Contribution to $\mathbb{K}$ Framework

Dorel Lucanu

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

22/06/2011, 8th MC Meeting, Action IC0701, Limerick
1. Introduction

2. K Framework
   - Motivation
   - K Framework Solution
   - Matching Logic

3. Contributions

4. Conclusion
Plan

1. Introduction

2. K Framework
   - Motivation
   - K Framework Solution
   - Matching Logic

3. Contributions

4. Conclusion
K Project

Started in 2003 by Grigore Rosu 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

UIUC team:
Chucky Ellison, Michael Ilseman, Patrick Meredith, Grigore Roșu, Traian Șerbănuță, Andrei Ștefănescu, David Lazar

UAIC team:
Andrei Arusoae, Irina Mariuca Asavoae, Mihai Asavoae, Gheorghe Grigoras, Dorel Lucanu, Radu Mereuta, Elena Naum
Plan

1. Introduction

2. K Framework
   - Motivation
   - K Framework Solution
   - Matching Logic

3. Contributions

4. Conclusion
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
Shortcomings of Existing Frameworks

- Hard to deal with control (except evaluation contexts)
  - halt, break/continue, exceptions
- Non-modular (except Modular SOS)
  - Adding new features require changing unrelated rules
- Lack of semantics for true concurrency (except CHAM)
  - Big-Step captures only all possible results of computation
    - Reduction approaches only give interleaving semantics
- Tedious to find next redex (except evaluation contexts)
  - One has to write the same descent rules for each construct
- Inefficient as interpreters (except for Big-Step SOS)
Roots are in Rewriting Semantics Project

Rewriting Logic

Small-Step SOS
Big-Step SOS
The Chemical Abstract Machine (CHAM)
Reduction Semantics with Evaluation Contexts
Modular SOS

Rewrite Semantics Project

[J. Meseguer, G. Roșu, T. Șerbănuță]
Distinguishable K Features

K technique:
for expressive, modular, versatile, and clear PL definitions

K rewriting:
more concurrent than regular rewriting

Representable (e.g., in RWL) for execution, testing and analysis purposes
Ingredients

- **Computations**
  - Sequences of tasks, including syntax
  - 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
the semantics is given by means of a set of K rules transforming the abstract syntax trees (ASTs) into results, eventually using some intermediate structures.

- the notion of result is a generic one: could be either the output, the result of a type-checking algorithm, the result of a static analyser/verifier and so on.

- the machine on which the programs are executed is abstractly described as a configuration of cells.

- K Rewrite Abstract Machine (KRAM) executes the rewrite rules in faithful way.
Running example: Cink

a kernel of C
- functions
- int expressions
- simple input/output
- basic flow control (if, if-else, while, sequential composition)
- pointers and arrays
- structures

in this talk
- a K semantic definition of Cink (without pointers and structures)
**K computations and K syntax**

### Computations
- Extend PL syntax with a “task sequentialization” operation
  - $t_1 \bowtie t_2 \bowtie \ldots \bowtie t_n$, where $t_i$ are computational tasks
- Computational tasks: pieces of syntax (with holes), closures, ...
- Mostly under the hood, via intuitive PL syntax annotations

### K Syntax: BNF syntax annotated with strictness

**Exp** ::=
- $Id$
- $Exp + Exp$ [strict] $ERed + ERed \iff ERed \bowtie \Box + ERed$
- $Exp = Exp$ [strict(2)] $E = ERed \iff ERed \bowtie E = \Box$
- $Stmt ::= Exp ;$ [strict] $ERed ; \iff ERed \bowtie \Box ;$
- $Stmt Stmt$ [seqstrict] $SRed S \iff SRed \bowtie \Box S$
Heating syntax through strictness rules

Computation

\[ y = x + 2 \; ; \; x = 7 \; ; \]

**K Syntax: BNF syntax annotated with strictness**

**Exp** ::= \( Id \)
- \( Exp + Exp \) \[ strict \]
- \( Exp = Exp \) \[ strict(2) \]

**Stmt** ::= \( Exp ; \) \[ strict \]
- \( Stmt \; Stmt \) \[ seqstrict \]

**ERed** + **ERed** \( \Leftrightarrow \) **ERed** \( \bowtie \Box + \) **ERed**

**E** = **ERed** \( \Leftrightarrow \) **ERed** \( \bowtie \Box \) **E** = \( \Box \)

**ERed** ; \( \Leftrightarrow \) **ERed** \( \bowtie \Box \) ;

**SRed** **S** \( \Leftrightarrow \) **SRed** \( \bowtie \Box \) **S**

D. Lucanu (UAIC)
Heating syntax through strictness rules

Computation

\[ y = x + 2 ; \ \bowtie \Box \ x = 7 ; \]

\textbf{K Syntax: BNF syntax annotated with strictness}

\begin{align*}
\text{Exp} & ::= \text{Id} \\
| \text{Exp} + \text{Exp} & \quad [\text{strict}] \\
| \text{Exp} = \text{Exp} & \quad [\text{strict}(2)] \\
\text{Stmt} & ::= \text{Exp} ; \\
| \text{Stmt} \text{ Stmt} & \quad [\text{seqstrict}] \\
\end{align*}

\begin{align*}
\text{ERed} + \text{ERed} & \iff \text{ERed} \bowtie \Box + \text{ERed} \\
E = \text{ERed} & \iff \text{ERed} \bowtie E = \Box \\
\text{ERed} ; & \iff \text{ERed} \bowtie \Box ; \\
\text{SRed} S & \iff \text{SRed} \bowtie \Box S
\end{align*}
Heating syntax through strictness rules

Computation

\[ y = x + 2 \xrightarrow{\square}; \xrightarrow{\square} x = 7; \]

K Syntax: BNF syntax annotated with strictness

\[
\begin{align*}
\text{Exp} &::= \text{Id} \\
| \quad \text{Exp} + \text{Exp} &\quad \text{[strict]} \\
| \quad \text{Exp} = \text{Exp} &\quad \text{[strict(2)]} \\
\text{Stmt} &::= \text{Exp} ; \\
| \quad \text{Stmt} \text{ Stmt} &\quad \text{[seqstrict]}
\end{align*}
\]

\[
\begin{align*}
\text{ERed} + \text{ERed} &\iff \text{ERed} \xrightarrow{\square} \square + \text{ERed} \\
\text{E} = \text{ERed} &\iff \text{ERed} \xrightarrow{\square} \text{E} = \square \\
\text{ERed} ; &\iff \text{ERed} \xrightarrow{\square} \square ; \\
\text{SRed} S &\iff \text{SRed} \xrightarrow{\square} \square S
\end{align*}
\]
Heating syntax through strictness rules

Computation

\[ x + 2 \leadsto y = \Box \leadsto \Box; \leadsto \Box x = 7; \]

*K Syntax: BNF syntax annotated with strictness*

\[
\begin{align*}
Exp & ::= \text{Id} \\
| & Exp + Exp \quad \text{[strict]} \\
| & Exp = Exp \quad \text{[strict(2)]} \\
Stmt & ::= Exp ; \quad \text{[strict]} \\
| & Stmt Stmt \quad \text{[seqstrict]}
\end{align*}
\]

\[
\begin{align*}
ERed + ERed & \iff ERed \leadsto \Box + ERed \\
E & = ERed \iff ERed \leadsto E = \Box \\
ERed ; & \iff ERed \leadsto \Box ; \\
SRed S & \iff SRed \leadsto \Box S
\end{align*}
\]
Heating syntax through strictness rules

Computation

\[ x \rightsquigarrow \Box + 2 \rightsquigarrow y = \Box \rightsquigarrow \Box; \rightsquigarrow \Box \ x = 7 \ ; \]

 Luft Syntax: BNF syntax annotated with strictness

\[
\begin{align*}
\text{Exp} &::= \text{Id} \\
| \quad \text{Exp} + \text{Exp} &\quad \text{[strict]} \quad \text{ERed} + \text{ERed} \quad \Leftarrow \quad \text{ERed} \rightsquigarrow \Box + \text{ERed} \\
| \quad \text{Exp} = \text{Exp} &\quad \text{[strict(2)]} \quad E = \text{ERed} \quad \Leftarrow \quad \text{ERed} \rightsquigarrow E = \Box \\
\text{Stmt} &::= \text{Exp} ; \\
| \quad \text{Stmt} \text{ Stmt} &\quad \text{[seqstrict]} \quad \text{ERed} ; \quad \Leftarrow \quad \text{ERed} \rightsquigarrow \Box ; \\
S &::= \text{Stmt} S \\
| \quad \text{SRed} &\quad \text{[seqstrict]} \quad S \text{Red} S \quad \Leftarrow \quad \text{SRed} \rightsquigarrow \Box S
\end{align*}
\]
Cink Configuration

\[ \begin{align*}
  y \leadsto x &= \Box + 1 \leadsto \ldots \\
  x \mapsto 2 &\quad y \mapsto 5 \\
  f \mapsto \text{int } f() \{ \ldots \}
\end{align*} \]
Cink (with pointers and arrays) Configuration
C Configuration
K rules: expressing natural language into rules

**Reading from environment**

If a local variable $X$ is the next thing to be processed . . .

. . . and if $X$ is mapped to a value $V$ in the environment . . .

. . . then process $X$, replacing it by $V$

\[
\begin{aligned}
  & k \\
  & x \dashv \Box + 2 \vdash y = \Box \vdash \Box ; \vdash \Box x = 7 \\
  & \text{env} \\
  & t \mapsto 7 \quad x \mapsto 3 \quad y \mapsto 4 \quad y \mapsto 8
\end{aligned}
\]
**K rules: expressing natural language into rules**

Focusing on the relevant part

**Reading from environment**

If a local variable $X$ is the next thing to be processed . . .

. . . and if $X$ is mapped to a value $V$ in the environment . . .

. . . then process $X$, replacing it by $V$
**K rules: expressing natural language into rules**

Unnecessary parts of the cells are abstracted away

Reading from environment

If a local variable $X$ is the next thing to be processed . . .

. . . and if $X$ is mapped to a value $V$ in the environment . . .

. . . then process $X$, replacing it by $V$
**K rules: expressing natural language into rules**

Underlining what to replace, writing the replacement under the line

**Reading from environment**

If a local variable $X$ is the next thing to be processed . . .

. . . and if $X$ is mapped to a value $V$ in the environment . . .

. . . then process $X$, replacing it by $V$
K rules: expressing natural language into rules

Configuration Abstraction: Keep only the relevant cells

Reading from environment

If a local variable \( X \) is the next thing to be processed . . .

. . . and if \( X \) is mapped to a value \( V \) in the environment . . .

. . . then process \( X \), replacing it by \( V \)
**K** rules: expressing natural language into rules

Generalize the concrete instance

**Reading from environment**

If a local variable $X$ is the next thing to be processed . . .

. . . and if $X$ is mapped to a value $V$ in the environment . . .

. . . then process $X$, replacing it by $V$
K rules: expressing natural language into rules

Voilà!

Reading from environment

If a local variable $X$ is the next thing to be processed . . .

. . . and if $X$ is mapped to a value $V$ in the environment . . .

. . . then process $X$, replacing it by $V$

\[
\begin{align*}
\text{graphic} & \quad \text{ASCII} \\
\text{rule } & <k>X \mapsto V<env> \text{ } X \mapsto V<env>
\end{align*}
\]
Rules at Work

\[
\frac{X}{V} \quad \xrightarrow{k} \quad X \mapsto V
\]

\[
\begin{align*}
T & \mapsto 7 \\
X & \mapsto 3 \\
y & \mapsto 4 \\
y & \mapsto 8 \\
\end{align*}
\]
Rules at Work

\[
\begin{array}{c}
X & \mapsto & V \\
\text{env} & \mapsto & V
\end{array}
\]

\[
\begin{align*}
3 & \mapsto \Box + 2 \mapsto y = \Box \mapsto \Box ; \mapsto \Box x = 7 \\
\text{env} & \mapsto t \mapsto 7 \mapsto x \mapsto 3 \mapsto y \mapsto 4 \mapsto y \mapsto 8
\end{align*}
\]
Rules at Work

\[ ERed + ERed \iff ERed \times \Box \; + \; ERed \]
Rules at Work

\[ ERed + ERed \iff ERed \bowtie \square + ERed \]
Rules at Work

\[ l_1 + l_2 \rightarrow l_1 + \text{Int } l_2 \]
Rules at Work

$I_1 + I_2 \rightarrow I_1 + \text{Int } I_2$

$$\begin{align*}
5 \vdash y &= \square \vdash \square ; \square \vdash x = 7 \\
\text{env} \quad t &\mapsto 7 \quad x &\mapsto 3 \quad y &\mapsto 4 \quad y &\mapsto 8
\end{align*}$$
Rules at Work

\[ E = E\text{Red} \quad \Rightarrow \quad E\text{Red} \rightsquigarrow E = \square \]
Rules at Work

\[ E = E_{\text{Red}} \implies E_{\text{Red}} \sim E = \square \]
Rules at Work

$$X = 1\quad \rightarrow\quad 7$$

$$y = 5 \quad ; \quad x = 7$$

$$t \rightarrow 7 \quad x \rightarrow 3 \quad y \rightarrow 4 \quad y \rightarrow 8$$
Rules at Work

\[ k \xrightarrow{X = 1} 1 \]

\[ env \quad X \xrightarrow{\cdot} \cdot \]

\[ k \xrightarrow{5 \xrightarrow{\cdot} \cdot ; \cdot \cdot \cdot x = 7} \]

\[ env \quad t \xrightarrow{7} \quad x \xrightarrow{3} \quad y \xrightarrow{5} \quad y \xrightarrow{8} \]
Program Logic in $\mathbb{K}$

An ideal PL Framework should serve as a program logic.
We can prove program correctness in $\mathbb{K}$ using Matching Logic (ML).

Matching Logic $\equiv \mathbb{K} + \text{FOL}$ [G. Roșu, W. Schulte, C. Ellyson, A, Ștefănescu]

- Formulae $\equiv \text{FOL}$ over configurations, called patterns
  (Configurations are allowed to contain variables)
- Models $\equiv$ Ground configurations
- Satisfaction $\equiv$ Matching for configurations, plus FOL for the rest

$\text{MatchC} \equiv$ Matching Logic for a (kernel of) C
MatchC: Example of Annotated Program

```c
struct ListNode* reverse(struct ListNode *x)
/*@ rule <k> $ => return p1; </k>
   <heap_> list(x)(A) => list(p1)(rev(A)) <_/heap> */
{
    struct ListNode *p;
    p = 0;
   /*@ inv <heap_> list(p)(?B), list(x)(?C) <_/heap>
    \ A = rev(?B) @ ?C*/
    while(x) {
        struct ListNode *y;
        y = x->next;
        x->next = p;
        p = x;
        x = y;
    }
    return p;
}
```

More examples on http://fsl.cs.uiuc.edu/index.php/Special:MLOnline

D. Lucanu (UAIC)
MatchC: Example of Annotated Program

```c
void readWrite(int n)
/*@ rule <k> $ => return; </k>
    <in> A => epsilon <_/in>
    <out_> epsilon => A </out>
    if n = len(A) */
{
   /*@ inv <in> ?B <_/in> <out_> ?A </out>
        ∧ n >= 0 ∧ len(?B) = n ∧ A = ?A @ ?B */
    while (n) {
        int t;

        scanf("%d", &t);
        printf("%d ", t);
        n -= 1;
    }
}
```

More examples on http://fsl.cs.uiuc.edu/index.php/Special:MLOnline
Plan

1. Introduction

2. K Framework
   - Motivation
   - K Framework Solution
   - Matching Logic

3. K ontributions

4. Conclusion
• Parsing (Radu Mereuță)
  a K definition uses both K syntax and PL syntax, hence a lot of ambiguities

• Contextual Transformers (Andrei Arusoaie)
  mapping rules on configurations according to Locality Principle

• Abstract Model Checking (Irina M. Asăvoae)
  collecting semantics, symbolic execution, model checking
  model-checking the object creation -based properties (joint work with F. de Boer, M. Bonsangue, J. Rot – CWI, LIACS)
Worst Case Execution Time Analysis for embedded systems (Mihai Asăvoae)

K semantics for hardware languages (SSRISC), abstraction and symbolic execution

Modeling and Metamodeling Languages (joint work with Vlad Rusu, INRIA Lille)

Automatic Instantiation of the Patterns from Annotations (Elena naum)
Plan

1. Introduction

2. K Framework
   - Motivation
   - K Framework Solution
   - Matching Logic

3. Kontributions

4. Conclusion
Formal semantics is useful and practical!
One can use an executable semantics of a language as is also for program verification
Giving a formal semantics is not necessarily painful, it can be fun if one uses the right tools
K Framework is scalable (C, Scheme, Verilog, Java etc)