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

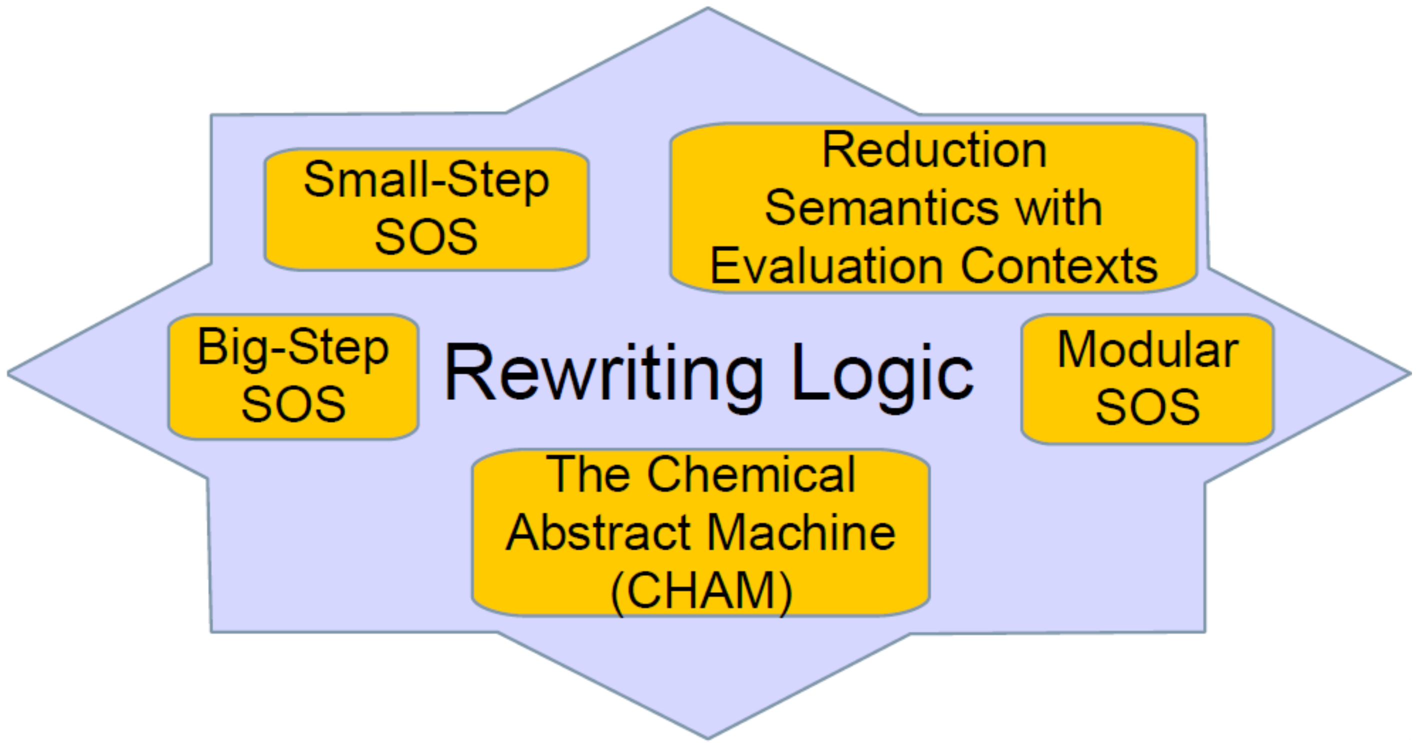
Why is Rewriting Logic Good for Programming Languages ?

- 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

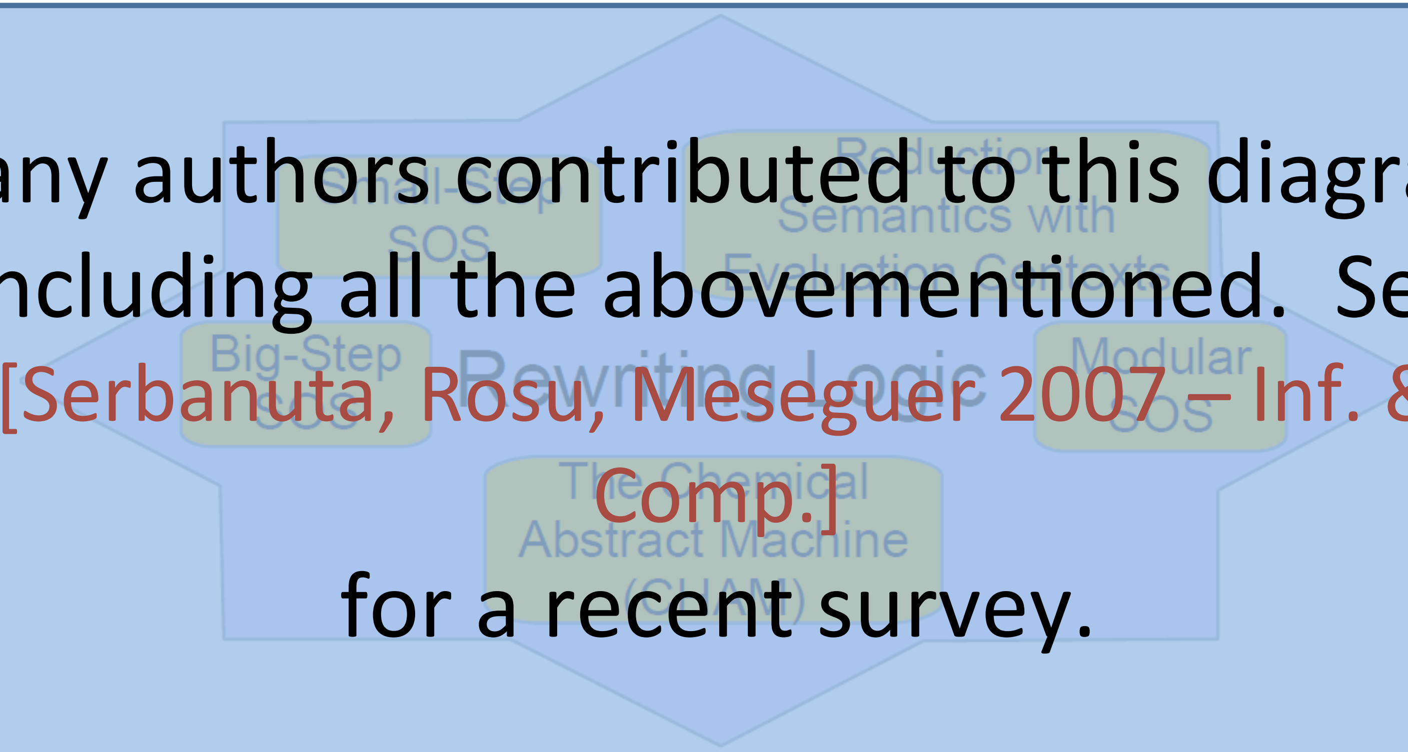
- Represent program state as configuration *term*
- Represent computational steps as *rewrite rules*
- Represent structural changes as *equations*
- These associate to any given configuration a *transition system*
  - In particular, it associates semantics to programs
- Resulting transition systems are 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

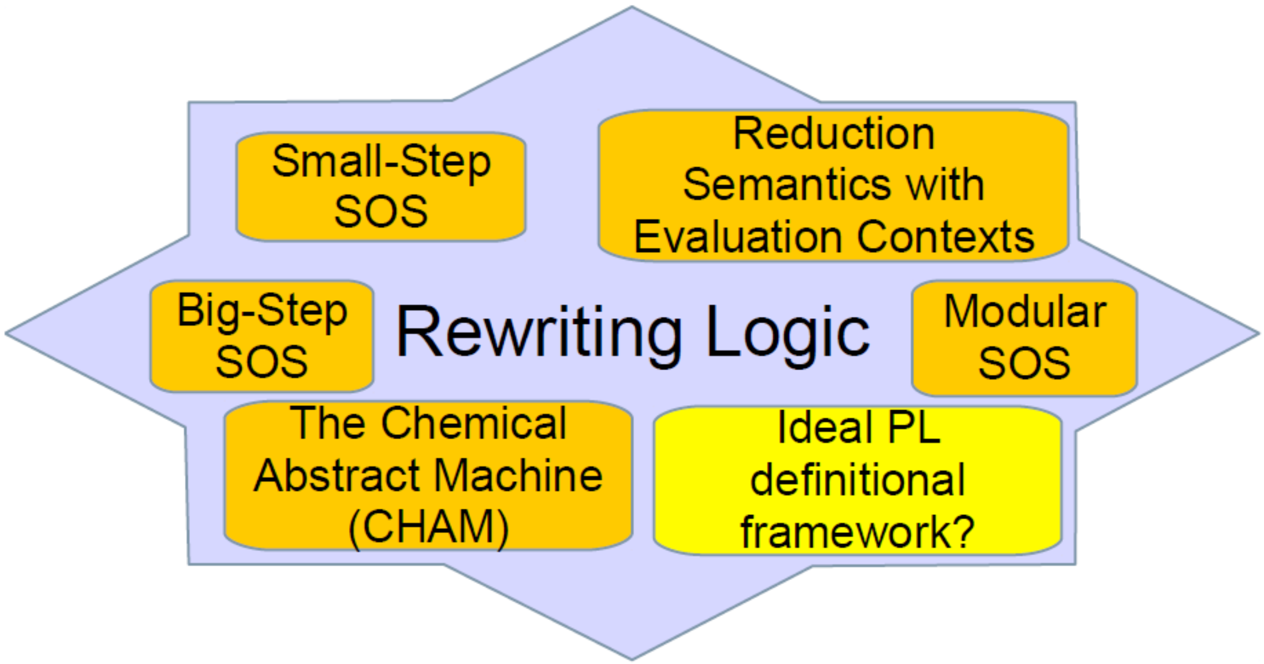
Existing Semantics Frameworks, From a Unified Perspective

- 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

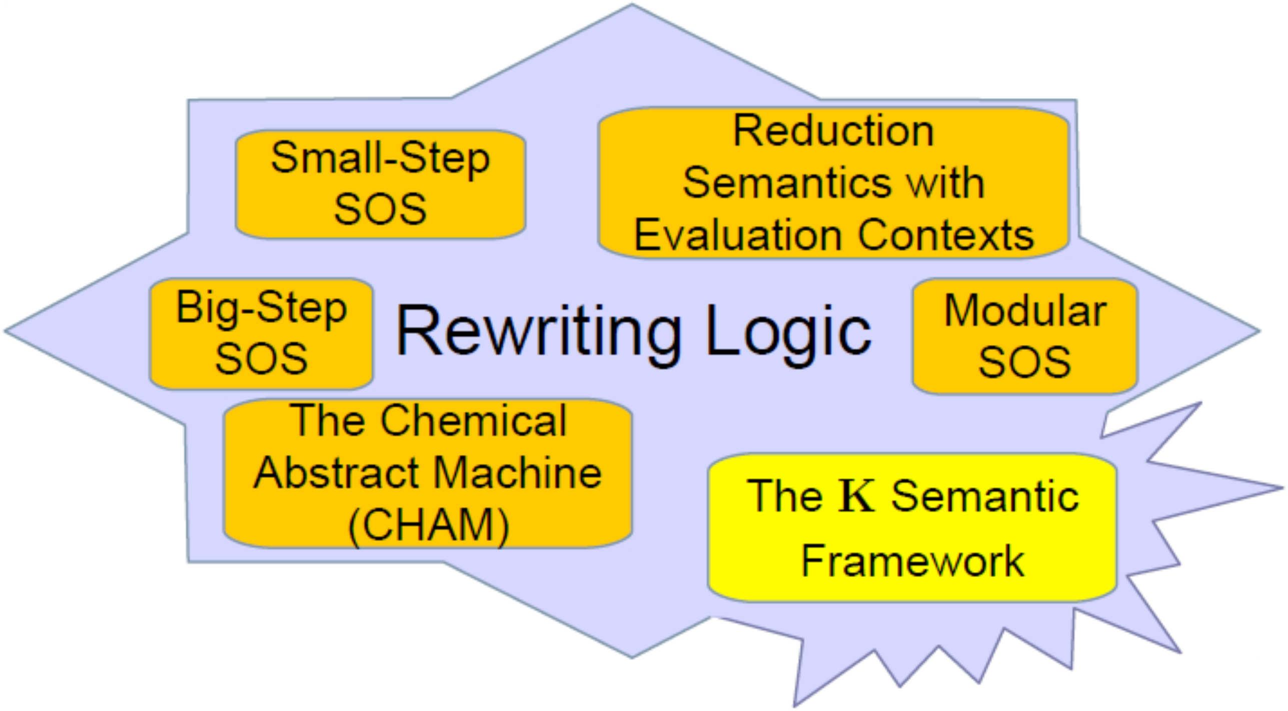
- Hard to deal with control (except evaluation contexts)
  - halt, break/continue, exceptions, callcc
- Non-modular (except Modular SOS)
  - Adding new features require changing unrelated rules
- Lack of semantics for true concurrency (except CHAM)
  - Big-Step captures only all possible results of computation
  - 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

$\mathbb{K}$ in a nutshell

$\mathbb{K}$omputations

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

$\mathbb{K}$onfigurations

- Multisets (bags) of nested cells
- High potential for concurrency and modularity

$\mathbb{K}$ rules

- Specify only what needed, precisely identify what changes
- More concise, modular, and concurrent than regular rewrite rules

Running example: KernelC

A subset of the C programming language

- Functions
- Memory allocation
- Pointer arithmetic
- Input/Output

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

Extended with concurrency features

- Thread creation
- Lock-based synchronization
- Thread join

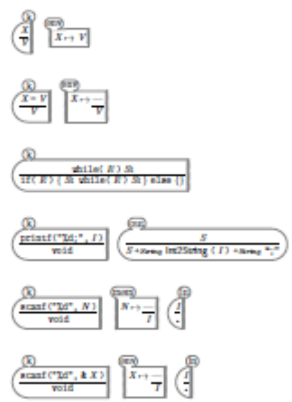
Running example: KernelC

Module KERNELC-SYNTAX
imports PL-ID+PL-INT
PointerId ::= *Id*
| * *PointerId* [strict]
Exp ::= *Int* | *PointerId* | *DeclId*
| * *Exp* [ditto]
| *Exp* + *Exp* [strict]
| *Exp* - *Exp* [strict]
| *Exp* == *Exp* [strict]
| *Exp* != *Exp* [strict]
| *Exp* <= *Exp* [strict]
| ! *Exp*
| *Exp* && *Exp*
| *Exp* || *Exp*
| *Exp* ? *Exp* : *Exp*
| *Exp* = *Exp* [strict(2)]
| printf("%d;", *Exp*) [strict]
| scanf("%d", *Exp*) [strict]
| & *Id*
| *Id* (*List*{*Exp*}) [strict(2)]
| *Id* ()
| *Exp* ++
| NULL
| free(*Exp*) [strict]
| (int*)malloc(*Exp* *sizeof(int)) [strict]
| *Exp* [*Exp*]
| spawn *Exp*
| acquire(*Exp*) [strict]
| release(*Exp*) [strict]
| join(*Exp*) [strict]
StmtList ::= *Stmt*
| *StmtList* *StmtList*
List{*Bottom*} ::= *Bottom*
| ()
| *List*{*Bottom*} , *List*{*Bottom*} [id: () strict hybrid assoc]
List{*PointerId*} ::= *PointerId* | *List*{*Bottom*}
| *List*{*PointerId*} , *List*{*PointerId*} [id: () ditto assoc]
List{*DeclId*} ::= *DeclId* | *List*{*Bottom*}
| *List*{*DeclId*} , *List*{*DeclId*} [id: () ditto assoc]
List{*Exp*} ::= *Exp* | *List*{*PointerId*} | *List*{*DeclId*}
| *List*{*Exp*} , *List*{*Exp*} [id: () ditto assoc]
DeclId ::= int *Exp*
| void *PointerId*
Stmt ::= *Exp* ; [strict]
| {}
| { *StmtList* }
| if(*Exp*) *Stmt*
| if(*Exp*) *Stmt* else *Stmt* [strict(1)]
| while(*Exp*) *Stmt*
| *DeclId* *List*{*DeclId*} { *StmtList* }
| *DeclId* *List*{*DeclId*} { *StmtList* return *Exp* ;}
| #include< *StmtList* >
Id ::= main
Pgm ::= #include<stdio.h>#include<stdlib.h> *StmtList*
end module

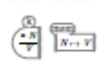
Module KERNELC-DESUGARED-SYNTAX
imports KERNELC-SYNTAX
macro: ! E = E ? 0 : 1
macro: E_1 && E_2 = E_1 ? E_2 : 0
macro: E_1 || E_2 = E_1 ? 1 : E_2
macro: if(E) St = if(E) St else {}
macro: NULL = 0
macro: I () = I (())
macro: DI L { Sts } = DI L { Sts return 0 ;}
macro: void PI = int PI
macro: int * PI = int PI
macro: #include< Sts > = Sts
macro: E_1 [E_2] = * E_1 + E_2
macro: int * PI = E = int PI = E
macro: E ++ = E = E + 1
end module

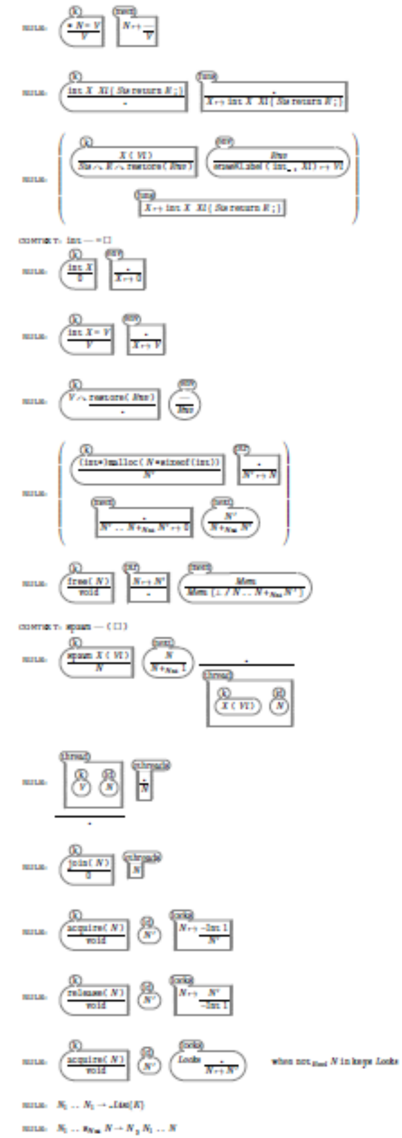
Module KERNELC-SEMANTICS
imports PL-CONVERSION+K+KERNELC-DESUGARED-SYNTAX
KResult ::= *List*{*Val*}
K ::= *List*{*Exp*} | *List*{*PointerId*} | *List*{*DeclId*} | *StmtList* | *Pgm* | *String*
| restore(*Map*)
Exp ::= *Val*
List{*Exp*} ::= *List*{*Val*}
Val ::= *Int*
| & *Id*
| void
List{*Val*} ::= *Val*
| *List*{*Val*} , *List*{*Val*} [id: () ditto assoc]
List{*K*} ::= *Nat* .. *Nat*
initial configuration:

K rules:
context: * □ = —
rule: I_1 == I_2 → Bool2Int ($I_1 ==_{Int} I_2$)
rule: I_1 != I_2 → Bool2Int ($I_1 !=_{Int} I_2$)
rule: I_1 + I_2 → $I_1 +_{Int} I_2$
rule: I_1 - I_2 → $I_1 -_{Int} I_2$
rule: I_1 <= I_2 → Bool2Int ($I_1 \leq_{Int} I_2$)
rule: _?_:_ → if(_)_else_
rule: if(I) — else St → St when $I ==_{Int} 0$
rule: if(I) St else — → St when not_{Bool} $I ==_{Int} 0$
rule: V ; → .



rule: #include<stdio.h>#include<stdlib.h> Sts → Sts
rule: { Sts } → Sts
rule: {} → .
rule: St Sts → St ↷ Sts



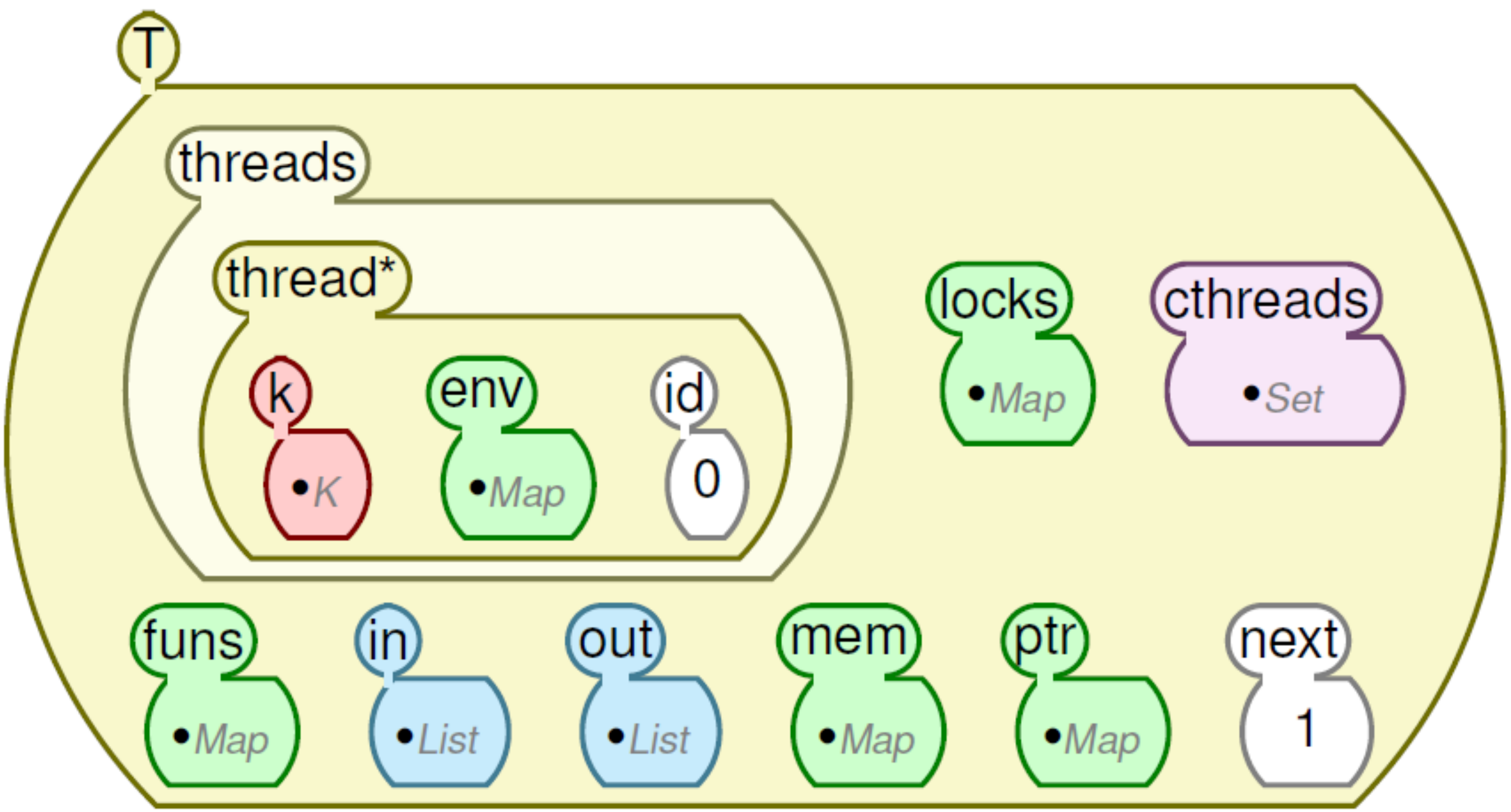


end module

Configurations—the running state of a program

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

Initial configuration for KernelC



$\mathbb{K}$ computations and $\mathbb{K}$ syntax

Computations

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

$\mathbb{K}$ Syntax: BNF syntax annotated with strictness

$$
\begin{array}{llrl}
Exp ::= & Id & & \\
 & \mid\ {*}\ Exp & [\text{strict}] & {*}\ ERed \rightleftharpoons ERed \curvearrowright {*}\ \square \\
 & \mid Exp = Exp & [\text{strict(2)}] & E = ERed \rightleftharpoons ERed \curvearrowright E = \square \\
Stmt ::= & Exp\ ; & [\text{strict}] & ERed\ ; \rightleftharpoons ERed \curvearrowright \square\ ; \\
 & \mid Stmt\ \ Stmt & [\text{seqstrict}] & SRed\ \ S \rightleftharpoons SRed \curvearrowright \square\ \ S
\end{array}
$$

Heating syntax through strictness rules

Computation

t = * x ; * x = * y ; * y = t ;

𝕂 Syntax: BNF syntax annotated with strictness

$Exp ::= Id$			
$\mid\ *\ Exp$	[strict]	$*\ ERed \rightleftharpoons ERed \curvearrowright *\ \square$
$\mid\ Exp = Exp$	[strict(2)]	$E = ERed \rightleftharpoons ERed \curvearrowright E = \square$
$Stmt ::= Exp\ ;$	[strict]	$ERed\ ; \rightleftharpoons ERed \curvearrowright \square\ ;$
$\mid\ Stmt\ \ Stmt$	[seqstrict]	$SRed\ \ S \rightleftharpoons SRed \curvearrowright \square\ \ S$

Heating syntax through strictness rules

Computation

$$t = {*}\,x\ ;\quad \curvearrowright\quad \square\ {*}\,x = {*}\,y\ ;\ \ {*}\,y = t\ ;$$

𝕂 Syntax: BNF syntax annotated with strictness

$Exp ::= Id$			
$\mid\ {*}\ Exp$	[strict]	${*}\ ERed$	$\rightleftharpoons\ ERed \curvearrowright {*}\ \square$
$\mid Exp = Exp$	[strict(2)]	$E = ERed$	$\rightleftharpoons\ ERed \curvearrowright E = \square$
$Stmt ::= Exp\ ;$	[strict]	$ERed\ ;$	$\rightleftharpoons\ ERed \curvearrowright \square\ ;$
$\mid Stmt\ \ Stmt$	[seqstrict]	$SRed\ \ S$	$\rightleftharpoons\ SRed \curvearrowright \square\ \ S$

Heating syntax through strictness rules

Computation

$$t = {*}\,x \;\curvearrowright\; \square\,; \;\curvearrowright\; \square\;\; {*}\,x = {*}\,y\;;\;\; {*}\,y = t\;;$$

$\mathbb{K}$ Syntax: BNF syntax annotated with strictness

$Exp ::= Id$		
$\mid\ {*}\ Exp$	[strict]	${*}\ ERed \rightleftharpoons ERed \curvearrowright {*}\ \square$
$\mid\ Exp = Exp$	[strict(2)]	$E = ERed \rightleftharpoons ERed \curvearrowright E = \square$
$Stmt ::= Exp\ ;$	[strict]	$ERed\ ; \rightleftharpoons ERed \curvearrowright \square\ ;$
$\mid\ Stmt\ \ Stmt$	[seqstrict]	$SRed\ \ S \rightleftharpoons SRed \curvearrowright \square\ \ S$

Heating syntax through strictness rules

Computation

$$* \, x \curvearrowright t = \square \curvearrowright \square ; \curvearrowright \square \; * x = * y \; ; \; * y = t \; ;$$

$\mathbb{K}$ Syntax: BNF syntax annotated with strictness

$Exp ::= Id$		
$\mid \; * \, Exp$	[strict]	$* \, ERed \rightleftharpoons ERed \curvearrowright * \, \square$
$\mid Exp = Exp$	[strict(2)]	$E = ERed \rightleftharpoons ERed \curvearrowright E = \square$
$Stmt ::= Exp \; ;$	[strict]	$ERed \; ; \rightleftharpoons ERed \curvearrowright \square \; ;$
$\mid Stmt \;\; Stmt$	[seqstrict]	$SRed \;\; S \rightleftharpoons SRed \curvearrowright \square \;\; S$

Heating syntax through strictness rules

Computation

$$x \curvearrowright {*}\,\square \curvearrowright t = \square \curvearrowright \square; \curvearrowright \square\ {*}\,x = {*}\,y\ ;\ {*}\,y = t\ ;$$

$\mathbb{K}$ Syntax: BNF syntax annotated with strictness

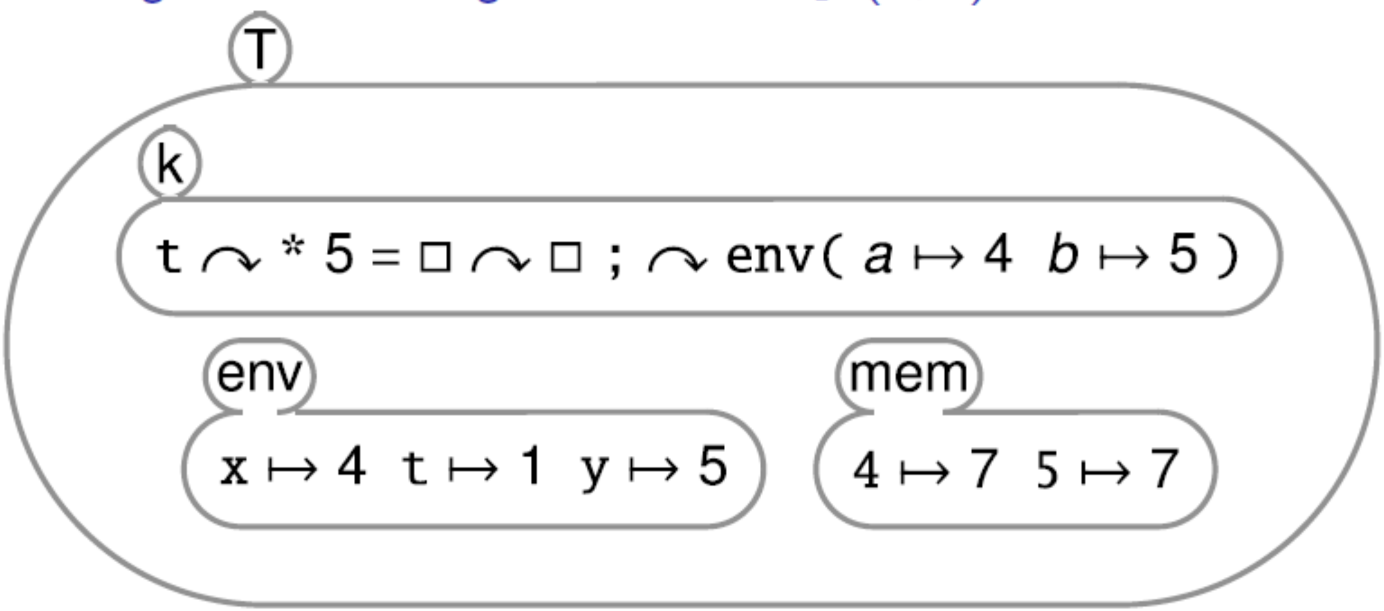
$$
\begin{array}{llll}
\mathit{Exp} ::= \mathit{Id} & & & \\
\quad\mid\ {*}\ \mathit{Exp} & [\text{strict}] & {*}\ \mathit{ERed} \rightleftharpoons \mathit{ERed} \curvearrowright {*}\ \square \\
\quad\mid \mathit{Exp} = \mathit{Exp} & [\text{strict(2)}] & E = \mathit{ERed} \rightleftharpoons \mathit{ERed} \curvearrowright E = \square \\
\mathit{Stmt} ::= \mathit{Exp}\ ; & [\text{strict}] & \mathit{ERed}\ ; \rightleftharpoons \mathit{ERed} \curvearrowright \square\ ; \\
\quad\mid \mathit{Stmt}\ \ \mathit{Stmt} & [\text{seqstrict}] & \mathit{SRed}\ \ S \rightleftharpoons \mathit{SRed} \curvearrowright \square\ \ S
\end{array}
$$

Example: running configuration

Swapping the values at locations 4 and 5 in memory

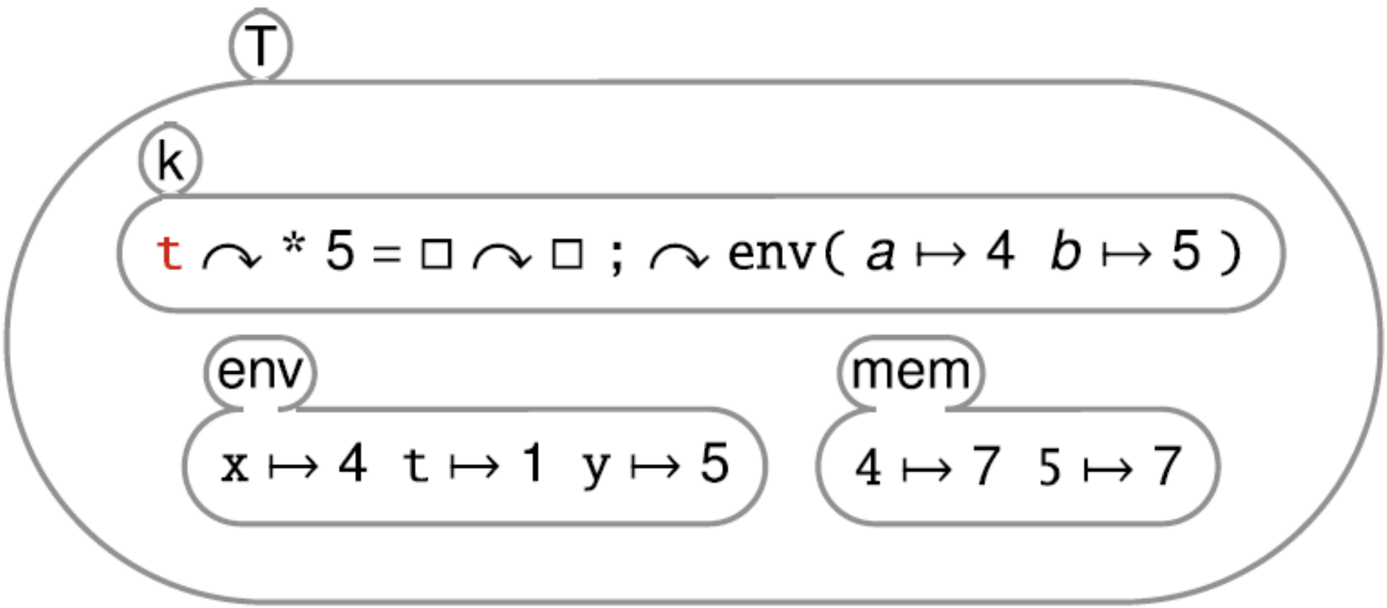
```
void swap(int * x, int * y){
 int t = * x;  * x = * y;  * y = t;
}
```

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



Example: running configuration

- Strictness rules (from annotations) extract the redex
- How do we specify next step?

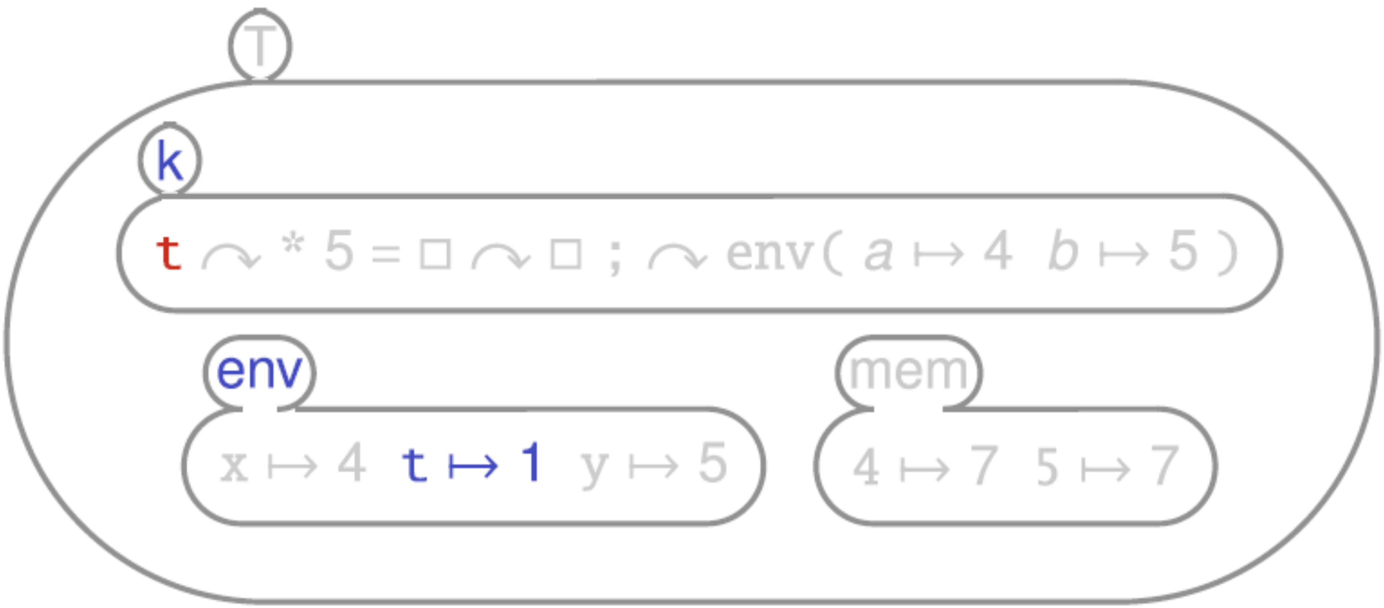


$\mathbb{K}$ rules: expressing natural language into rules

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



$\mathbb{K}$ rules: expressing natural language into rules

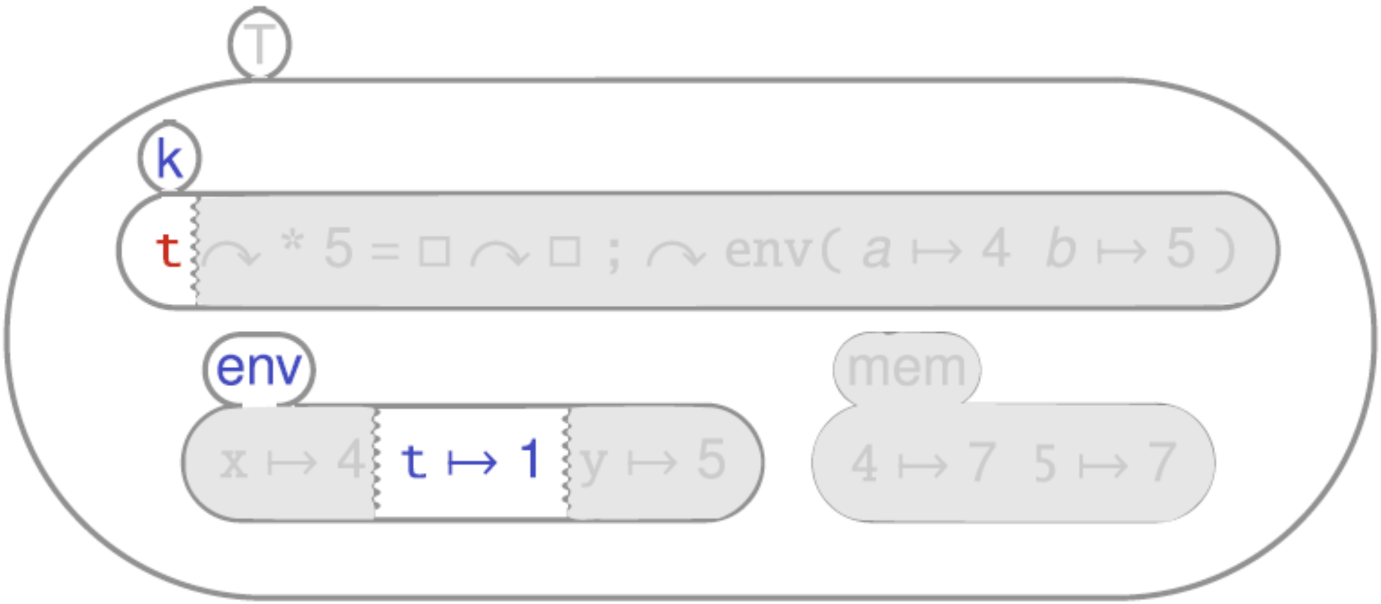
Unnecessary parts of the cells are abstracted away

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

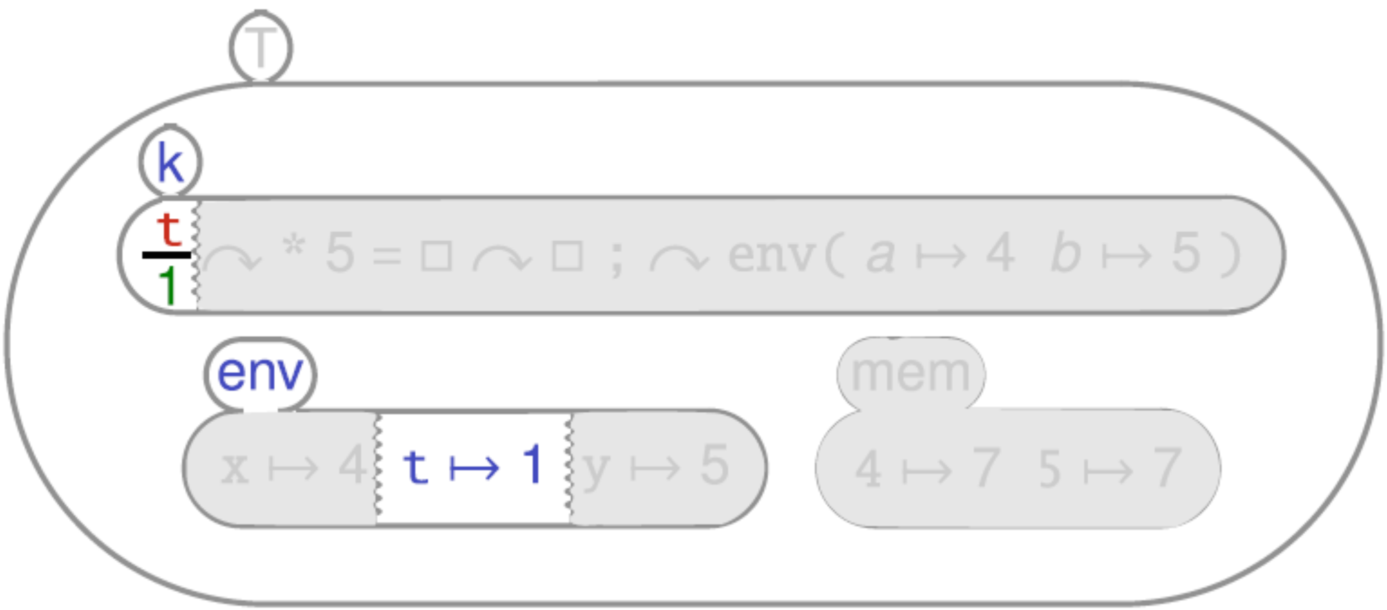
Underlining what to replace, writing the replacement under the line

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

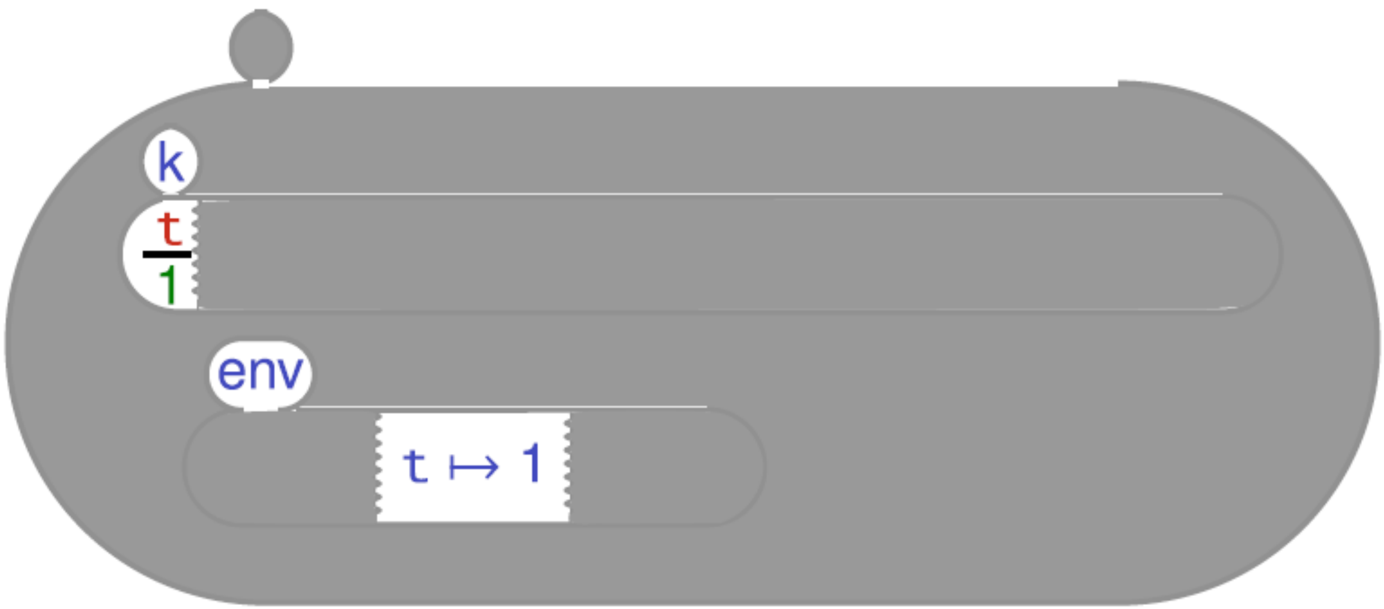


𝕂 rules: expressing natural language into rules

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



$\mathbb{K}$ rules: expressing natural language into rules

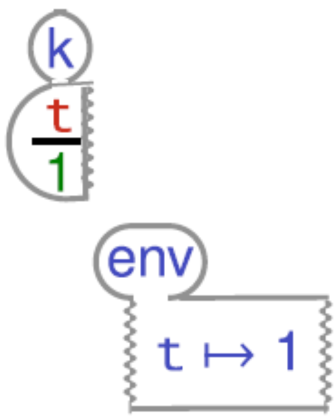
Configuration Abstraction: Keep only the relevant cells

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

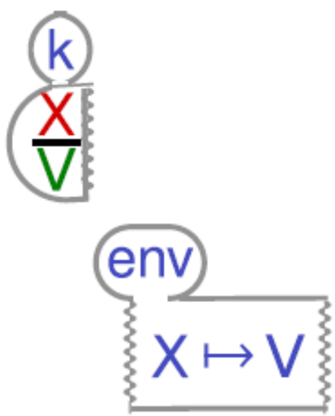


𝕂 rules: expressing natural language into rules

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



$\mathbb{K}$ rules: expressing natural language into rules

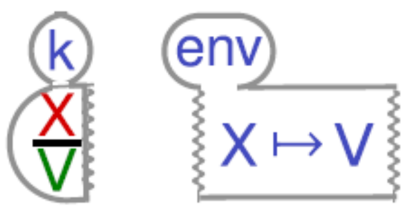
Voilà!

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 . . .

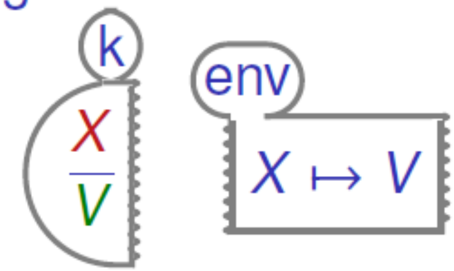
. . . then process X, replacing it by V

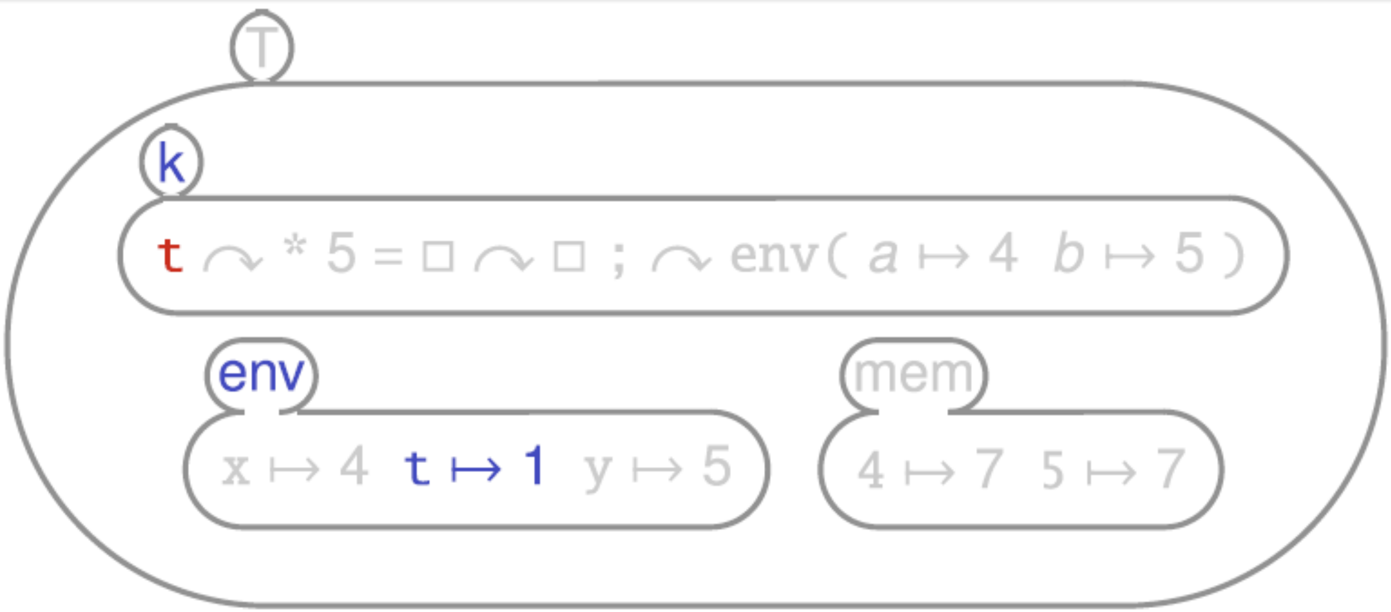


$\mathbb{K}$ rules: expressing natural language into rules

Summary

Reading from environment

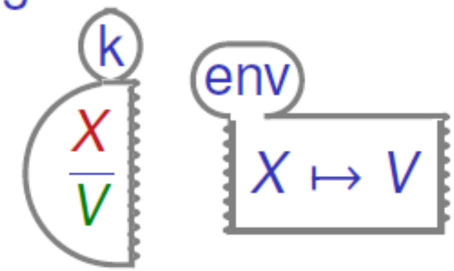


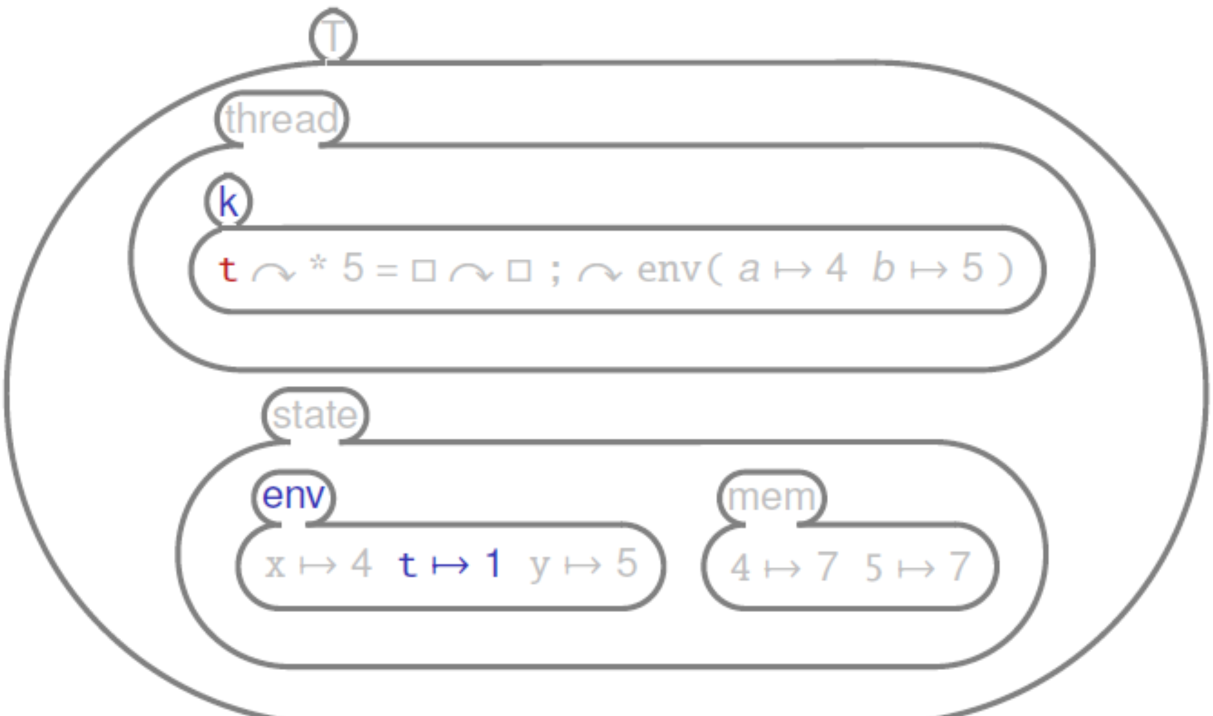


Configuration abstraction enhances modularity

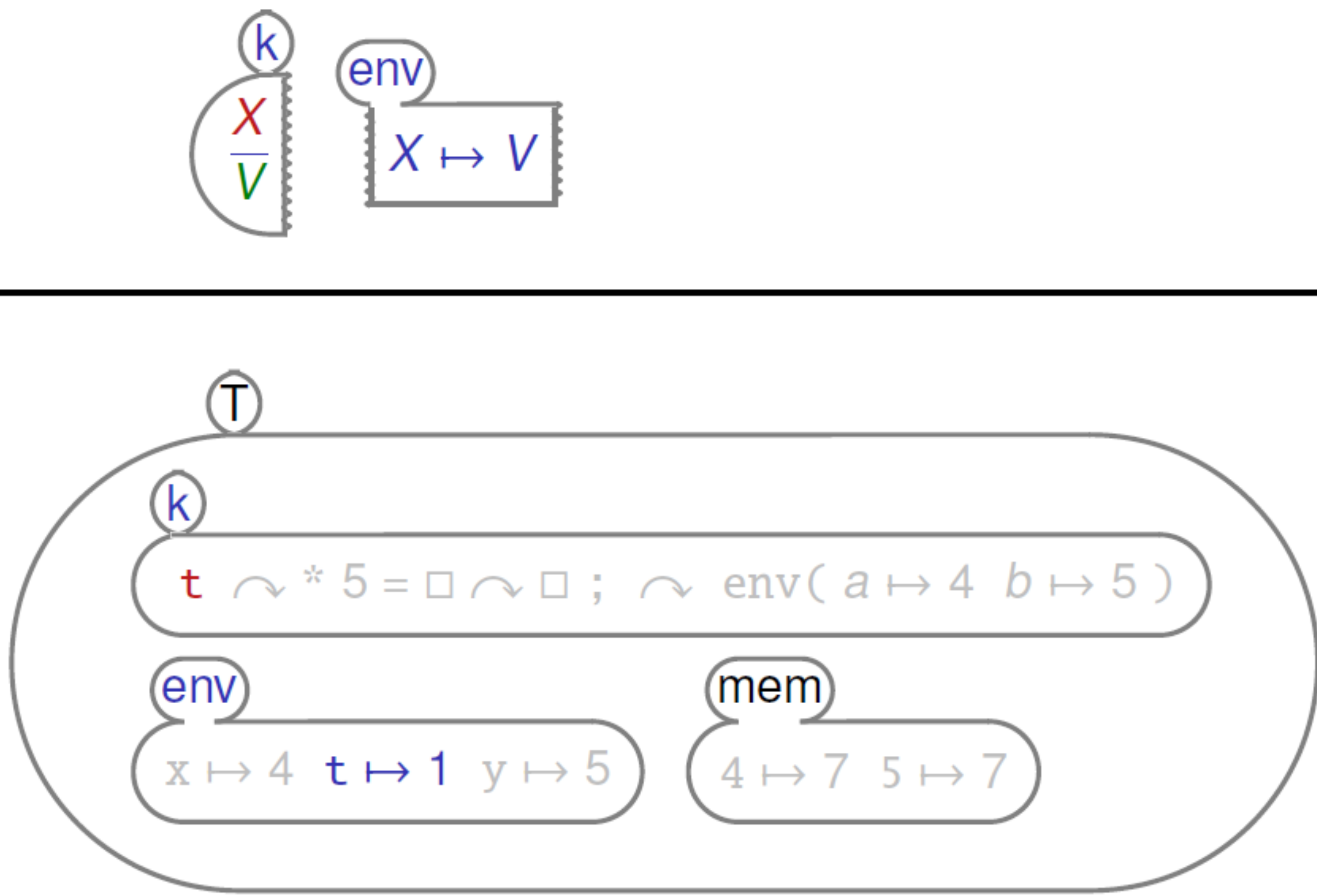
At least as modular as Modular SOS

Reading from environment

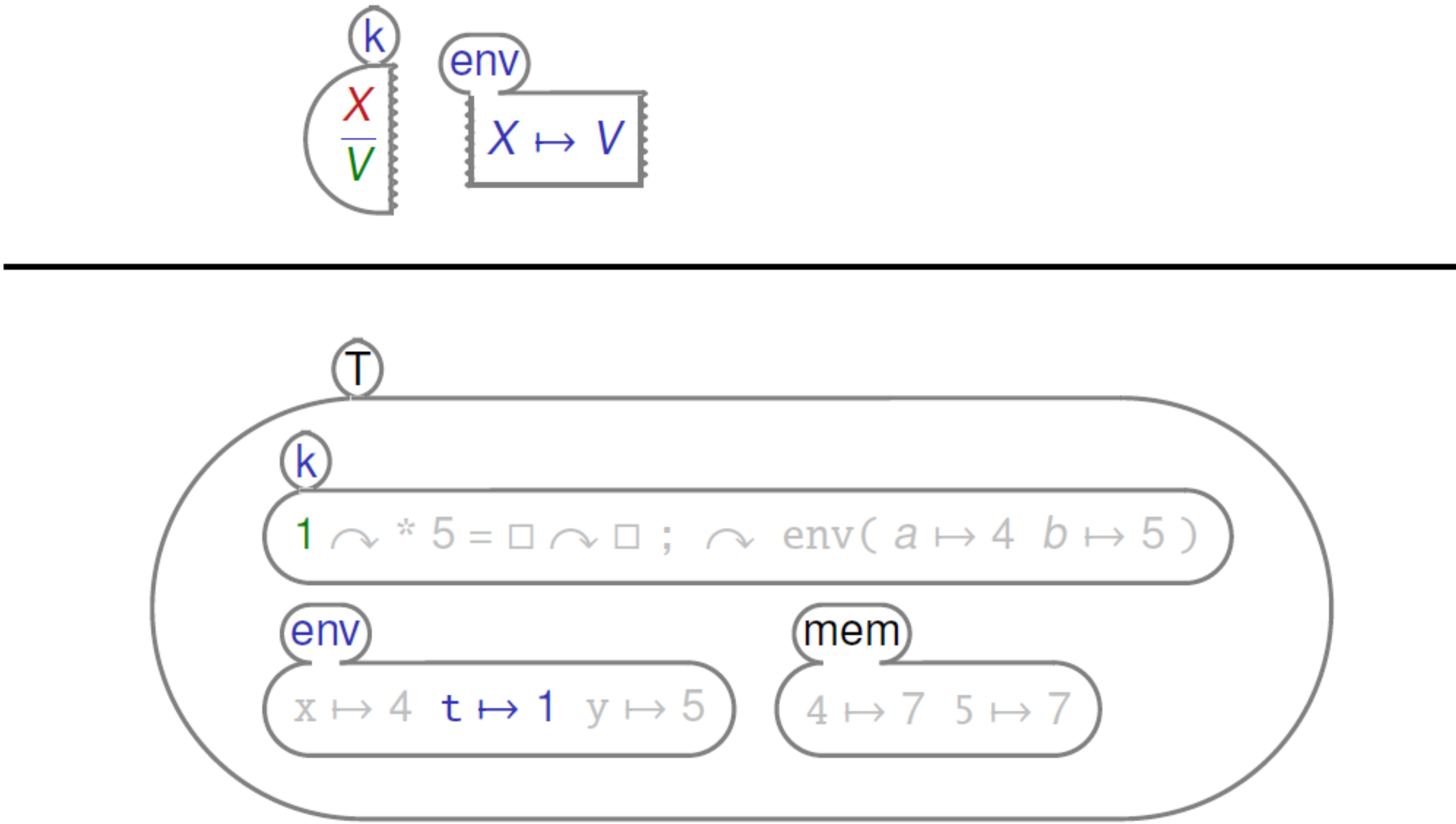




Applying 𝕂 rules: Reading from environment



Applying 𝕂 rules: Reading from environment

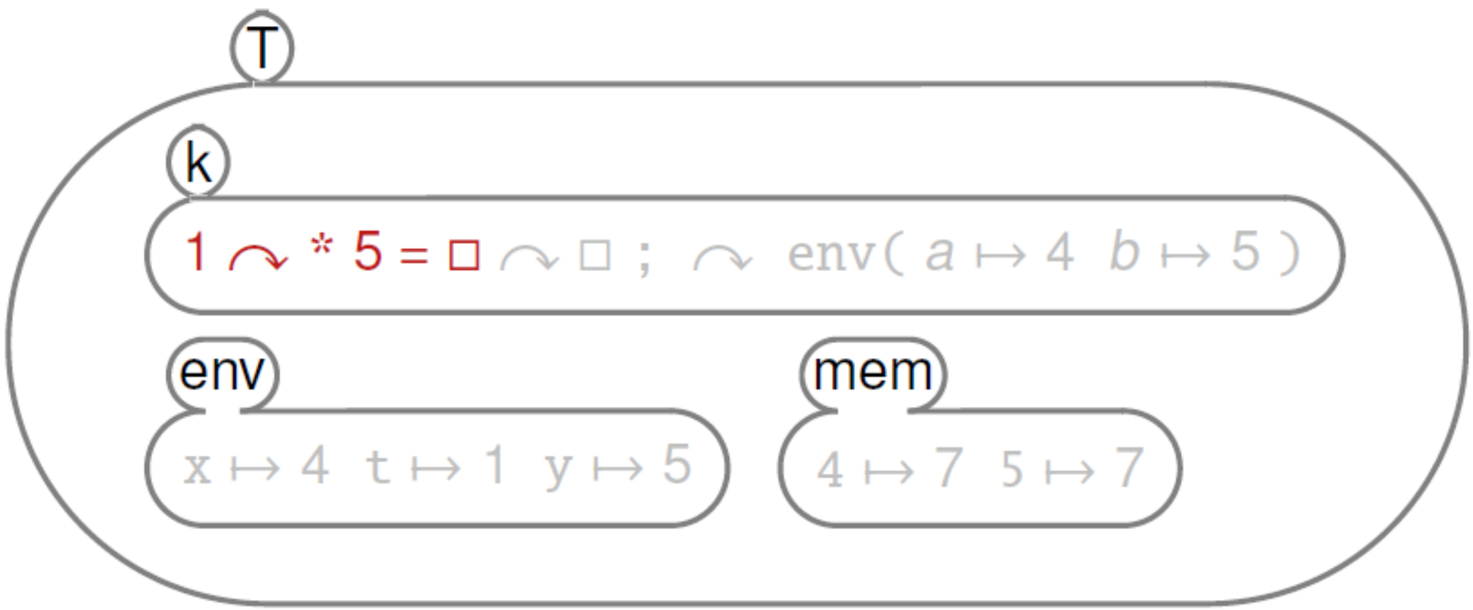


Applying $\mathbb{K}$ rules: Strictness of assignment

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

which is syntactic sugar for:

$$E = Redex \rightleftharpoons Redex \curvearrowright E = \square$$

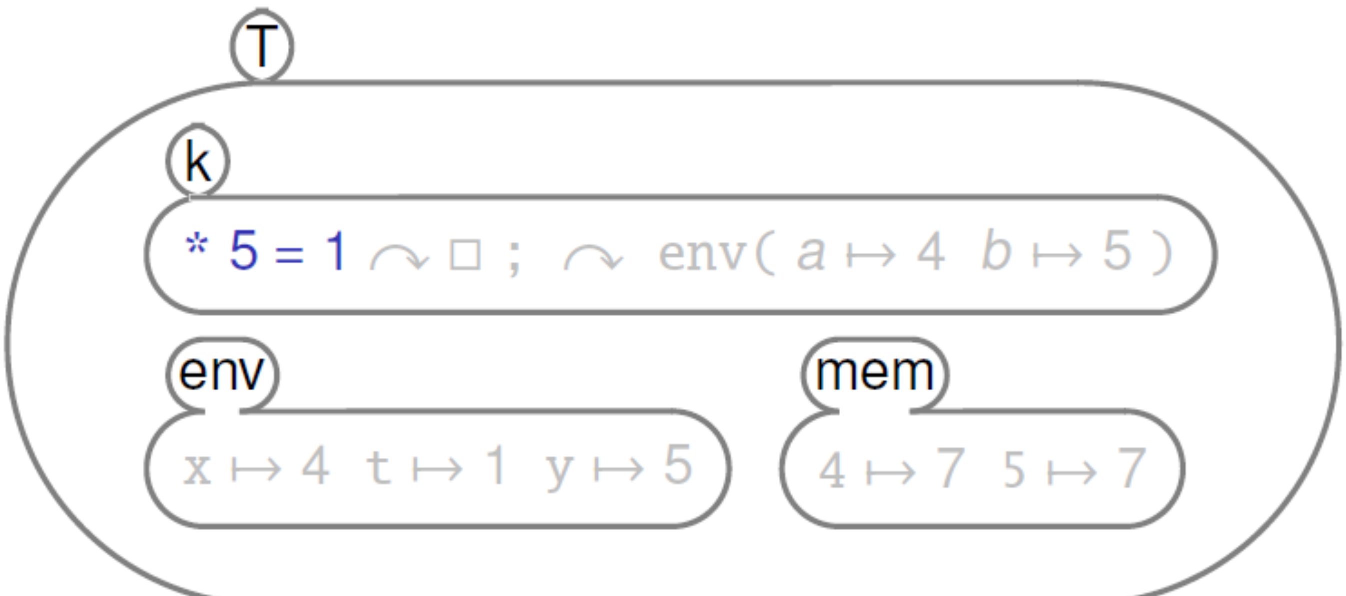


Applying $\mathbb{K}$ rules: Strictness of assignment

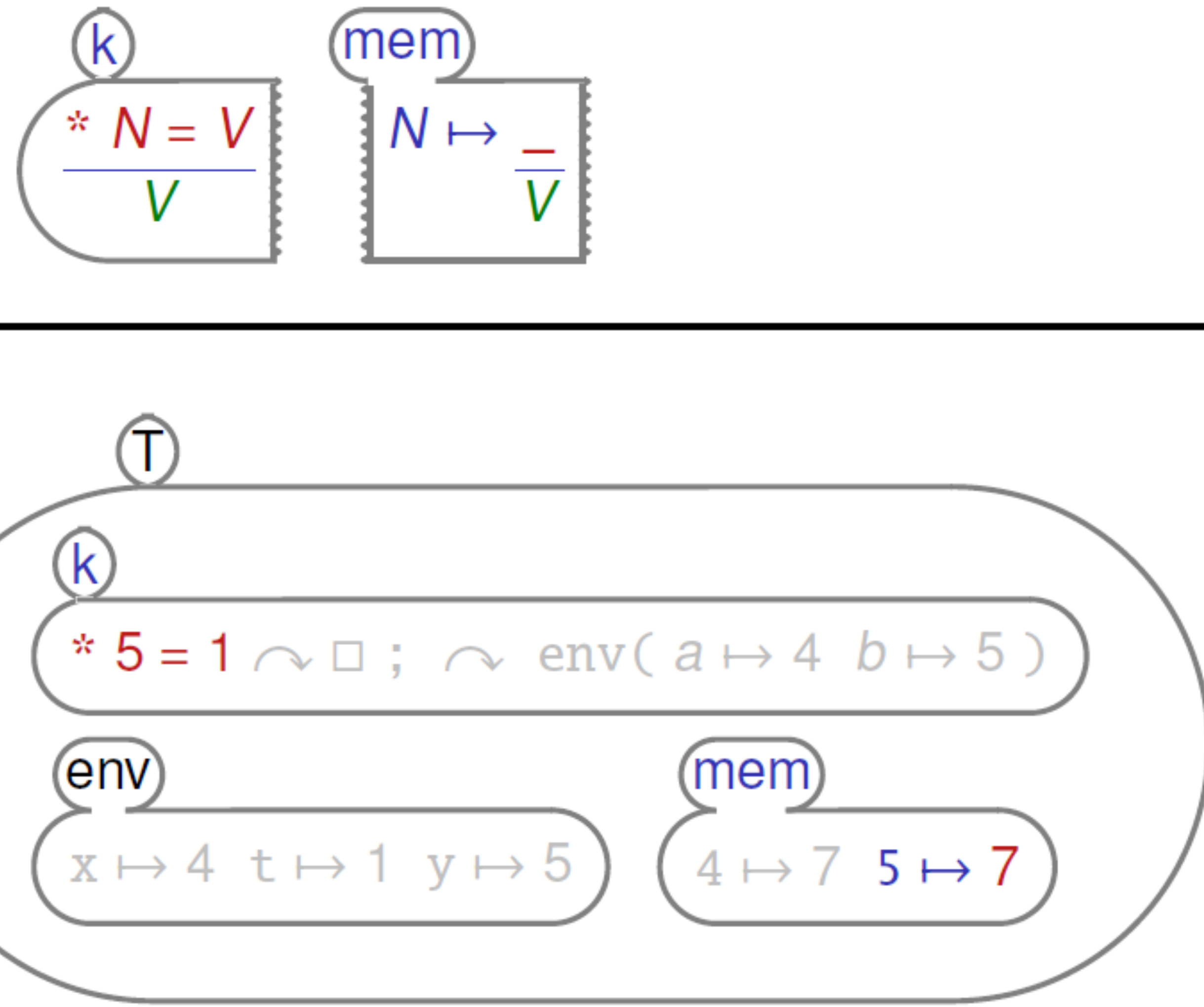
$Exp ::= Exp = Exp$ [strict(2)]

which is syntactic sugar for:

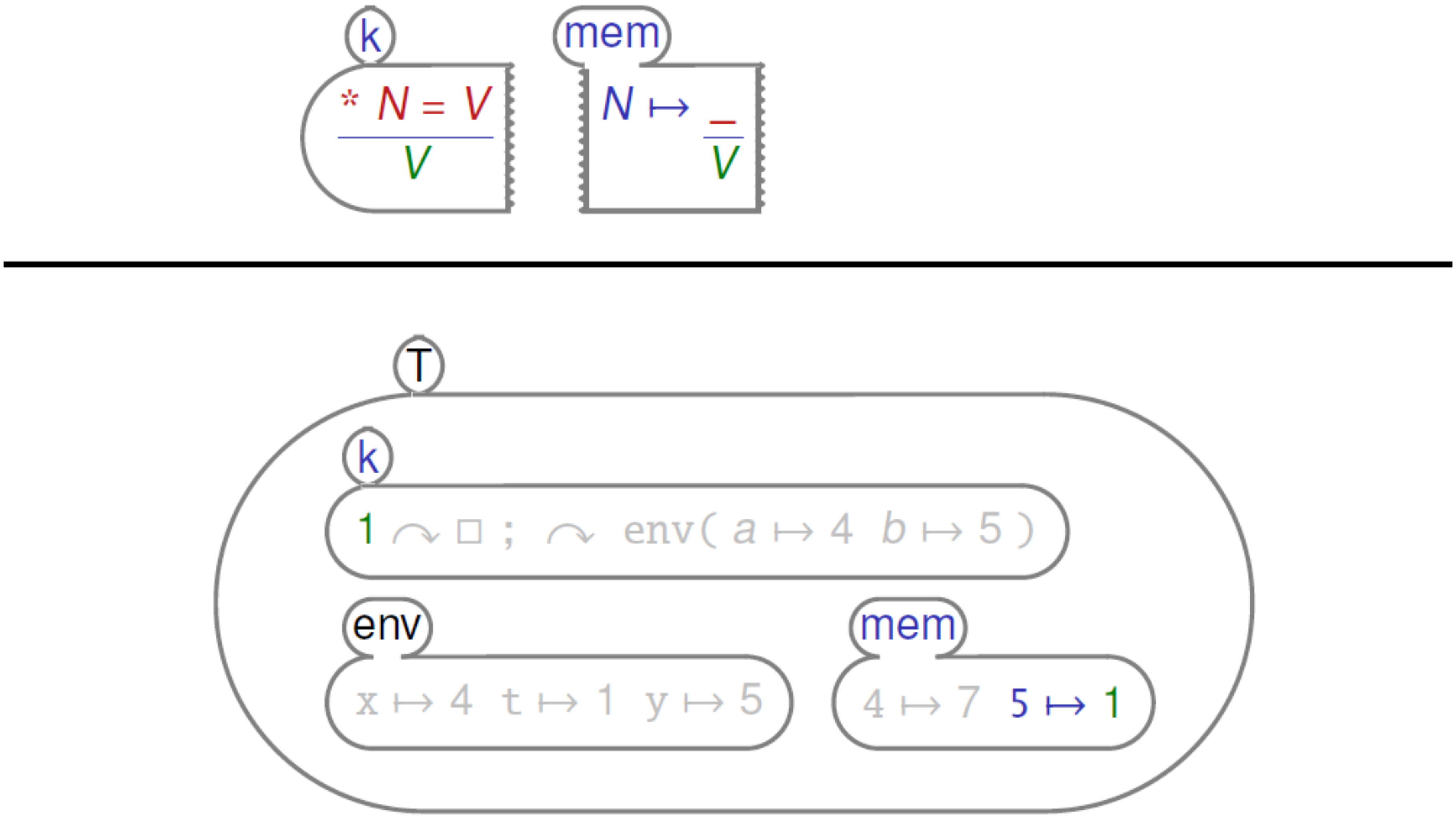
$$E = Redex \rightleftharpoons Redex \curvearrowright E = \square$$



Applying 𝕂 rules: Updating memory



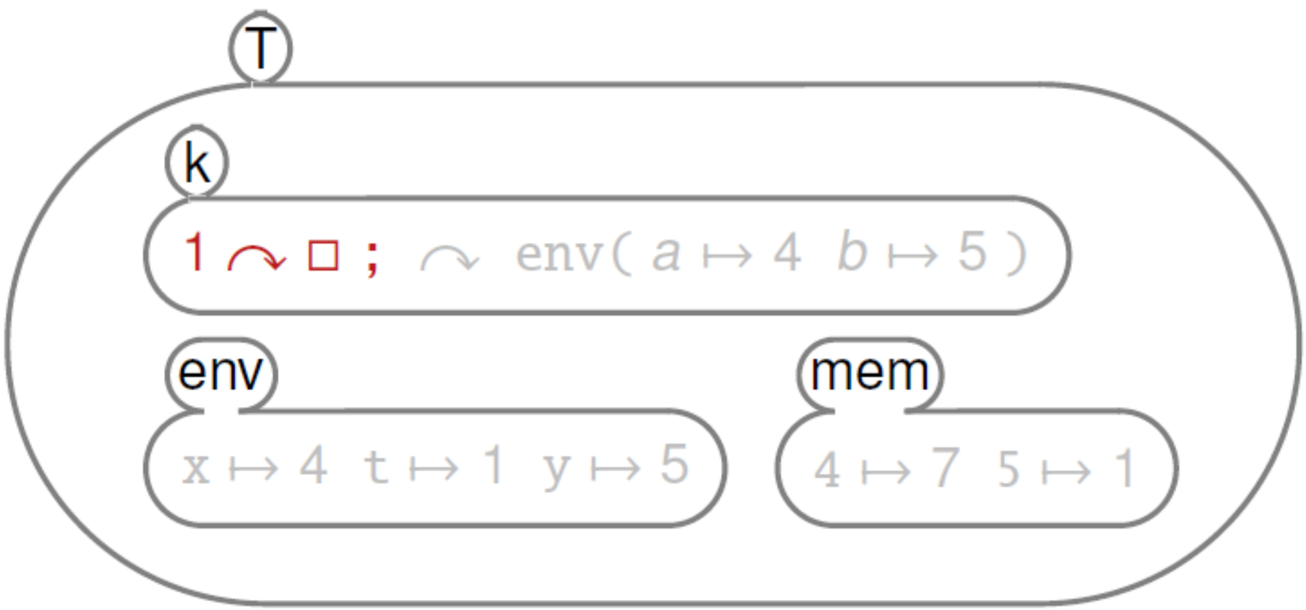
Applying 𝕂 rules: Updating memory



Applying $\mathbb{K}$ rules: Semantics for expression statements

Strictness : $Stmt ::= Exp$; [strict]

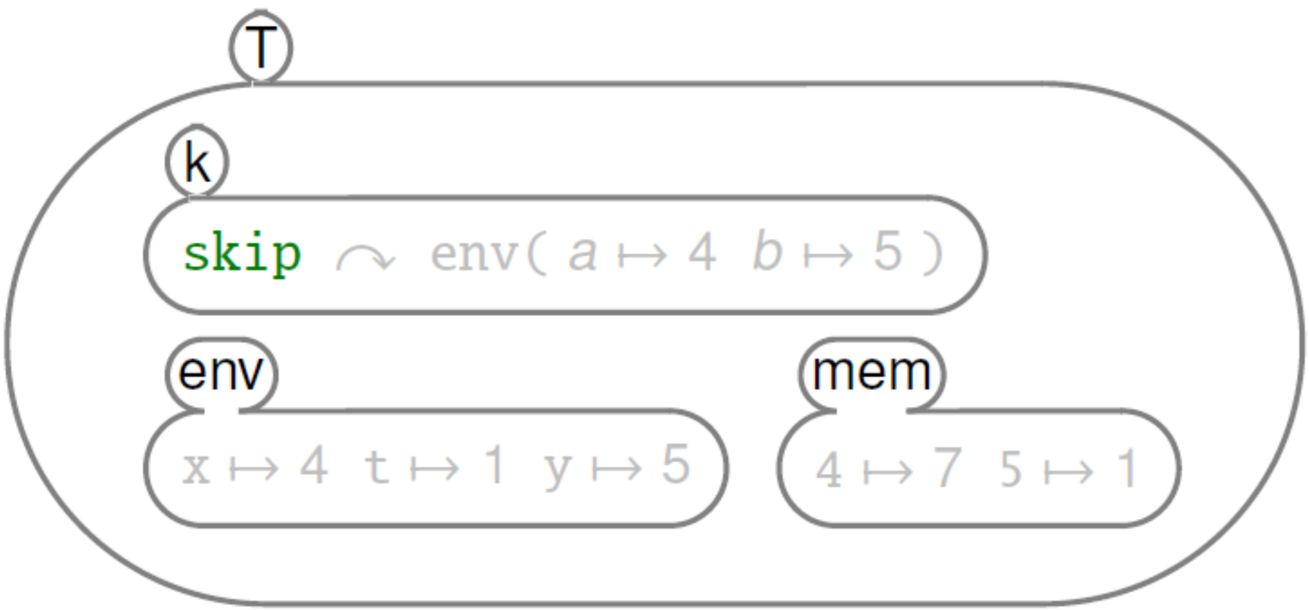
Semantic rule: V ; $\rightharpoonup$ `skip`



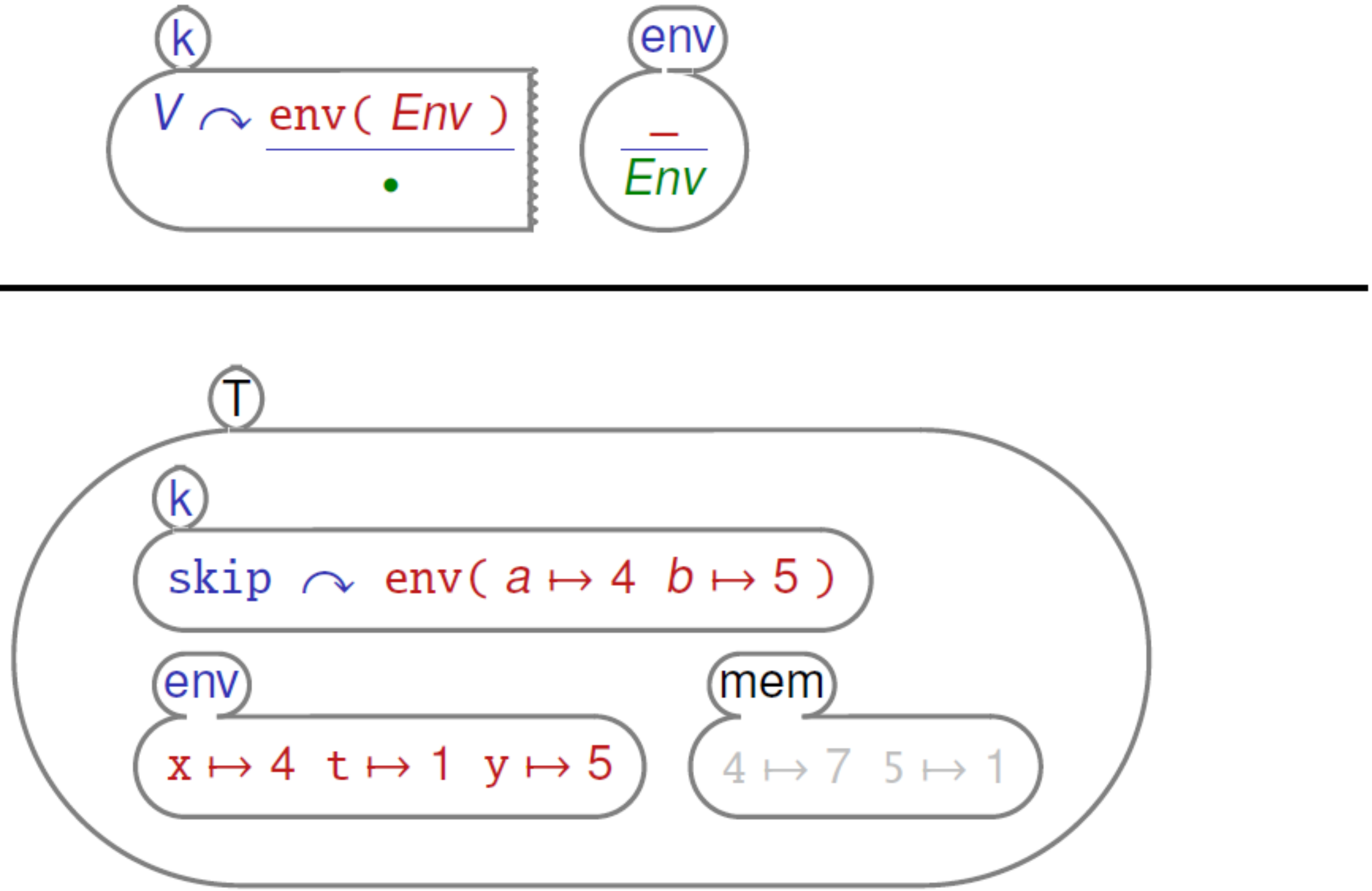
Applying $\mathbb{K}$ rules: Semantics for expression statements

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

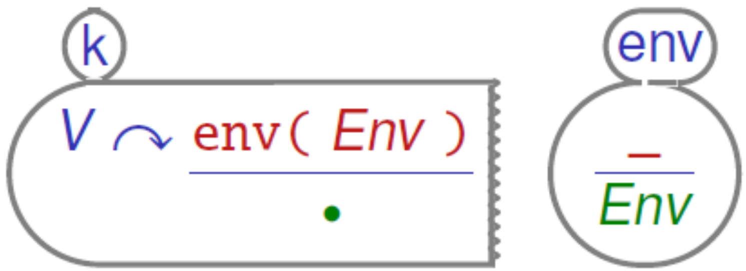
Semantic rule: V ; $\rightharpoonup$ `skip`

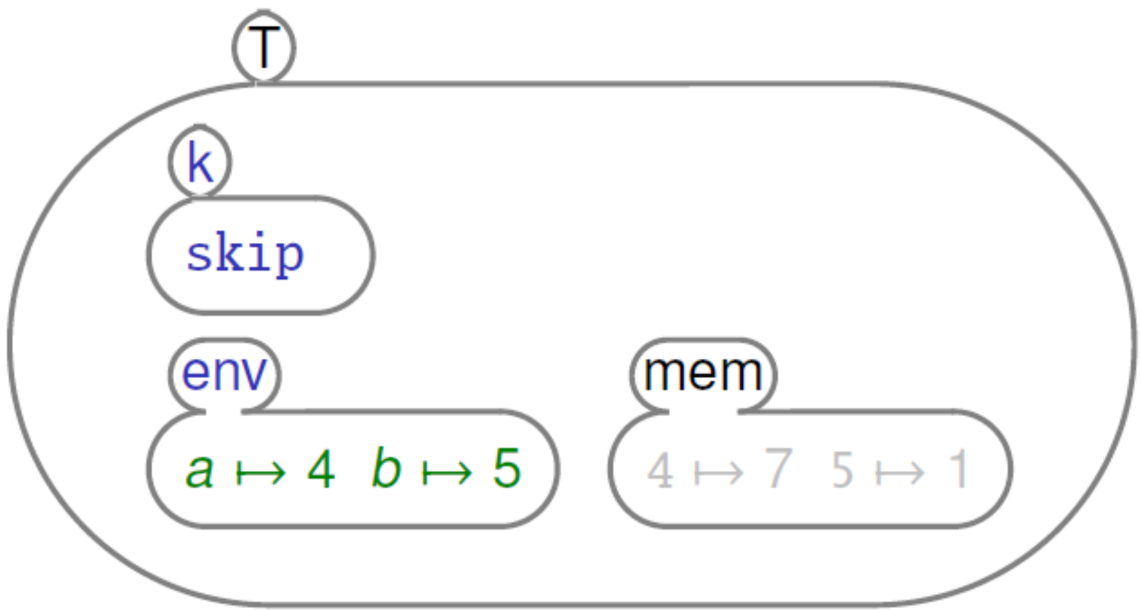


Applying 𝕂 rules: Recovering environment



Applying $\mathbb{K}$ rules: Recovering environment





$\mathbb{K}$'s definitional power

More expressive than Reduction with Evaluation Contexts

Some of $\mathbb{K}$'s strengths

- Dealing with control
- Task synchronization
- Defining reflective features

Definitional Power: Control

Call with current continuation

Passing computation as value:

$$\frac{\texttt{callcc}\ V}{V\ \ cc(K)} \curvearrowright K$$

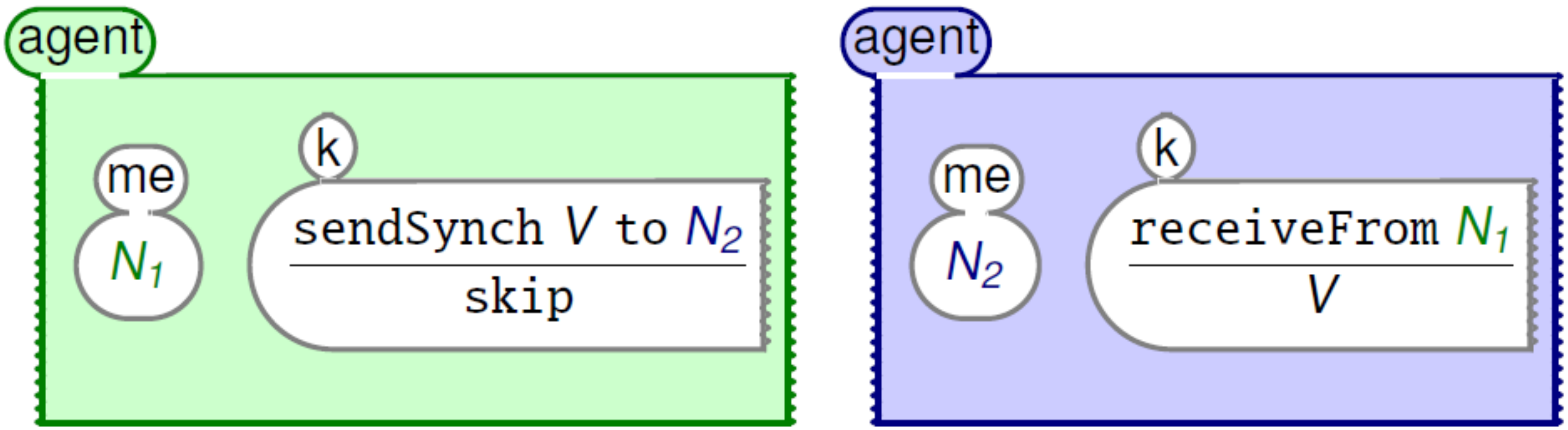
Applying computation as function:

$$\frac{cc(K)\ \ V \curvearrowright \_}{V \curvearrowright K}$$

- 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 Syntax = AST

- $K ::= KLabel(List\{K\}) \ \mid \bullet_{K} \ \mid K \curvearrowright K$
  $List\{K\} ::= K \ \mid \bullet_{List\{K\}} \ \mid List\{K\} \ , \ List\{K\}$
- Everything else (language construct, builtin constant) is a K label

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

Reflection through AST manipulation

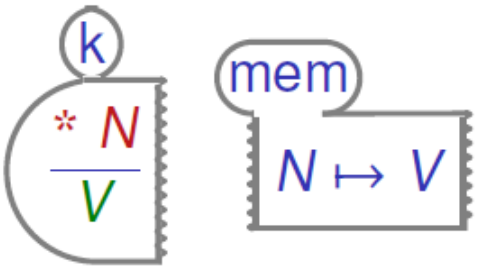
- Generic AST visitor pattern
- Code generation: quote/unquote
- Binder independent substitution

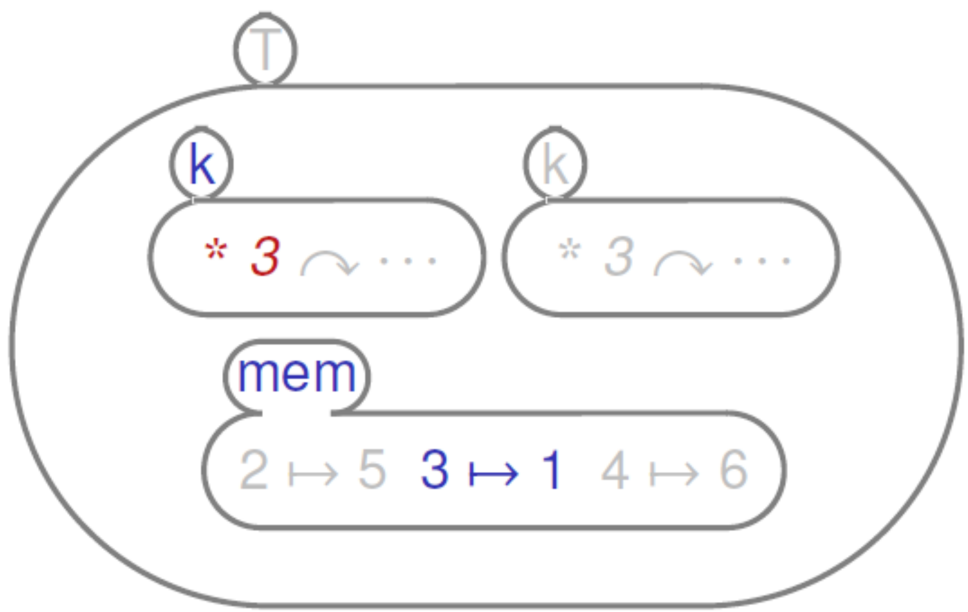
The concurrency of the $\mathbb{K}$ framework

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

𝕂 rules enhance concurrency

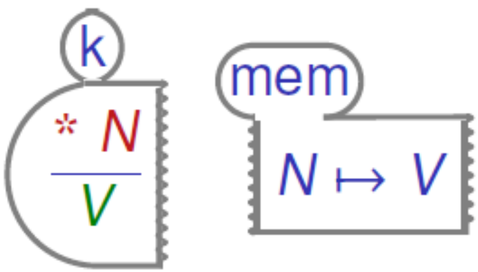
Concurrent reads

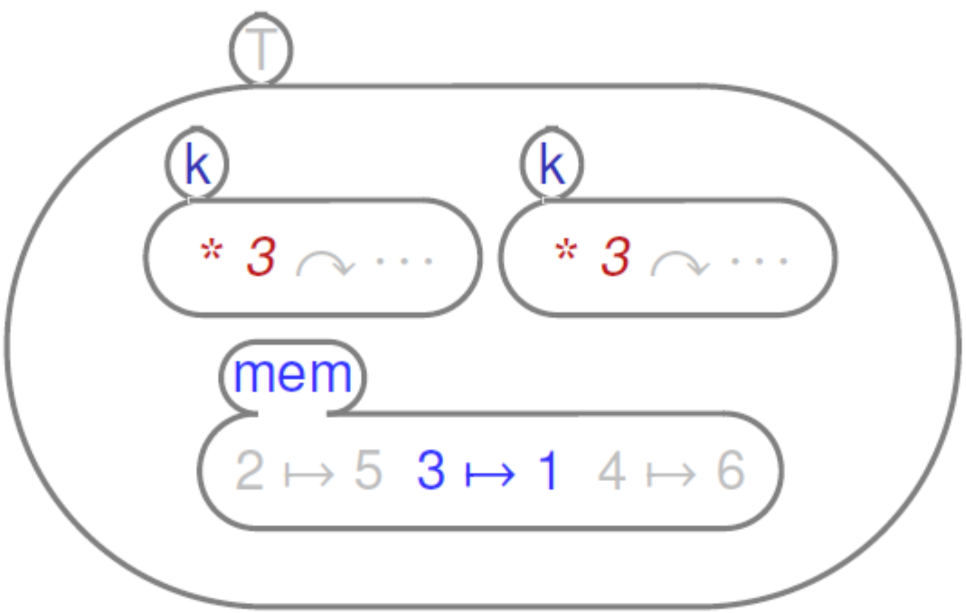




$\mathbb{K}$ rules enhance concurrency

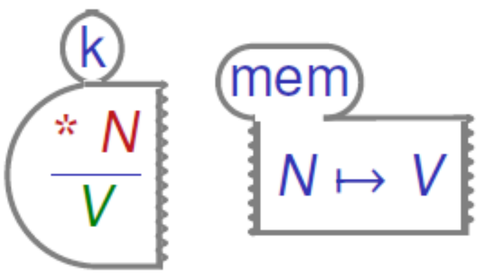
Concurrent reads

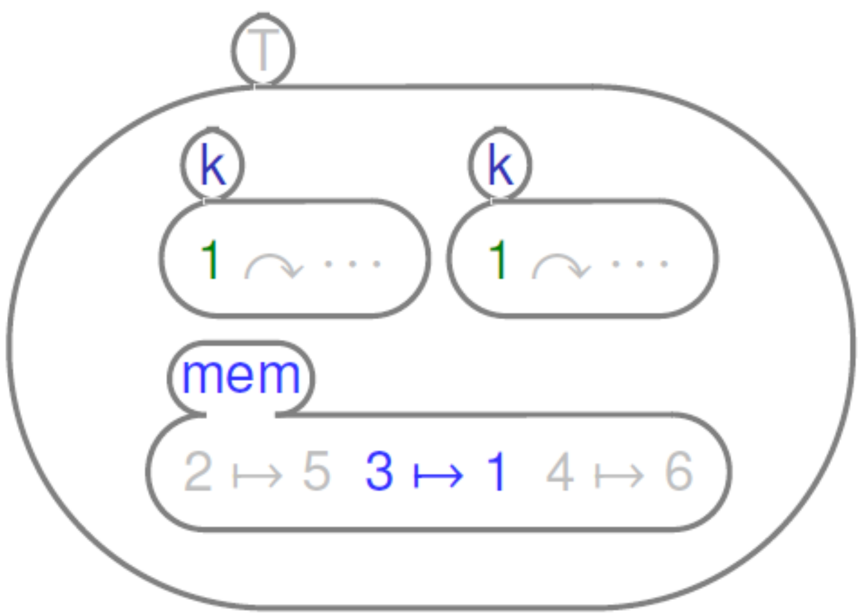




𝕂 rules enhance concurrency

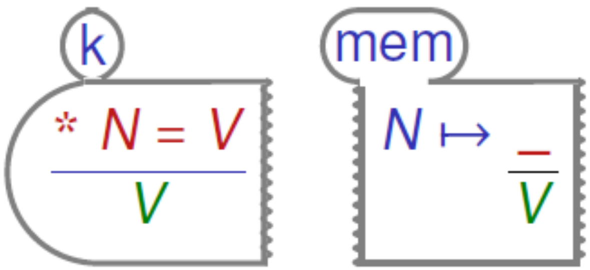
Concurrent reads

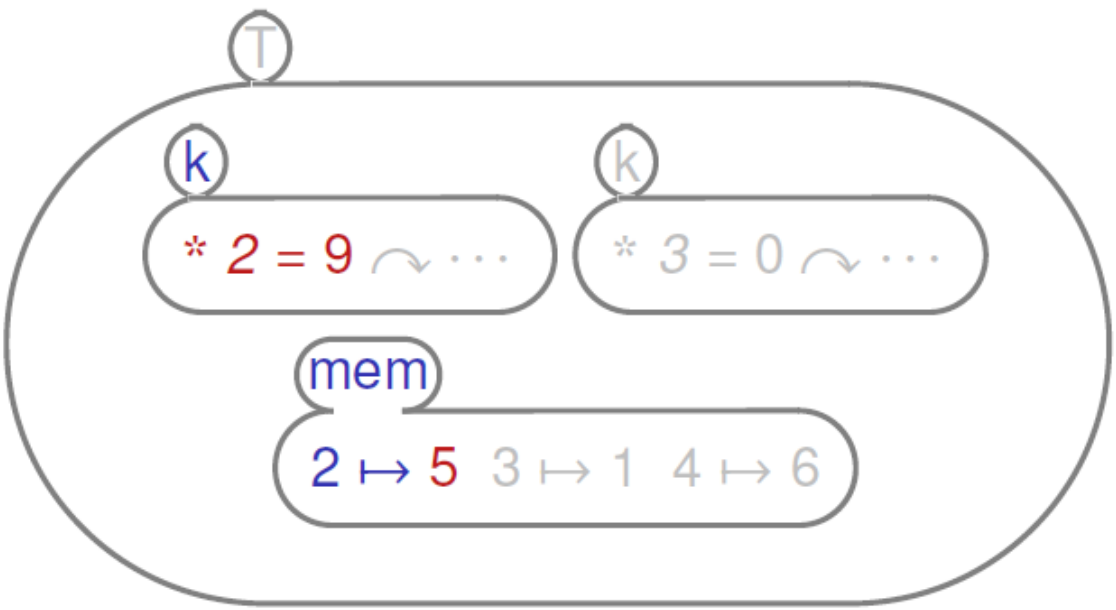




$\mathbb{K}$ rules enhance concurrency

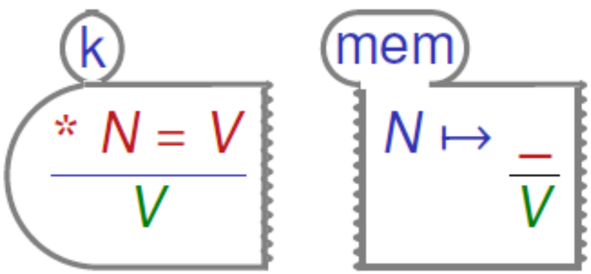
Concurrent updates (on distinct locations)

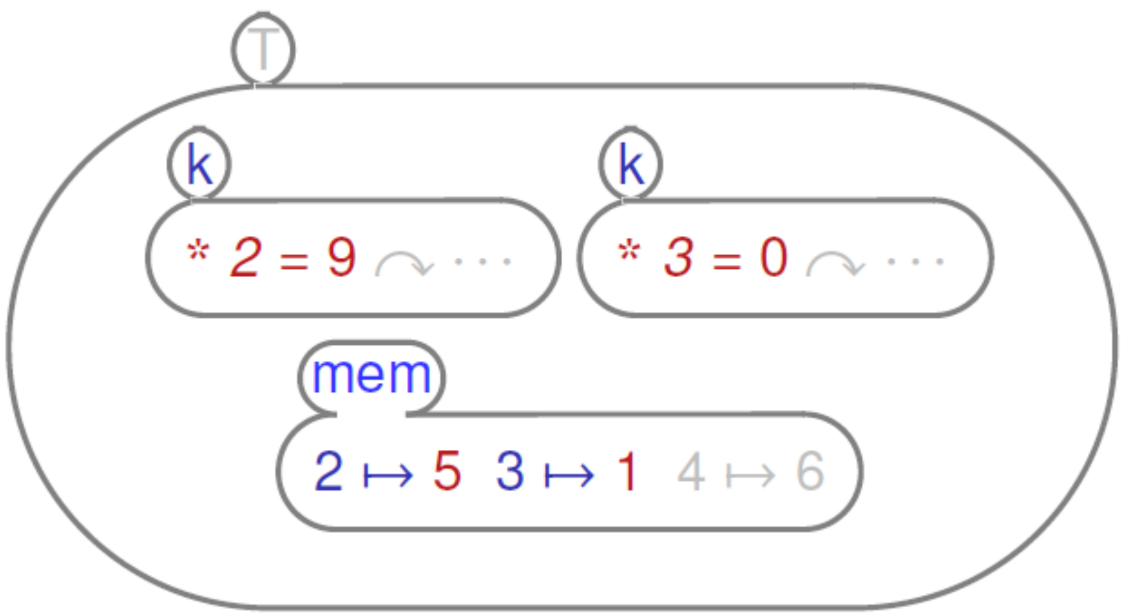




$\mathbb{K}$ rules enhance concurrency

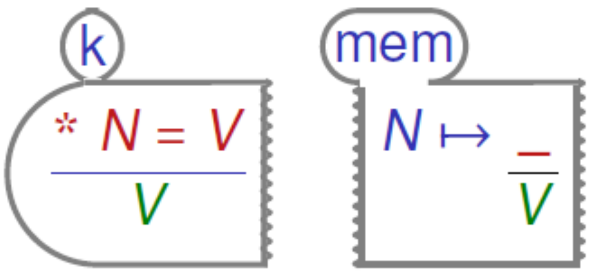
Concurrent updates (on distinct locations)

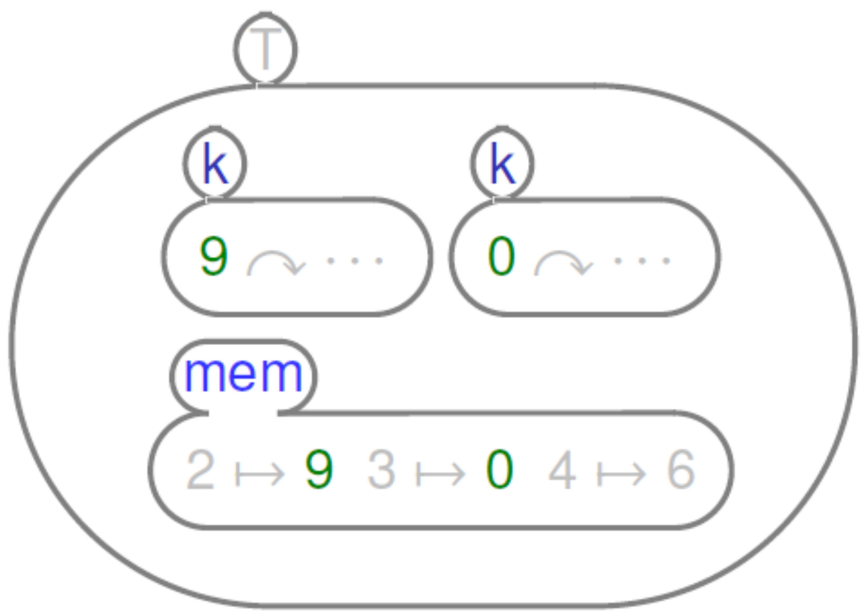




𝕂 rules enhance concurrency

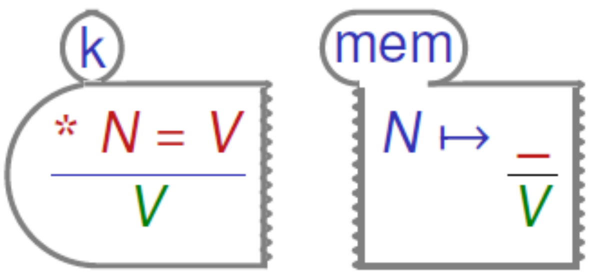
Concurrent updates (on distinct locations)

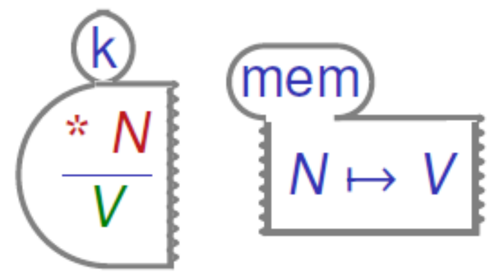


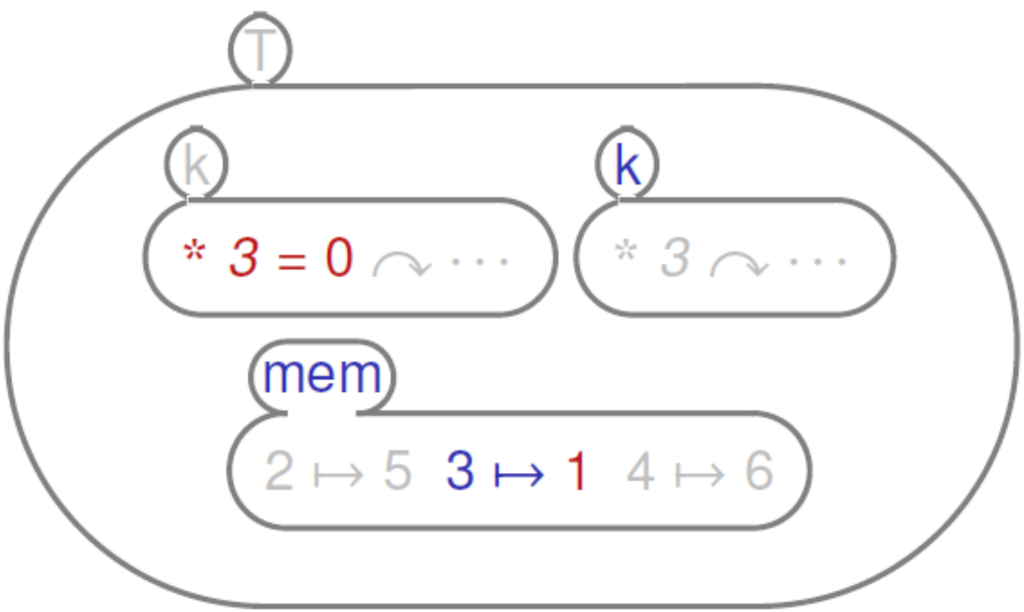


𝕂 rules enhance concurrency

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







Formally capturing the concurrency of $\mathbb{K}$

Idea: Give semantics to $\mathbb{K}$ rewriting through graph rewriting

- $\mathbb{K}$ 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 $\mathbb{K}$ rewriting

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

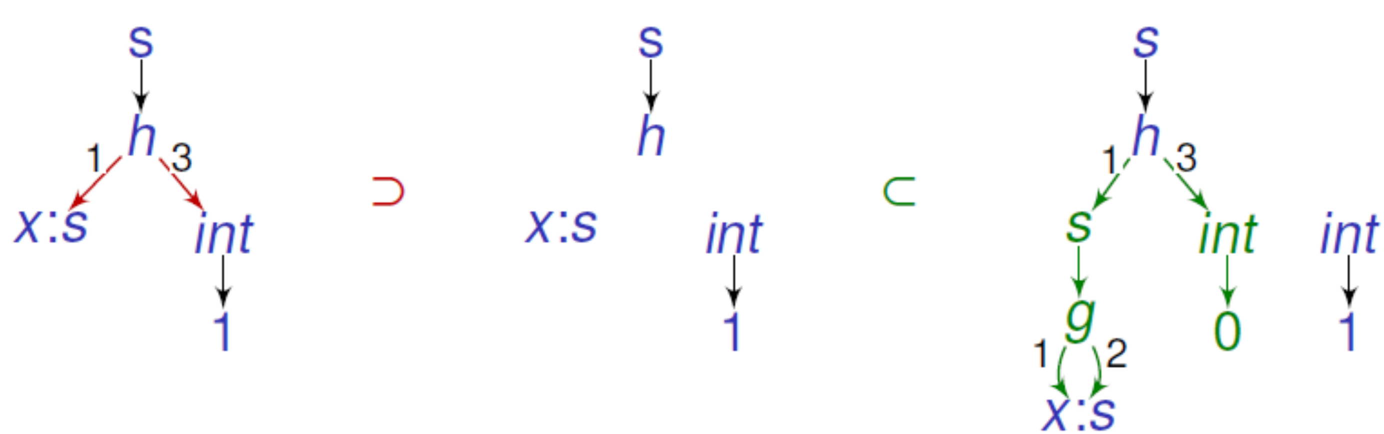
$\mathbb{K}$ rule ρ

$$h\left(\frac{x}{g(x,x)}, \_, \frac{1}{0}\right)$$

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

$$h(x, y, 1) \to h(g(x, x), y, 0)$$

Corresponding graph rewrite rule $K2G(\rho)$



$\mathbb{K}$ rewriting

Definition

Let $\mathbb{S}$ be a $\mathbb{K}$ rewrite system and t be a term. Then

$$t \overset{\mathbb{S}}{\underset{\mathbb{K}}{\Rrightarrow}} t' \quad \text{iff} \quad K2G(t) \xRightarrow[\mathit{Graph}]{K2G(\mathbb{S})} H \text{ such that } \mathrm{term}(H) = t'$$

Theorem (Correctness w.r.t. rewriting)

Soundness: *If* $t \overset{\rho}{\underset{\mathbb{K}}{\Rrightarrow}} t'$ *then* $t \xRightarrow[\mathit{Rew}]{K2R(\rho)} t'$.

Completeness: *If* $t \xRightarrow[\mathit{Rew}]{K2R(\rho)} t'$ *then* $t \overset{\rho}{\underset{\mathbb{K}}{\Rrightarrow}} t'$.

Serializability: *If* $t \overset{\rho_1 + \cdots + \rho_n}{\underset{\mathbb{K}}{\Rrightarrow}} t'$ *then* $t \overset{\rho_1{}^*}{\underset{\mathbb{K}}{\Rrightarrow}} \cdots \overset{\rho_n{}^*}{\underset{\mathbb{K}}{\Rrightarrow}} t'$ *(and thus,* $t \xRightarrow[\mathit{Rew}]{*} t'$*).*

Representing $\mathbb{K}$ into RWL

Core of the K-Maude tool

Faithfully, in two steps

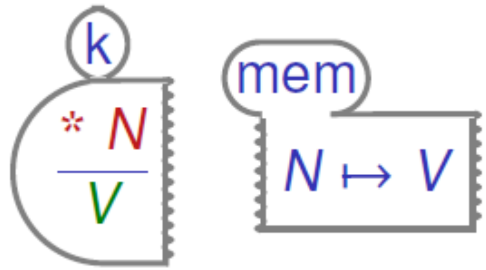
- Represent $\mathbb{K}$ rewriting as $\mathbb{K}$ graph rewriting
- RWL captures graph rewriting as theories over concurrent objects
  - Theoretical, non executable
- Useful to reason about the amount of concurrency in a $\mathbb{K}$ definition

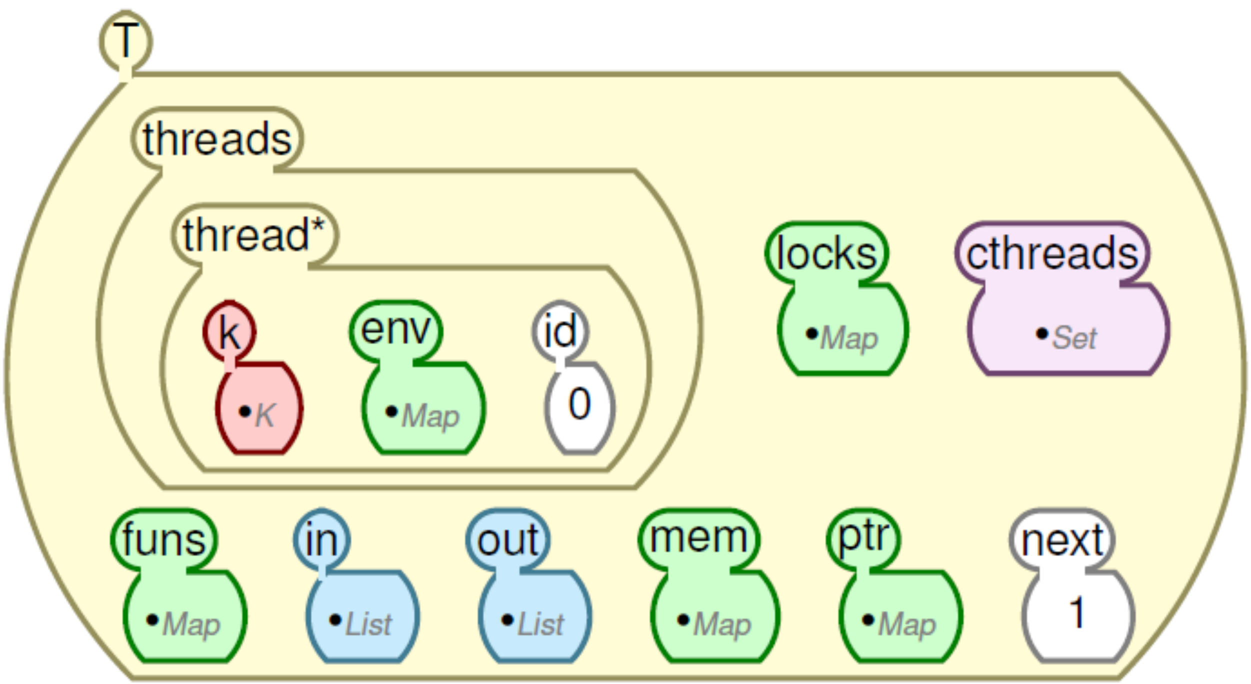
Directly—Implemented in the K-Maude tool

- 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 $\mathbb{K}$ into RWL (core of K-Maude)

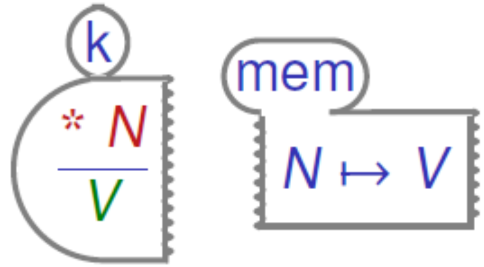
Dereferencing rule



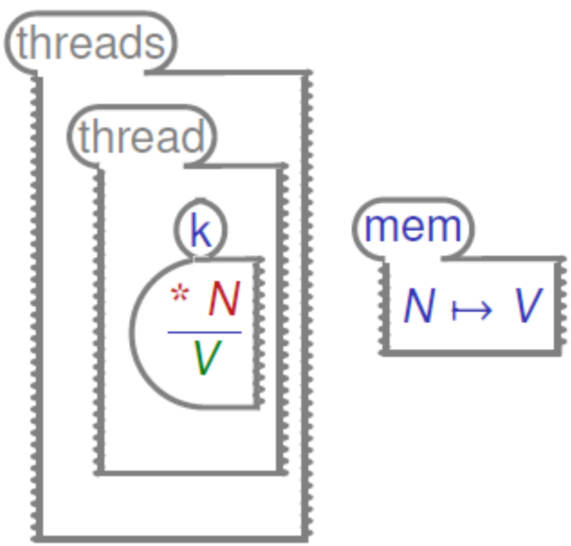


Direct embedding of $\mathbb{K}$ into RWL (core of K-Maude)

Dereferencing rule

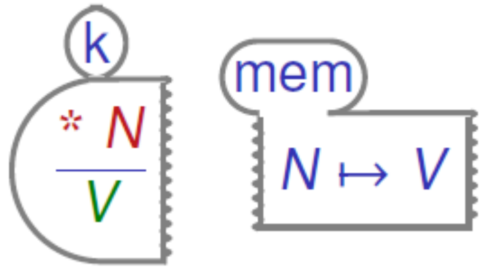


Configuration-concretized version of the rule



Direct embedding of 𝕂 into RWL (core of K-Maude)

Dereferencing rule



Flattening the 𝕂 rule to a rewrite rule

```
rl  <threads> <thread>  <k> * N ↷ ?1:K </k> ?2:Bag </thread> ?3:Bag </threads>
     <mem> N ↦ V  ?4:Map </mem>
=> <threads> <thread>  <k>  V  ↷ ?1:K </k> ?2:Bag </thread> ?3:Bag </threads>
     <mem> N ↦ V  ?4:Map </mem>
```

K-Maude overview [Șerbănuță, Roșu, 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 Ilseman, David Lazar
Javascript	Maurice Rabb
Scheme	Patrick Meredith
X10	Milos Gligoric
Matching Logic	Elena Naum, Andrei Ştefănescu
CEGAR	Irina Asăvoae, Mihail Asăvoae
Teaching	Dorel Lucanu, Grigore Roşu
Interface	Andrei Arusoaie, Michael Ilseman

Summary of contributions

$\mathbb{K}$: 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 $\mathbb{K}$ definitions

- $\mathbb{K}$ 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 $\mathbb{K}$ as a programming language
  - Compiling $\mathbb{K}$ 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 $\mathbb{K}$ definitions