$\mathbb{K}$ 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

Plan

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.

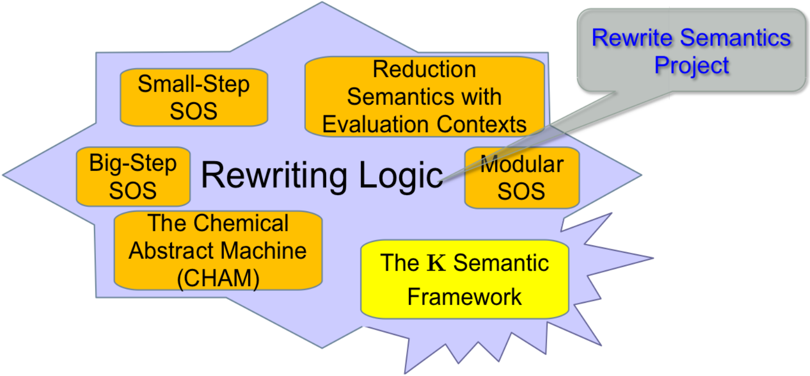
Joint work between Formal Systems Laboratory (FSL) from University of Illinois at Urbana-Champaign (UIUC) lead by Grigore Roșu and Formal Methods in Software Engineering (FMSE) from Al. I. Cuza University (UAIC) lead by presenter

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

$\mathbb{K}$ Roots are in Rewriting Semantics Project



[J. Meseguer, G. Roșu, T. Șerbănuță]

$\mathbb{K}$ at work

DEMO with online-interface

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

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

$\mathbb{K}$ 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
- $\mathbb{K}$ rules
  - Specify only what needed, precisely identify what changes
  - More concise, modular, and concurrent than regular rewrite rule

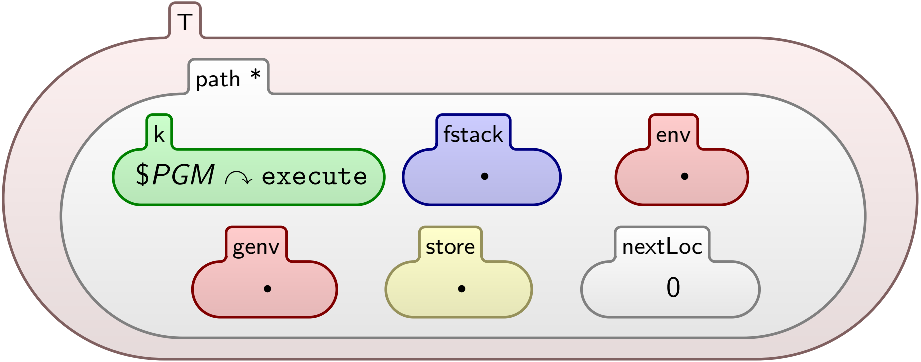
Plan

Running example: Boolean Programs (BP)

```
Exp ::= #Bool
      | #Id
      | Exp | Exp [strict]
      | Exp & Exp [strict]
      | Exp ^ Exp [strict]
      | Exp = Exp [strict]
      | Exp != Exp [strict]
      | Exp -> Exp [strict(1)]
      | ! Exp [strict]
Decider ::= ?
          | Exp
```

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

BP Configuration



BP Rules: assignment and nondeterministic operator

assignment

RULE $\langle \dfrac{X := V\ ;}{\bullet} \cdots \rangle_{\mathsf{k}}$ $\langle \cdots X \mapsto L \cdots \rangle_{\mathsf{env}}$ $\langle \cdots L \mapsto \dfrac{K}{V} \cdots \rangle_{\mathsf{store}}$

nondeterministic operator

RULE $\langle \langle ? \curvearrowright K \rangle_{\mathsf{k}}\ C \rangle_{\mathsf{path}} \Rightarrow \langle \langle \mathsf{true} \curvearrowright K \rangle_{\mathsf{k}}\ C \rangle_{\mathsf{path}}\ \langle \langle \mathsf{false} \curvearrowright K \rangle_{\mathsf{k}}\ C \rangle_{\mathsf{path}}$

BP Rules: if-then-else-endif

RULE k: $\dfrac{\texttt{if}(\ \mathsf{true}\ )\texttt{then}\ Ss\ \texttt{else}\ Ss'\ \texttt{endif}}{Ss}$

RULE k: $\dfrac{\texttt{if}(\ \mathsf{false}\ )\texttt{then}\ Ss\ \texttt{else}\ Ss'\ \texttt{endif}}{Ss'}$

BP Rules: function call

RULE

k: $$\frac{(F(Xs)\,.\,Ss)\ (\ Vs\ );\ \curvearrowright K}{\texttt{bindto(}\ Xs\ ,\ Vs\ \texttt{)}\ \curvearrowright Ss \curvearrowright \texttt{return;}}$$

fstack: $$\frac{\bullet}{(\ Env\ ,\ K\ )}$$

env: $$\frac{Env}{GEnv}$$

genv: $$GEnv$$

BP Rules: function return

RULE

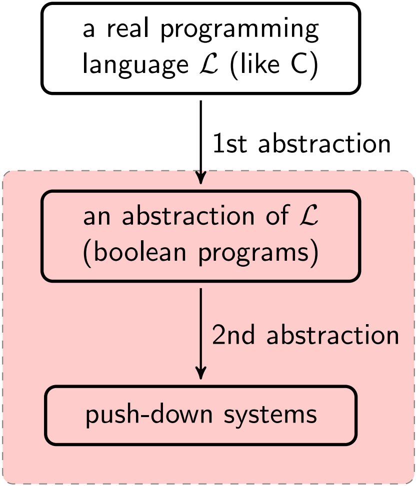
k: $\dfrac{\texttt{return}\ V\ ;\ \curvearrowright K'}{V \curvearrowright K}$

fstack: $\dfrac{(\ Env\ ,\ K\ )}{\bullet}$

env: $\dfrac{\text{—}}{Env}$

Plan

PDS-based Analysis

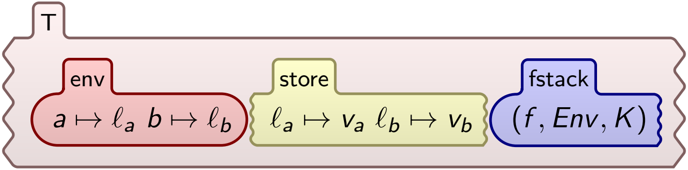


𝕂 Definitions as Push-down Specifications

Main idea

- a ground 𝕂 configuration is abstracted into a PDS configuration
    - PDS configurations are very easy to express in 𝕂
- 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

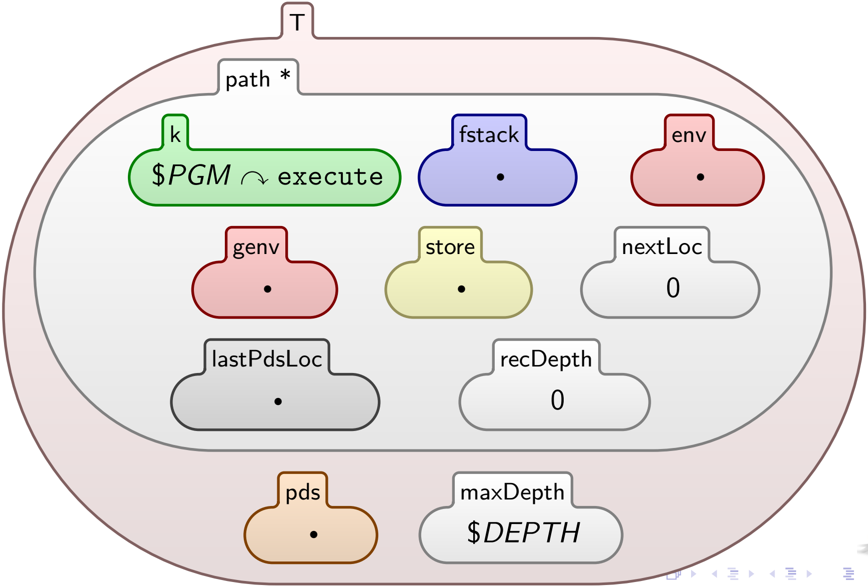
𝕂 configuration ⇒ PDS Configuration



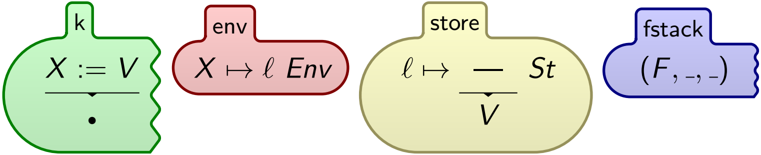
$$\langle a \mapsto v_a \; b \mapsto v_b, f \rangle$$

Notation: $state(a \mapsto \ell_a \; b \mapsto \ell_b, \ell_a \mapsto v_a \; \ell_b \mapsto v_b) = a \mapsto v_a \; b \mapsto v_b$

BP-PDS: configuration



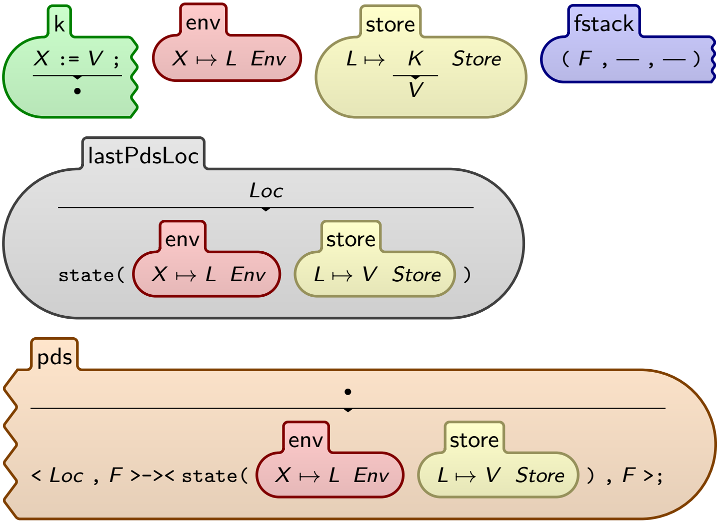
$\mathbb{K}$ rules $\Rightarrow$ PDS rules: assignment



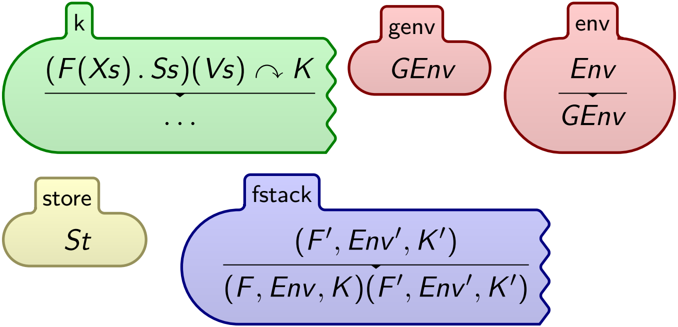
$$\langle currentLocation, F\rangle \hookrightarrow \langle X \mapsto V\ state(Env, St), F\rangle$$

$X \mapsto V\ state(Env, St)$ becomes the new currentLocation

BP-PDS: assignment



$\mathbb{K}$ rules $\Rightarrow$ PDS rules: function call



$$\langle currentLocation, F' \rangle \hookrightarrow \langle state(GEnv, St), F\ \ F' \rangle$$

$state(GEnv, St)$ becomes the new currentLocation

BP-PDS: function call

k
$$\frac{(F(Xs)\,.\,Ss)\ (\ Vs\ );\ \curvearrowright K}{\texttt{bindto(}\ Xs\ ,\ Vs\ \texttt{)}\ \curvearrowright Ss \curvearrowright \texttt{return;}}$$

fstack
$$\frac{(\ F'\ ,\ Env'\ ,\ K'\ )}{(\ F\ ,\ Env\ ,\ K\ )\ \ (\ F'\ ,\ Env'\ ,\ K'\ )}$$

env
$$\frac{Env}{GEnv}$$

genv $GEnv$

store $Store$

lastPdsLoc
$$\frac{Loc}{\texttt{state(}\ \text{env: } GEnv\ \ \text{store: } Store\ \texttt{)}}$$

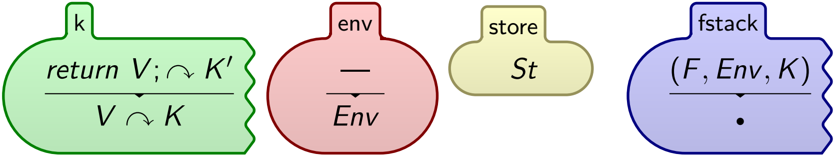
pds
$$\frac{\bullet}{<\ Loc\ ,\ F'\ \texttt{>-><}\ \texttt{state(}\ \text{env: } GEnv\ \ \text{store: } Store\ \texttt{)}\ ,\ F \curvearrowright F'\ \texttt{>;}}$$

recDepth
$$\frac{D}{D +_{Int} 1}$$

maxDepth Max

$D <_{Int} Max$

$\mathbb{K}$ rules $\Rightarrow$ PDS rules: function return



$$\langle currentLocation, F\rangle \hookrightarrow \langle state(Env, St), \varepsilon\rangle$$

$state(Env, St)$ becomes the new currentLocation

BP-PDS: function return

k
$$\frac{\texttt{return}\ V\ ;\ \curvearrowright K'}{V \curvearrowright K}$$

fstack
$$\frac{(\ F\ ,\ Env\ ,\ K\ )}{\bullet}$$

env
$$\frac{—}{Env}$$

store
$$Store$$

lastPdsLoc
$$\frac{Loc}{\texttt{state(}\ \text{env: } Env\ \ \text{store: } Store\ \texttt{)}}$$

recDepth
$$\frac{D}{D\ -_{Int}\ 1}$$

pds
$$\frac{\bullet}{<\ Loc\ ,\ F\ \texttt{>-><}\ \texttt{state(}\ \text{env: } Env\ \ \text{store: } Store\ \texttt{)}\ ,\ \texttt{eps}\ \texttt{>;}}$$

Plan

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 .
  eq baut =
    (accept@init, px, accept@init) ;
     emptyTrSet .

  op accept@init : -> State .
  eq 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

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!