AN EXAMPLE-BASED INTRODUCTION TO K

Dorel Lucanu

Alexandru Ioan Cuza University, Iasi, Romania

dlucanu@info.uaic.ro

Sinaia, March 1st 2011

Joint work with Grigore Rosu and Traian Serbanuta

CONTEXT

A short motivation, K and DAK projects

K Project

- Started in 2003 by Grigore Rosu at UIUC, motivated mainly by teaching programming languages and noticing that the existing semantic frameworks have limitations

- Project thesis:
  - Rewriting gives an appropriate environment to formally define the semantics of real-life programming languages and to test and analyze programs written in those languages.
- UIUC team
  - Chucky Ellison, Michael Ilseman, Patrick Meredith, Grigore Rosu, Traian Serbanuta, Andrei Stefanescu

DAK

- DAK is a Romanian funded project
- DAK goal: to contribute at the development of the K framework (semantics execution engine, analysis tools, definition of languages)
- Grigore Rosu is the external expert
  - a strong cooperation between the two groups from UIUC and UAIC
- UAIC team:
  - Andrei Arusoae, Irina Asavoae, Mihai Asavoae, Gheorghe Grigoras, Dorel Lucanu, Radu Mereuta, Elena Naum

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

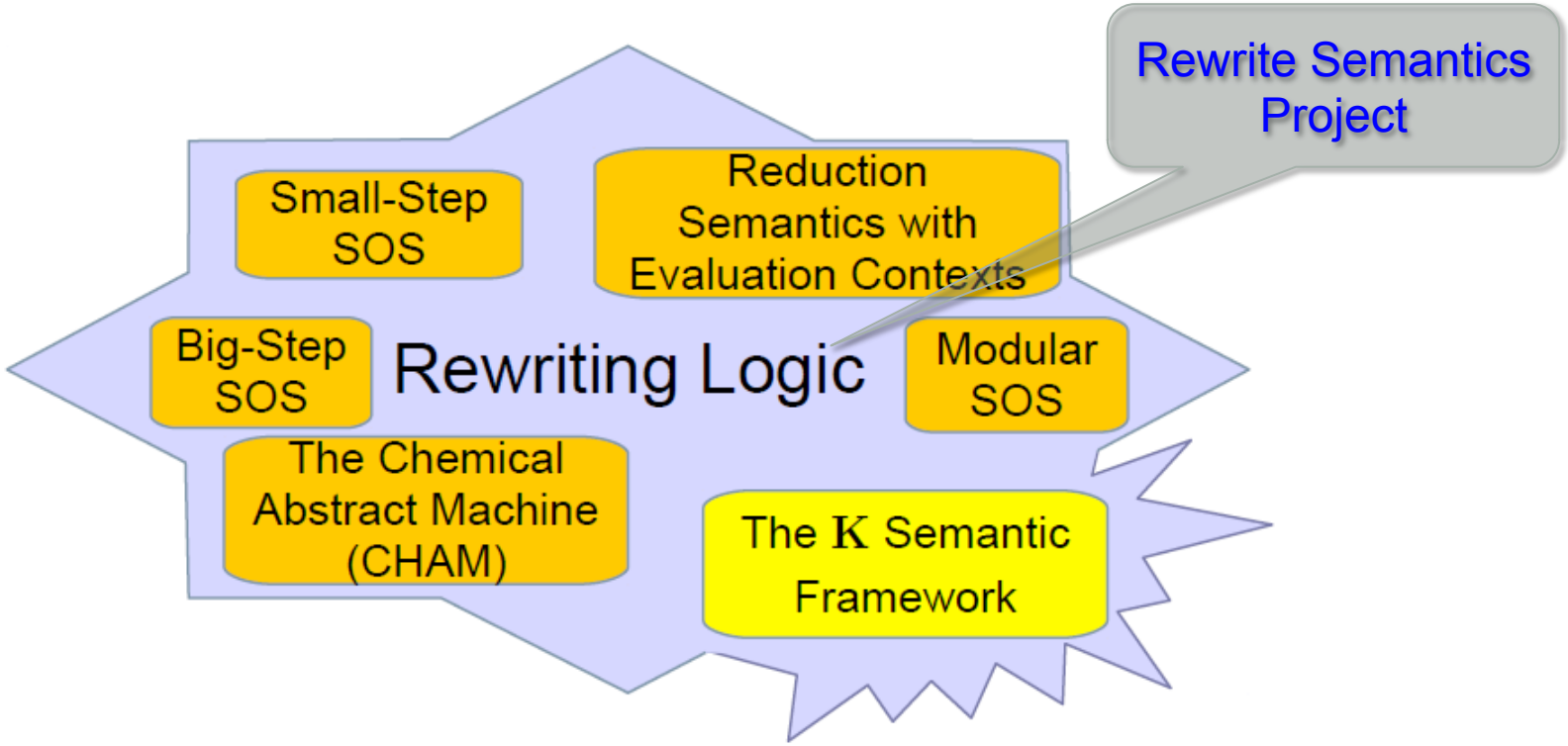
Shortcomings of Existing Frameworks

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

K FRAMEWORK

based on Grigore's and Traian's presentations and Cink example

The $\mathbb{K}$ Framework



The $\mathbb{K}$ framework

- $\mathbb{K}$ technique: for expressive, modular, versatile, and clear PL definitions
- $\mathbb{K}$ 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

K in a nutshell (cont.)

- the semantics is given by means of a set of rewrite rules transforming the abstract syntax trees (ASTs) into results, eventually using some intermediate structures
- the notion of result is a generic one: could be either the output, the result of a type-checking algorithm, the result of a static analyser/verifier and so on
- the machine on which the programs are executed is abstractly described as a configuration of cells
- examples of cells: computation steps, environment, memory, call stack, formulas to be verified
- K Rewrite Abstract Machine (KRAM) executes the rewrite rules in faithful way

Running example: Cink

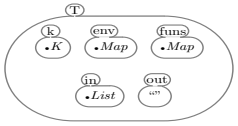
- a kernel of C
  - functions
  - int expressions
  - input/output
  - basic flow control (if, if-else, while, sequential composition)
  - pointers and arrays
  - structures
- in this talk
  - a K semantic definition of Cink (without pointers and structures)
  - a static analyzer derived from K definition (infeasible paths, infinite loops, reading non-initialized variables, …)

K definition of Cink

MODULE CINK-SYNTAX
IMPORTS PL-ID+PL-INT
DeclId ::= `int` *Exp*
| `void` *Id*
Exp ::= *Int*
| *Id*
| *Exp* `+` *Exp* [strict]
| *Exp* `-` *Exp* [strict]
| *Exp* `*` *Exp* [strict]
| *Exp* `>` *Exp* [strict]
| *Exp* `=` *Exp* [strict(2)]
| `printf("%d;",` *Exp* `)` [strict]
| `scanf("%d",&` *Id* `)`
| *Id* `(` *List*{*Exp*} `)` [strict(2)]
| *Id* `()`
| *DeclId*
Id ::= `main`
Stmt ::= *Exp* `;` [strict]
| `{}`
| `{` *StmtList* `}`
| `if(` *Exp* `)` *Stmt*
| `if(` *Exp* `)` *Stmt* `else` *Stmt* [strict(1)]
| `while(` *Exp* `)` *Stmt*
| `return` *Exp* `;`
| *DeclId* `(` *List*{*DeclId*} `){` *StmtList* `}`
| *DeclId* `(){` *StmtList* `}`
StmtList ::= *Stmt*
| *StmtList* *StmtList*
Pgm ::= *StmtList*
List{*Bottom*} ::= `.Bottom`
| *Bottom*
| *List*{*Bottom*} `,` *List*{*Bottom*} [id: `.Bottom` strict hybrid assoc]
List{*Id*} ::= *Id*
| *List*{*Bottom*}
| *List*{*Id*} `,` *List*{*Id*} [id: `.Bottom` ditto assoc]
List{*DeclId*} ::= *DeclId*
| *List*{*Bottom*}
| *List*{*DeclId*} `,` *List*{*DeclId*} [id: `.Bottom` ditto assoc]
List{*Exp*} ::= *Exp*
| *List*{*Id*}
| *List*{*DeclId*}
| *List*{*Exp*} `,` *List*{*Exp*} [id: `.Bottom` ditto assoc]
END MODULE

MODULE CINK-DESUGARED-SYNTAX
IMPORTS CINK-SYNTAX
MACRO: `if(` *E* `)` *St* = `if(` *E* `)` *St* `else {}`
MACRO: *I* `()` = *I* `(` `.Bottom` `)`
MACRO: *DeclId* `(){` *Stmts* `}` = *DeclId* `(` `.Bottom` `){` *Stmts* `}`
MACRO: `void` *X* = `int` *X*
MACRO: `int` *X* `=` *E* `;` = `int` *X* `;` *X* `=` *E* `;`
END MODULE

MODULE CINK-SEMANTICS
IMPORTS K-SHARED
IMPORTS PL-CONVERSION+K+CINK-DESUGARED-SYNTAX
Val ::= *Int*
| `void`
Exp ::= *Val*
List{*Val*} ::= *List*{*Val*} `,` *List*{*Val*} [id: `.Bottom` ditto assoc]
| *Val*
List{*Exp*} ::= *List*{*Val*}
KResult ::= *List*{*Val*}
K ::= *List*{*Exp*}
| *List*{*Id*}
| *List*{*DeclId*}
| *StmtList*
| *Pgm*
| *String*
Nat ::= `initialLoc`
K ::= `initial`
| `restore(` *Map* `)`
| `increment(` *Nat* `,` *Nat* `)`
| `endOfFunction`
List{*K*} ::= *Nat* `..` *Nat*
| `varNameList(` *List*{*K*} `)`
INITIAL CONFIGURATION:



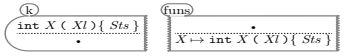
RULE: I_1 + I_2 ⇀ I_1 $-_{Int}$ I_2

RULE: I_1 - I_2 ⇀ I_1 $-_{Int}$ I_2

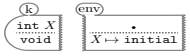
RULE: I_1 * I_2 ⇀ I_1 $*_{Int}$ I_2

RULE: I_1 > I_2 ⇀ `Bool2Int (` I_1 $>_{Int}$ I_2 `)`

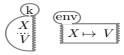
FUN-DECL RULE:



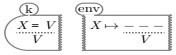
VAR-DECL RULE:



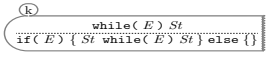
MEM-LOOKUP RULE:



MEM-UPDATE RULE:



WHILE RULE:



IF-FALSE RULE: `if(` I `)` $---$ `else` St ⇀ St when $I ==_{Int} 0$

IF-TRUE RULE: `if(` I `)` St `else` $---$ ⇀ St when $\neg_{Bool} I ==_{Int} 0$

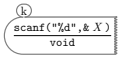
INSTR-EXPR RULE: V `;` ⇀ •

BLOCK RULE: `{` Sts `}` ⇀ Sts

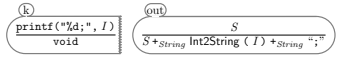
BLOCK-EMPTY RULE: `{}` ⇀ •

SEQ-COMP RULE: St Sts ⇀ St ↷ Sts

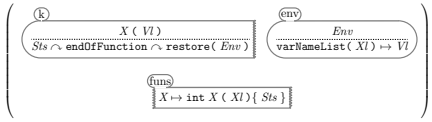
READ-LOCAL RULE:



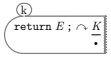
PRINT RULE:



FUN-CALL RULE:

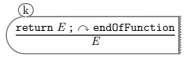


RETURN-MIDDLE RULE:

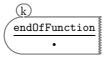


when $K \neq_{Bool}$ `endOfFunction`

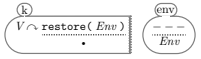
RETURN-LAST RULE:



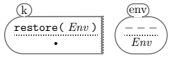
NO-RETURN RULE:



NONVOID-FUN-RETURN RULE:



VOID-FUN-RETURN RULE:



RULE: N_1 `..` N_1 ⇀ *List*{*K*}

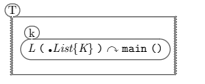
RULE: N_1 `..` s_{Nat} N ⇀ N , N_1 `..` N

RULE: `varNameList(` *Kl* `)` ⇀ `eraseKLabel (` `int_` `,` *Kl* `)`

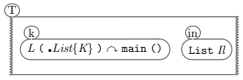
END MODULE

MODULE CINK
IMPORTS K-SHARED
IMPORTS CINK-SEMANTICS+CINK-PROGRAMS+CINK-SYNTAX
Bag ::= `run(` *KLabel* `)`
| `run(` *KLabel* `,` *List*{*K*} `)`

RULE: `run(` *L* `)` ⇀



RULE: `run(` *L* `,` *Il* `)` ⇀



END MODULE

$\mathbb{K}$ 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

$Exp ::= Id$		
$\mid * Exp$	[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 ; * x = * y ; * y = t ;

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

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

Heating syntax through strictness rules

Computation

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

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

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

Heating syntax through strictness rules

Computation

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

$\mathbb{K}$ 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

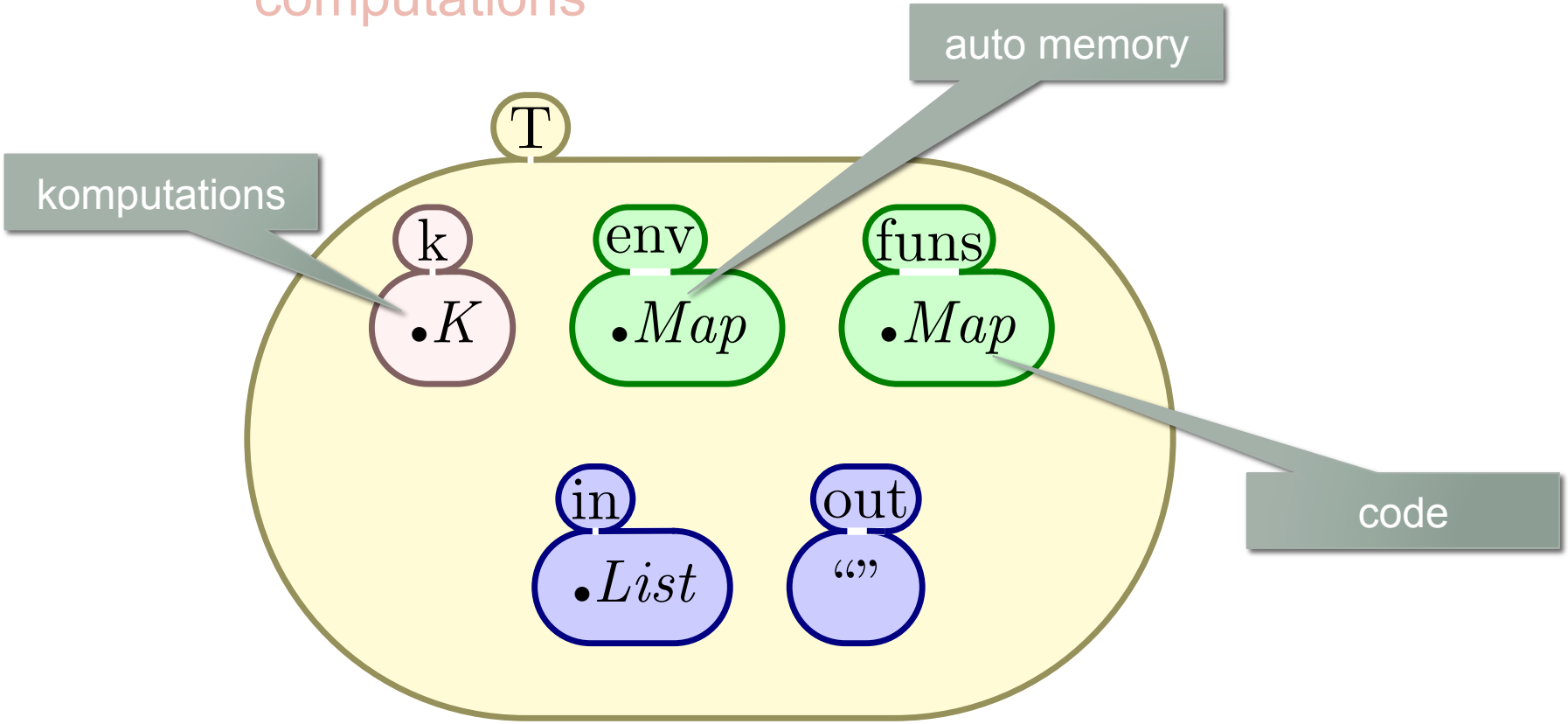
$$x \curvearrowright *\,\square \curvearrowright t = \square \curvearrowright \square; \curvearrowright \square\ *\,x = *\,y\ ;\ *\,y = t\ ;$$

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

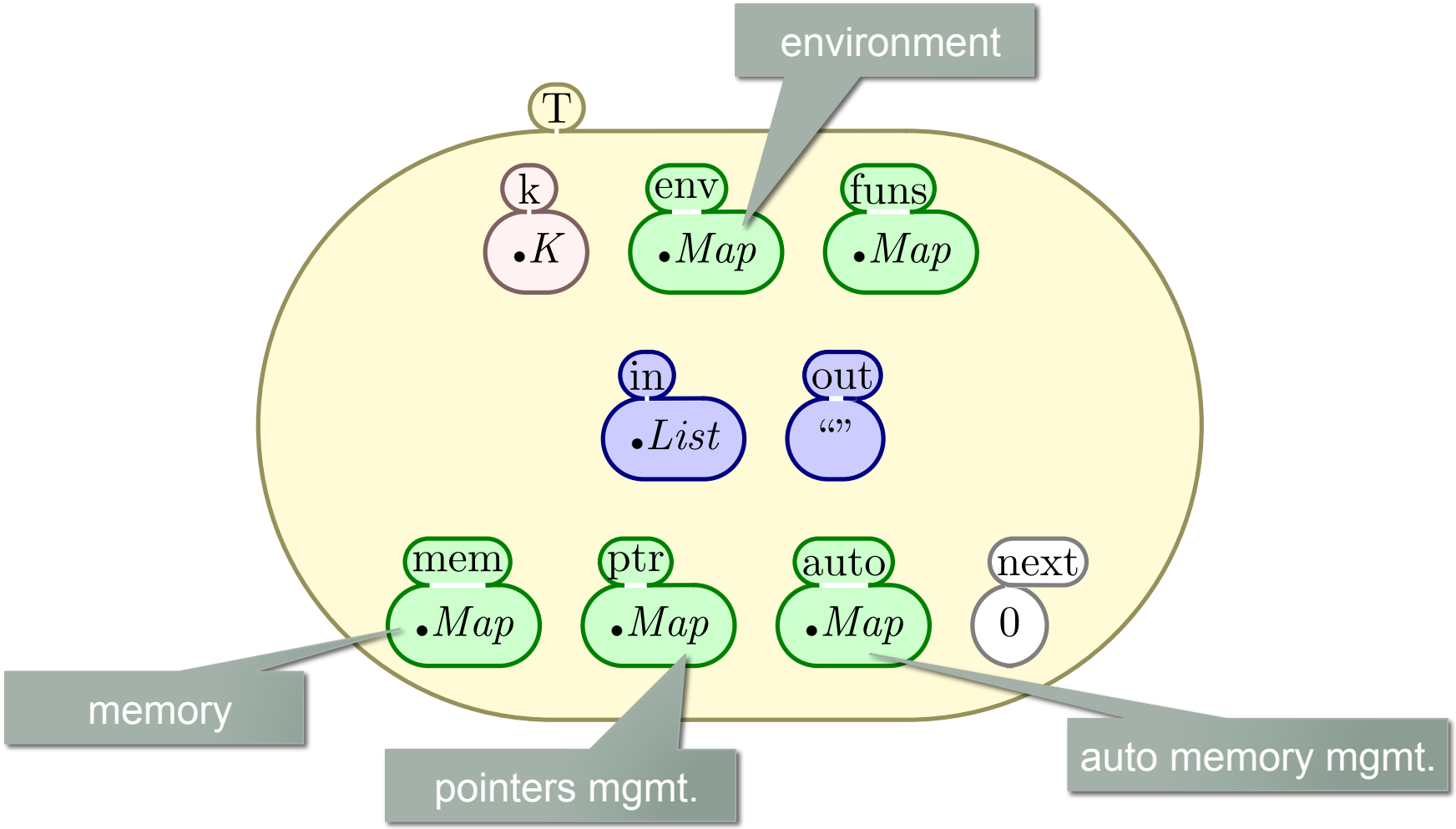
Exp ::= *Id*		
| * *Exp*	[strict]	$*\,ERed \rightleftharpoons ERed \curvearrowright *\,\square$
| *Exp* = *Exp*	[strict(2)]	$E = ERed \rightleftharpoons ERed \curvearrowright E = \square$
Stmt ::= *Exp* ;	[strict]	$ERed\ ; \rightleftharpoons ERed \curvearrowright \square\ ;$
| *Stmt* *Stmt*	[seqstrict]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

Configuration for Cink

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



Configuration for Cink with pointers

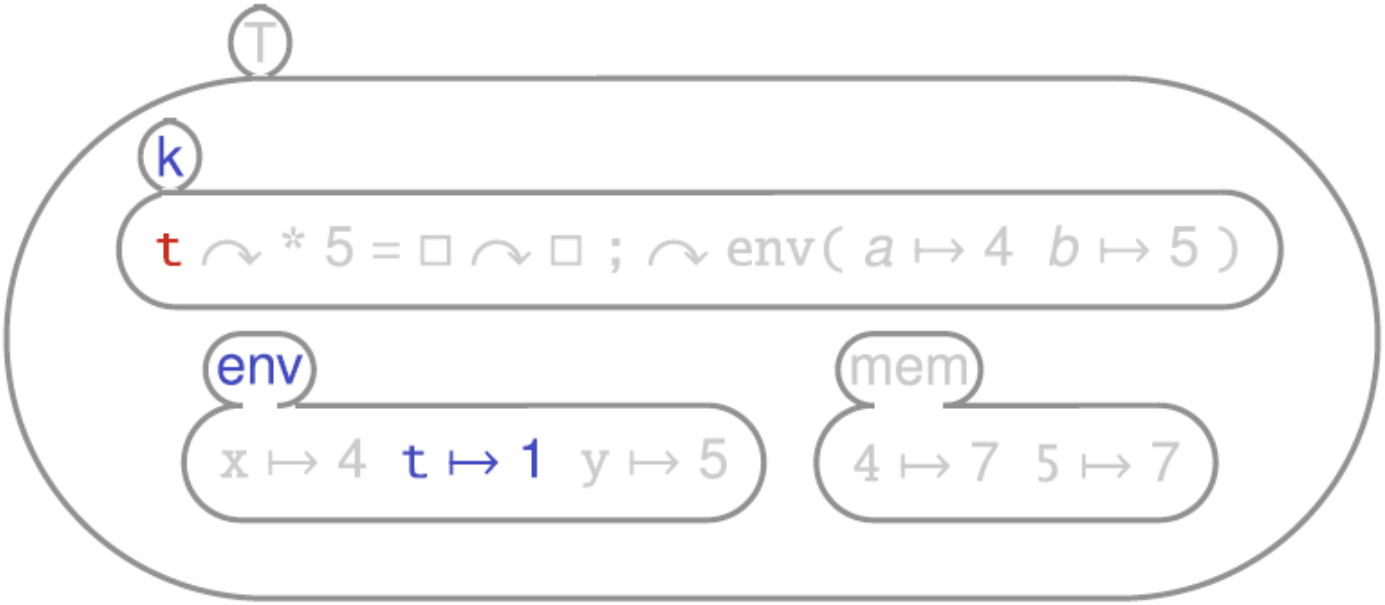


$\mathbb{K}$ rules: expressing natural language into rules

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

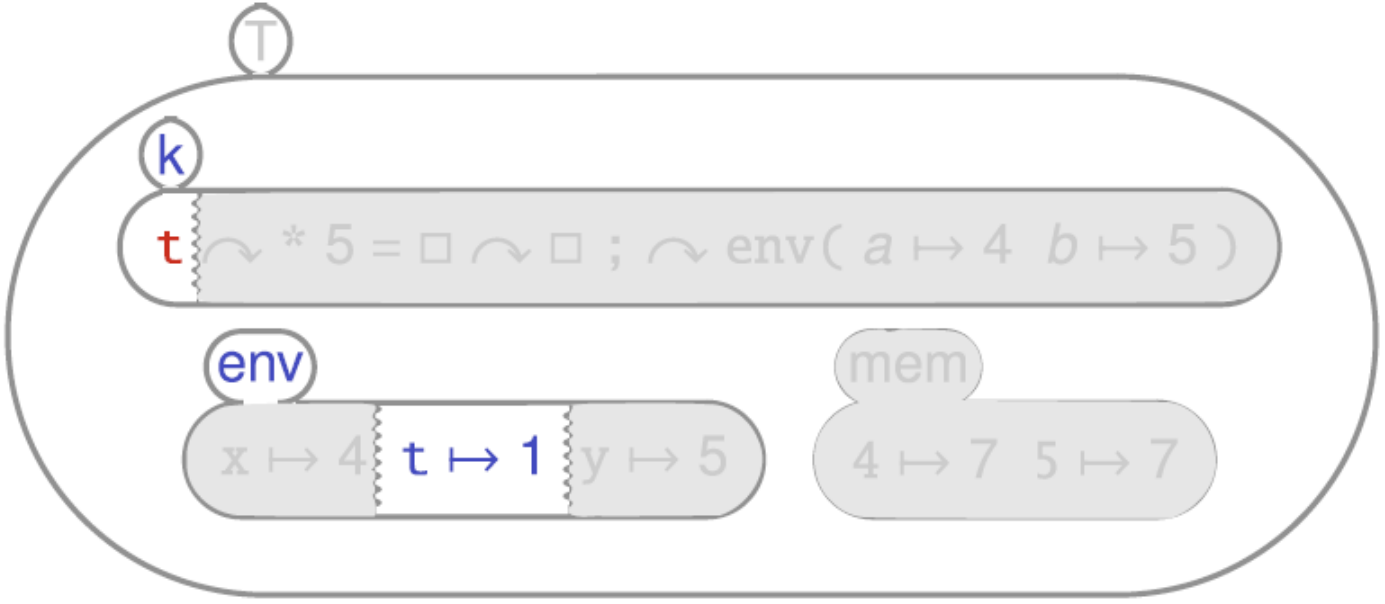


$\mathbb{K}$ rules: expressing natural language into rules

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

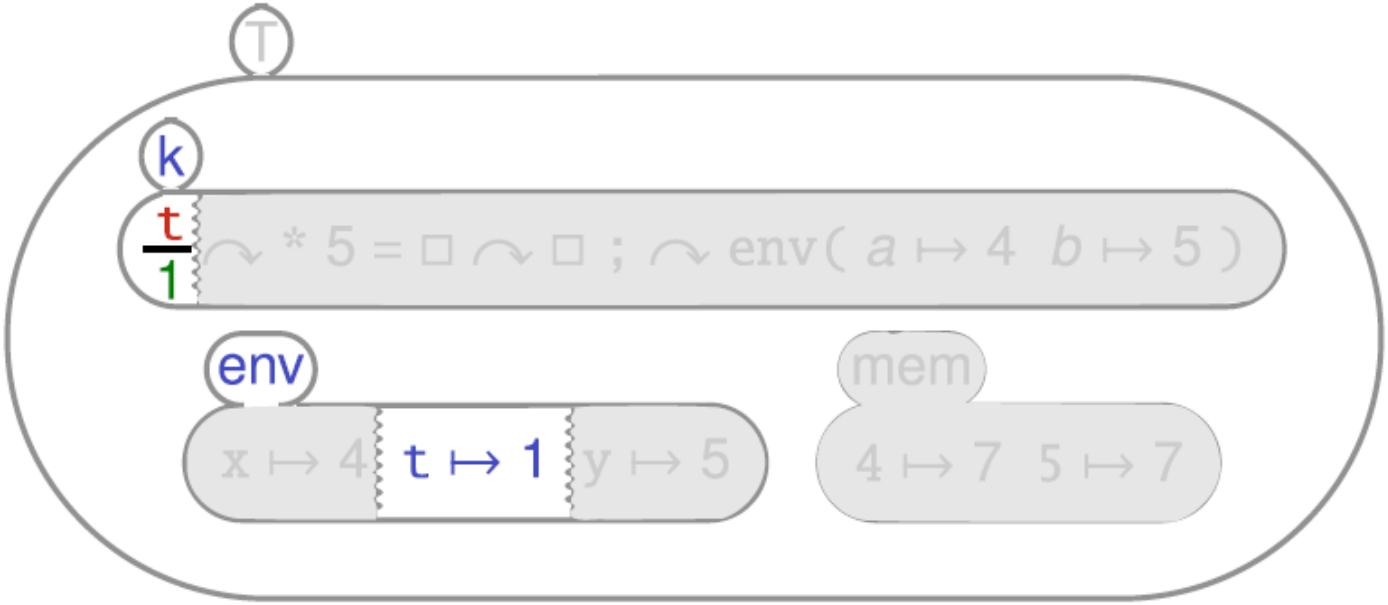


𝕂 rules: expressing natural language into rules

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

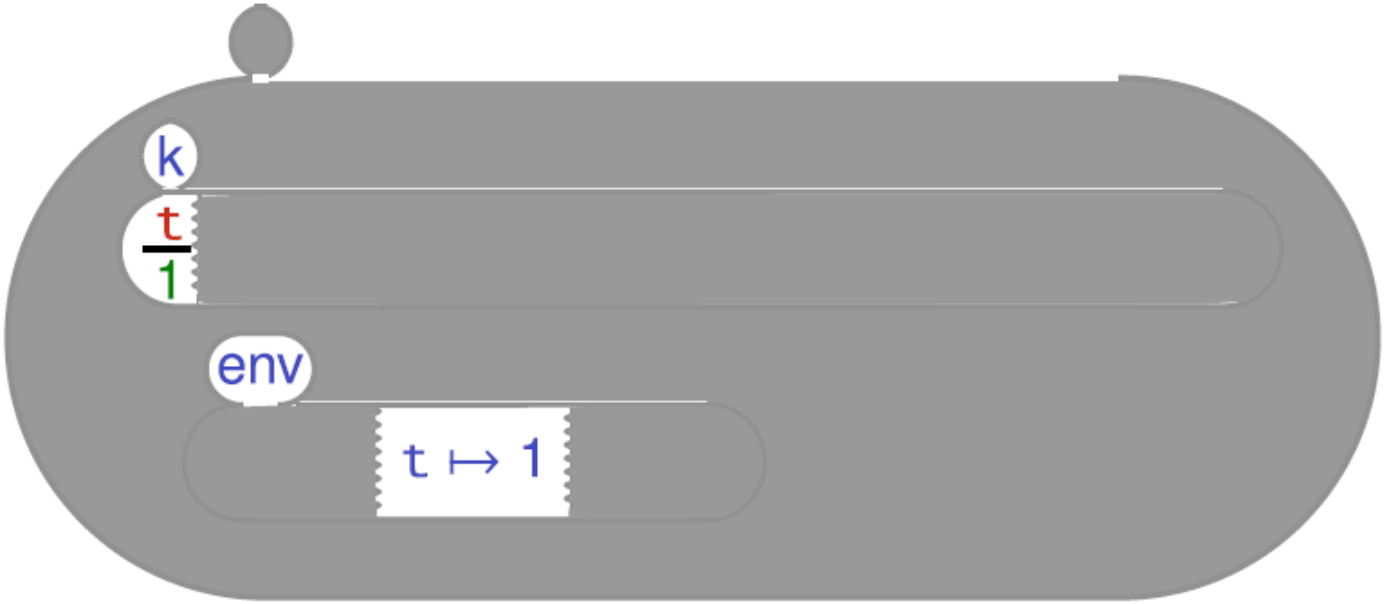


$\mathbb{K}$ rules: expressing natural language into rules

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

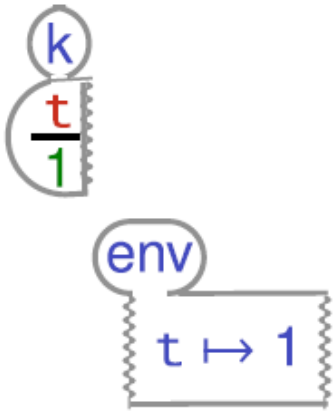


$\mathbb{K}$ rules: expressing natural language into rules

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

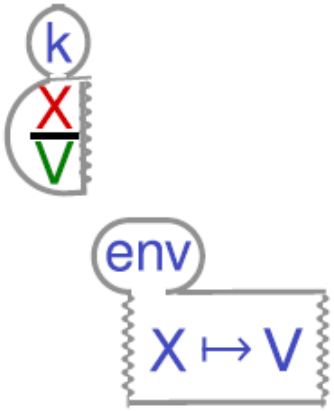


$\mathbb{K}$ rules: expressing natural language into rules

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

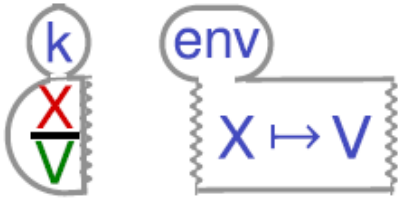


$\mathbb{K}$ rules: expressing natural language into rules

Voilà!

Reading from environment

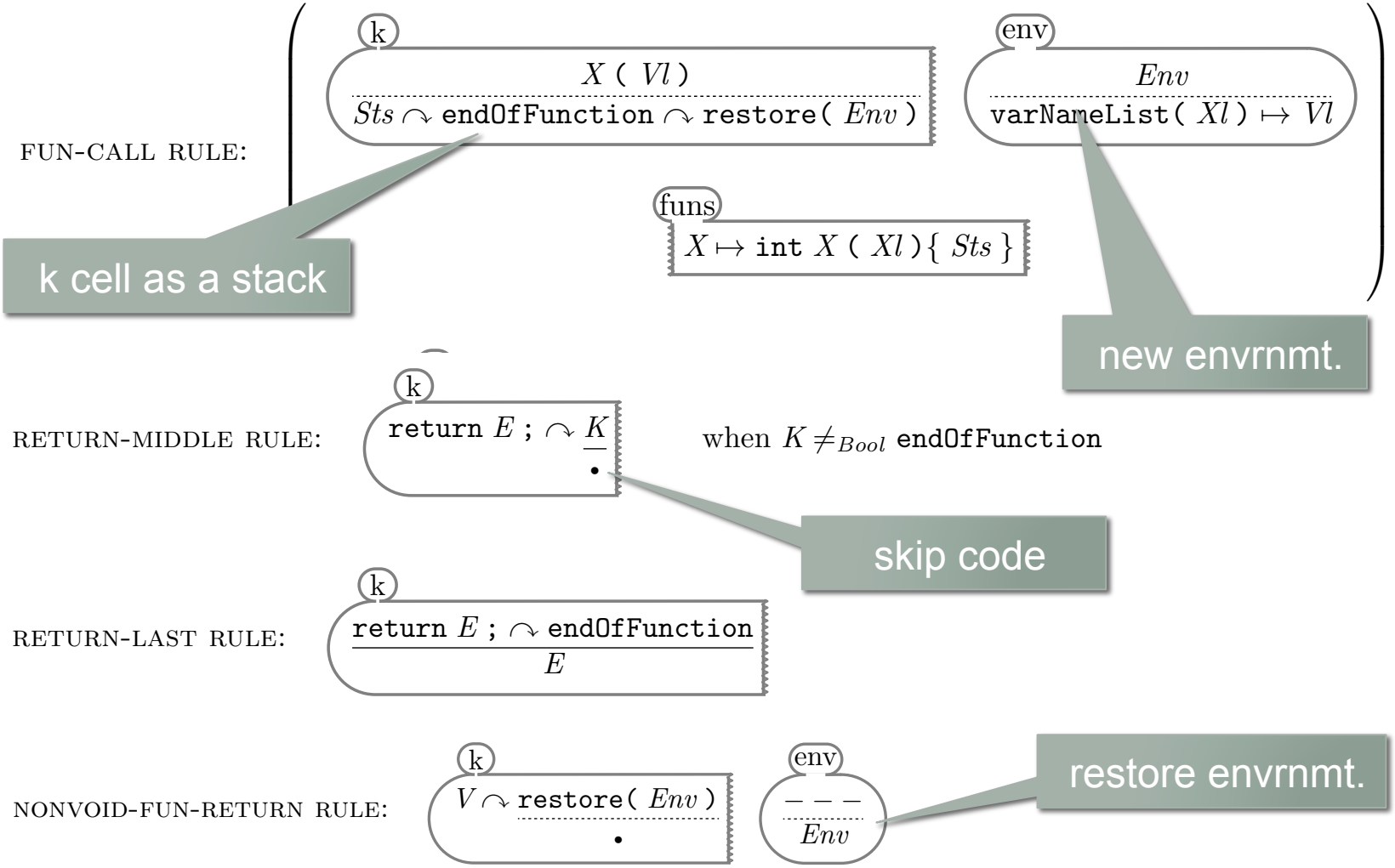
If a local variable X is the next thing to be processed . . .
. . . and if X is mapped to a value V in the environment . . .
. . . then process X, replacing it by V



- ASCII notation:

```
rule <k> X:Id => V <_/k>
     <env_> X |-> V <_/env>
```

Examples of rules



DEMO

FROM PL DEFINITION TO A STATIC ANALYZER

A simple static analyzer for Cink

From the definition to semantic tools

- We may take the advantage of having *a formal definition of PL* and build analyzing and verification tools which are sound w.r.t. the formal definition
- it is recommended to have just one formal definition
- for all tools, it can be proved the soundness w.r.t. this definition
- in this talk we present a static analyzer for Cink, able to discover infinite cycles, unfeasible paths in the flow graph, reading uninitiated variables
- the analyser is obtained by transforming the concrete semantics into a symbolic execution

Symbolic values

- we extend Int with symbolic values SymInt
- the value of a variable can be an axpression

  Int SymInt < ExpInt
- we assume a decision procedure SOLVER s.t.

$$\texttt{SOLVER} \models \texttt{s@lve(}\ EB\ \texttt{)} \Rightarrow \texttt{s@t}$$

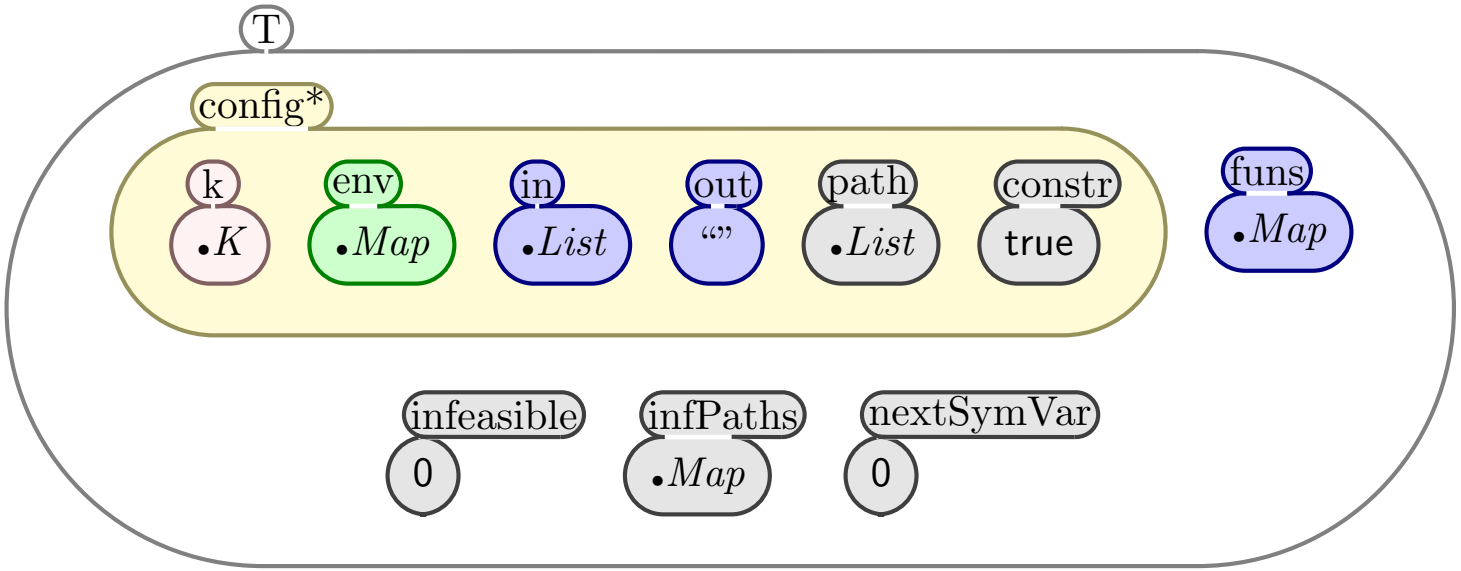
iff EB is satisfiable, and

$$\texttt{SOLVER} \models \texttt{s@lve(}\ EB\ \texttt{)} \Rightarrow \texttt{uns@t}$$

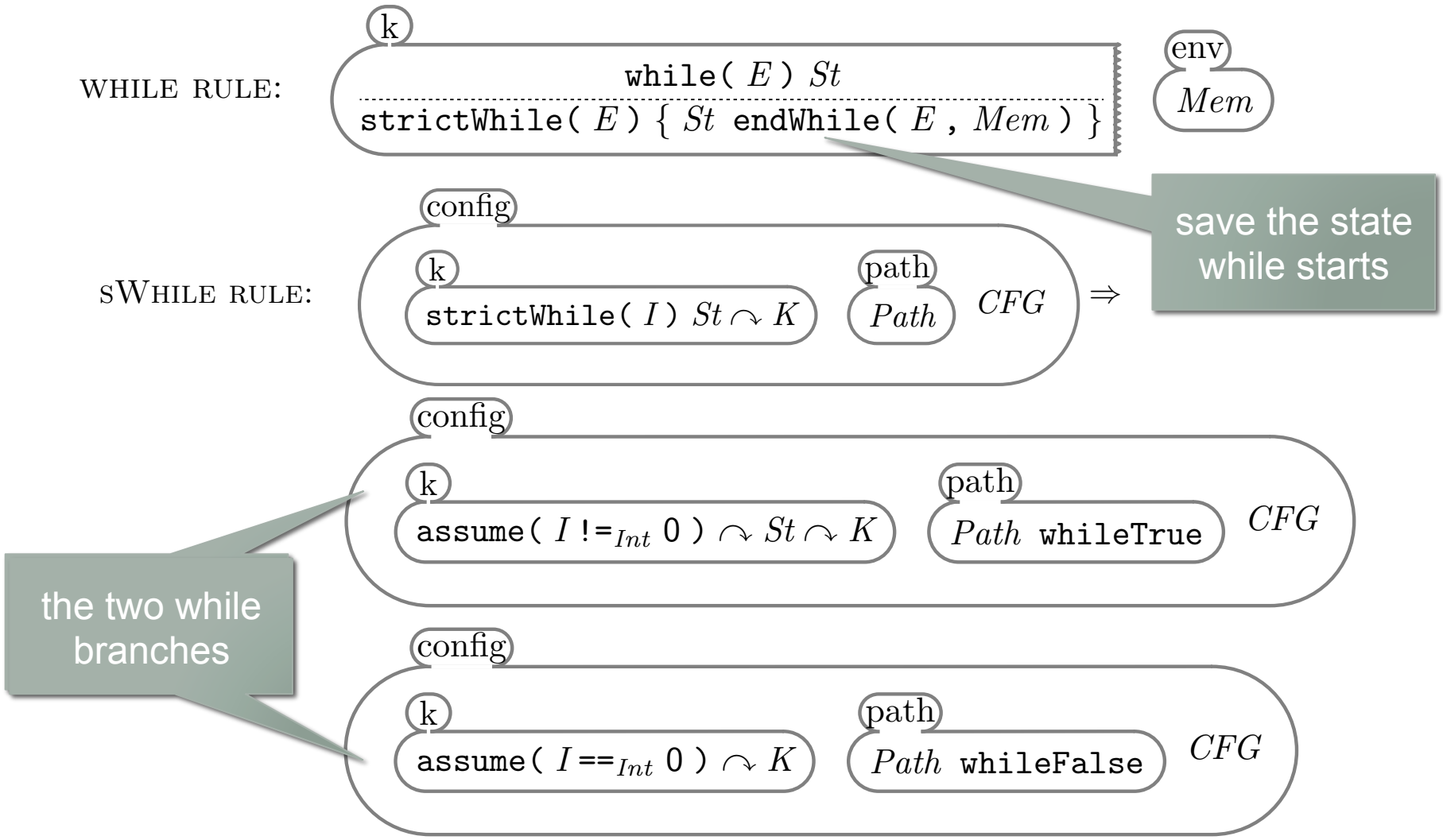
iff EB is satisfiable

Configuration for symbolic execution

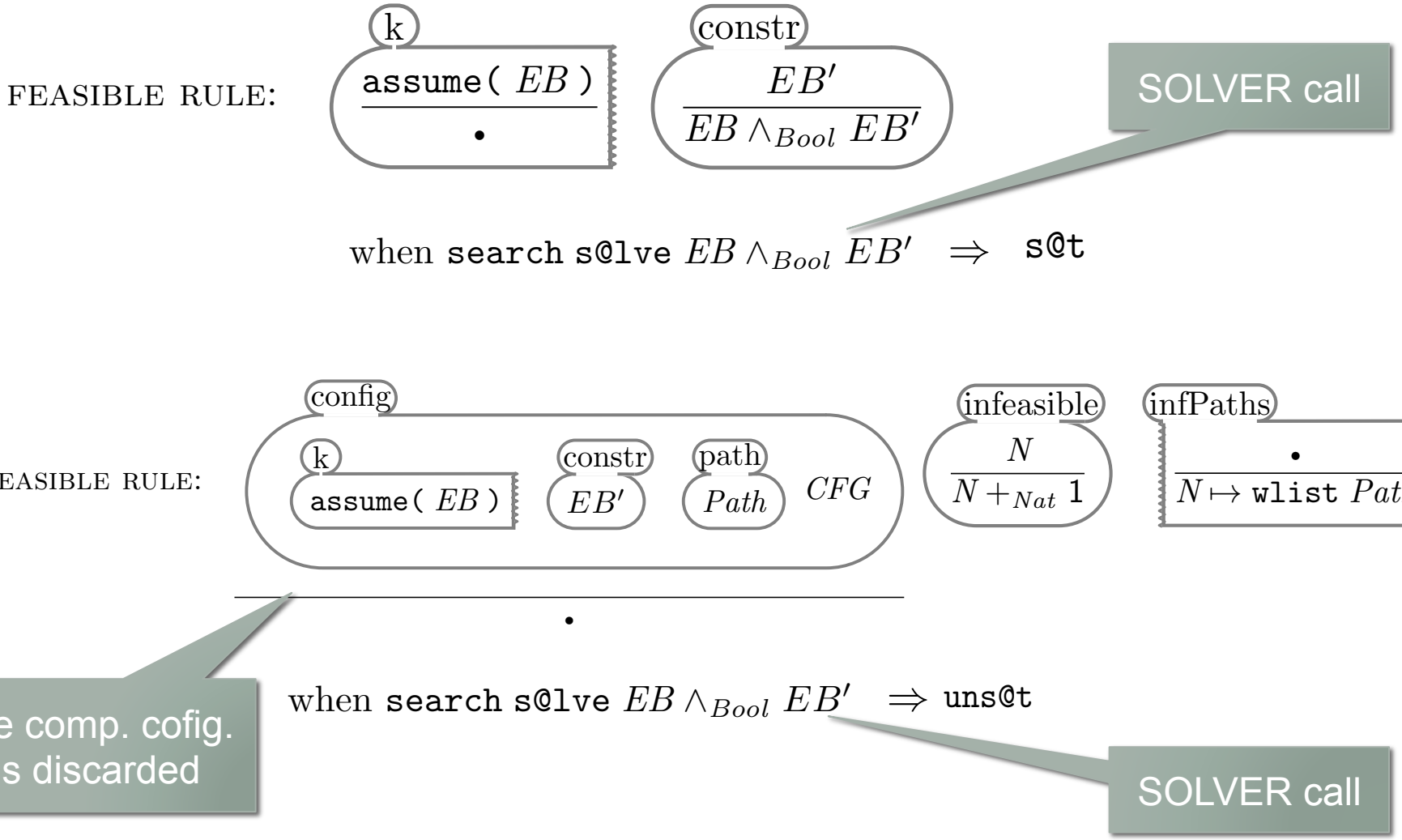
- We reorganize the configuration by
  - allowing many configurations (one for each path in the flow graph),
  - adding cells forconstraints (path formulas),
  - cells for counting and storing unfeasible paths
  - a cell supporitng to generate new symbolic values



Symbolic definition of while

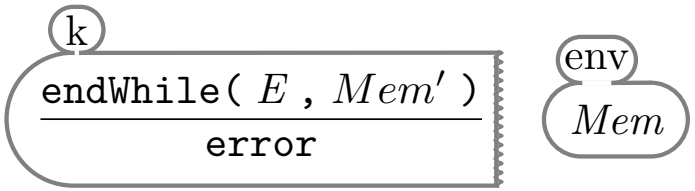


Discovering unfeasible paths



Discovering infinite loops

INFINITE-CYCLE RULE:



when $\texttt{getReduct(}\ Mem\ \texttt{, getVar(}\ E\ \texttt{)}\ \texttt{)} =_{Bool} \texttt{getReduct(}\ Mem'\ \texttt{, getVar(}\ E\ \texttt{)}\ \texttt{)}$

memory of a given set of vars

variables of expression E

DEMO

Conclusion

- K Framework
  - ExpressiveModular—at least as Modular SOS
  - Concurrent
  - Concise
- K Maude
  - a prototype for executing and analyzing K definitions
- Future work
  - improve K Maude tool
  - more formal definitions for real PLs
  - analysis and verification semantics (Matching Logic)