An Overview of the K Framework

Grigore Rosu
University of Illinois at Urbana-Champaign
(joint work with Traian Florin Serbanuta)
Challenges in Programming Language Design / Semantics / Analysis

• Programming languages are continuously born, updated and extended
  – C#, CIL; Java memory model, Scheme R6RS, C1X
  – Concurrency is the norm, not the exception

• Executable specifications could help
  – Design and maintain mathematical definitions
  – Easily test/analyze language updates/extensions
  – Explore/Abstract non-deterministic executions
K Project

• Started in 2003, 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.
Overview

• Rewriting logic semantics project
  – How it all started ...

• K framework
  – K definitional style
  – K concurrent rewriting

• Example
  – Challenge language
Rewriting Logic Semantics Project

• Rewriting Logic (RWL)
  – Meseguer 1992

• Rewriting Logic Semantics (RLS) Project

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

• Participants (probably incomplete list):
  – Wolfgang Ahrendt, Musab Al-Turki, Marcelo d’Amorim, Irina M. Asavoae, Mihai Asavoae, 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, Jose Meseguer, Elena Naum, Olaf Owe, Stefan Reich, Grigore Rosu, Andreas Roth, Juan Santa-Cruz, Ralf Sasse, Wolfram Schulte, Koushik Sen, Andrei Stefanescu, Mark-Oliver Stehr, Carolyn Talcott, Prasanna Thati, Traian Serbanuta, Ram Prasad Venkatesan, Alberto Verdejo
Why is Rewriting Logic Good for Programming Languages?

- **Executability**
  - Language definitions turn into interpreters
- **Concurrency**
  - The norm rather than the exception
- **Equational abstraction**
  - Collapse state space through equations
- **Generic tools (built around Maude)**
  - Execution, tracing, debugging, state space search, model checker, inductive theorem prover
Guidelines for Defining Programming Languages in RWL

• Represent program state as configuration term
• Represent computational steps as rewrite rules
• Represent structural changes as equations
• These associate to any given configuration a transition system
  – In particular, it associates semantics to programs
• Resulting transition systems are amenable to exploration, search and model checking
• Great, but it does not tell us how to do it
Conventional semantic frameworks become RWL definitional methodologies

• This allows to define a language using one’s favorite semantic style, and then to execute, explore or model check it using RWL
Conventional semantic frameworks become RWL definitional methodologies

Many authors contributed to this diagram, including all the abovementioned. See [Serbanuta, Rosu, Meseguer 2007 – Inf. & Comp.] for a recent survey.

- This allows to define a language using one’s favorite semantic style, and then to execute, explore or model check it using RWL
Existing Semantics Frameworks, From a Unified Perspective

• The unified view of semantic frameworks within RWL also allows to better examine them and understand their limitations.

• For example, can existing styles define *real* programming languages on a regular basis, as opposed to only toy languages?
  – No, but their combined strengths might
Shortcomings of Existing Frameworks

• Hard to deal with control (except evaluation contexts)
  – halt, break/continue, exceptions, callcc
• 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)
Towards an Ideal PL Definitional Framework

- Our initial goal was to search for an ideal language definitional framework within RWL
  - At least as expressive as evaluation contexts
  - At least as modular as Modular SOS
  - At least as concurrent as the CHAM
The K Framework

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
- Pointer arithmetic
- Input/Output

```c
void arrCpy(int * a, int * b) {
    while (* a ++ = * b ++) {}
}
```

Extended with concurrency features
- Thread creation
- Lock-based synchronization
- Thread join
Running example: KERNELC
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
  \[ t_1 \rightsquigarrow t_2 \rightsquigarrow \ldots \rightsquigarrow t_n \], where \( t_i \) are computational tasks
- Computational tasks: pieces of syntax (with holes), closures, \ldots
- Mostly under the hood, via intuitive PL syntax annotations

K Syntax: BNF syntax annotated with strictness

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

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

Computation

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

K Syntax: BNF syntax annotated with strictness

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

\[
\begin{align*}
    * \text{ERed} & \Leftrightarrow \text{ERed} \sim * \Box \\
    \text{E} & = \text{ERed} \Leftrightarrow \text{ERed} \sim \text{E} = \Box \\
    \text{ERed} & ; \Leftrightarrow \text{ERed} \sim \Box ; \\
    \text{SRed} \ S & \Leftrightarrow \text{SRed} \sim \Box \ S
\end{align*}
\]
Heating syntax through strictness rules

Computation

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

\K Syntax: BNF syntax annotated with strictness

\[ \text{Exp} ::= \text{Id} \]
\[ \quad | \quad \ast \; \text{Exp} \quad \text{[strict]} \]
\[ \quad | \quad \text{Exp} = \text{Exp} \quad \text{[strict(2)]} \]

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

\[ \ast \; \text{ERed} \quad \Rightarrow \quad \text{ERed} \; \rightsquigarrow \; \ast \; \square \]
\[ E = \text{ERed} \quad \Rightarrow \quad \text{ERed} \; \rightsquigarrow \; E = \square \]
\[ \text{ERed} ; \quad \Rightarrow \quad \text{ERed} \; \rightsquigarrow \; \square \; ; \]
\[ \text{SRed} \; S \quad \Rightarrow \quad \text{SRed} \; \rightsquigarrow \; \square \; S \]
Heating syntax through strictness rules

Computation

\[ t = * x \quad \Rightarrow \quad \Box; \quad \Rightarrow \quad \Box * x = * y ; * y = t ; \]

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

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

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

\[ * ERed \ \Leftrightarrow \ ERed \sim * \Box \]
\[ E = ERed \ \Leftrightarrow \ ERed \sim E = \Box \]
\[ ERed ; \ \Rightarrow \ ERed \sim \Box ; \]
\[ SRed \ S \ \Leftrightarrow \ SRed \sim \Box \ S \]
Heating syntax through strictness rules

Computation

\[ * x \sim t = \Box \sim \Box ; \sim \Box * x = * y ; * y = t ; \]

K Syntax: BNF syntax annotated with strictness

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

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

Computation

\[ X \leadsto \star \square \leadsto t = \square \leadsto \square ; \leadsto \square \star x = \star y ; \star y = t ; \]

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

\[ Exp ::= \textbf{id} \]
\[ | * \ Exp \quad \textbf{[strict]} \quad * \ ERed \ \Rightarrow \ ERed \leadsto * \ \square \]
\[ | Exp = Exp \quad \textbf{[strict(2)]} \quad E = ERed \ \Rightarrow \ ERed \leadsto E = \square \]
\[ Stmt ::= Exp ; \quad \textbf{[strict]} \quad ERed ; \quad \Rightarrow \ ERed \leadsto \square ; \]
\[ | Stmt \ Stmt \quad \textbf{[seqstrict]} \quad SRed S \quad \Rightarrow \ SRed \leadsto \square \ S \]
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 ≃ * 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 \texttt{redex}
- How do we specify next step?

\[
t \Rightarrow * \ 5 = \square \Rightarrow \square ; \Rightarrow \texttt{env}( \ a \mapsto 4 \ b \mapsto 5 )
\]

\[
\begin{array}{ccc}
\text{env} & \text{mem} \\
4 \mapsto 7 & 5 \mapsto 7 \\
\end{array}
\]

\[
\begin{array}{ccc}
x \mapsto 4 & t \mapsto 1 & y \mapsto 5 \\
\end{array}
\]
**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 $V$

$$t \overline{=} \times 5 = \square \; ; \; \overline{=} \; \text{env}(\ a \mapsto 4 \ b \mapsto 5)$$

**env**

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

**mem**

$4 \mapsto 7 \quad 5 \mapsto 7$
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 $V$
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

\[ k \vdash \begin{array}{c} X \Rightarrow V \\ \end{array} \]

\[ t \sim * 5 = \square \sim \square ; \sim \text{env}( a \leftrightarrow 4 \ b \leftrightarrow 5 ) \]

\[ \text{env} \]

\[ x \leftrightarrow 4 \ t \leftrightarrow 1 \ y \leftrightarrow 5 \]

\[ \text{mem} \]

\[ 4 \leftrightarrow 7 \ 5 \leftrightarrow 7 \]
Configuration abstraction enhances modularity

At least as modular as Modular SOS

Reading from environment

```
X ⊨ V
```

```
t ⊨ * 5 = □ ⊨ □ ; ⊨ env( a ⊨ 4 b ⊨ 5 )
```

```
x ⊨ 4 t ⊨ 1 y ⊨ 5
```

```
4 ⊨ 7 5 ⊨ 7
```

```k```
```
X ⊨ V
```

```k```
```
t ⊨ * 5 = □ ⊨ □ ; ⊨ env( a ⊨ 4 b ⊨ 5 )
```

```k```
```
x ⊨ 4 t ⊨ 1 y ⊨ 5
```

```k```
```
4 ⊨ 7 5 ⊨ 7
```
Applying $\mathbf{K}$ rules: Reading from environment

\[ \mathbf{k} \]

\[ \frac{X}{V} \]

\[ X \leftrightarrow V \]

\[ T \]

\[ k \]

\[ t \sim \cdot 5 = \square \sim \square ; \sim \text{env} ( a \mapsto 4, b \mapsto 5 ) \]

\[ \text{env} \]

\[ x \mapsto 4, t \mapsto 1, y \mapsto 5 \]

\[ \text{mem} \]

\[ 4 \mapsto 7, 5 \mapsto 7 \]
Applying \( \mathcal{K} \) rules: Reading from environment

\[ \frac{X}{\top} \]

\[ X \leftrightarrow \top \]

\[ 1 \mapsto \ast \ 5 = \square \mapsto \square \ ; \ \sim \ env( a \mapsto 4 \ \ b \mapsto 5 ) \]

\[ env \]

\[ x \leftrightarrow 4 \ \ t \leftrightarrow 1 \ \ y \leftrightarrow 5 \]

\[ mem \]

\[ 4 \leftrightarrow 7 \ 5 \leftrightarrow 7 \]
Applying K rules: Strictness of assignment

```
Exp ::= Exp = Exp

which is syntactic sugar for:

E = Redex \implies Redex \sim E = □
```

Diagram:

```
1 \sim * 5 = □ \sim □ ; \sim env( a \mapsto 4 \ b \mapsto 5 )
```

```
env
x \mapsto 4 \ t \mapsto 1 \ y \mapsto 5
```

```
mem
4 \mapsto 7 \ 5 \mapsto 7
```
Applying $\mathcal{K}$ rules: Strictness of assignment

$$\text{Exp} ::= \text{Exp} = \text{Exp} \quad \text{[strict(2)]}$$

which is syntactic sugar for:

$$E = \text{Redex} \quad \Rightarrow \quad \text{Redex } \sim E = \Box$$

---

Diagram:

- $T$
- $k$
- $5 = 1 \sim \Box$
- $\sim env(a \mapsto 4 \ b \mapsto 5)$
- $\text{env}$
  - $x \mapsto 4 \ t \mapsto 1 \ y \mapsto 5$
- $\text{mem}$
  - $4 \mapsto 7 \ 5 \mapsto 7$
Applying $\mathbf{K}$ rules: Updating memory

\[ \frac{\text{N} = \text{V}}{\text{V}} \]

\[ \text{N} \mapsto \_ \mapsto \text{V} \]

\[ \text{5} = 1 \sim \square ; \sim \text{env} \left( \begin{array}{c} a \mapsto 4 \ b \mapsto 5 \end{array} \right) \]

\[ \text{env} \]

\[ \begin{array}{c} x \mapsto 4 \ t \mapsto 1 \ y \mapsto 5 \end{array} \]

\[ \text{mem} \]

\[ \begin{array}{c} 4 \mapsto 7 \ 5 \mapsto 7 \end{array} \]
Applying $\text{K}$ rules: Updating memory

$N = V$  
$\frac{k}{\text{mem}}$  
$N \mapsto _{\_}$  
$V$

$\frac{1 \sim \square; \sim \text{env}(a \mapsto 4 \ b \mapsto 5)}{\text{env}}$  
$x \mapsto 4 \ t \mapsto 1 \ y \mapsto 5$

$\frac{4 \mapsto 7 \ 5 \mapsto 1}{\text{mem}}$
Applying $\mathbf{K}$ rules: Semantics for expression statements

Strictness: $Stmt ::= Exp ;$  
Semantic rule: $V ; \rightsquigarrow \text{skip}$

```
1 \sim \Box ; \sim \text{env}( a \mapsto 4 \ b \mapsto 5 )
```

```
\text{env}:
\begin{align*}
x & \mapsto 4 \\
t & \mapsto 1 \\
y & \mapsto 5
\end{align*}
```

```
\text{mem}:
\begin{align*}
4 & \mapsto 7 \\
5 & \mapsto 1
\end{align*}
```
Applying $\mathbf{K}$ rules: Semantics for expression statements

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

Semantic rule: $V ; \rightarrow \text{skip}$
Applying $\mathbb{K}$ rules: Recovering environment

$V \sim env(Env)$

$\frac{}{Env}$

$\frac{skip \sim env(a \mapsto 4 \quad b \mapsto 5)}{env}$

$\frac{x \mapsto 4 \quad t \mapsto 1 \quad y \mapsto 5}{mem}$

$\frac{4 \mapsto 7 \quad 5 \mapsto 1}{mem}$
Applying \( K \) rules: Recovering environment

\[
V \sim \text{env}(Env) \\
\_ \\
Env
\]

\[
T \\
k \\
\text{skip} \\
\text{env} \\
a \mapsto 4 \quad b \mapsto 5 \\
\text{mem} \\
4 \mapsto 7 \quad 5 \mapsto 1
\]
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:**

\[
\text{callcc } V \leadsto K \\
\frac{}{V \leadsto \text{cc}(K)}
\]

**Applying computation as function:**

\[
\text{cc}(K) \quad V \leadsto - \\
\frac{}{V \leadsto K}
\]

- 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

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

\[ K ::= K\text{Label}(\text{List}\{K\}) \mid \cdot_K \mid K \bowtie K \]
\[ \text{List}\{K\} ::= K \mid \cdot_{\text{List}\{K\}} \mid \text{List}\{K\}, \text{List}\{K\} \]

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

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

Reflection through AST manipulation

- Generic AST visitor pattern
- Code generation: quote/unquote
- Binder independent substitution
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

\[ \frac{k \quad N}{\mathit{mem} \quad N \leftrightarrow V} \]
**K rules enhance concurrency**

Concurrent reads

```
* N \rightarrow V
```

```
mem
```

```
2 \leftrightarrow 5 3 \leftrightarrow 1 4 \leftrightarrow 6
```
K rules enhance concurrency

Concurrent reads

\( \frac{k}{*N} \)

\( N \stackrel{\leftarrow}{\rightarrow} V \)

\( \frac{mem}{1 \sim \ldots} \)

\( 2 \leftrightarrow 5 \quad 3 \leftrightarrow 1 \quad 4 \leftrightarrow 6 \)
K rules enhance concurrency

Concurrent updates (on distinct locations)

\[ \frac{N = V}{V} \]

\[ N \mapsto \_ \mapsto V \]

\[ \begin{align*}
 k & \quad 2 = 9 \quad \cdots \\
 mem & \quad 2 \mapsto 5 \quad 3 \mapsto 1 \quad 4 \mapsto 6 \\
 k & \quad 3 = 0 \quad \cdots
\end{align*} \]
K rules enhance concurrency

Concurrent updates (on distinct locations)

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

\[ N \leftrightarrow \_ \leftrightarrow V \]

\[ \text{mem} \]

\[ \begin{array}{c}
\text{k} \\
* 2 = 9 \sim \cdots \\
\text{mem} \\
2 \leftrightarrow 5 \quad 3 \leftrightarrow 1 \\
\text{k} \\
* 3 = 0 \sim \cdots \\
\end{array} \]

\[ 4 \leftrightarrow 6 \]
K rules enhance concurrency

Concurrent updates (on distinct locations)

\[
\begin{align*}
\text{mem} & \quad N \leftrightarrow \_ \\
\text{k} & \quad N = V \\
\hline
& \quad V \\
\end{align*}
\]

\[
\begin{align*}
\text{mem} & \quad 2 \leftrightarrow 9 \\
\text{k} & \quad 9 \leftrightarrow \ldots \\
\text{k} & \quad 0 \leftrightarrow \ldots \\
\text{mem} & \quad 3 \leftrightarrow 0 \\
\end{align*}
\]

4 \leftrightarrow 6
K rules enhance concurrency

No dataraces: rule instances can overlap only on read-only part

\[
\text{k} \quad \begin{array}{c}
\text{mem} \\
N \mapsto V
\end{array} \\
\frac{N = V}{V}
\]

\[
\text{k} \\
\text{mem} \\
N \mapsto V
\]

\[
\text{mem} \\
2 \mapsto 5, \quad 3 \mapsto 1, \quad 4 \mapsto 6
\]

\[
\begin{array}{c}
\text{k} \\
\frac{N}{V}
\end{array}

\begin{array}{c}
\frac{3 = 0}{\ldots}
\end{array}

\begin{array}{c}
\frac{3}{\ldots}
\end{array}
\]
Formally capturing the concurrency of $\mathcal{K}$

Idea: Give semantics to $\mathcal{K}$ rewriting through graph rewriting
- $\mathcal{K}$ 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 $\mathcal{K}$ rewriting
\[ h(\underbrace{x}, \underbrace{\_}, \underbrace{1}) \]
\[ g(x, x) \quad \underbrace{0} \]

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: If $t \xrightarrow{\rho} t'$ then $t \xrightarrow{\text{K2R}(\rho)} t'$.

Completeness: If $t \xrightarrow{\text{Rew}} t'$ then $t \xrightarrow{\rho} t'$.

Serializability: If $t \xrightarrow{\rho_1 + \cdots + \rho_n} t'$ then $t \xrightarrow{\rho_1^*} \cdots \xrightarrow{\rho_n^*} t'$ (and thus, $t \xrightarrow{\ast} t'$).
Representing $\mathbb{K}$ into RWL

Core of the K-Maude tool

Faithfully, in two steps

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

Directly—Implemented in the K-Maude tool

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

Dereferencing rule

$$\frac{k}{\text{mem}} \quad \frac{N}{N \leftrightarrow V}$$
Direct embedding of \( K \) into RWL (core of K-Maude)

Dereferencing rule

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

Dereferencing rule

Flattening the $K$ rule to a rewrite rule

\[
\begin{align*}
rl & \quad <\text{threads}> <\text{thread}> \quad <k> \ast N \rightsquigarrow ?1:K <</k> \quad ?2:\text{Bag} <</\text{thread}> \quad ?3:\text{Bag} <</\text{threads}> \\
& \quad <\text{mem}> N \leftrightarrow V \quad ?4:\text{Map} <</\text{mem}> \\
=& \quad <\text{threads}> <\text{thread}> \quad <k> \quad V \rightsquigarrow ?1:K <</k> \quad ?2:\text{Bag} <</\text{thread}> \quad ?3:\text{Bag} <</\text{threads}> \\
& \quad <\text{mem}> N \leftrightarrow V \quad ?4:\text{Map} <</\text{mem}> 
\end{align*}
\]
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-\LaTeX{} compiler—typesetting ASCII $\mathbb{K}$
- Graphical representation, for presentations
- Mathematical representation, for papers
K-Maude Demo?
From PL definitions to runtime analysis tools
K-Maude community
http://k-framework.googlecode.com

Current K-Maude projects

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

Matching Logic  Elena Naum, Andrei Stefănescu
CEGAR    Irina Asăvoae, Mihail Asăvoae

Teaching  Dorel Lucanu, Grigore Roșu
Interface Andrei Arusoaie, Michael Ilsemann
Summary of contributions

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

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 K

Definitions of real languages “by the book”

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

Analysis tools and techniques

- Static Policy Checker for C [Hills, Chen, Roşu, 2008]
- Memory Safety [Roşu, Schulte, Şerbănuţă, 2009]
- Type Soundness [Ellison, Şerbănuţă, Roşu, 2008]
- Matching Logic [Roşu, 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 $\mathbf{K}$ as a programming language
  - Compiling $\mathbf{K}$ 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 $\mathbf{K}$ definitions