K-Maude

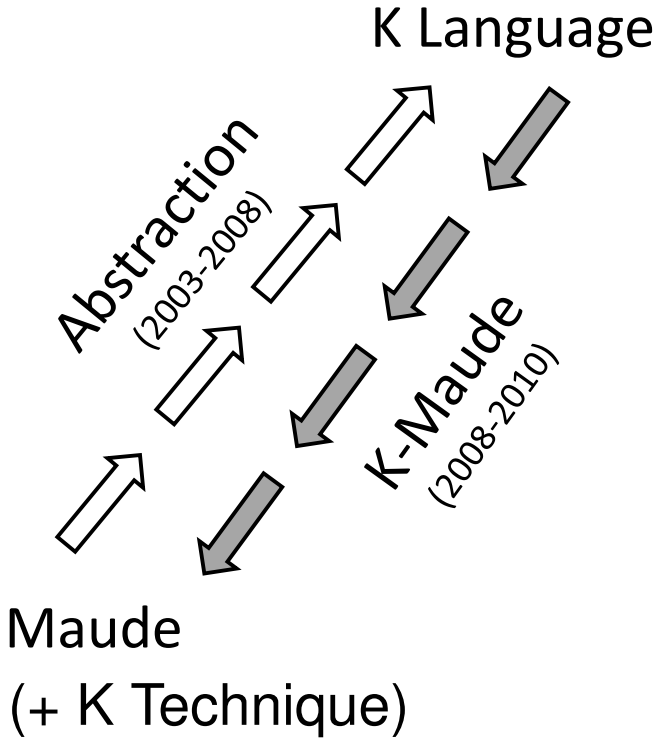
A Rewriting Based Tool for Semantics of Programming Languages

Traian Florin Șerbănuță and Grigore Roșu

University of Illinois at Urbana-Champaign

WRLA 2010

Overview



K-Maude: A prototype implementation of the K framework in Maude.

- The K framework consists of:
  - A rewriting technique for defining programming lanugages
  - A specialized language to simplify (and enhance) the K technique
- K-Maude extends Maude language to support the K language

Rewriting Logic Semantics Project

Aim: Use RWL to define (and analyze) programming languages

- Rewriting logic faithfully captures PL semantic frameworks
  - SOS [*Verdejo,Martí-Olliet,Meseguer*], [*Meseguer,Braga*]
  - Chemical Abstract Machine [*Meseguer*]
  - MSOS [*Meseguer,Braga*], the Maude MSOS tool [*Braga,Chalub*]
  - Reduction semantics with evaluation contexts [*Șerbănuță,Roșu,Meseguer*]
- However, they are captured together with their limitations

Motivational question:

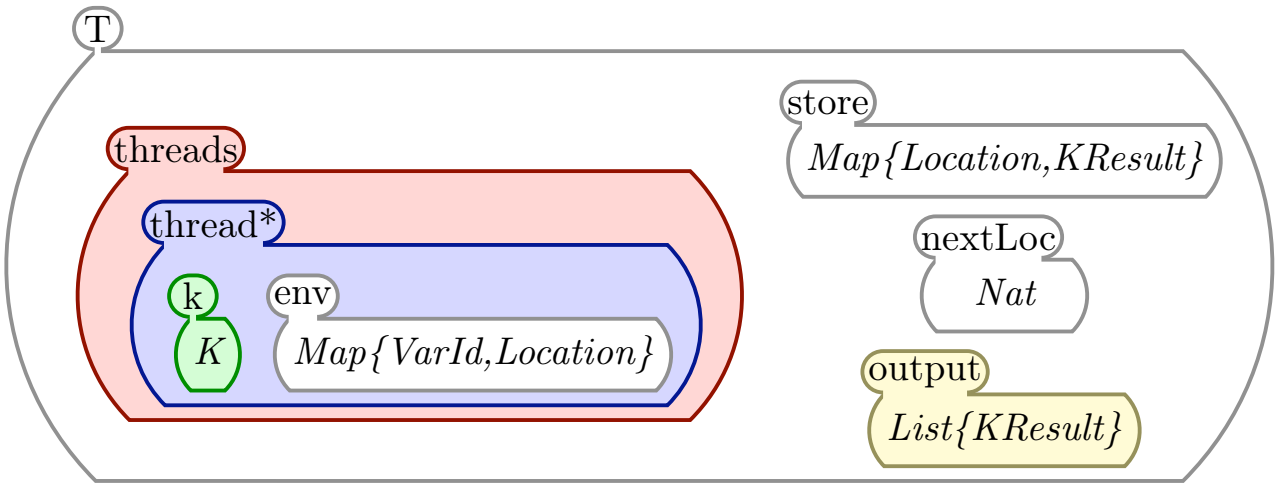
Can RWL inspire a better language definitional framework?

The K Technique

- Flexible, extensible, configurations as multi-sets of nested cells
  - Cells can contain (Multi-)Sets, Lists, Maps, or computations
- Computation as a list of $\curvearrowright$-separated tasks
  - Next task always at top of the list
  - Easy to define control-intensive features like halt,call/cc
- Rewriting modulo ACU to improve modularity
  - Specify only what is needed from a cell for a semantic rule
  - Abstract the remainder of the cell with variables
- Was used to give definitions in Maude for Java 1.4, Scheme, Beta

Memory access in a multi-threaded language

Configuration



Variable Lookup

$$\langle\langle\langle X \curvearrowright K\rangle_{\mathsf{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathsf{env}}\ T\rangle_{\mathsf{thread}}\ Ts\rangle_{\mathsf{threads}}\ \langle L \mapsto V\ \sigma\rangle_{\mathsf{store}}$$
$$\rightarrow\quad \langle\langle\langle V \curvearrowright K\rangle_{\mathsf{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathsf{env}}\ T\rangle_{\mathsf{thread}}\ Ts\rangle_{\mathsf{threads}}\ \langle L \mapsto V\ \sigma\rangle_{\mathsf{store}}$$

Memory access in a multi-threaded language

Variable Lookup

$$\begin{array}{rl} & \langle\langle\langle X \curvearrowright K\rangle_{\mathsf{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathsf{env}}\ T\rangle_{\mathsf{thread}}\ Ts\rangle_{\mathsf{threads}}\ \langle L \mapsto V\ \sigma\rangle_{\mathsf{store}} \\ \rightarrow & \langle\langle\langle V \curvearrowright K\rangle_{\mathsf{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathsf{env}}\ T\rangle_{\mathsf{thread}}\ Ts\rangle_{\mathsf{threads}}\ \langle L \mapsto V\ \sigma\rangle_{\mathsf{store}} \end{array}$$

Assignment

- Executing the assignment

$$\begin{array}{rl} & \langle\langle\langle X := V \curvearrowright K\rangle_{\mathsf{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathsf{env}}\ T\rangle_{\mathsf{thread}}\ Ts\rangle_{\mathsf{threads}}\ \langle L \mapsto V'\ \sigma\rangle_{\mathsf{store}} \\ \rightarrow & \langle\langle\langle\ \ K\rangle_{\mathsf{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathsf{env}}\ T\rangle_{\mathsf{thread}}\ Ts\rangle_{\mathsf{threads}}\ \langle L \mapsto V\ \ \sigma\rangle_{\mathsf{store}} \end{array}$$

Memory access in a multi-threaded language

Variable Lookup

$$\begin{array}{rl} & \langle\langle\langle X \curvearrowright K\rangle_{\mathsf{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathsf{env}}\ T\rangle_{\mathsf{thread}}\ Ts\rangle_{\mathsf{threads}}\ \langle L \mapsto V\ \sigma\rangle_{\mathsf{store}} \\ \rightarrow & \langle\langle\langle V \curvearrowright K\rangle_{\mathsf{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathsf{env}}\ T\rangle_{\mathsf{thread}}\ Ts\rangle_{\mathsf{threads}}\ \langle L \mapsto V\ \sigma\rangle_{\mathsf{store}} \end{array}$$

Assignment

- Executing the assignment

$$\begin{array}{rl} & \langle\langle\langle X := V \curvearrowright K\rangle_{\mathsf{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathsf{env}}\ T\rangle_{\mathsf{thread}}\ Ts\rangle_{\mathsf{threads}}\ \langle L \mapsto V'\ \sigma\rangle_{\mathsf{store}} \\ \rightarrow & \langle\langle\langle\ \ K\rangle_{\mathsf{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathsf{env}}\ T\rangle_{\mathsf{thread}}\ Ts\rangle_{\mathsf{threads}}\ \langle L \mapsto V\ \ \sigma\rangle_{\mathsf{store}} \end{array}$$

- Evaluating the right hand side (strictness)

$$\langle X := Ke \curvearrowright K\rangle_{\mathsf{k}} \rightharpoonup \langle Ke \curvearrowright X := [\,]\ \curvearrowright K\rangle_{\mathsf{k}} \text{ if } not(Ke :: KResult)$$
$$\langle X := Ke \curvearrowright K\rangle_{\mathsf{k}} \leftharpoondown \langle Ke \curvearrowright X := [\,]\ \curvearrowright K\rangle_{\mathsf{k}} \text{ if } Ke :: KResult$$

The K Language

Aim: Optimize the language definition process

- Automation: generate common fixed form rules from annotations
- Conciseness: redundancy might lead to copy errors
- Modularity: language features definitions resilient through structural changes in the configuration
- Simplicity: intuitive and avoiding unnecessary encodings

Automation: reduce strictness rules to annotations

Problem: Strictness rules are very common and very mechanical

- Most expressions are strict in their defining context

$\langle X := Ke \curvearrowright K \rangle_k \rightharpoonup \langle Ke \curvearrowright X := [\,] \curvearrowright K \rangle_k$ if $not(Ke :: KResult)$

$\langle X := Ke \curvearrowright K \rangle_k \leftharpoondown \langle Ke \curvearrowright X := [\,] \curvearrowright K \rangle_k$ if $Ke :: KResult$

Solution: Automatically generate strictness rules from annotations

$$Stmt ::= VarId := AExp \; [strict(2)]$$

Conciseness: in-place rewriting

Problem: Redundancy — rhs has to re-copy things matched by the lhs

$$\begin{array}{rl} & \langle\langle\langle X := V \curvearrowright K\rangle_{\mathrm{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathrm{env}}\ T\rangle_{\mathrm{thread}}\ Ts\rangle_{\mathrm{threads}}\ \langle L \mapsto V'\ \sigma\rangle_{\mathrm{store}} \\ \rightarrow & \langle\langle\langle \qquad\qquad K\rangle_{\mathrm{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathrm{env}}\ T\rangle_{\mathrm{thread}}\ Ts\rangle_{\mathrm{threads}}\ \langle L \mapsto V\ \ \sigma\rangle_{\mathrm{store}} \end{array}$$

Solution: Make rules more local

$$\langle\langle\langle \underbrace{X := V}_{\cdot} \curvearrowright K\rangle_{\mathrm{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathrm{env}}\ T\rangle_{\mathrm{thread}}\ Ts\rangle_{\mathrm{threads}}\ \langle L \mapsto \underbrace{V'}_{V}\ \sigma\rangle_{\mathrm{store}}$$

Modularity: Context Transformers

Problem: Changes of configuration structure inflict unneeded changes in semantic rules

$$\langle\langle\langle \underline{X := V} \curvearrowright K\rangle_{\mathsf{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathsf{env}}\ T\rangle_{\mathsf{thread}}\ Ts\rangle_{\mathsf{threads}}\ \langle L \mapsto \underline{V'}\ \sigma\rangle_{\mathsf{store}}$$
$$\cdot \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad V$$

Solution: Liberate rules from structural constraints

- Use a global configuration structure...

$$\langle\langle\langle\langle K\rangle_{\mathsf{k}}\ \langle Map\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}*}\rangle_{\mathsf{threads}}\langle Map\rangle_{\mathsf{store}}\langle Nat\rangle_{\mathsf{nextLoc}}\langle List\rangle_{\mathsf{output}}\rangle_{\mathsf{T}}$$

- ...to infer missing context for all rules in the definition

$$\langle \underline{X := V} \curvearrowright K\rangle_{\mathsf{k}}\ \langle X \mapsto L\ \rho\rangle_{\mathsf{env}}\ \langle L \mapsto \underline{V'}\ \sigma\rangle_{\mathsf{store}}$$
$$\cdot \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad V$$

Simplicity: Get rid of contextual (named) variables

Problem: Variables unused by the rule are unnecessary

$$\langle \underline{X := V} \curvearrowright K \rangle_{\text{k}} \langle X \mapsto L\ \rho \rangle_{\text{env}} \langle L \mapsto \underline{V'}\ \sigma \rangle_{\text{store}}$$

Solution: Abstract cell variables using the structure of the cell. Abstract other unused variables to anonymous variables

$$\langle \underline{X := V} \cdots \rangle_{\text{k}} \langle \cdots\ X \mapsto L\ \cdots \rangle_{\text{env}} \langle \cdots\ L \mapsto \underline{\_}\ \cdots \rangle_{\text{store}}$$

All in one step

Assignment using plain K-technique in RWL

- Strictness
  $\langle X := Ke \curvearrowright K \rangle_k \rightharpoonup \langle Ke \curvearrowright X := [\,] \curvearrowright K \rangle_k$ if $not(Ke :: KResult)$
  $\langle X := Ke \curvearrowright K \rangle_k \leftharpoondown \langle Ke \curvearrowright X := [\,] \curvearrowright K \rangle_k$ if $Ke :: KResult$
- Executing the assignment

$$\langle\langle\langle X := V \curvearrowright K\rangle_k \langle X \mapsto L\ \rho\rangle_{env}\ T\rangle_{thread}\ Ts\rangle_{threads} \langle L \mapsto V'\ \sigma\rangle_{store}$$
$$\rightarrow \quad \langle\langle\langle \qquad\qquad K\rangle_k \langle X \mapsto L\ \rho\rangle_{env}\ T\rangle_{thread}\ Ts\rangle_{threads} \langle L \mapsto V\ \ \sigma\rangle_{store}$$

Assignment using the K language

- Strictness: $Stmt ::= VarId := AExp\ [strict(2)]$
- Executing the assignment

$$\langle \underline{X := V}_{\cdot} \ \cdots\rangle_k \langle \cdots\ X \mapsto L\ \cdots\rangle_{env} \langle \cdots\ L \mapsto \underline{\_}_{V}\ \cdots\rangle_{store}$$

- Using global configuration term
  $\langle\langle\langle\langle K\rangle_k\ \langle Map\rangle_{env}\rangle_{thread*}\rangle_{threads}\langle Map\rangle_{store}\langle Nat\rangle_{nextLoc}\langle List\rangle_{output}\rangle_T$

IMP++ Annotated Syntax

AExp	::=	*Int* | *VarId*	
	|	*AExp* + *AExp*	[*strict*]
	|	*AExp* / *AExp*	[*strict*]
BExp	::=	*Bool*	
	|	*AExp* <= *AExp*	[*seqstrict*]
	|	`not` *BExp*	[*strict*]
	|	*BExp* `and` *BExp*	[*strict*(1)]
Stmt	::=	`skip` | *Stmt* `;` *Stmt*	
	|	*VarId* `:=` *AExp*	[*strict*(2)]
	|	`if` *BExp* `then` *Stmt* `else` *Stmt*	[*strict*(1)]
	|	`while` *BExp* `do` *Stmt*	
	|	`print` *AExp*	[*strict*]
	|	`var` *VarId* `;` *Stmt*	

Intuition for strictness

$$\begin{array}{rcll} \mathit{Stmt} & ::= & \mathit{VarId} := \mathit{AExp} & [\mathit{strict}(2)] \\ \mathit{AExp} & ::= & \mathit{AExp} + \mathit{AExp} & [\mathit{strict}] \end{array}$$

- Corresponding to descent rules in SOS

$$\frac{\langle A, \sigma \rangle \rightarrow \langle A', \sigma' \rangle}{\langle X := A, \sigma \rangle \rightarrow X := A', \sigma' \rangle}$$

- Corresponding to evaluation contexts in reduction semantics

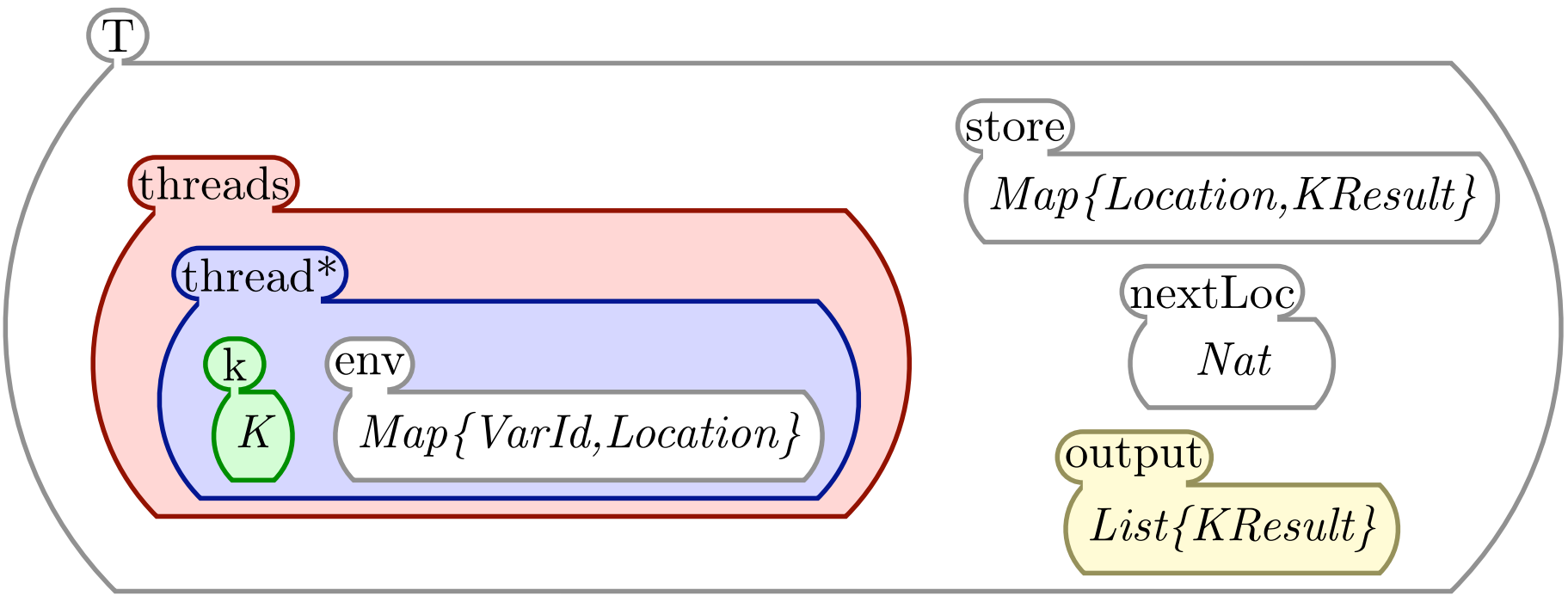
$$\begin{array}{rcl} C_{\mathit{AExp}} & ::= & C_{\mathit{AExp}} + \mathit{AExp} \\ & | & \mathit{AExp} + C_{\mathit{AExp}} \end{array}$$

IMP Semantics (I)

$$i_1 + i_2 \rightarrow i_1 +_{Int} i_2$$
$$i_1 \ / \ i_2 \rightarrow i_1 /_{Int} i_2 \quad \text{where } i_1 \neq 0$$
$$i_1 <= i_2 \rightarrow i_1 \leq_{Int} i_2$$
$$\texttt{not}\ t \rightarrow \neg_{Bool}\ t$$
$$\mathit{true}\ \texttt{and}\ b \rightarrow b$$
$$\mathit{false}\ \texttt{and}\ b \rightarrow \texttt{false}$$
$$\texttt{skip} \rightarrow \cdot$$
$$s_1\ \texttt{;}\ s_2 \rightharpoonup s_1 \curvearrowright s_2$$
$$\texttt{if}\ \mathit{true}\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \rightarrow s_1$$
$$\texttt{if}\ \mathit{false}\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \rightarrow s_2$$

IMP++ Configuration

$\langle\langle\langle\langle K\rangle_{\text{k}}\ \langle Map\rangle_{\text{env}}\rangle_{\text{thread}*}\rangle_{\text{threads}}\langle Map\rangle_{\text{store}}\langle Nat\rangle_{\text{nextLoc}}\langle List\rangle_{\text{output}}\rangle_{\text{T}}$



IMP Semantics (II)

$$\langle \underline{x} \cdots \rangle_{\mathsf{k}} \langle \cdots\ x \mapsto l \cdots \rangle_{\mathsf{env}} \langle \cdots\ l \mapsto v \cdots \rangle_{\mathsf{store}}$$
$$\overline{i}$$

$$\langle \underline{x := v} \cdots \rangle_{\mathsf{k}} \langle \cdots\ x \mapsto l \cdots \rangle_{\mathsf{env}} \langle \cdots\ l \mapsto \_ \cdots \rangle_{\mathsf{store}}$$
$$\cdot \qquad\qquad \overline{v}$$

$$\langle \underline{\texttt{while}\ b\ \texttt{do}\ s} \cdots \rangle_{\mathsf{k}}$$
$$\texttt{if}\ b\ \texttt{then}\ (s\ ;\ \texttt{while}\ b\ \texttt{do}\ s)\ \texttt{else}\ \cdot$$

$$\langle \texttt{print}\ v \cdots \rangle_{\mathsf{k}} \langle \cdots\ \underline{\cdot} \rangle_{\mathsf{out}}$$
$$\overline{v}$$

$$\langle \underline{\texttt{var}\ x\ ;\ s} \cdots \rangle_{\mathsf{k}} \langle \underline{\rho} \rangle_{\mathsf{env}} \langle \cdots\ \underline{\cdot} \cdots \rangle_{\mathsf{store}} \langle \underline{l} \rangle_{\mathsf{nextLoc}}$$
$$s \curvearrowright env(\rho) \qquad \rho[l/x] \qquad l \mapsto 0 \qquad l + 1$$

$$\langle \underline{env(\rho)} \cdots \rangle_{\mathsf{k}} \langle \underline{\_} \rangle_{\mathsf{env}}$$
$$\cdot \qquad\qquad \overline{\rho}$$

K-Maude in a nutshell

- Provides infrastructure for the K technique in Maude
  - Defines modules for computations, lists, sets, maps, and configurations
- Defines a way to specify K language definitions in Maude
  - Non executable, but abiding to the Maude language syntax
- Compiles K language definitions using the K technique in Maude
  - Obtained definitions are executable and amenable to search/model checking

IMP language in K-Maude — what to look for

Syntax

- Strictness annotations as metadata
- All syntax is flattened into the computation sort K

Semantics

- Structure of configuration declared as a term
  - In current version using mb configuation _ : KSentence .[1]
- K (structural) and (computational) rules
  - In current version using mb rule _ : KSentence .[1]
  - structural rules distinguished by metadata
  - K in-place rewriting $\frac{L}{R}$ is written in K-Maude as $[L \Rightarrow R]$
- Anonymous variable ? and open-ended cells ...

[1] Gray part omitted in the sequel

Syntax

```
sorts AExp BExp Stmt Pgm .
subsort VarId Int < AExp .
op _+_ : AExp AExp ->AExp [prec 33 gather (E e) metadata "strict"] .
op _/_ : AExp AExp ->AExp [prec 31 gather (E e) metadata "strict"] .
subsort Bool < BExp .
op _<=_ : AExp AExp ->BExp [prec 37 metadata "seqstrict"] .
op not_ : BExp -> BExp [prec 53 metadata "strict"] .
op _and_ : BExp BExp ->BExp [prec 55 metadata "strict(1)"] .
op skip : -> Stmt .
op _:=_ : VarId AExp -> Stmt [prec 40 metadata "strict(2)"] .
op _;_ : Stmt Stmt -> Stmt [prec 60 gather (e E)] .
op if_then_else_ : BExp Stmt Stmt ->Stmt [prec 59 metadata "strict(1)"] .
op while_do_ : BExp Stmt ->Stmt [prec 59] .
ops print_ : AExp -> Stmt [prec 59 metadata "strict" format (b o d)] .
op var_;_ : VarId Stmt -> Stmt [prec 70] .
```

Syntax Becomes Computation

- All syntactic sorts are made part of the computation sort

  subsort AExp BExp Stmt Pgm ⟨*K* .

- Values need to be distinguised as result computations

  subsorts Int Bool ⟨ *KResult* .

Semantics (I)

```
rule [I1 + I2 ⇒ I1 +Int I2] .
rule [I1 / I2 ⇒ I1 /Int I2] if I2 =/=Bool 0 .
rule [I1 <= I2 ⇒ I1 <=Int I2] .
rule [not(T) ⇒ notBool T] .
rule [true and B ⇒ B] .
rule [false and ? ⇒ false] .
rule [skip ⇒ .k] .
rule [S1 ; S2 ⇒ S1 ↷ S2] .
rule [if true then S1 else ? ⇒S1] .
rule [if false then ? else S2 ⇒S2] .
```

Configuration

- Configuration is a (multi-level) (multi-)set of cells
- A cell ⟨_⟩_⟨/_⟩ has a *CellLabel* as identifier (same on both ends)
- A cell can contain either a computation K, a Map, a Set, or a List.
- In configuration terms "★" denotes potential multiplicity

```
var Sigma : Map . var K : K .
ops output threads thread env store nextLoc : → CellLabel .
configuration
  ⟨ T ⟩ ⟨ threads ⟩ ⟨ thread * ⟩
          ⟨ k ⟩ K ⟨/ k ⟩ ⟨ env ⟩ Env ⟨/ env ⟩
        ⟨/ thread * ⟩ ⟨/ threads ⟩
        ⟨ store ⟩ Store ⟨/ store ⟩ ⟨ nextLoc ⟩ N ⟨/ nextLoc ⟩
        ⟨ output ⟩ Output ⟨/ output ⟩
  ⟨/ T ⟩ .
```

Semantics (II)

rule ⟨ k ⟩ [X ⇒ I] ...⟨/ k ⟩ ⟨ env ⟩...X ↦L ...⟨/ env ⟩
⟨ store ⟩...L ↦ I ...⟨/ store ⟩ .
rule ⟨ k ⟩ [X := I ⇒ .k] ...⟨/ k ⟩ ⟨ env ⟩...X ↦L ...⟨/ env ⟩
⟨ store ⟩...L ↦ [? ⇒ I] ...⟨/ store ⟩ .
rule ⟨ k ⟩[`while` B `do` S ⇒ `if` B `then` S ; `while` B `do` S `else` `skip`]...⟨/ k ⟩.
rule ⟨ k ⟩[`print`(I) ⇒ .k]...⟨/ k ⟩ ⟨ output ⟩...[.*I* ⇒ *I*(I)] ⟨/ output ⟩ .
rule ⟨ k ⟩[**var** X ; S ⇒ S ↷ env(Env)]...⟨/ k ⟩
⟨ env ⟩[Env ⇒ Env[N / X]]⟨/ env ⟩ ⟨ store ⟩...[.m ⇒ N ↦ 0]...⟨/ store ⟩
⟨ nextLoc ⟩[N ⇒ N +Int 1]⟨/ nextLoc ⟩ .
rule ⟨ k ⟩[env(Env) ⇒ .k]...⟨/ k ⟩ ⟨ env ⟩[? ⇒ Env]⟨/ env ⟩ .

Compilation process

- K-Maude takes a Maude module containing a K definition . . .
- . . . and (meta-)transforms it into an executable Maude module implementing the desired definition
- Compilation split in small steps
  - It applies in reverse order all K-language transformations.
- Each step takes a Maude module and produces a Maude module

The assignment rule (this time in K-Maude)

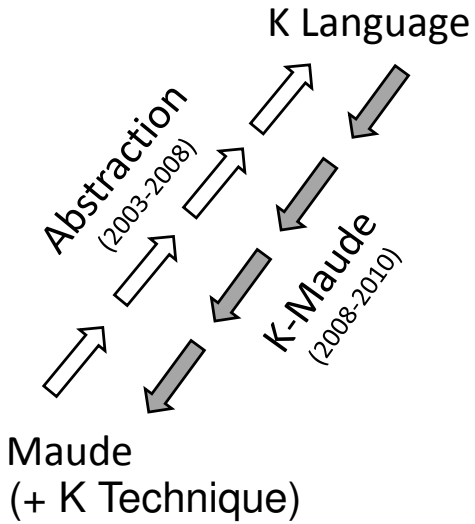
Original K-Maude rule:

rule ⟨ k ⟩ [X := I ⇒ .k] ...⟨/ k ⟩ ⟨ env ⟩...X ↦L ...⟨/ env ⟩
⟨ store ⟩...L ↦ [? ⇒ I] ...⟨/ store ⟩ .

Generated (executable, abstract) Maude rule

rl ⟨ threads ⟩ ?13:*Set* ⟨ thread ⟩ ?14:*Set*
⟨ k ⟩ "_:=_"(("VarId" X:VarId)(. kl),(" Int " I : Int)(. kl)) ⇝ ?15:*K* ⟨/ k ⟩
⟨ env ⟩ ?16:*Map* ("VarId" X:VarId)(. kl) ↦ (" Int " L: Int)(. kl) ⟨/ env ⟩
⟨/ thread ⟩ ⟨/ threads ⟩
⟨ store ⟩ ?17:*Map* ("Int" L: Int)(. kl) ↦ ?18:*K* ⟨/ store ⟩
⇒ ⟨ threads ⟩ ?13:*Set* ⟨ thread ⟩ ?14:*Set*
⟨ k ⟩ ?15:*K* ⟨/ k ⟩
⟨ env ⟩ ?16:*Map* ("VarId" X:VarId)(. kl) ↦ (" Int " L: Int)(. kl) ⟨/ env ⟩
⟨/ thread ⟩ ⟨/ threads ⟩
⟨ store ⟩ ?17:*Map* ("Int" L: Int)(. kl) ↦ (" Int " I : Int)(. kl) ⟨/ store ⟩
[metadata "computational **rule**"] .

Conclusions



- K framework:
  - Takes the strengths of other PL semantic frameworks...
  - ...and adds to them the strength of rewriting.
- K-Maude:
  - Faithful implementation of the K Language in Maude
  - Compiles K definitions into executable/analyzable Maude definitions
- Future Work:
  - (Re-)integrate with Full Maude
  - Backward translations to give answers and errors at the top level

Required modules

including PL–INT . ––– Builtin Integers
including PL–VARID . ––– Builtin Variable identifiers

including *K* . ––– Basic structure of the computation

including CONFIG . ––– Predefined cells and **configuration** infrastructure

including *K*–RULES . ––– *K* rules and in–place rewriting

including *K*–CONFIG . ––– to allow specifying **configuration** as a term

including *K*–ANONYMOUS–VARIABLES . ––– introduces anonymous variabl

including *K*–OPEN–CELLS . ––– to use open cells lik ⟨_⟩. . ._⟨_⟩

including *K*–MAP–EXTRAS . ––– to do bulk assigments in the map

What K-Maude does?

- K-Maude takes a Maude module containing a K definition . . .
- . . . and transforms it into an executable Maude module implementing the desired definition
- Compilation split in small steps
- Each step takes a Maude module and produces a Maude Module

Case study: Assignment in a multi-threaded language

Original K-Maude definition

```
sorts AExp Stmt . subsort VarId Int ⟨ AExp .
subsorts AExp Stmt ⟨K . subsort Int ⟨ KResult .
op _:=_ : VarId AExp → Stmt [prec 40 metadata "strict(2)"] .
mb rule ⟨ k ⟩ [X:VarId := I:Int ⇒ .k] ...⟨/ k ⟩
        ⟨ env ⟩... X:VarId ↦L:Int ...⟨/ env ⟩
        ⟨ store ⟩... L:Int ↦[? ⇒ I:Int] ...⟨/ store ⟩
        : KSentence .

mb configuration
  ⟨ T ⟩ ⟨ threads ⟩ ⟨ thread * ⟩
          ⟨ k ⟩ K:K ⟨/ k ⟩ ⟨ env ⟩ Env:Map ⟨/ env ⟩
        ⟨/ thread * ⟩ ⟨/ threads ⟩
        ⟨ store ⟩ Store:Map ⟨/ store ⟩ ⟨ nextLoc ⟩ N:Nat ⟨/ nextLoc ⟩
        ⟨ output ⟩ Output:List ⟨/ output ⟩
  ⟨/ T ⟩ ⟨ result ⟩ Output:List ⟨/ result ⟩
  : KSentence .
```

Resolving opened cells

```
sorts AExp Stmt . subsort VarId Int < AExp .
subsorts AExp Stmt <K . subsort Int < KResult .
op _:=_ : VarId AExp -> Stmt [prec 40 metadata "strict(2)"] .
mb rule < k > [X:VarId := I:Int => .k] ...</ k >
        < env >... X:VarId |->L:Int ...</ env >
        < store >... L:Int |->[? => I:Int] ...</ store >
        : KSentence .

mb configuration
  < T > < threads > < thread * >
            < k > K:K </ k > < env > Env:Map </ env >
        </ thread * > </ threads >
        < store > Store:Map </ store > < nextLoc > N:Nat </ nextLoc >
        < output > Output:List </ output >
  </ T > < result > Output:List </ result >
  : KSentence .
```

Opened cells resolved

```
sorts AExp Stmt . subsort VarId Int < AExp .
subsorts AExp Stmt < K . subsort Int < KResult .
op _:=_ : VarId AExp -> Stmt [prec 40 metadata "strict(2)"] .
mb rule < k > [X:VarId := I:Int => .k] ~> ?:K </ k >
        < env > ?:Map X:VarId |-> L:Int </ env >
        < store > ?:Map L:Int |-> [? => I:Int] </ store >
        : KSentence .

mb configuration
  < T > < threads > < thread * >
            < k > K:K </ k > < env > Env:Map </ env >
        </ thread * > </ threads >
        < store > Store:Map </ store > < nextLoc > N:Nat </ nextLoc >
        < output > Output:List </ output >
  </ T > < result > Output:List </ result >
  : KSentence .
```

Turning anonymous constants into anonymous variables

```
sorts AExp Stmt . subsort VarId Int < AExp .
subsorts AExp Stmt < K . subsort Int < KResult .
op _:=_ : VarId AExp -> Stmt [prec 40 metadata "strict(2)"] .
mb rule <k> [X:VarId := I:Int => .k] ~> ?:K </k>
        <env> ?:Map X:VarId |-> L:Int </env>
        <store> ?:Map L:Int |-> [? => I:Int] </store>
        : KSentence .

mb configuration
  <T> <threads> <thread>* <k> K:K </k> <env> Env:Map </env>
                </thread>* </threads>
      <store> Store:Map </store> <nextLoc> N:Nat </nextLoc>
      <output> Output:List </output>
  </T> <result> Output:List </result>
  : KSentence .
```

No more anonymous constants

```
sorts AExp Stmt . subsort VarId Int ⟨ AExp .
subsorts AExp Stmt ⟨K . subsort Int ⟨KResult .
op _:=_ : VarId AExp → Stmt [prec 40 metadata "strict(2)"] .
mb rule ⟨ k ⟩ [X:VarId := I:Int ⇒ .k] ↷ ?:K ⟨/ k ⟩
        ⟨ env ⟩ ?:Map X:VarId ↦L:Int ⟨/ env ⟩
        ⟨ store ⟩ ?:Map L:Int ↦ [?:Int ⇒ I:Int] ⟨/ store ⟩
        : KSentence .

mb configuration
  ⟨ T ⟩ ⟨ threads ⟩ ⟨ thread * ⟩
          ⟨ k ⟩ K:K ⟨/ k ⟩ ⟨ env ⟩ Env:Map ⟨/ env ⟩
        ⟨/ thread * ⟩ ⟨/ threads ⟩
        ⟨ store ⟩ Store:Map ⟨/ store ⟩ ⟨ nextLoc ⟩ N:Nat ⟨/ nextLoc ⟩
        ⟨ output ⟩ Output:List ⟨/ output ⟩
  ⟨/ T ⟩ ⟨ result ⟩ Output:List ⟨/ result ⟩
  : KSentence .
```

Merging syntax sorts into K

```
sorts AExp Stmt . subsort VarId Int < AExp .
subsorts AExp Stmt < K. subsort Int <KResult .
op _:=_ : VarId AExp ->Stmt [prec 40 metadata "strict(2)"] .
mb rule <k> [X:VarId := I:Int => .k] ~> ?:K </k>
        <env> ?:Map X:VarId |->L:Int </env>
        <store> ?:Map L:Int |-> [?:Int => I:Int] </store>
        : KSentence .

mb configuration
  <T> <threads> <thread>* <k> K:K </k> <env> Env:Map </env>
      </thread>* </threads>
      <store> Store:Map </store> <nextLoc> N:Nat </nextLoc>
      <output> Output:List </output>
  </T> <result> Output:List </result>
  : KSentence .
```

Syntax sorts are gone

```
subsort VarId Int ⟨K .
subsort Int ⟨ KResult .
op _:=_ : VarId K → K [prec 40 metadata "strict(2)"] .
mb rule ⟨ k ⟩ [X:VarId := I:Int ⇒ .k] ↷ ?:K ⟨/ k ⟩
        ⟨ env ⟩ ?:Map X:VarId ↦L:Int ⟨/ env ⟩
        ⟨ store ⟩ ?:Map L:Int ↦ [?:Int ⇒ I:Int] ⟨/ store ⟩
        : KSentence .

mb configuration
  ⟨ T ⟩ ⟨ threads ⟩ ⟨ thread * ⟩
          ⟨ k ⟩ K:K ⟨/ k ⟩ ⟨ env ⟩ Env:Map ⟨/ env ⟩
        ⟨/ thread * ⟩ ⟨/ threads ⟩
        ⟨ store ⟩ Store:Map ⟨/ store ⟩ ⟨ nextLoc ⟩ N:Nat ⟨/ nextLoc ⟩
        ⟨ output ⟩ Output:List ⟨/ output ⟩
  ⟨/ T ⟩ ⟨ result ⟩ Output:List ⟨/ result ⟩
  : KSentence .
```

Adding Proper computations

```
subsort VarId Int ⟨K .
subsort Int ⟨ KResult .
op _:=_ : VarId K → K [prec 40 metadata "strict(2)"] .
mb rule ⟨ k ⟩ [X:VarId := I:Int ⇒ .k] ↷ ?:K ⟨/ k ⟩
        ⟨ env ⟩ ?:Map X:VarId ↦L:Int ⟨/ env ⟩
        ⟨ store ⟩ ?:Map L:Int ↦ [?:Int ⇒ I:Int] ⟨/ store ⟩
        : KSentence .

mb configuration
  ⟨ T ⟩ ⟨ threads ⟩ ⟨ thread * ⟩
          ⟨ k ⟩ K:K ⟨/ k ⟩ ⟨ env ⟩ Env:Map ⟨/ env ⟩
        ⟨/ thread * ⟩ ⟨/ threads ⟩
        ⟨ store ⟩ Store:Map ⟨/ store ⟩ ⟨ nextLoc ⟩ N:Nat ⟨/ nextLoc ⟩
        ⟨ output ⟩ Output:List ⟨/ output ⟩
  ⟨/ T ⟩ ⟨ result ⟩ Output:List ⟨/ result ⟩
  : KSentence .
```

All non-result computations are proper

```
subsort VarId < KProper .
subsort Int < KResult .
op _:=_ : VarId K -> KProper [prec 40 metadata "strict(2)"] .
mb rule <k> [X:VarId := I:Int => .k] ~> ?:K </k>
        <env> ?:Map X:VarId |-> L:Int </env>
        <store> ?:Map L:Int |-> [?:Int => I:Int] </store>
        : KSentence .

mb configuration
  <T> <threads> <thread>* </k> K:K </k> <env> Env:Map </env>
      </thread>* </threads>
      <store> Store:Map </store> <nextLoc> N:Nat </nextLoc>
      <output> Output:List </output>
  </T> <result> Output:List </result>
  : KSentence .
```

Inferring missing configuration

```
subsort VarId < KProper .
subsort Int < KResult .
op _:=_ : VarId K -> KProper [prec 40 metadata "strict(2)"] .

mb rule <k> [X:VarId := I:Int => .k] ~> ?:K </k>
        <env> ?:Map X:VarId |-> L:Int </env>
        <store> ?:Map L:Int |-> [?:Int => I:Int] </store>
        : KSentence .

mb configuration
  <T> <threads> <thread>* </k> K:K </k> <env> Env:Map </env>
      </thread>* </threads>
      <store> Store:Map </store> <nextLoc> N:Nat </nextLoc>
      <output> Output:List </output>
  </T> <result> Output:List </result>
  : KSentence .
```

Configuration no longer needed

```
subsort VarId < KProper .
subsort Int < KResult .
op _:=_ : VarId K -> KProper [prec 40 metadata "strict(2)"] .

mb rule <threads> ?:Set
          <thread> ?:Set
            <k> [X:VarId := I:Int => .k] ~> ?:K </k>
            <env> ?:Map X:VarId |-> L:Int </env>
          </thread>
        </threads>
        <store> ?:Map L:Int |-> [?:Int => I:Int] </store>
        : KSentence .
```

Resolve anonymous variables

```
subsort VarId ⟨ KProper .
subsort Int ⟨ KResult .
op _:=_ : VarId K → KProper [prec 40 metadata "strict(2)"] .

mb rule ⟨ threads ⟩ ?:Set
          ⟨ thread ⟩ ?:Set
            ⟨ k ⟩ [X:VarId := I:Int ⇒ .k] ↷ ?:K ⟨/ k ⟩
            ⟨ env ⟩ ?:Map X:VarId ↦L:Int ⟨/ env ⟩
          ⟨/ thread ⟩
        ⟨/ threads ⟩
        ⟨ store ⟩ ?:Map L:Int ↦[?:Int ⇒I:Int] ⟨/ store ⟩
        : KSentence .
```

Anonymous variables become real variables

```
subsort VarId < KProper .
subsort Int < KResult .
op _:=_ : VarId K -> KProper [prec 40 metadata "strict(2)"] .

mb rule <threads> ?13:Set
          <thread> ?14:Set
            <k> [X:VarId := I:Int => .k] ~> ?15:K </k>
            <env> ?16:Map X:VarId |-> L:Int </env>
          </thread>
        </threads>
        <store> ?17:Map L:Int |-> [?18:Int => I:Int] </store>
        : KSentence .
```

Generating Maude rules from K rules

```
subsort VarId < KProper .
subsort Int < KResult .
op _:=_ : VarId K -> KProper [prec 40 metadata "strict(2)"] .

mb rule < threads > ?13:Set
          < thread > ?14:Set
            < k > [X:VarId := I:Int => .k] ~> ?15:K </ k >
            < env > ?16:Map X:VarId |-> L:Int </ env >
          </ thread >
        </ threads >
        < store > ?17:Map L:Int |-> [?18:Int => I:Int] </ store >
        : KSentence .
```

K rules are gone

```
subsort VarId < KProper .
subsort Int < KResult .
op _:=_ : VarId K -> KProper [prec 40 metadata "strict(2)"] .

rl < threads > ?13:Set < thread > ?14:Set
       < k > X:VarId := I:Int ~> ?15:K </ k >
       < env > ?16:Map X:VarId |-> L:Int </ env >
   </ thread > </ threads >
   < store > ?17:Map L:Int |-> ?18:Int </ store >
=> < threads > ?13:Set < thread > ?14:Set
       < k > .k ~> ?15:K </ k >
       < env > ?16:Map X:VarId |-> L:Int </ env >
   </ thread > </ threads >
   < store > ?17:Map L:Int |-> I:Int </ store > .
```

. . . K abstract syntax

```
op VarId : VarId → KProperLabel .
op Int : Int → KResultLabel .
op "_:=_" : → KProperLabel [prec 40 metadata "arity 2 strict (2)"] .

rl ⟨ threads ⟩ ?13:Set ⟨ thread ⟩ ?14:Set
       ⟨ k ⟩ "_:=_"(VarId(X:VarId)(.kl), Int(I:Int)(.kl)) ↷ ?15:K ⟨/ k ⟩
       ⟨ env ⟩ ?16:Map VarId(X:VarId)(.kl) ↦ Int(L:Int)(.kl) ⟨/ env ⟩
   ⟨/ thread ⟩ ⟨/ threads ⟩
   ⟨ store ⟩ ?17:Map Int(L:Int)(.kl) ↦ Int(?18:Int)(.kl) ⟨/ store ⟩
⇒ ⟨ threads ⟩ ?13:Set ⟨ thread ⟩ ?14:Set
       ⟨ k ⟩ .k ↷ ?15:K ⟨/ k ⟩
       ⟨ env ⟩ ?16:Map VarId(X:VarId)(.kl) ↦ Int(L:Int)(.kl) ⟨/ env ⟩
   ⟨/ thread ⟩ ⟨/ threads ⟩
   ⟨ store ⟩ ?17:Map Int(L:Int)(.kl) ↦ Int(I:Int)(.kl) ⟨/ store ⟩ .
```

. . . Resolving strictness annotations

```
op VarId : VarId → KProperLabel .
op Int : Int → KResultLabel .
op "_:=_" : → KProperLabel [prec 40 metadata "arity 2 strict(2)"] .

rl ⟨ threads ⟩ ?13:Set ⟨ thread ⟩ ?14:Set
       ⟨ k ⟩ "_:=_"(VarId(X:VarId)(. kl), Int(I:Int)(. kl)) ↷ ?15:K ⟨/ k ⟩
       ⟨ env ⟩ ?16:Map VarId(X:VarId)(.kl) ↦ Int(L:Int)(. kl) ⟨/ env ⟩
   ⟨/ thread ⟩ ⟨/ threads ⟩
   ⟨ store ⟩ ?17:Map Int(L:Int)(. kl) ↦ Int(?18:Int)(. kl) ⟨/ store ⟩
⇒ ⟨ threads ⟩ ?13:Set ⟨ thread ⟩ ?14:Set
       ⟨ k ⟩ .k ↷ ?15:K ⟨/ k ⟩
       ⟨ env ⟩ ?16:Map VarId(X:VarId)(.kl) ↦ Int(L:Int)(. kl) ⟨/ env ⟩
   ⟨/ thread ⟩ ⟨/ threads ⟩
   ⟨ store ⟩ ?17:Map Int(L:Int)(. kl) ↦ Int(I:Int)(. kl) ⟨/ store ⟩ .
```

Strictness annotations became strict contexts

```
op VarId : VarId → KProperLabel .
op Int : Int → KResultLabel .
op "_:=_" : → KProperLabel [prec 40] .

mb context "_:=_"(K1:K,[]) : KSentence .

rl ⟨ threads ⟩ ?13:Set ⟨ thread ⟩ ?14:Set
       ⟨ k ⟩ "_:=_"(VarId(X:VarId), Int(I:Int)) ↷ ?15:K ⟨/ k ⟩
       ⟨ env ⟩ ?16:Map VarId(X:VarId) ↦ Int(L:Int) ⟨/ env ⟩
   ⟨/ thread ⟩ ⟨/ threads ⟩
   ⟨ store ⟩ ?17:Map Int(L:Int) ↦ Int(?18:Int) ⟨/ store ⟩
⇒ ⟨ threads ⟩ ?13:Set ⟨ thread ⟩ ?14:Set
       ⟨ k ⟩ .k ↷ ?15:K ⟨/ k ⟩
       ⟨ env ⟩ ?16:Map VarId(X:VarId) ↦ Int(L:Int) ⟨/ env ⟩
   ⟨/ thread ⟩ ⟨/ threads ⟩
   ⟨ store ⟩ ?17:Map Int(L:Int) ↦ Int(I:Int) ⟨/ store ⟩ .
```

Resolving strict contexts

```
op VarId : VarId → KProperLabel .
op Int : Int → KResultLabel .
op "_:=_" : → KProperLabel [prec 40] .

mb context "_:=_"(K1:K,[]) : KSentence .

rl ⟨ threads ⟩ ?13:Set ⟨ thread ⟩ ?14:Set
       ⟨ k ⟩ "_:=_"(VarId(X:VarId), Int(I:Int)) ↷ ?15:K ⟨/ k ⟩
       ⟨ env ⟩ ?16:Map VarId(X:VarId) ↦ Int(L:Int) ⟨/ env ⟩
   ⟨/ thread ⟩ ⟨/ threads ⟩
   ⟨ store ⟩ ?17:Map Int(L:Int) ↦ Int(?18:Int) ⟨/ store ⟩
⇒ ⟨ threads ⟩ ?13:Set ⟨ thread ⟩ ?14:Set
       ⟨ k ⟩ .k ↷ ?15:K ⟨/ k ⟩
       ⟨ env ⟩ ?16:Map VarId(X:VarId) ↦ Int(L:Int) ⟨/ env ⟩
   ⟨/ thread ⟩ ⟨/ threads ⟩
   ⟨ store ⟩ ?17:Map Int(L:Int) ↦ Int(I:Int) ⟨/ store ⟩ .
```

Strict contexts transalte to strictness equations

```
op VarId : VarId → KProperLabel .
op Int : Int → KResultLabel .
op "_:=_" : → KProperLabel [prec 40] .
eq ⟨ k ⟩ "_:=_"(K1:K, Kcxt:KProper) ↷ Rest:K ⟨/ k ⟩
 = ⟨ k ⟩ Kcxt:KProper ↷ "_ :=[]"( K1:K) ↷ Rest:K ⟨/ k ⟩.
eq ⟨ k ⟩ Kcxt:KResult ↷ "_ :=[]"( K1:K) ↷ Rest:K ⟨/ k ⟩
 = ⟨ k ⟩ "_:=_"(K1:K, Kcxt:KResult) ↷ Rest:K ⟨/ k ⟩.
rl ⟨ threads ⟩ ?13:Set ⟨ thread ⟩ ?14:Set
       ⟨ k ⟩ "_:=_"(VarId(X:VarId), Int(I:Int)) ↷ ?15:K ⟨/ k ⟩
       ⟨ env ⟩ ?16:Map VarId(X:VarId) ↦ Int(L:Int) ⟨/ env ⟩
   ⟨/ thread ⟩ ⟨/ threads ⟩
   ⟨ store ⟩ ?17:Map Int(L:Int) ↦ Int(?18:Int) ⟨/ store ⟩
⇒ ⟨ threads ⟩ ?13:Set ⟨ thread ⟩ ?14:Set
       ⟨ k ⟩ .k ↷ ?15:K ⟨/ k ⟩
       ⟨ env ⟩ ?16:Map VarId(X:VarId) ↦ Int(L:Int) ⟨/ env ⟩
   ⟨/ thread ⟩ ⟨/ threads ⟩
   ⟨ store ⟩ ?17:Map Int(L:Int) ↦ Int(I:Int) ⟨/ store ⟩ .
```

Getting K-Maude

Google code project page

- `http://k-framework.googlecode.com`
- svn access:
  `http://k-framework.googlecode.com/svn/k-maude/trunk`
- reporting issues:
  `http://code.google.com/p/k-framework/issues`
- Actively developing, please update regularly

Compiling K-Maude definitions into Maude

- Currently supporting one-module definitions only
- First make sure Maude loads your definition
- kcompile.sh tool
  usage: tools/kcompile.sh <source_file>[.maude] [<module_name>]
  If <module_name> is not specified, is asumed to be allcaps(<source_file>). <source_file> should ensure that a module <module_name> is loaded. Module <module_name> should contain the entire definition of the language. Output is printed in <source_file>-compiled.maude.
  If an error occurs in the compilation process (including warnings such as Module redefined), the script will stop, displaying the input, the (maybe partial) generated output, and the error/warning messages reported by Maude.

Outline

2002-2003

eq evalExps(exps(X := E) env([X,L] Env) cont(C) S) =
evalExps(exps(E) cont(L → int(1) → C) env([X,L] Env) S) .
eq evalCont(cont(VI → LI → C) store(St) S) =
evalCont(cont(C) store(St[LI ← VI]) S) .

With carried-along environment (seemed to have been faster)

```
rl k([(set X = E) @ ([X,L] Env)] -> K) =
   k([E @ ([X,L] Env)] -> L => int(1) -> K) .
rl k(V -> L => K), m(M) => k(K), m(M[L <- V]) .
```

ow [wq dop

2002-2003

Continuation is maintained separate from the expressions

```
eq evalExps(exps(X := E) env([X,L] Env) cont(C) S) =
   evalExps(exps(E) cont(L -> int(1) -> C) env([X,L] Env) S)
eq evalCont(cont(Vl -> Ll -> C) store(St) S) =
   evalCont(cont(C) store(St[Ll <- Vl]) S) .
```

wij diow udwq

Computation as continuation

Strictness as heating and cooling

Configuration as hierarchical multi-set

Abstract syntax instead of concrete syntax

Strictness annotations

Local rewriting

Context Inference

Anonymous variables and cell comprehension

Syntax with annotations

Generalized Strictness

Configuration

K rules

Conclusions