Int} 1 \\
\]

\[
\text{maxDepth} \\
\text{Max} \\
D < \text{Int} \text{ Max} \\
\]
\[ \text{K rules } \Rightarrow \text{ PDS rules: function return} \]

\[ \begin{align*}
\text{return } V; & \sim K' \\
V & \sim K
\end{align*} \]

\[ \begin{align*}
\text{env} & \sim \text{Env} \\
\text{store} & \sim \text{St}
\end{align*} \]

\[ (F, \text{Env}, K) \]

\[ \langle \text{currentLocation}, F \rangle \leftrightarrow \langle \text{state(Env, St)}, \varepsilon \rangle \]

\textit{state(Env, St) becomes the new currentLocation}
BP-PDS: function return

\[
\frac{\text{return } V ; K'}{V \bowtie K'}
\]

\[
\frac{\text{fstack} (F, Env, K)}{Env}
\]

\[
\frac{\text{env}}{\text{store Store}}
\]

\[
\frac{\text{lastPdsLoc}}{\text{Loc}}
\]

\[
\frac{\text{state} (Env, Store)}{\text{recDepth}}
\]

\[
\frac{D}{D \rightarrow^* \text{Int 1}}
\]

\[
\text{pds}
\]

\[
< \text{Loc, } F > \rightarrow < \text{state} (Env, Store), \text{eps} >;
\]
Plan

1. Introduction to $\mathbb{K}$

2. $\mathbb{K}$ Definition of Boolean Programs

3. $\mathbb{K}$ Definitions producing Push-down Systems

4. Model-checking the Push-down System

5. Conclusion
Example of BP program

s() begin
    if (?)
        then return;
    else s(); x := ! x;
    endif
end

decl x;

main() begin
    x := false;
    s();
end
BP-PDS at work

$krun --k-definition=bp-pds.k --DEPTH="ListItem(100)" recp.bp
<T>
  <path>
    <k>
      s :: lambda( , if( ? )then return nothing ; else s ( ); x := ! x ;
    </k>
    ...
  </path>
  <pds>
    < ., "bot" > -> < "x" |-> "undefined", "main" "bot" > ;
    < "x" |-> "undefined", "main" > -> < "x" |-> false, "main" > ;
    < "x" |-> false, "main" > -> < "x" |-> false, "s" "main" > ;
    ...
  </pds>
</T>
The result PDS can be model-checked

- model-checking problem for (finite) PDS is decidable
- the existing model-checkers, e.g., Moped, cannot be used directly because the PDS location are structured (include information about memory)
- we wrote a prototype in Maude of the model-checking algorithm for PDS (version with reachability post-automaton)
- pros: more flexibility in defining atomic propositions
- cons: less efficient due to implementation at equational level (no BDD or other optimizations)
Running our model-checker: step 1

- produce the PDS as a Maude module
- define the satisfaction predicate for atomic state proposition

```
$ ./createpds bp-pds.k recp.bp
Saved in recp-pds.maude.
$less recp-pds.maude
load pdspostaut.maude
mod PDS is including PDSMC + STRING .
  ...
  op pds : -> SetRule .
  eq pds =
  *** load rules here
  < .,"bot" > -> < "x" |-> "undefined" ,"main" "bot" > ;
  ...
  ops px : -> Prop .
  eq (("x" |-> true) M:Map, A:Alph) |= px = true .
  ...
endm
```
Running our model-checker: step 2

- produce BA for (negated) LTL formula as a Maude module

```
$ ./createltl recp-pds.maude "[] px"
Saved in recp-ltl.maude
$less recp-ltl.maude
load recp-pds.maude
mod BA-MAUDE is including PDS .
   --- LTL formula: [] px
   op baut : -> SetTrans .
exeq baut =
      (accept@init, px, accept@init) ;
      emptyTrSet .

   op accept@init : -> State .
exeq isAcc(accept@init) = true .
endm
```
Running our model-checker: step 3

- Launch Maude over the generated module
- apply the model-checker algorithm over the initial configuration

```$maude recp-ltl.maude
Maude> red pdsModelCheck(pds, baut, (., accept@init, "bot")).
reduce in BA-MAUDE : pdsModelCheck(pds, baut, (.,accept@init,"bot")).
rewrites: 106 in 0ms cpu (8ms real) (1060000 rewrites/second)
result SetConfig: (true).SetConfig```
Plan

1. Introduction to K

2. K Definition of Boolean Programs

3. K Definitions producing Push-down Systems

4. Model-checking the Push-down System

5. Conclusion
Conclusion

- $\mathbb{K}$ is suitable for defining language abstractions
  - this is exhibited by the $\mathbb{K}$ definition of boolean programs
- $\mathbb{K}$ definition can be refined to produce further abstractions
  - we showed how the semantics of Boolean programs can produce push-down systems
- the final abstractions can be checked with specialized algorithms
  - we wrote a Maude prototype for model-checking PDSs
- this is a really WORK IN PROGRESS!!!
- future work: define the PDS model-checker directly in $\mathbb{K}$
Questions?

Thanks!