\( \textbf{K}: \) A Rewriting Approach to Concurrent Programming Language Design and Semantics

—PhD Thesis Defense—

Traian Florin Șerbănuță

University of Illinois at Urbana-Champaign

Thesis advisor: Grigore Roșu
Committee members: Thomas Ball
Darko Marinov
José Meseguer
Madhusudan Parthasarathy
Rewriting is a natural environment to formally define the semantics of real-life concurrent programming languages and to test and analyze programs written in those languages.
Motivation: pervasive computing
Challenges in PL design and analysis

PLs need to be designed, updated, and extended
- C# and CIL; new Java memory model, Scheme R6RS, C1X
- Concurrency must become the norm

“External” non-determinism makes traditional testing difficult
- Concurrency and communication (scheduler specific)
- Under-specification for optimization purposes (compiler specific)

Executable formal definitions can help
- Design and maintain mathematical definitions of languages
- Easily test and analyze language updates or extensions
- Explore and/or abstract nondeterministic executions
Outline and Contributions

This dissertation re-affirms

1. Rewriting logic (RWL) as a powerful meta-logical framework for PL
   - Executable, with generic and efficient tool support

This dissertation proposes

2. K: the most comprehensive PL definitional framework based on RWL
   - Expressive, concurrent, modular, intuitive
3. A true concurrency with resource sharing semantics for K
4. K-Maude as a tool mechanizing the representation of K in RWL
   - Execute, explore, analyze K definitions

Demo: exploring concurrency in K-Maude

- Defining dataraces and verifying datarace freeness
- Experimenting with relaxed memory models (x86-TSO)
My research

Rewriting & Programming languages

Specifying and verifying concurrency
2010: J.LAP; 2008: ICSE, WMC.

Foundations

Collaborators
Feng Chen, Camelia Chira, Chucky Ellison, Regina Frei, Mark Hills, Giovanna Di Marzo Serugendo, José Meseguer, Andrei Popescu, Grigore Roșu, Wolfram Schulte, Virgil Nicolae Șerbănuță, Gheorghe Ștefănescu.
Rewriting logic semantics project

*Meseguer, Roșu, 2004, 2006, 2007*

**Goal**

Advance the use of rewriting logic for defining programming languages, and for executing and analyzing programs written in them.

**Some people involved in the Rewriting Logic Semantics Project**

Wolfgang Ahrendt, Musab Al-Turki, Marcelo d’Amorim, Irina M. Asăvoae, Mihai Asăvoae, Eyvind W. Axelsen, Christiano Braga, Illiano Cervesato, Fabricio Chalub, Feng Chen, Manuel Clavel, Chucky Ellison, Azadeh Farzan, Alejandra Garrido, Mark Hills, Michael Ilseman, Einar Broch Johnsen, Ralph Johnson, Michael Katelman, Laurentiu Leustean, Dorel Lucanu, Narciso Martí-Oliet, Patrick Meredith, Elena Naum, Olaf Owe, Stefan Reich, Andreas Roth, Juan Santa-Cruz, Ralf Sasse, Wolfram Schulte, Koushik Sen, Andrei Ștefănescu, Mark-Oliver Stehr, Carolyn Talcott, Prasanna Thati, Ram Prasad Venkatesan, Alberto Verdejo
Why is RWL good for programming languages?

- **Executability**: definitions are interpreters
- **Concurrency**: the norm rather than the exception
- **Equational abstraction**: collapse state space through equations
- **Generic tools** (built around the Maude system):
  - Execution, tracing and debugging
  - State space exploration
  - LTL model checker
  - Inductive theorem prover
Guidelines for defining programming languages in RWL

- Represent the state of a running program as a configuration term
- Represent rules of execution as rewrite rules and equations
  - Equations express structural changes and irrelevant steps
  - Rewrite rules express relevant computational steps (transitions)
- Execution: transition-sequence between equivalence classes of states
- State space: transition system
  - amenable to exploration and model checking
Guidelines for defining programming languages in RWL

- Represent the state of a running program as a configuration term
- Represent rules of execution as rewrite rules and equations
  - Equations express structural changes and irrelevant steps
  - Rewrite rules express relevant computational steps (transitions)
- Execution: transition-sequence between equivalence classes of states
- State space: transition system
  - amenable to exploration and model checking

This sounds great! But... we need methodologies.
PL definitional frameworks become RWL methodologies

[Șerbănuță, Roșu, Meseguer, 2007]

Programming language definitional styles can be faithfully captured as a particular definitional methodologies within RWL. (based on prior work by [Meseguer, 1992] [Marti-Oliet, Meseguer,1993] [Meseguer, Braga, 2004])

Best of both worlds

- Write definitions using your favorite PL framework style and notation
- Execute and analyze them through their RWL representation
Existing definitional frameworks at a closer look

Can existing styles define (and execute) real programming languages?
No, but their combined strengths might be able to.

Shortcomings

- Hard to deal with control (except for evaluation contexts)
  - break/continue, exceptions, halt, call/cc
- Modularity issues (except for Modular SOS)
  - Adding new features require changing unrelated rules
- Lack of semantics for true concurrency (except for CHAM)
  - Big-Step captures only the set of all possible results of computation
  - Approaches based on reduction only give interleaving semantics
- Tedious to find next redex (except for evaluation contexts)
  - one has to write essentially the same descent rules for each construct
- Inefficient for direct use as interpreters (except for Big-Step SOS)
Towards an ideal PL definitional framework

Goal: search for an ideal definitional framework based on RWL

- At least as expressive as Reduction with Evaluation Contexts
- At least as modular as Modular SOS
- At least as concurrent as the CHAM
The **K** Framework

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

**Rewriting Logic**

---

**The **K** framework**

- **K technique**: for expressive, modular, versatile, and clear PL definitions
- **K rewriting**: more concurrent than regular rewriting
- Representable in RWL for execution, testing and analysis purposes
K in a nutshell

Komputations

- Sequences of tasks, including syntax
- Capture the sequential fragment of programming languages
- Syntax annotations specify order of evaluation

Konfigurations

- 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 rules
Running example: KERNELC

A subset of the C programming language

- Functions
- Memory allocation: void arrCpy(int * a, int * b) {
  while (* a ++ = * b ++) {}
}
- Pointer arithmetic
- Input/Output

Extended with concurrency features

- Thread creation
- Lock-based synchronization
- Thread join
Running example: **KERNELC**

**Module KERNELC-SYNTAX**

```k
imports PL-ID=PL-INT

pointerId := Id

Expr := Int | PointerId | DeclId
    | * Expr [strict]
    | int | malloc | int* | (expr) [strict]
    | id | (id) [strict(2)]
    | if (Expr) , [ strict]
    | (,,) | (,,) | (,,) | (,,) | (,,) | (,,) | (,,) | (,,) |
    | print (*id*, *Exp) [strict]
    | stmt (*id*, *Exp) [strict]
    | & | id | (id) [strict(2)]
    | if | 0
    | null
    | free (Expr) [strict]
    | (int | malloc | int* | (malloc(int)) [strict]
    | Exp | (Exp)
    | spawn
    | acquire (Exp) [strict]
    | release (Exp) [strict]
    | join

StmtList := Stmt | StmtList

List[Bottom] := Bottom
    | ()
    | List[Bottom] , List[Bottom] [id: ()] [strict hybrid assoc]

List[PointerId] := PointerId | List[PointerId]
    | List[PointerId] , List[PointerId] [id: ()] [strict assoc]

List[DeclId] := DeclId | List[DeclId]
    | List[DeclId] , List[DeclId] [id: ()] [strict assoc]

List[Exp] := Exp | List[PointerId] | List[DeclId]
    | List[Exp] , List[Exp] [id: ()] [strict assoc]

DeclId := id

Stmt := id
    | if (Expr) Stmt
    | if (Expr) Stmt

Exp := Val
    | Val | Val

Id := main

Pym := #include<stdio.h>

END MODULE
```

**Module KERNELC-SEMANTICS**

```k
imports PL-CONVERSION & KERNELC-DESUGARED-SYNTAX

Kw := LoF (id)
K := LoF (Expr) 
LoF (id) | LoF (PointerId) | LoF (DeclId) | Stmts

Stmt := if (Expr) , [ stmt]

Exp := Val

Id := main
```

**Module KERNELC-DESUGARED-SYNTAX**

```k
macros

| E | E | 0 | 1
| E | E | 0 | 1
| E | E | 0 | 1
| E | E | 0 | 1
| E | E | 0 | 1
| E | E | 0 | 1
| E | E | 0 | 1
| E | E | 0 | 1
| E | E | 0 | 1
```

---

Traian Florin Șerbănuță (UIUC)  Programming Language Semantics using K
Configurations—the running state of a program

- Nested multisets (bags) of labeled cells
  - containing lists, sets, bags, maps and computations

Initial configuration for KERNELC
**K computations and K syntax**

### Computations
- Extend PL syntax with a “task sequentialization” operation
  - \( t_1 \leadsto t_2 \leadsto \ldots \leadsto t_n \), 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 = Exp [strict(2)]

**Stmt** ::= Exp ; [strict]

- Stmt Stmt [seqstrict]

**ERed** ⇐ ERed \( \leadsto \) * □

- E = ERed \( \Leftarrow \) ERed \( \leadsto \) E = □

- ERed ; \( \Leftarrow \) ERed \( \leadsto \) □ ;

- SRed S \( \Leftarrow \) SRed \( \leadsto \) □ S
Heating syntax through strictness rules

Computation

\[ t = \ast x ; \ast x = \ast y ; \ast y = t ; \]

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

\[
\begin{align*}
\text{Exp} & ::= \text{Id} \\
& | \ast \text{Exp} \quad \text{[strict]} \\
& | \text{Exp} = \text{Exp} \quad \text{[strict(2)]} \\
\text{Stmt} & ::= \text{Exp} ; \quad \text{[strict]} \\
& | \text{Stmt} \text{ Stmt} \quad \text{[seqstrict]} \\
* \text{ERed} & \Leftrightarrow \text{ERed} \rightsquigarrow \ast \Box \\
E & = \text{ERed} \Leftrightarrow \text{ERed} \rightsquigarrow E = \Box \\
\text{ERed} ; & \Leftrightarrow \text{ERed} \rightsquigarrow \Box ; \\
\text{SRed S} & \Leftrightarrow \text{SRed} \rightsquigarrow \Box \text{ S}
\end{align*}
\]
Heating syntax through strictness rules

Computation

\[ t = * x ; \quad \rightsquigarrow \quad \square * x = * y ; \quad * y = t ; \]

\[ \text{Syntax: BNF syntax annotated with strictness} \]

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

[strict] \[ \text{ERed} \quad \Rightarrow \quad \text{ERed} \, \rightsquigarrow \, * \, \square \]

[strict(2)] \[ \text{E} = \text{ERed} \quad \Rightarrow \quad \text{ERed} \, \rightsquigarrow \, E = \square \]

[seqstrict] \[ \text{ERed} \; ; \quad \Rightarrow \quad \text{ERed} \, \rightsquigarrow \, \square \; ; \]

[strict] \[ \text{SRed} \quad S \quad \Rightarrow \quad \text{SRed} \, \rightsquigarrow \, \square \; S \]
Heating syntax through strictness rules

Computation

\[ t = * x \quad \Leftarrow \quad \square ; \quad \Leftarrow \quad \square \quad * x = * y ; \quad * y = t ; \]

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

| Exp ::= Id                  | * ERed ⇔ ERed \sim * \square       |
| | * Exp                    | E = ERed ⇔ ERed \sim E = \square |
| | * Exp = Exp             | ERed ; ⇔ ERed \sim \square ;       |
| Stmt ::= Exp ;            | SRed S ⇔ SRed \sim \square S       |
| | Stmt Stmt              | [strict(2)]                        |
|                          | [strict]                           |
|                          | [seqstrict]                         |
Heating syntax through strictness rules

**Computation**

\[ * \ x \ \cdot\ c\; \rightarrow \; t = \Box \ \cdot\ c\; ; \ \rightarrow \; \Box \ ; \ \rightarrow \; \Box \ * \ x = * \ y \ ; \ * \ y = t \ ; \]

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

\[\begin{align*}
\text{Exp} &::= \text{Id} \\
& \mid * \text{Exp} \quad \text{[strict]} \\
& \mid \text{Exp} = \text{Exp} \quad \text{[strict(2)]} \\
\text{Stmt} &::= \text{Exp} ; \\
& \mid \text{Stmt} \; \text{Stmt} \quad \text{[seqstrict]} \\
\text{ERed} &\equiv \text{ERed} \ \cdot\ c\; \equiv \ * \Box \\
E &= \text{ERed} \ \Rightarrow \text{ERed} \ \cdot\ c\; \equiv \ E = \Box \\
\text{ERed} ; &\equiv \text{ERed} \ \cdot\ c\; ; \\
\text{SRed} \; S &\equiv \text{SRed} \ \cdot\ c\; \equiv \ S
\end{align*}\]
Heating syntax through strictness rules

Computation

\[ x \rightsquigarrow * \square \rightsquigarrow t = \square \rightsquigarrow \square ; \rightsquigarrow \square * x = * y ; * y = t ; \]

\[ \text{\texttt{K} Syntax: BNF syntax annotated with strictness} \]

\[ \text{Exp ::= Id} \]
\[ | \ * \ Exp \ [\text{strict}] \]
\[ | \ Exp = \ Exp \ [\text{strict}(2)] \]

\[ \text{Stmt ::= Exp ;} \]
\[ | \ Stmt \ Stmt \ [\text{seqstrict}] \]

\[ * \ ERed \ \Rightarrow \ ERed \rightsquigarrow * \square \]
\[ E = ERed \ \Rightarrow \ ERed \rightsquigarrow E = \square \]
\[ ERed ; \ \Rightarrow \ ERed \rightsquigarrow \square ; \]
\[ SRed \ S \ \Rightarrow \ SRed \rightsquigarrow \square \ S \]
Example: running configuration
Swapping the values at locations 4 and 5 in memory

```c
void swap(int * x, int * y){
    int t = * x; * x = * y; * y = t;
}
```

Possible Configuration during a call to `swap(a, b)`:

```
T

k

t ⊆ * 5 = □ ⊆ □ ; ⊆ env( a ⟷ 4  b ⟷ 5 )

env
x ⟷ 4  t ⟷ 1  y ⟷ 5

mem
4 ⟷ 7  5 ⟷ 7
```
Example: running configuration

- Strictness rules (from annotations) extract the redex
- How do we specify next step?
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 $V$
The \( \mathcal{K} \) Framework \( \mathcal{K} \) 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**

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 \( V \)
K rules: expressing natural language into rules

Summary

Reading from environment

```
x ↦ V
```

```
t ≈ * 5 = □ ≈ □ ; ≈ env( a ↦ 4 , b ↦ 5 )
```

```
x ↦ 4 , t ↦ 1 , y ↦ 5
```

```
4 ↦ 7 , 5 ↦ 7
```

Traian Florin Șerbănuta (UIUC)
Programming Language Semantics using K

18 / 39
Configuration abstraction enhances modularity
At least as modular as Modular SOS

Reading from environment

\[
\begin{align*}
X & \mapsto V \\
env & (a \mapsto 4, b \mapsto 5)
\end{align*}
\]
Applying $\mathbb{K}$ rules: Reading from environment

$X \mapsto V$
Applying $\mathbb{K}$ rules: Reading from environment

$\mathbb{K}$ rules

$X \mapsto V$

$\mathbb{K}$

$X \mapsto V$

$\mathbb{K}$

$1 \sim * 5 = \square \sim \square ; \sim \text{env}(a \mapsto 4 \ b \mapsto 5)$

$\text{env}$

$x \mapsto 4 \ t \mapsto 1 \ y \mapsto 5$

$\text{mem}$

$4 \mapsto 7 \ 5 \mapsto 7$
Applying $\mathbb{K}$ rules: Strictness of assignment

$$\text{Exp} ::= \text{Exp} = \text{Exp}$$

which is syntactic sugar for:

$$E = \text{Redex} \iff \text{Redex } \overset{*}{\sim} E = \square$$

```
1 \overset{*}{\sim} 5 = \square \overset{*}{\sim} \square ; \overset{*}{\sim} \text{env}( a \mapsto 4 \ b \mapsto 5 )
```

```
\text{env} \quad \text{mem}
\begin{array}{ccc}
x & \mapsto & 4 \\
t & \mapsto & 1 \\
y & \mapsto & 5 \\
\end{array}
\begin{array}{ccc}
4 & \mapsto & 7 \\
5 & \mapsto & 7 \\
\end{array}
```
Applying $\kappa$ rules: Strictness of assignment

$Exp ::= Exp = Exp$  \[strict(2)\]

which is syntactic sugar for:

$E = \text{Redex} \Leftrightarrow \text{Redex} \rightsquigarrow E = \square$

![Diagram showing the K framework rules and examples of strictness of assignment](image)
Applying $\mathbf{K}$ rules: Updating memory

\[
\begin{align*}
\text{Applying } & \quad \text{K rules: Updating memory} \\
* \quad N &= \mathbf{V} \\
\frac{}{\mathbf{V}} \\
\text{mem} & \\
N &\mapsto \_ \\
\mathbf{V} \\
\end{align*}
\]

\[
\begin{align*}
\text{Applying } & \quad \text{K rules: Updating memory} \\
* \quad 5 &= 1 \\
\text{env} & \\
x &\mapsto 4 \\
t &\mapsto 1 \\
y &\mapsto 5 \\
\text{mem} & \\
4 &\mapsto 7 \\
5 &\mapsto 7 \\
\end{align*}
\]
Applying $\mathbb{K}$ rules: Updating memory

\[
\begin{align*}
* & \quad N = V \\
\frac{}{V} & \quad N \mapsto _V
\end{align*}
\]
Applying $\mathcal{K}$ rules: Semantics for expression statements

Strictness: $Stmt ::= Exp$;  \[\text{[strict]}\]

Semantic rule: $V ; \rightarrow \text{skip}$
Applying \(\mathcal{K}\) rules: Semantics for expression statements

Strictness: \(Stmt ::= Exp\);  
Semantic rule: \(V ; \rightarrow \text{skip}\)
Applying $\mathbb{K}$ rules: Recovering environment
Applying \(K\) rules: Recovering environment

\[
V \sim env(Env) \\
\quad \Downarrow \\
\cdot
\]

\[
env \quad \Downarrow \\
Env
\]

\[
T \\
\quad \Downarrow \\
\quad \Downarrow \\
k \quad skip \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quiet
**K’s definitional power**

More expressive than Reduction with Evaluation Contexts

Some of K’s strengths

- Dealing with control
- Task synchronization
- Defining reflective features
Definitional Power: Control

Call with current continuation

Passing computation as value:

Applying computation as function:

Impossible to capture in frameworks without evaluation contexts

Why? Execution context is logical context, not observable from within
Definitional Power: Synchronization

Synchronous communication

- Hard, if not impossible to capture in other definitional frameworks
  - (might work for toy languages: CCS, π-calculus)
- Both redexes must be matched simultaneously
  - Easy in $\mathbb{K}$ as redex is always at top of computation
Definitional Power: Syntactic Reflection

\[ \text{\textbf{K} Syntax} = \text{AST} \]

- \[ K ::= KLabel(List\{K\}) \mid \bullet_{K} \mid K \leadsto K \]
  \[ List\{K\} ::= K \mid \bullet_{List\{K\}} \mid List\{K\}, List\{K\} \]

- Everything else (language construct, builtin constant) is a \text{K} label

\[ a + 3 \equiv \_ + \_ (a(\bullet_{List\{K\}}), 3(\bullet_{List\{K\}})) \]

Reflection through AST manipulation

- Generic AST visitor pattern
- Code generation: quote/unquote
- Binder independent substitution
- Check dissertation for more details
The concurrency of the K framework

- Truly concurrent (even more concurrent than the CHAM framework)
- Captures concurrency with sharing of resources.
**K rules enhance concurrency**

**Concurrent reads**

\[ \begin{array}{c}
\text{k} \\
\text{mem}
\end{array} \xrightarrow{\ast} \begin{array}{c}
N \\
\text{mem}
\end{array} \xrightarrow{\ast} \begin{array}{c}
V \\
N \leftrightarrow V
\end{array} \xrightarrow{\ast} \begin{array}{c}
2 \\
\rightarrow 5
\end{array} \xrightarrow{\ast} 
\begin{array}{c}
6
\end{array} \xrightarrow{\ast} 
\begin{array}{c}
3 \\
1
\end{array} \xrightarrow{\ast} \begin{array}{c}
4
\end{array} \xrightarrow{\ast} \begin{array}{c}
\cdot \\
\cdot
\end{array} \]
\( \text{K rules enhance concurrency} \)

**Concurrent reads**

\[
\begin{array}{c}
\text{k} \\
\begin{array}{c}
* \ N \\
\Downarrow \\
V
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{mem} \\
\begin{array}{c}
N \leftrightarrow V
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{T} \\
\begin{array}{c}
\text{k}
\text{k}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{mem} \\
\begin{array}{c}
2 \leftrightarrow 5 \\
3 \leftrightarrow 1 \\
4 \leftrightarrow 6
\end{array}
\end{array}
\]

\[
\begin{array}{c}
* \ 3 \sim \cdots \\
* \ 3 \sim \cdots
\end{array}
\]
K rules enhance concurrency

Concurrent reads

\[
\begin{align*}
\text{k} & \quad \text{mem} \\
\star N & \quad V \\
\end{align*}
\]
The \( \mathcal{K} \) Framework

Concurrency

\( \mathcal{K} \) rules enhance concurrency

Concurrent updates (on distinct locations)

\[ * \quad N = V \quad \frac{\Delta}{V} \]

\[ \quad \text{mem} \quad N \leftrightarrow \_ \quad \frac{\Delta}{V} \]

\[ k \]

\[ T \]

\[ k \]

\[ * \quad 2 = 9 \quad \sim \cdots \]

\[ \text{mem} \]

\[ 2 \leftrightarrow 5 \quad 3 \leftrightarrow 1 \quad 4 \leftrightarrow 6 \]

\[ k \]

\[ * \quad 3 = 0 \quad \sim \cdots \]
K rules enhance concurrency

Concurrent updates (on distinct locations)

\[
\begin{align*}
N &\rightarrow V \\
N &\rightarrow V
\end{align*}
\]
The $\mathbb{K}$ Framework

Concurrency

$\mathbb{K}$ rules enhance concurrency

Concurrent updates (on distinct locations)

\[ N = V \quad \implies \quad V \]

\[ N \leftrightarrow _{\mathit{mem}} \]

\[ \begin{array}{c}
  \mathit{k} \\
  \overset{\top}{\mathit{mem}} \\
  9 \sim \cdots \\
  \mathit{mem} \\
  2 \leftrightarrow 9 \quad 3 \leftrightarrow 0 \quad 4 \leftrightarrow 6
\end{array} \]

Traian Florin Șerbănuță (UIUC)

Programming Language Semantics using $\mathbb{K}$

PhD Thesis, University of Illinois, December 2010
K rules enhance concurrency

No dataraces: rule instances can overlap only on read-only part
Formally capturing the concurrency of $K$

Idea: Give semantics to $K$ rewriting through graph rewriting
- $K$ rules resemble graph rewriting rules
- Graph rewriting captures concurrency with sharing of context

Results: A new formalism of term-graph rewriting
- Sound and complete w.r.t. term rewriting
- Capturing the intended concurrency of $K$ rewriting
Formally capturing the concurrency of \( \mathbb{K} \) graph rewriting

\( \mathbb{K} \) graph rules: a new kind of term graph rewriting rules

\[ h(\overbrace{x}, \_, 1) \]
\[ g(x, x) \\
0 \]

\( \mathbb{K} \) rule \( \rho \)

Direct representation as a rewrite rule \( K2R(\rho) \)

\[ h(x, y, 1) \rightarrow h(g(x, x), y, 0) \]

Corresponding graph rewrite rule \( K2G(\rho) \)
K rewriting

Definition
Let S be a K rewrite system and t be a term. Then

\[ t \xrightarrow{S} t' \iff K2G(t) \xrightarrow{K2G(S)} \text{Graph} H \text{ such that } \text{term}(H) = t' \]

Theorem (Correctness w.r.t. rewriting)

Soundness: \( t \xrightarrow{K} t' \) then \( t \xrightarrow{K2R(\rho)} t' \).

Completeness: \( t \xrightarrow{K2R(\rho)} t' \) then \( t \xrightarrow{\rho} t' \).

Serializability: \( t \xrightarrow{\rho_1 + \cdots + \rho_n} t' \) then \( t \xrightarrow{\rho_1^*} \cdots \xrightarrow{\rho_n^*} t' \) (and thus, \( t \xrightarrow{\ast} t' \)).
Instead of proof

Quite technical and complex

- 12 pages of definitions, constructions, and preparatory results

Getting there

- $\mathbb{K}$ term graphs and $\mathbb{K}$ graph rewrite rules
- $\mathbb{K}$ term graphs are stable under concurrent applications of $\mathbb{K}$ rules
  - If their instances only overlap on read-only part
  - If they do not introduce cycles
- Serializability based on graph rewriting serializability
  
  [Ehrig, Kreowski, 1976]
- $\mathbb{K}$ graph rewriting conservatively extends Jungle rewriting
- Jungle rewriting is sound and complete w.r.t. rewriting
  
  [Holland, Plump, 1991; Plump 1999]
Representing $\mathcal{K}$ into RWL

Core of the K-Maude tool

Faithfully, in two steps

- Represent $\mathcal{K}$ rewriting as $\mathcal{K}$ graph rewriting
- RWL captures graph rewriting as theories over concurrent objects
  - Theoretical, non executable
- Useful to reason about the amount of concurrency in a $\mathcal{K}$ definition

Directly—Implemented in the K-Maude tool

- Straight-forward, forgetting read-only information of $\mathcal{K}$ rules
- Executable, with full access to rewriting logic’s tools
- Loses the one-step concurrency of $\mathcal{K}$
  - Nevertheless, it is sound for all other analysis purposes
Direct embedding of \( \mathbb{K} \) into RWL (core of K-Maude)

Dereferencing rule

\[
\begin{array}{c}
\text{Der} & \frac{k}{\text{mem}} & \frac{N}{V} \\
* & N & \rightarrow \; V
\end{array}
\]

\( T \)

threads

\[ \text{thread}^* \]

\begin{align*}
\text{k} & \rightarrow \mathbb{K} \\
\text{env} & \rightarrow \text{Map} \\
\text{id} & \rightarrow 0 \\
\text{locks} & \rightarrow \text{Map} \\
\text{ctreads} & \rightarrow \text{Set} \\
\text{funs} & \rightarrow \text{Map} \\
\text{in} & \rightarrow \text{List} \\
\text{out} & \rightarrow \text{List} \\
\text{mem} & \rightarrow \text{Map} \\
\text{ptr} & \rightarrow \text{Map} \\
\text{next} & \rightarrow 1
\end{align*}
Direct embedding of $\mathbb{K}$ into RWL (core of K-Maude)

Dereferencing rule

Configuration-concretized version of the rule
Representing $K$ into RWL (core of K-Maude)

Dereferencing rule

Flattening the $K$ rule to a rewrite rule

$$\text{rl} \quad \text{<threads>} \quad \text{<thread>} \quad <k> \star N \leadsto ?1:K \quad ?2:Bag \quad ?3:Bag \quad \text{<threads>}$$
$$\quad <\text{mem}> \quad N \mapsto V \quad ?4:Map \quad <\text{mem}>$$

$$\Rightarrow \quad \text{<threads>} \quad \text{<thread>} \quad <k> \quad V \leadsto ?1:K \quad ?2:Bag \quad ?3:Bag \quad \text{<threads>}$$
$$\quad <\text{mem}> \quad N \mapsto V \quad ?4:Map \quad <\text{mem}>$$
K-Maude overview [Şerbănuţă, Roşu, 2010]

K-Maude compiler: 22 stages ~ 8k lines of Maude code

- Transforming $\mathbb{K}$ rules in rewrite rules (6 stages)
- Strictness rules generation (3 stages)
- Flattening syntax to AST form (10 stages)
- Interface (3 stages)

K-$\mathbb{LATEX}$ compiler—typesetting ASCII $\mathbb{K}$

- Graphical representation, for presentations (like this defense)
- Mathematical representation, for papers (like the dissertation)
K-Maude Demo?
From PL definitions to runtime analysis tools
K-Maude Demo: Datarace freeness

- Begin with a definition of \texttt{KERNELC}, a subset of C
  - functions, memory allocation, pointers, input/output
- Extend it with concurrency features (threads, locks, join)
  - without changing anything but the configuration
- Specify dataraces and explore executions for datarace freeness
  - adding only two structural rules, for write-write and write-read conflicts
- Case study: Bank account with buggy transfer function
  - Detect the race using a test driver
  - Fix the race and verify the fix
  - Show that the fix introduces a deadlock
  - Fix the fix and verify it is now datarace and deadlock free
K-Maude Demo: Experimenting with memory models

- Change the memory model for concurrent KERNELC
- Use a relaxed memory model inspired by x86-TSO
  - Threads act like processors, local variables as registers
  - Synchronization constructs (thread create, end, join) generate fences
  - Rules faithfully capture the natural language description
  - Only the rules specifically involved need to be changed
- Case study: Analyzing Peterson’s algorithm on both memory models
  - first model, being sequentially consistent, ensures mutual exclusion
  - second model fails to ensure it
K-Maude community

http://k-framework.googlecode.com

Current K-Maude projects

- **C** Chucky Ellison
- **Haskell** Michael Ilseman, David Lazar
- **Javascript** Maurice Rabb
- **Scheme** Patrick Meredith
- **X10** Milos Gligoric

Matching Logic  Elena Naum, Andrei Ștefănescu

CEGAR  Irina Asăvoae, Mihail Asăvoae

Teaching  Dorel Lucanu, Grigore Roșu

Interface  Andrei Arusoaie, Michael Ilseman
Summary of contributions

\textbf{K: a framework for defining real programming languages}
- Expressive—at least as Reduction with evaluation contexts
- Modular—at least as Modular SOS
- Concurrent—more than CHAM
- Concise, intuitive

\textbf{K-Maude: a tool for executing and analyzing K definitions}
- K definitions become testing and analysis tools
- Strengthens the thesis that RWL is amenable for PL definitions
Related work using $\mathbb{K}$

Definitions of real languages “by the book”

- **Java** [Farzan, Chen, Meseguer, Roșu, 2004]
- **Scheme** [Meredith, Hills, Roșu, 2007]
- **Verilog** [Meredith, Katelman, Meseguer, Roșu, 2010]
- **C** [Ellison, Roșu, 2011?]

Analysis tools and techniques

- **Static Policy Checker for C** [Hills, Chen, Roșu, 2008]
- **Memory Safety** [Roșu, Schulte, Șerbănuță, 2009]
- **Type Soundness** [Ellison, Șerbănuță, Roșu, 2008]
- **Matching Logic** [Roșu, Ellison, Schulte, 2010]
- **CEGAR with predicate abstraction** [Asăvoae, Asăvoae, 2010]
Future Work

Rewriting & Programming languages
- Long list of feature requests for K-Maude
- Use of $K$ as a programming language
  - Compiling $K$ definitions for faster (and concurrent) execution
- Proving meta-properties about languages

Specifying and verifying concurrency
- Relaxed memory models

Foundations
- Explore non-serializable concurrency for rewriting
- Models for $K$ definitions