# K: The Concurrent Rewrite Abstract Machine

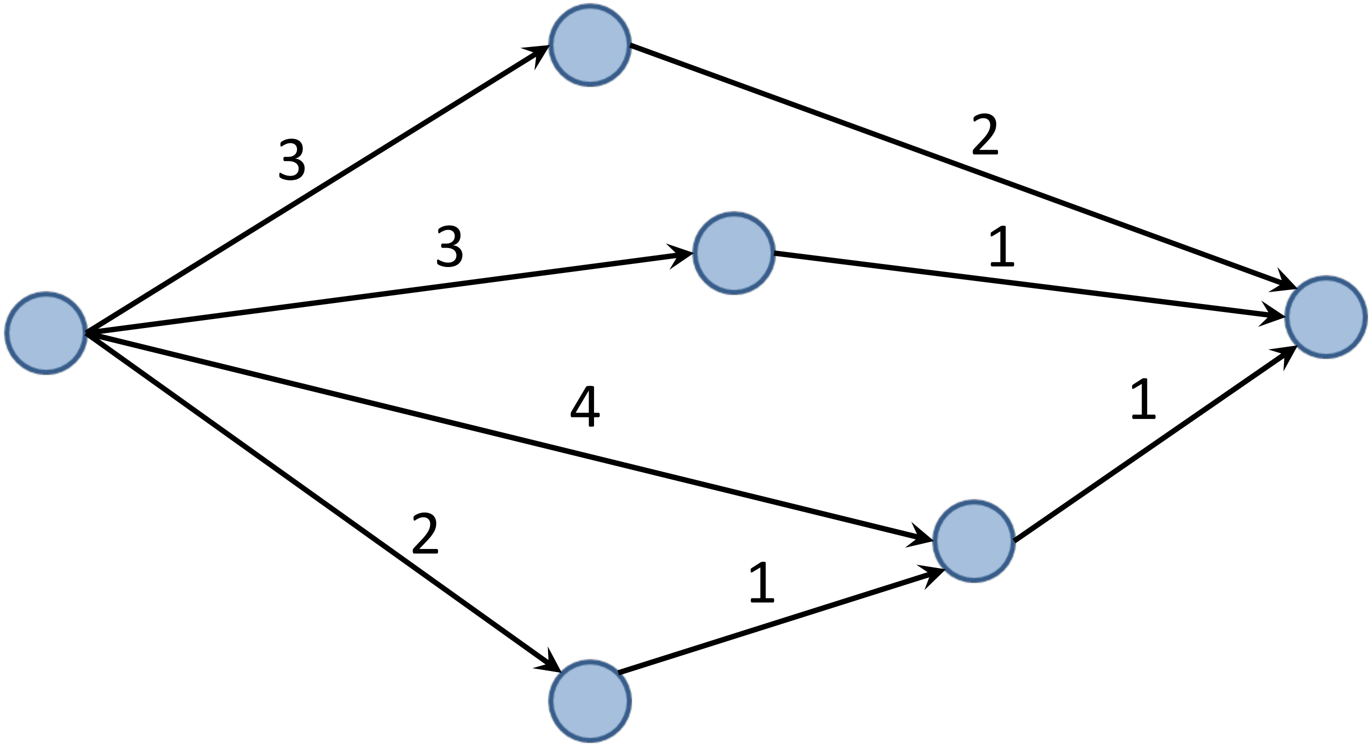
Grigore Rosu

# Note to the reader

- There are many “explanation slides” that I do not use in presentations about K; instead, I include those explanations as part of the talk
- These explanations are here only to better explain the slides to those who just read them
- Please let me know if I should explain better certain parts of this presentation
- Feel free to contact me for pointers to work on K

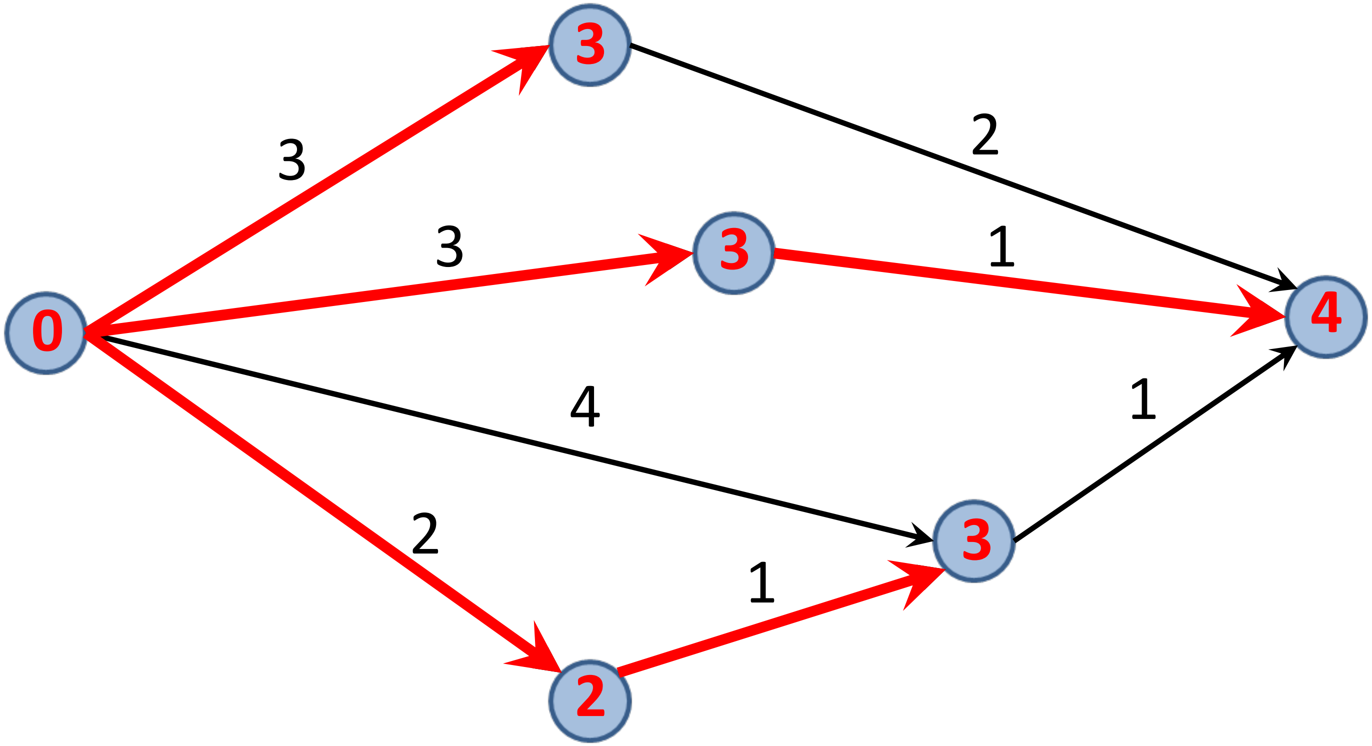
# Dijkstra's Algorithm

All shortest paths in a graph



# Dijkstra’s Algorithm

All shortest paths in a graph



# Dijkstra’s Algorithm in K

- All shortest distances in a graph, concurrently
- Hold pairs **(node,cost)** in a multi-set soup

**(start,0)**

**$(x,c_x)$** **…**

**$(y,c_y)$**

- For each edge $\mathbf{x} \xrightarrow{t} \mathbf{y}$ add a ***K-rule***

$$\left\langle (x,\ c_x) \quad (y,\ \underset{t\,+\,c_x}{\underline{\ c_y\ }}\ ) \right\rangle \quad \text{when } t + c_x < c_y$$

# Explanation for previous slide

- The rule

$$\langle\!|\; (x, c_x) \quad (y,\ \underset{t + c_x}{\underline{c_y}}\ ) \;|\!\rangle \quad \text{when } t + c_x < c_y$$

reads as follows:

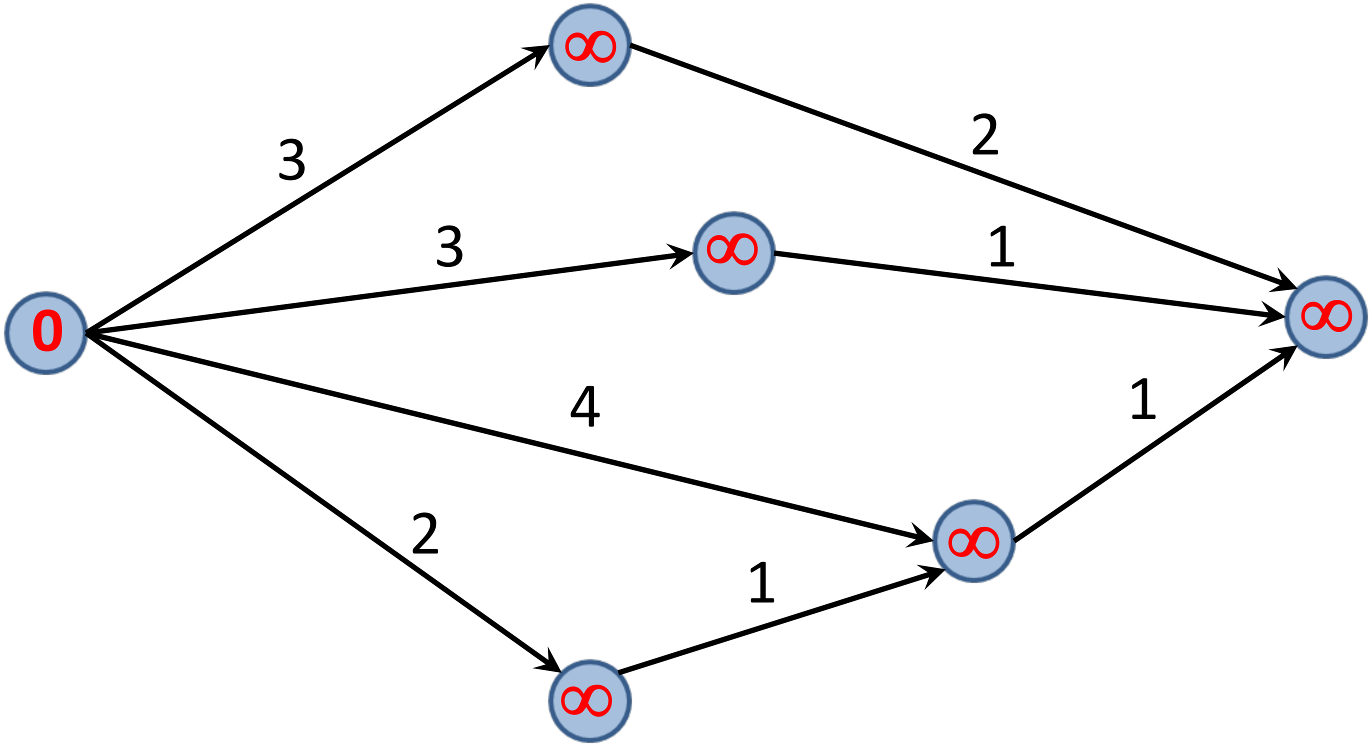
  - Whenever two pairs $(x,c_x)$ and $(y,c_y)$ can be found in the multiset soup (the $\langle\!|$ and $|\!\rangle$ are open soup boundaries) such that $t + c_x < c_y$, replace $c_y$ by $t + c_x$ (K-rules change the underlined subterms as indicated below the line) ; note that in K, unlike in term rewriting, concurrent rule applications can share subterms which are not underlined

# Explanation for next slides

- Making use of sharing information, K defines a concurrent rewriting relation ⇛ , which allows for sharing of read-only subterms (i.e., not underlined)
- **t** ⇛ **t’** means that **t** may concurrently rewrite to **t’**; K does not enforce maximal concurrent rewriting on purpose (it would be easy to add rewriting strategies, including maximal concurrent rewriting, but we do not do it for the time being)
- Next, $\textcircled{c}$ means a pair **(node,c)** in the soup (i.e., the current cost of **node** is **c**), and $\textcircled{c_1} \rightarrow \textcircled{c_2}$ means that a K rule is applied matching the two involved pairs, and the cost of the target node is $\mathbf{c_2}$ after the rule is applied

# Run 1 in K

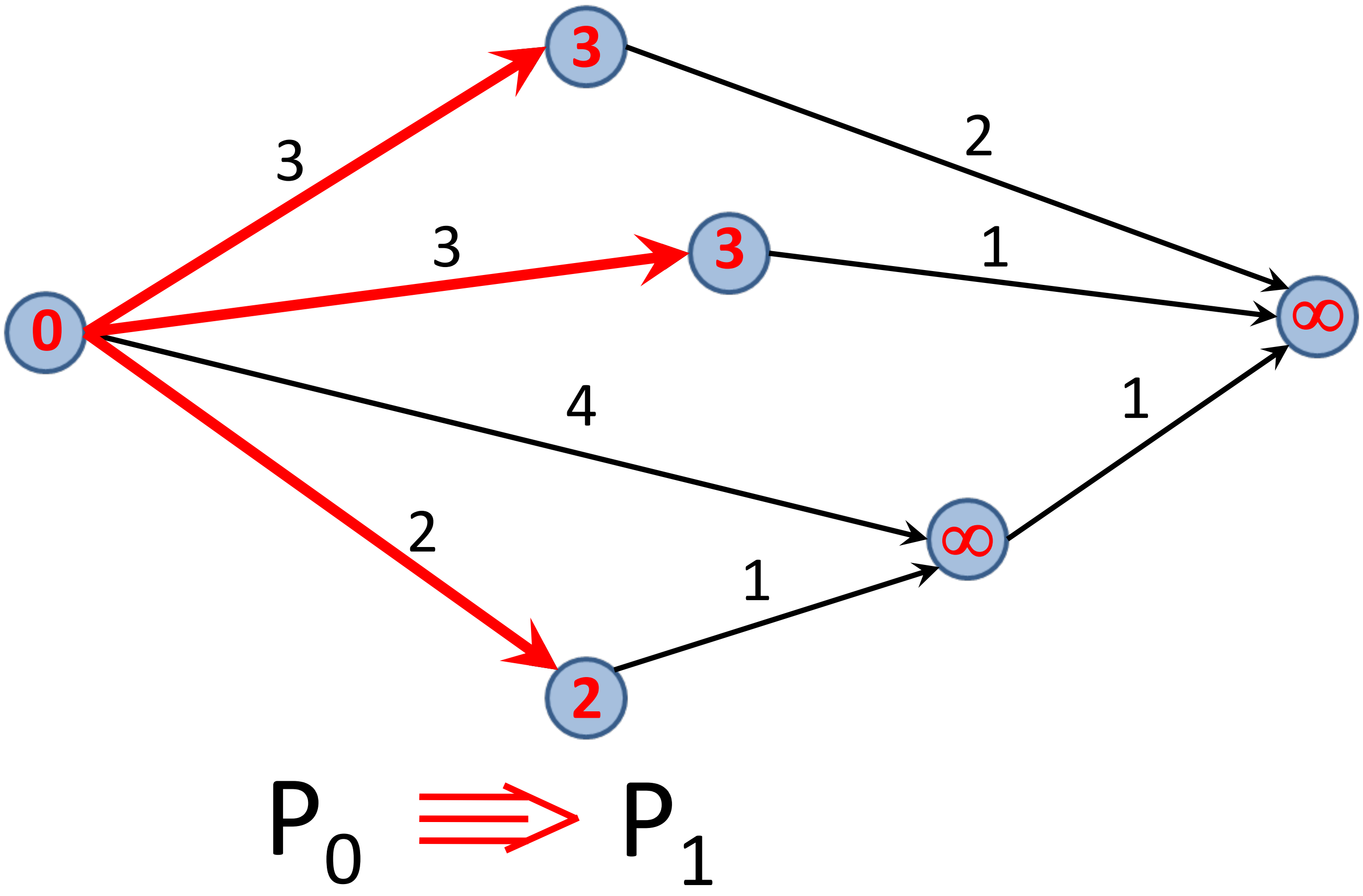
All shortest distances … directly



$P_0$

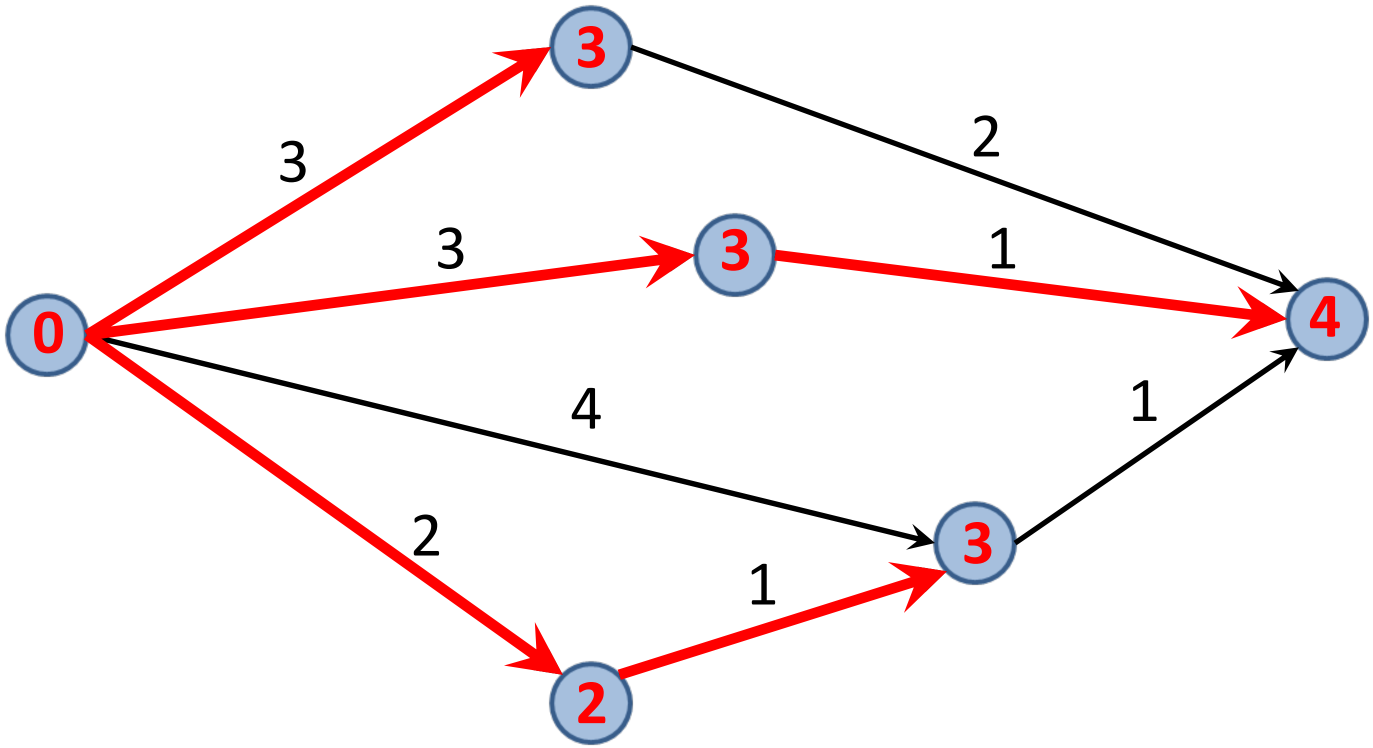
# Run 1 in K
# Concurrent Step 1

All shortest distances … directly



# Run 1 in K
# Concurrent Step 2

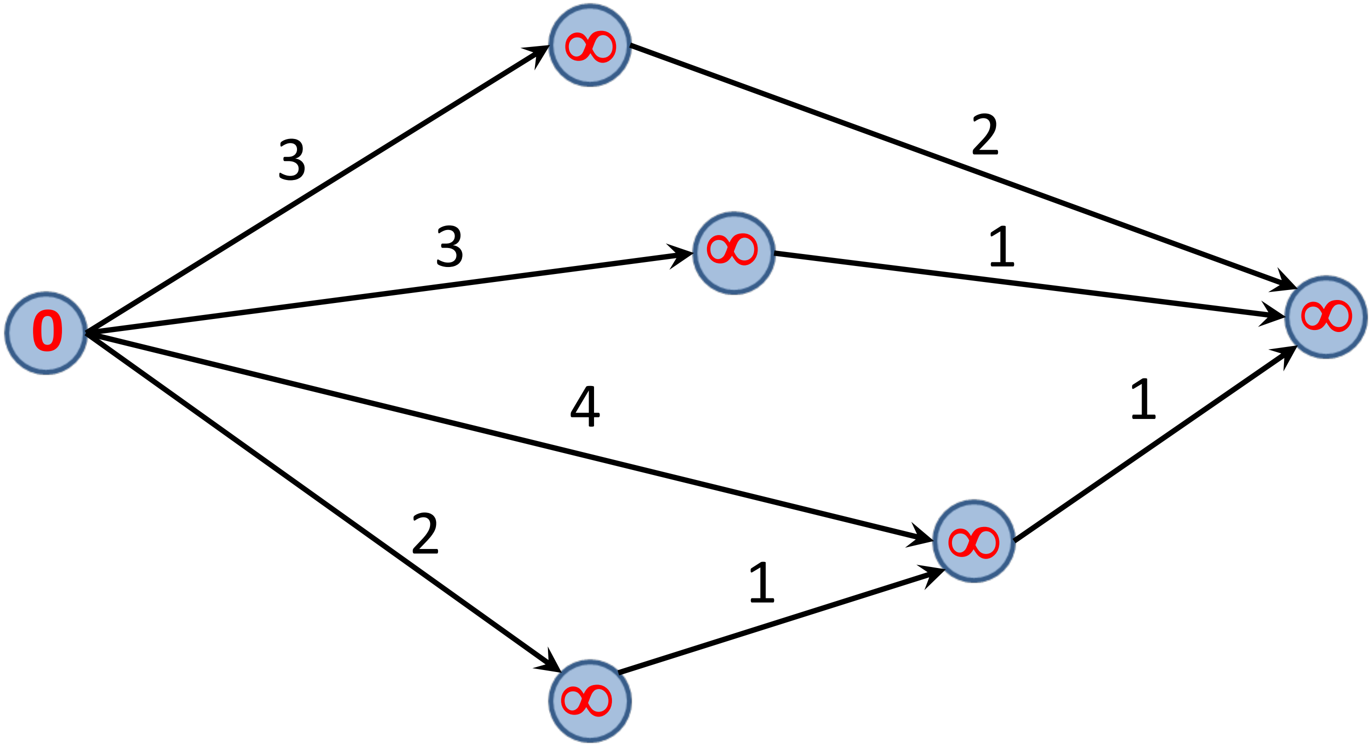
All shortest distances … directly



$P_0 \Rrightarrow P_1 \Rrightarrow P_2$

# Run 2 in K

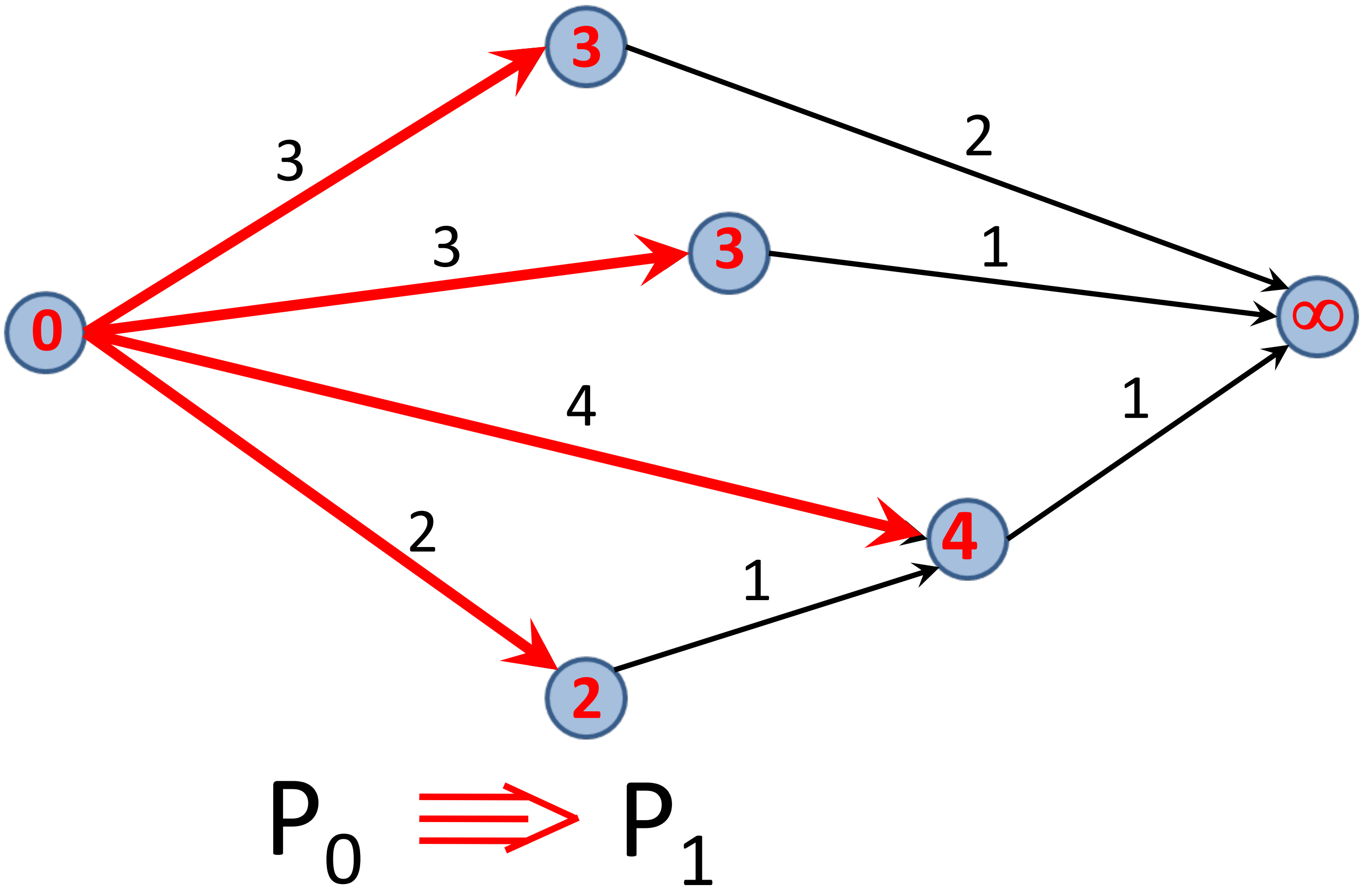
All shortest distances … eventually (max concurrency)



$P_0$

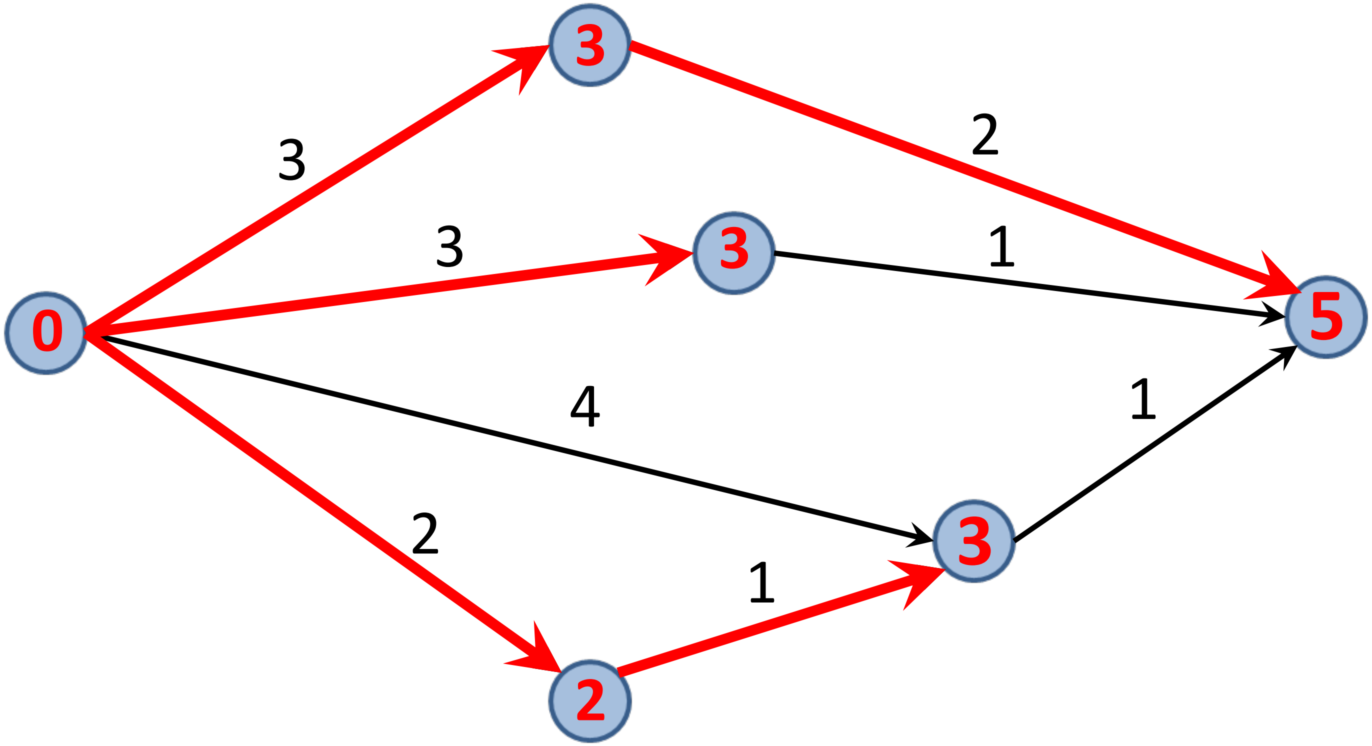
# Run 2 in K
# Concurrent Step 1

All shortest distances … eventually (max concurrency)



# Run 2 in K
# Concurrent Step 2

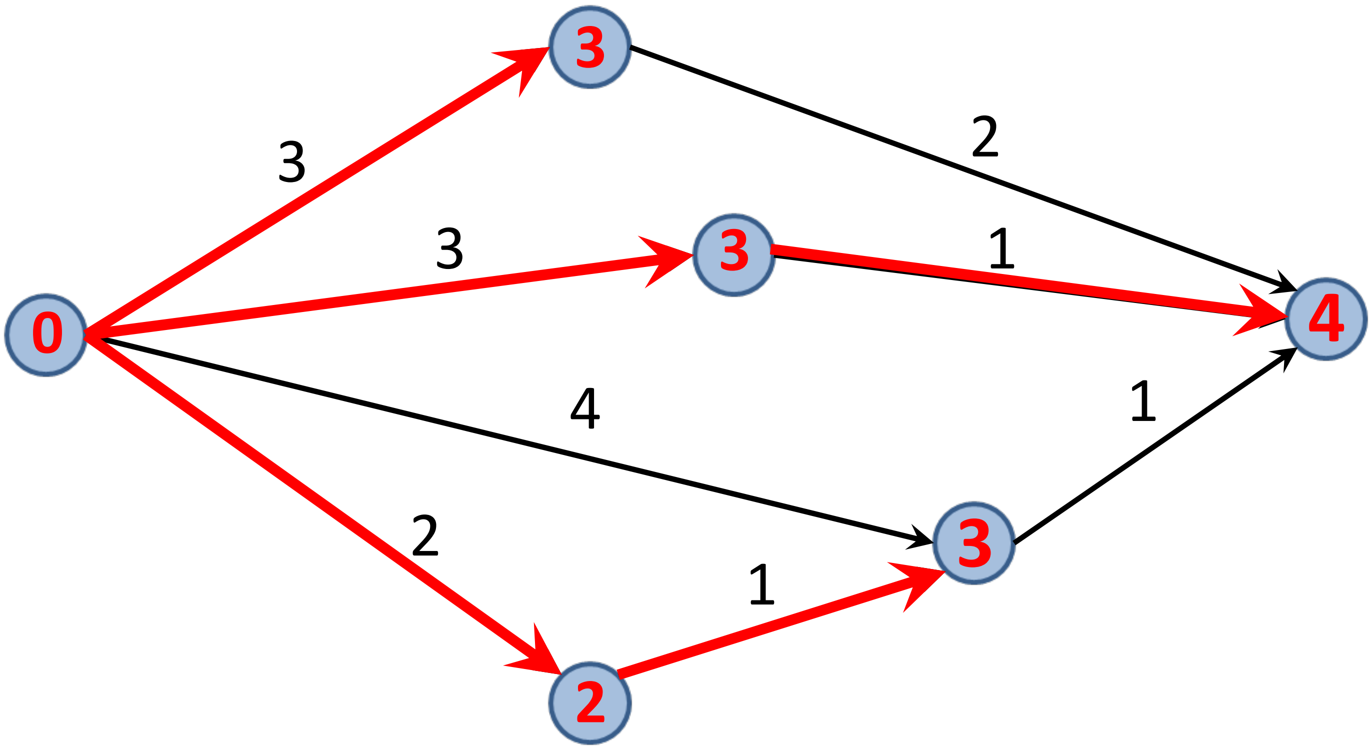
All shortest distances … eventually (max concurrency)

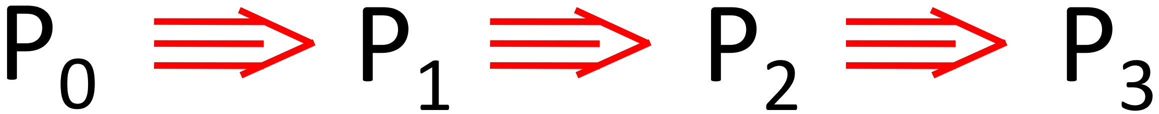


$P_0 \Rightarrow P_1 \Rightarrow P_2$

# Run 2 in K
# Concurrent Step 3

All shortest distances … eventually (max concurrency)





# Explanation on the two runs above

- Since K does not enforce maximal concurrent rewriting, the first run showed how one can directly calculate all the minimal distances in the graph using only two concurrent steps
- The second run was greedy, maximizing the number of concurrent applications of rules; consequently, it ended up using three concurrent steps instead of two
- Morale: it is hard to find optimal scheduling of concurrent rule applications; different implementations may choose different strategies; we prefer to let this issue open, so we do not enforce any particular concurrent rewrite strategy in K

# Dijkstra's Algorithm Correctness

- Termination: each rule decreases a cost
- Confluence: critical pairs joinable
- Thus, unique normal forms
- Normal form = all shortest distances
  - Build a rewrite sequence corresponding to some shortest paths; canonicity guarantees the rest
- If one wants to find all shortest paths as well, then one needs to also keep a parent to each node; however, the rewrite system is not confluent then, because there may be multiple shortest path solutions

# Motivation for K

- Teaching Programming Languages
- Why K? We found no formalism to define everything we wanted, including:
  - Operational semantics
    - Including concurrency, callcc and other existing PL features
  - Efficient interpreters at no additional expense
  - Program analyzers based on semantics of PL
    - Symbolic execution, model checkers, theorem provers, …
  - Type systems, type checkers, type inferencers
  - Visualization

# Demo

- Go to http://fsl.cs.uiuc.edu/index.php/Special:MaudeStepperOnline
- Select some languages from the left menu and
  - run them (this shows the interpreter capability)
  - run the stepper (go through the program exec)
  - run the graph (see all the statespace)

# We Tried to Use the Following … and Failed

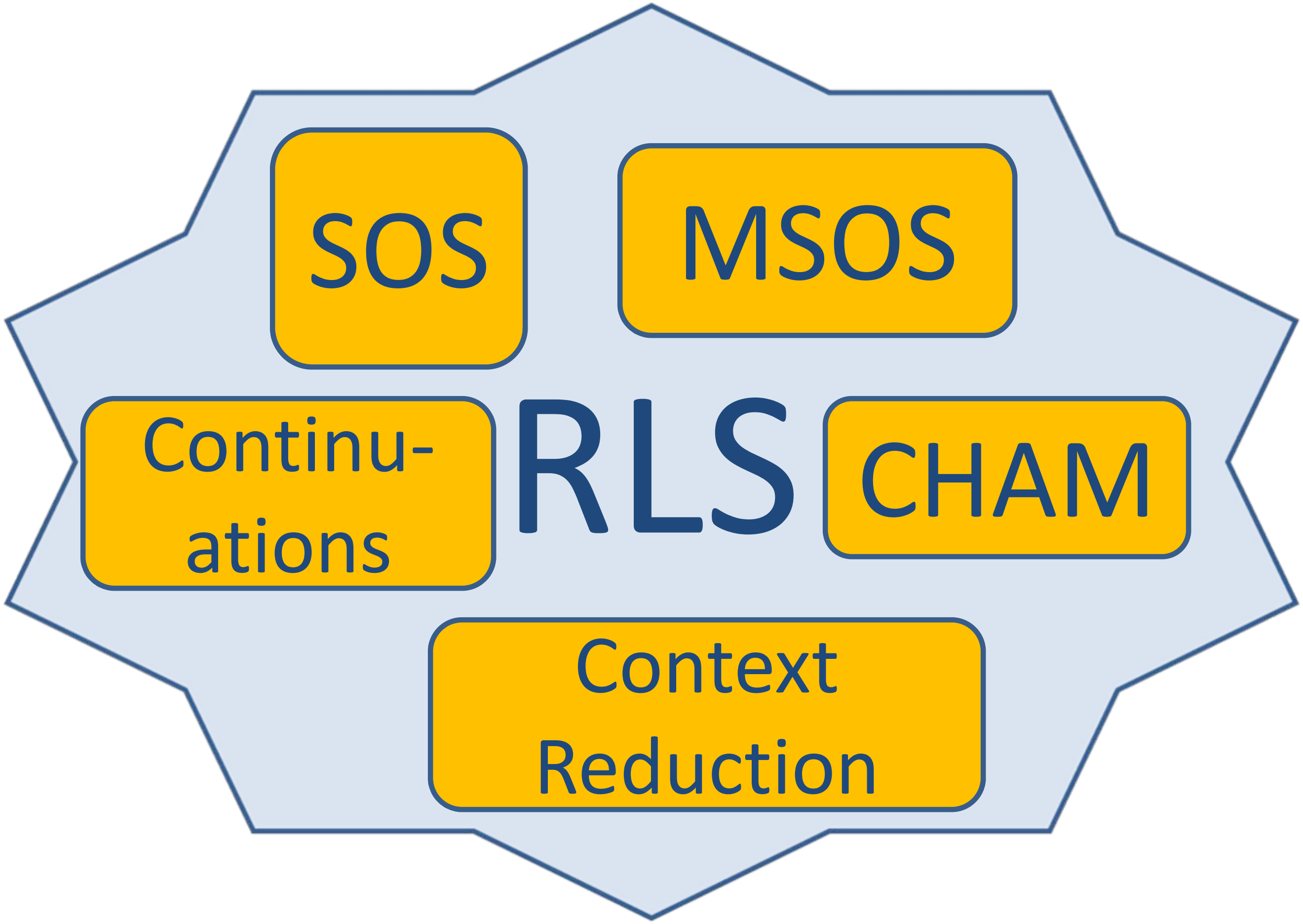
- SOS
  - Non-modular, rigidity to syntax, cannot define existing language features, only interleaving semantics for concurrency, slow; we want a framework where definition = implementation and everything else, i.e., a language definition should serve all the purposes, not only some purposes
- MSOS
  - Partially solves only the non-modularity problem of SOS; still not fully modular (aspects, etc); slow; no support for program analysis
- Evaluation contexts
  - Does not support environment-based definitions, still only interleaving semantics; no support for program analysis (model checking, symbolic execution); slow
- CHAM
  - Claims true concurrency … but not when rules share data; no implementation available and hard to implement; airlock is expensive
- Continuations
  - Mainly implementation technique; interleaving concurrency semantics; little to no support for program analysis (model checking, symbolic execution, etc.); not used for defining type systems; we want an ideal definitional technique, which can be used for anything related to languages, including typing

# Explanation for next two slides

- They show the way we used the various formalisms mentioned above
- We faithfully embedded each in rewriting logic and then used the latter to execute them
- Faithful embedding means that the resulting rewriting theory captures the original one step-for-step
- This is different from “encodings”, which typically change the computation granularity of the source framework

# Rewriting Logic Semantics
# - Ecumenical Definitional Framework -

- Serbanuta, Rosu, Meseguer: Info&Comp 2008



SOS

MSOS

Continu-ations

RLS

CHAM

Context Reduction

# Example (and similarly for all approaches) SOS as a methodological fragment of RLS

*SOS:*

$$\frac{C_1 \xrightarrow{l_1} C_1',\ C_2 \xrightarrow{l_2} C_2',\ \ldots,\ C_n \xrightarrow{l_n} C_n'}{C \xrightarrow{l} C'}$$

*RLS*$_{SOS}$*:*

$$\{C\} \to \{l, C'\} \ \text{ if } \ \{C_1\} \to \{l_1, C_1'\} \ \wedge \ \{C_2\} \to \{l_2, C_2'\} \ \wedge \cdots \wedge \{C_n\} \to \{l_n, C_n'\}$$

Theorem:

$$SOS \vdash C \xrightarrow{l} C' \quad \Longleftrightarrow \quad RLS_{SOS} \vdash \{C\} \to \{l, C'\}$$

# What does it actually mean?

- One can use one’s favorite definitional approach within RL
  - Use RL’s generic tools and techniques
- However:
  - RL does not tell you *how* to define a language
  - RL has all the advantages and disadvantages of the adopted definitional methodology

# How does K relate to RLS?

- Extended fragment
- Unconditional, but “more concurrent”
- K currently executed using RL (Maude)

# Success Stories

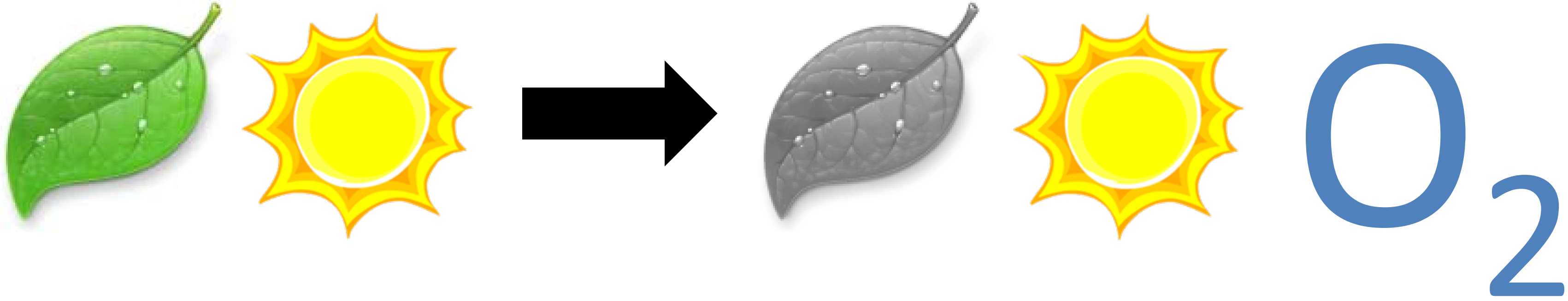
- Complete real languages defined using K
  - Java 1.4, Scheme, Beta
  - Started, to be completed: SML, Haskell, C
- Large research and classroom languages
  - SIMPLE, KOOL, FUN, LOGIK
- Competitive resulting interpreters and tools
  - Our Scheme ~10 times slower than Dr. Scheme
  - JavaFAN model checker faster than JavaPathFinder
  - Polymorphic type system faster than SML's

# K Specific Features

## Explicit Data Sharing

Why?  *To Increase Concurrency*

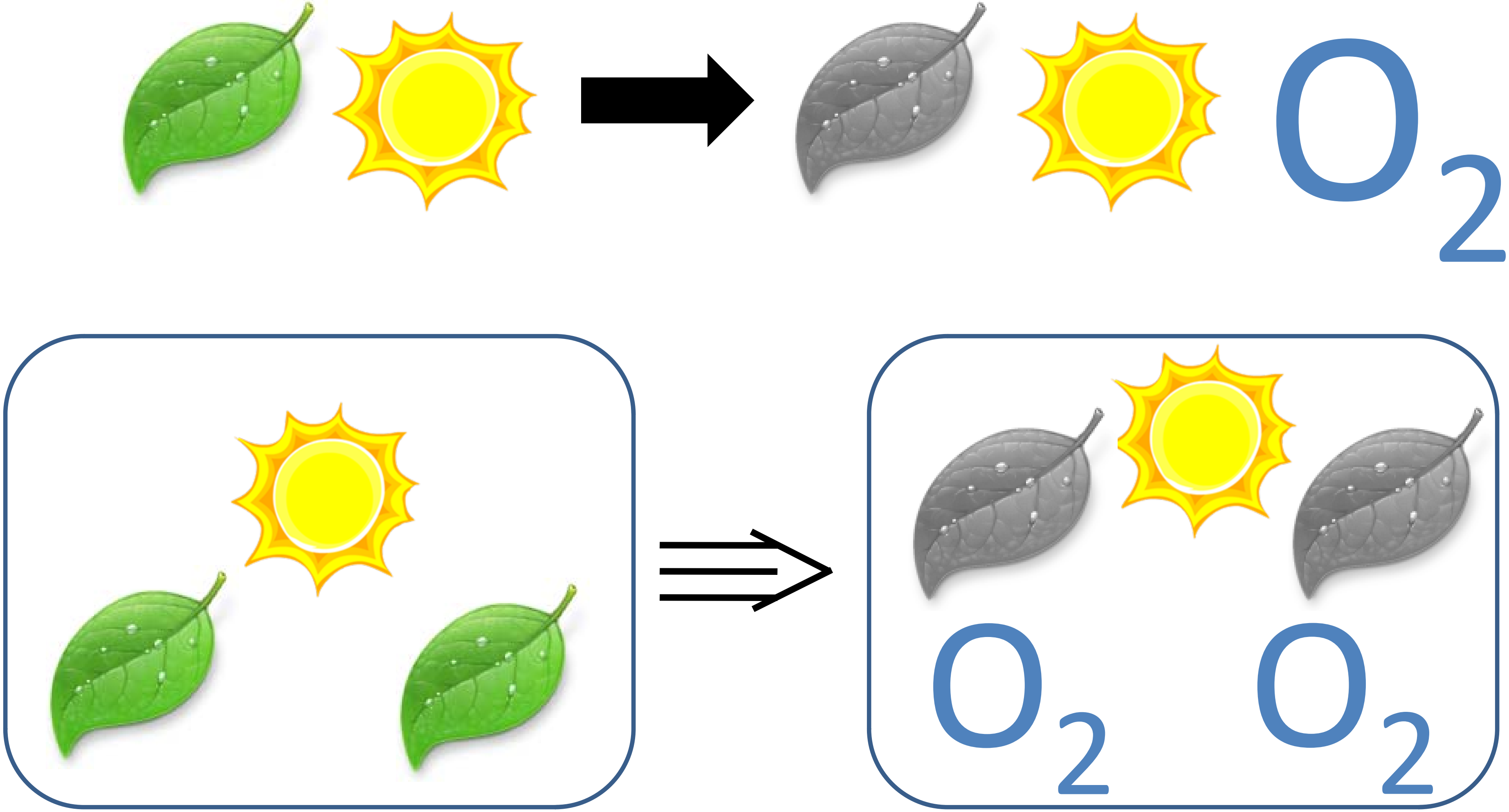
# Why Explicit Data Sharing?
# Example: Resource Sharing



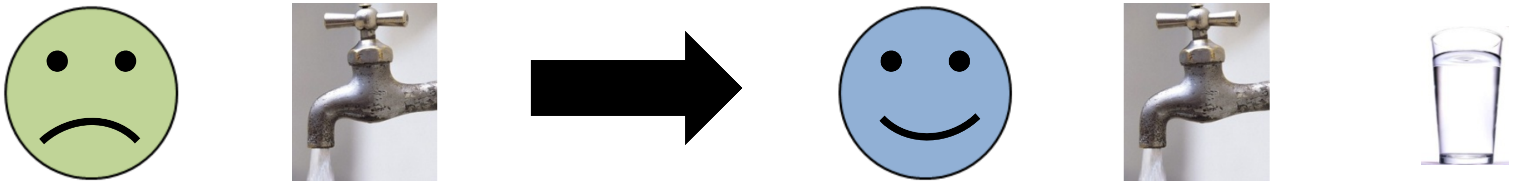
$O_2$

- We want photosynthesis to apply concurrently in spite of the fact that the sun is shared by all rule instances (that is, rules overlap!)

# Why Explicit Data Sharing?
# Example: Resource Sharing

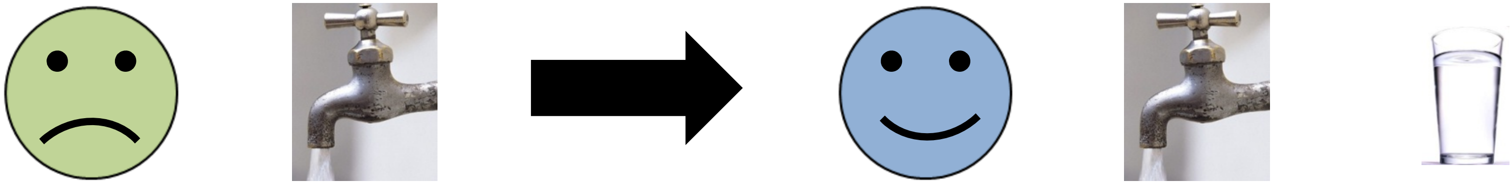


# Why Explicit Data Sharing?
# Example: Mutual Exclusion

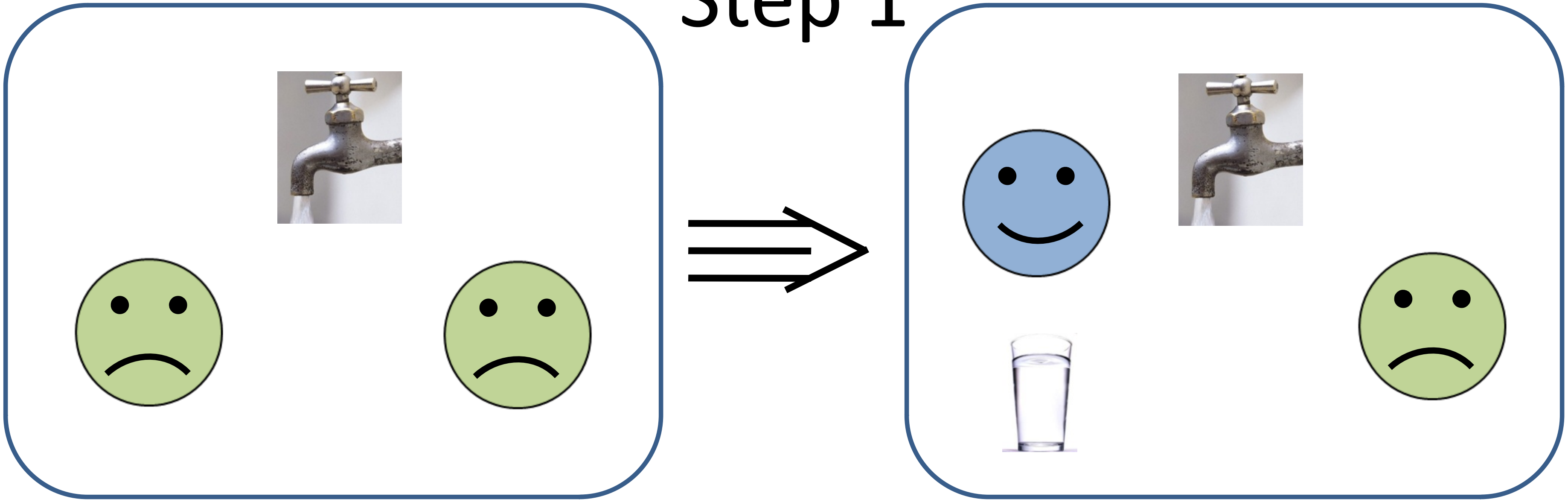


- Access to critical resource (water faucet here) cannot be concurrent, by design.
- Takes two steps to get two glasses of water, in spite of potential for concurrent execution

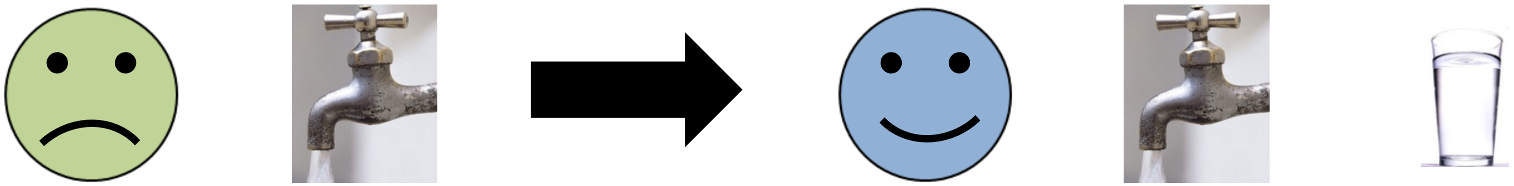
# Why Explicit Data Sharing?
# Example: Mutual Exclusion



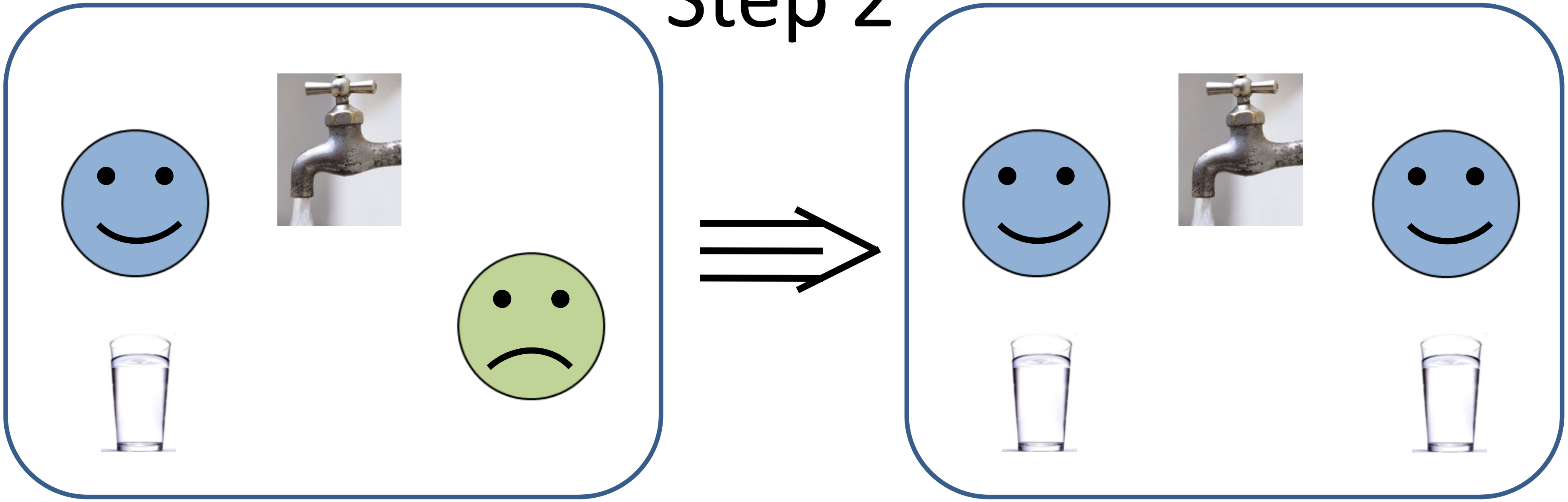
Step 1



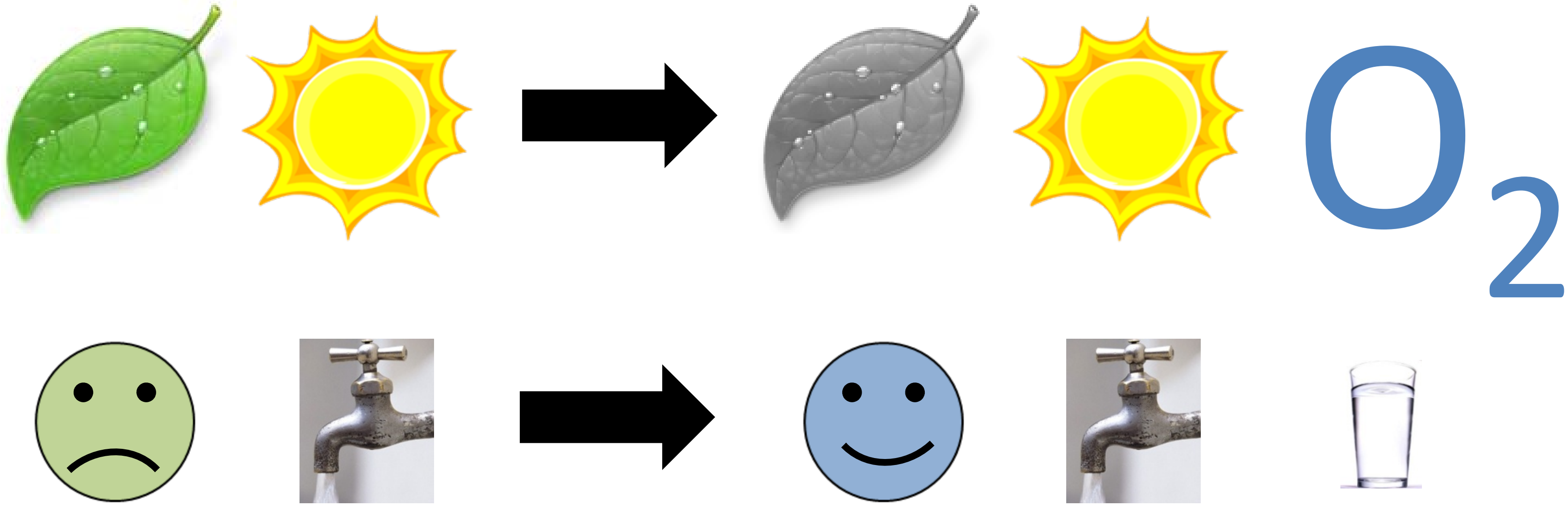
# Why Explicit Data Sharing?
# Example: Mutual Exclusion



Step 2



# Conventional Rewrite Rules Are Not Expressive Enough for Concurrency



- As conventional rewrite rules, the two rules above are identical (leaf -> face, sun -> water, …)
- Yet, we want them to have totally different meaning wrt concurrency semantics!

# K Rules – First Iteration

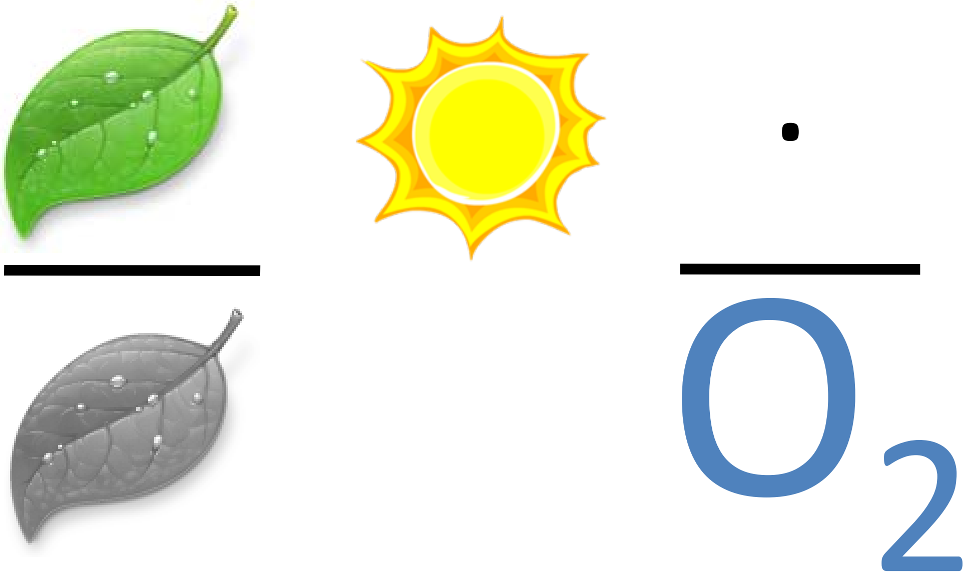
- *K rules* mention the shared context only once:

$$C[\frac{t_1}{t'_1}, \frac{t_2}{t'_2}, ..., \frac{t_n}{t'_n}]$$

instead of

$$C[t_1, t_2, ..., t_n] \rightarrow C[t'_1, t'_2, ..., t'_n]$$

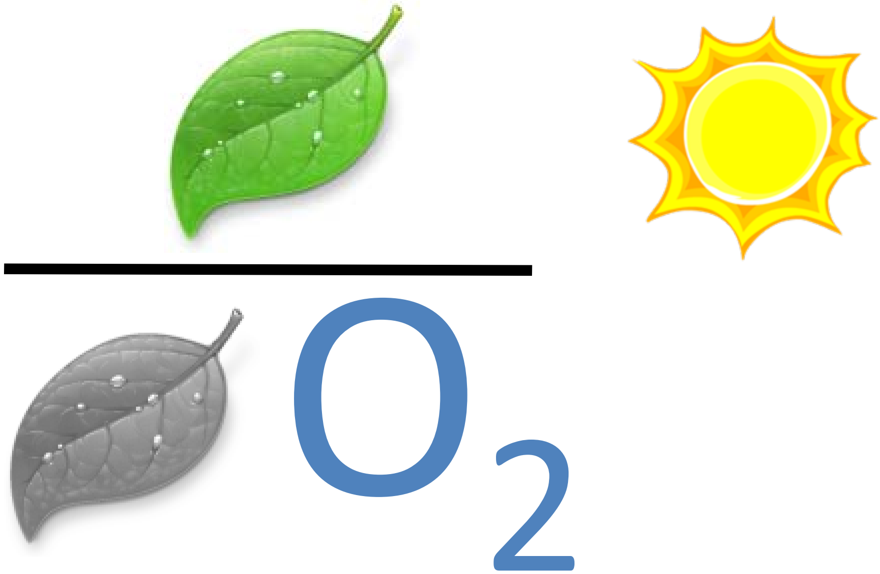
# Example of K Rule
# Resource Sharing



$$\frac{\cdot}{O_2}$$

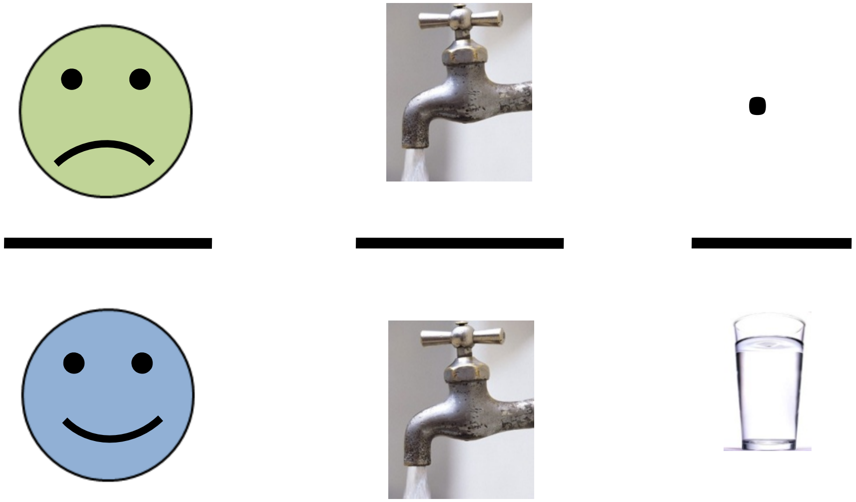
The dot "." is the unit of both bags and lists

# Example of K Rule
# Resource Sharing – Alternative rule



$O_2$

# Example of K Rule
# Mutual Exclusion



# K Specific Features

## Explicit Data Liberation

Why? *To Increase Concurrency*

# Why Explicit Data Liberation Concurrency Unconstrained by Matching

- Joe and Ann, make unconditional promises:
  - Joe: Ann, for you, I’ll be an ideal Joe (say Joe’)
  - Ann: Joe, for you, I’ll be an ideal Ann (say Ann’)

**couple(Joe, Ann) → couple(Joe’, Ann)**
**couple(Joe, Ann) → couple(Joe, Ann’)**

- Standard term rewriting does not allow **couple(Joe, Ann)** to evolve to **couple(Joe’, Ann’)**

# Why Explicit Data Liberation Concurrency Unconstrained by Matching

- Explicit context sharing does not help:

$$\mathbf{couple(}\underset{\mathbf{Joe'}}{\underline{\mathbf{Joe}}}\mathbf{,\ Ann)}$$

$$\mathbf{couple(Joe,\ }\underset{\mathbf{Ann'}}{\underline{\mathbf{Ann}}}\mathbf{)}$$

- The two rules above cannot apply concurrently because each changes the context of the other
- Same happens when defining concurrent languages: Joe and Ann can be threads accessing different locations in a shared store

# Explicit Data Liberation in K

- Positions can be explicitly liberated
  - Notation: overline them!

$$\mathbf{couple}(\underset{\mathbf{Joe'}}{\underline{\mathbf{Joe}}}, \overline{\mathbf{Ann}}) \qquad \mathbf{couple}(\overline{\mathbf{Joe}}, \underset{\mathbf{Ann'}}{\underline{\mathbf{Ann}}})$$

- Liberated positions
  - Used for concurrent matching, … but
  - Allowed to be changed by the matching rules

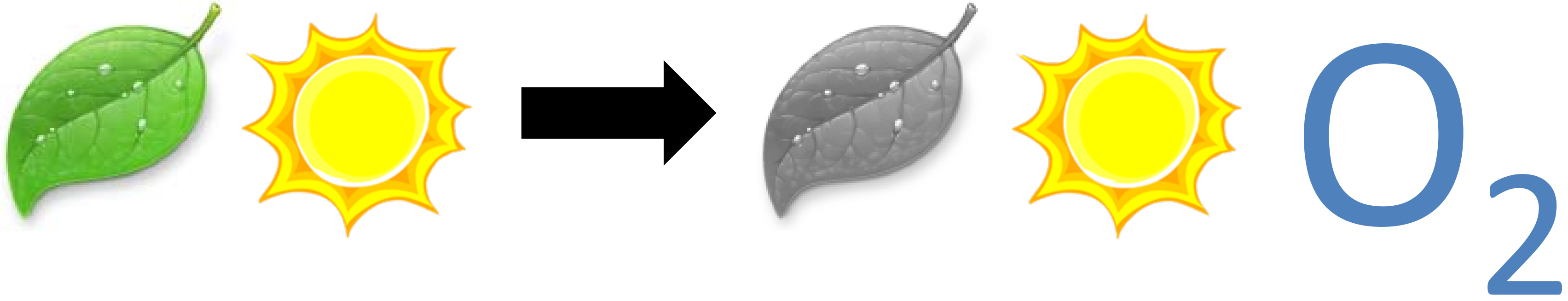
$$\mathbf{couple}(\mathbf{Joe}, \mathbf{Ann}) \Rrightarrow \mathbf{couple}(\mathbf{Joe'}, \mathbf{Ann'})$$

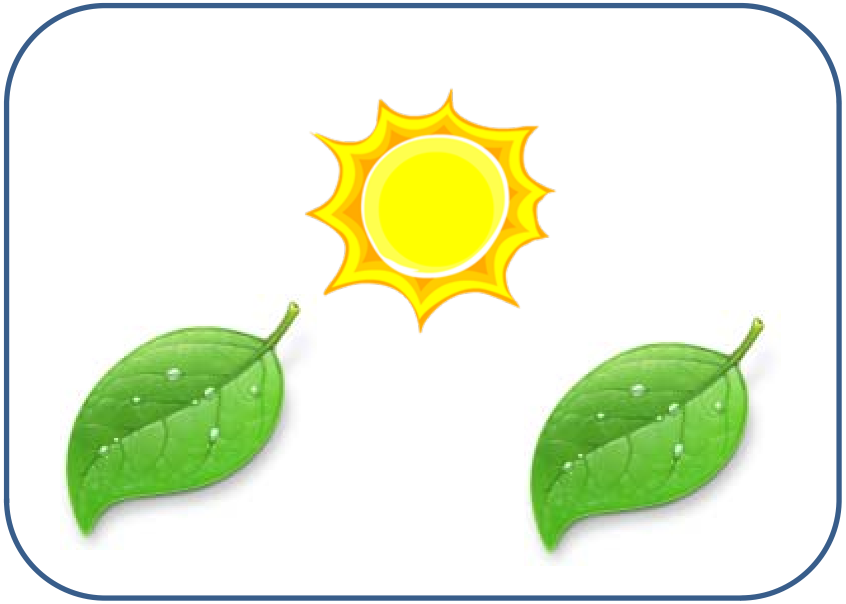
- Major Difference between K and rewriting
  - Like causal atomicity versus serializability

# K Specific Features

## List and Bag Cells

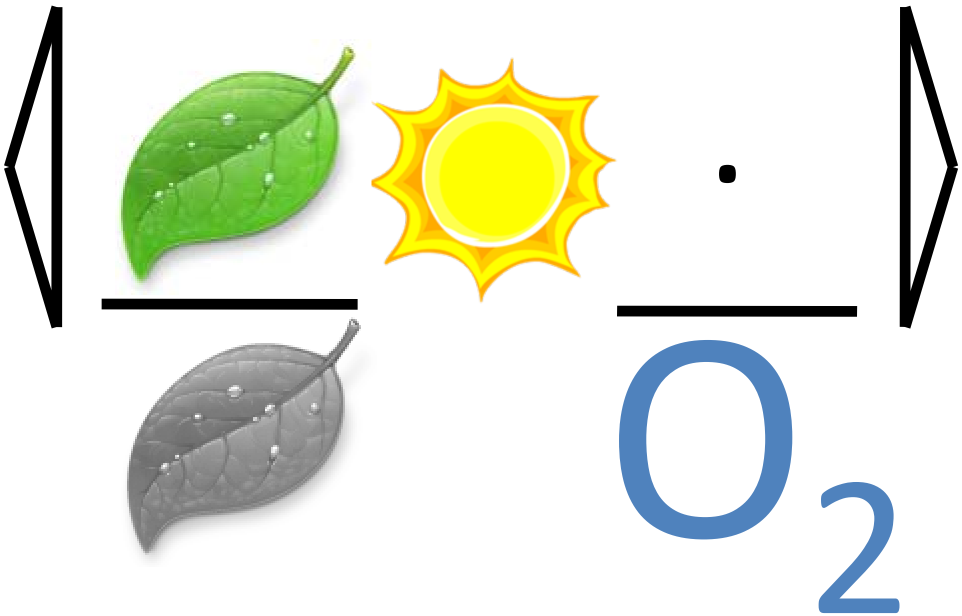
# Rewriting Modulo … Insufficient





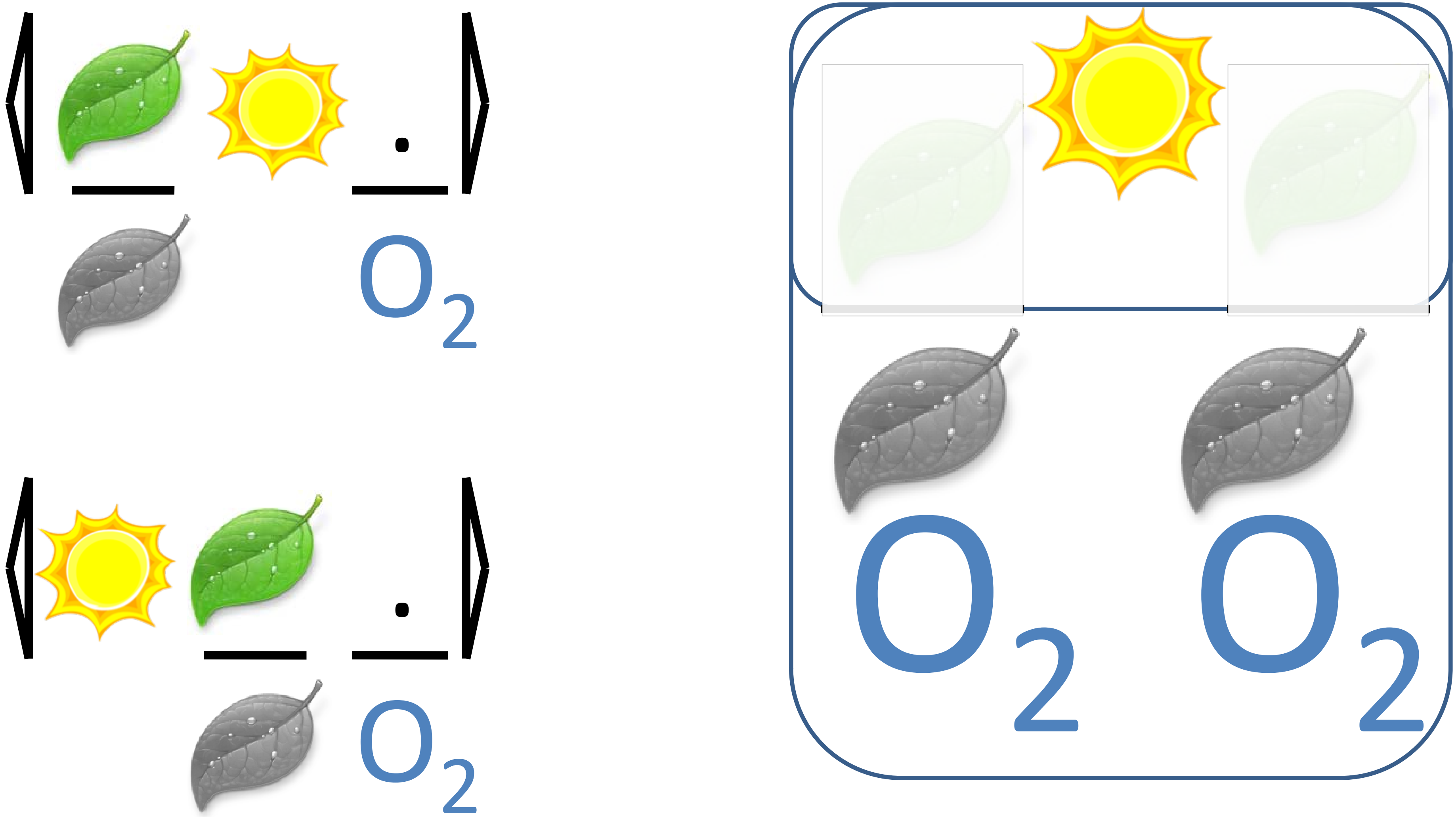
No way to rearrange soup so that one can apply two rules concurrently; one cannot use idempotency of sun, as “unexpected” concurrent behaviors could happen if other rules were around, e.g., an “eclipse” rule; think of sun as a shared store.

# Special Support for Lists and Bags in K



Angular separators mean “inside”; desugared into a finite number of multiset equivalent rules

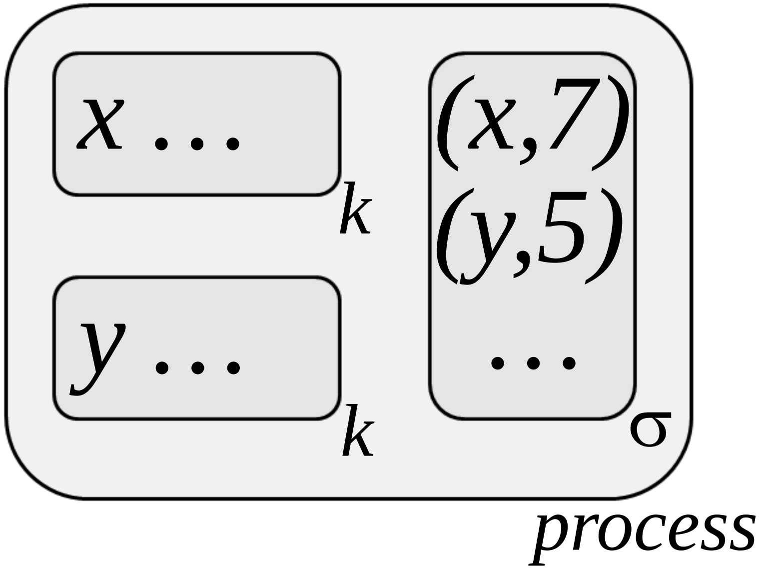
# Special Support for Lists and Bags in K



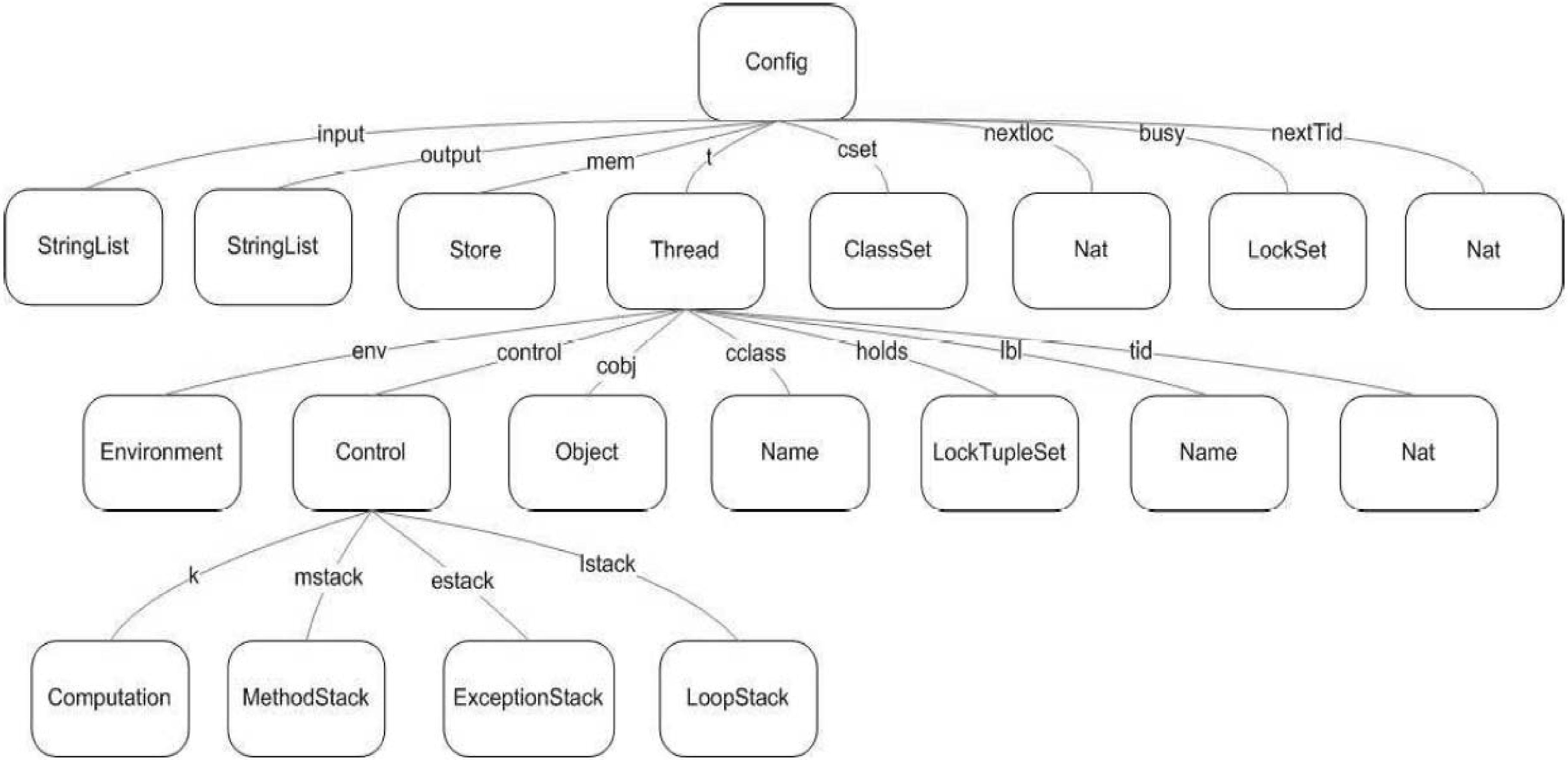
# Special Support for Lists and Bags in K

- "Cell" separators also used for lists
- If a separator is round, it means "end"; if angular it means "and so on in that direction"
- Separators can be indexed and nested

$$\left\langle\!\left| \;\; \left(\!\left| \underset{7}{\underline{x}} \right|\!\right\rangle_{k} \;\; \left\langle\!\left| (x,7) \right|\!\right\rangle_{\sigma} \;\; \right|\!\right\rangle_{process}$$



# Configurations = Nested Lists and Bags



KOOL configuration “soup”

K Specific Features

# Computations and Tasks

# Computations and Tasks

- Computations are lists of tasks as follows

$$T_1 \curvearrowright T_2 \curvearrowright \cdots \curvearrowright T_n$$

- Produced by *heating/cooling equations*

$$a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$$
$$a_1 + a_2 \rightleftharpoons a_2 \curvearrowright a_1 + \square$$

# Computational Equivalence Classes

$$x * (y + 2)$$

$$x \curvearrowright (\Box * (y + 2))$$

$$x \curvearrowright (\Box * (y \curvearrowright (\Box + 2)))$$

$$x \curvearrowright (\Box * (2 \curvearrowright (y + \Box)))$$

$$(y + 2) \curvearrowright (x * \Box)$$

$$y \curvearrowright (\Box + 2) \curvearrowright (x * \Box)$$

$$2 \curvearrowright (y + \Box) \curvearrowright (x * \Box)$$

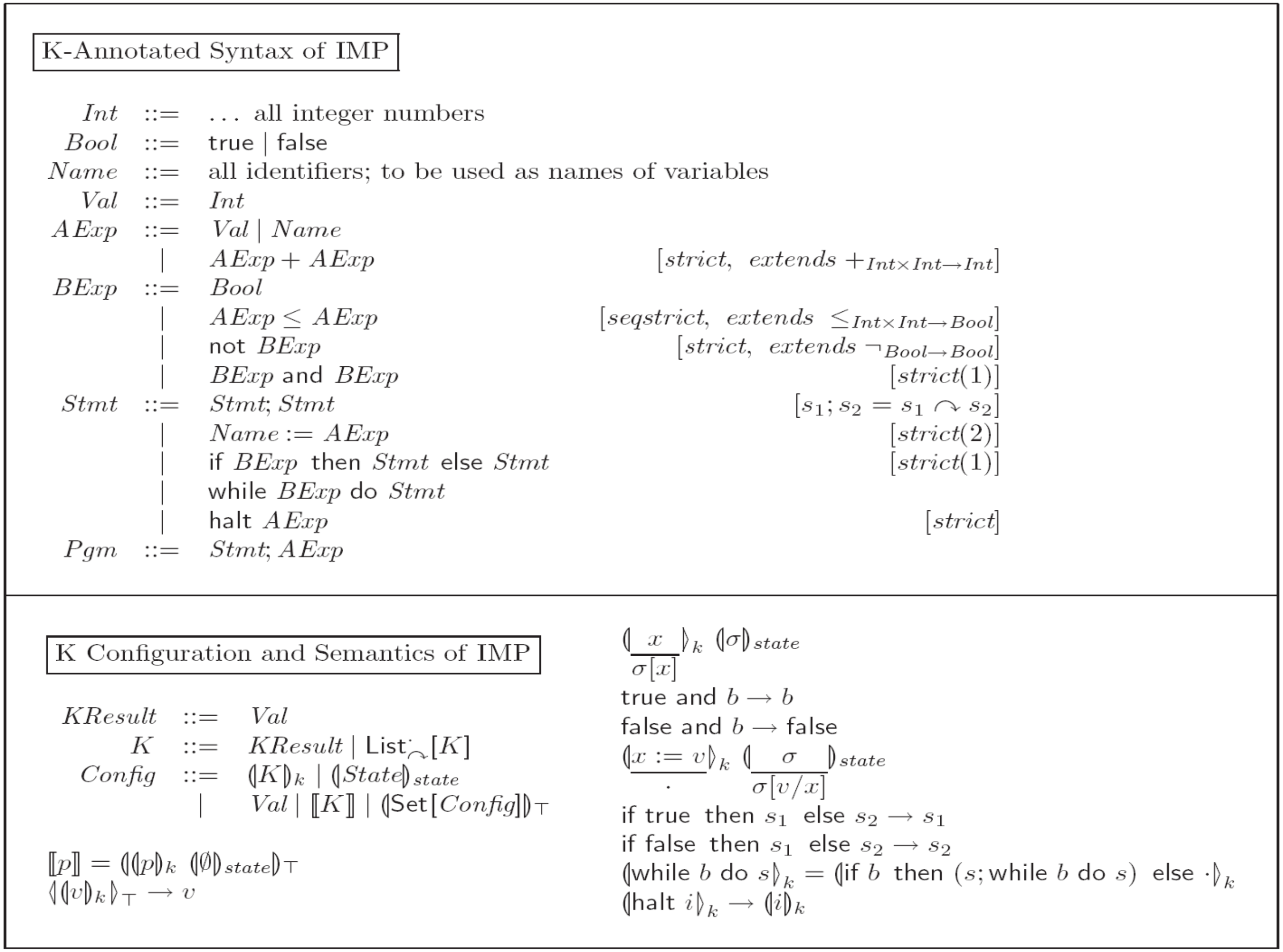
$$x * (y \curvearrowright (\Box + 2))$$

$$x * (2 \curvearrowright (y + \Box))$$

# The K-CHALLENGE

- An experimental programming language intended to challenge definitional frameworks
- Starts with simple imperative language
- Keeps adding features, modularly (in K)
- All existing frameworks, except K, fail: they can either not define certain features at all, or, if they can, they do it non-modularly

# K-CHALLENGE: Start with IMP



K-Annotated Syntax of IMP

$$
\begin{array}{rcll}
\mathit{Int} & ::= & \ldots \text{ all integer numbers} & \\
\mathit{Bool} & ::= & \mathsf{true} \mid \mathsf{false} & \\
\mathit{Name} & ::= & \text{all identifiers; to be used as names of variables} & \\
\mathit{Val} & ::= & \mathit{Int} & \\
\mathit{AExp} & ::= & \mathit{Val} \mid \mathit{Name} & \\
 & \mid & \mathit{AExp} + \mathit{AExp} & [\mathit{strict},\ \mathit{extends}\ +_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
\mathit{BExp} & ::= & \mathit{Bool} & \\
 & \mid & \mathit{AExp} \leq \mathit{AExp} & [\mathit{seqstrict},\ \mathit{extends}\ \leq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathsf{not}\ \mathit{BExp} & [\mathit{strict},\ \mathit{extends}\ \neg_{\mathit{Bool}\to\mathit{Bool}}] \\
 & \mid & \mathit{BExp}\ \mathsf{and}\ \mathit{BExp} & [\mathit{strict}(1)] \\
\mathit{Stmt} & ::= & \mathit{Stmt}; \mathit{Stmt} & [s_1; s_2 = s_1 \curvearrowright s_2] \\
 & \mid & \mathit{Name} := \mathit{AExp} & [\mathit{strict}(2)] \\
 & \mid & \mathsf{if}\ \mathit{BExp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt} & [\mathit{strict}(1)] \\
 & \mid & \mathsf{while}\ \mathit{BExp}\ \mathsf{do}\ \mathit{Stmt} & \\
 & \mid & \mathsf{halt}\ \mathit{AExp} & [\mathit{strict}] \\
\mathit{Pgm} & ::= & \mathit{Stmt}; \mathit{AExp} &
\end{array}
$$

K Configuration and Semantics of IMP

$$
\begin{array}{rcl}
\mathit{KResult} & ::= & \mathit{Val} \\
K & ::= & \mathit{KResult} \mid \mathsf{List}^{\cdot}_{\curvearrowright}[K] \\
\mathit{Config} & ::= & \llparenthesis K \rrparenthesis_k \mid \llparenthesis \mathit{State} \rrparenthesis_{\mathit{state}} \\
 & \mid & \mathit{Val} \mid [\![K]\!] \mid \llparenthesis \mathsf{Set}[\mathit{Config}] \rrparenthesis_{\top}
\end{array}
$$

$[\![p]\!] = \llparenthesis \llparenthesis p \rrparenthesis_k\ \llparenthesis \emptyset \rrparenthesis_{\mathit{state}} \rrparenthesis_{\top}$

$\langle\!| \llparenthesis v \rrparenthesis_k |\!\rangle_{\top} \longrightarrow v$

$\llparenthesis \underline{x} \rangle\!\rangle_k\ \llparenthesis \sigma \rrparenthesis_{\mathit{state}}$ with $x$ rewritten to $\sigma[x]$:

$$\llparenthesis \underset{\sigma[x]}{\underline{\ x\ }} \rangle_k\ \llparenthesis \sigma \rrparenthesis_{\mathit{state}}$$

$\mathsf{true\ and}\ b \longrightarrow b$

$\mathsf{false\ and}\ b \longrightarrow \mathsf{false}$

$$\llparenthesis \underset{\cdot}{\underline{x := v}} \rangle_k\ \llparenthesis \underset{\sigma[v/x]}{\underline{\ \ \sigma\ \ }} \rrparenthesis_{\mathit{state}}$$

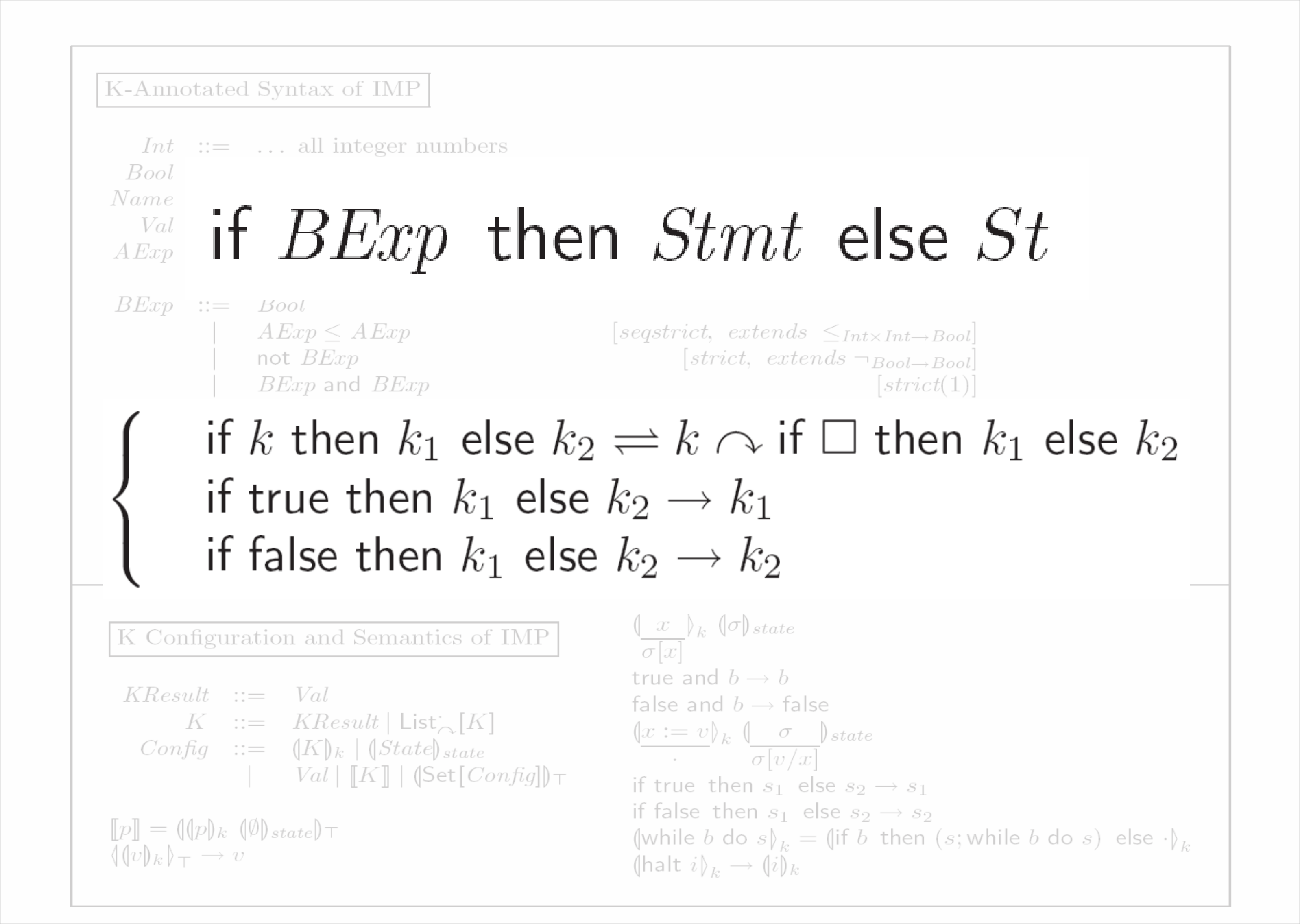
$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \longrightarrow s_1$

$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \longrightarrow s_2$

$\llparenthesis \mathsf{while}\ b\ \mathsf{do}\ s \rangle_k = \llparenthesis \mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else}\ \cdot \rangle_k$

$\llparenthesis \mathsf{halt}\ i \rangle_k \longrightarrow \llparenthesis i \rrparenthesis_k$

# K-CHALLENGE: Start with IMP



K-Annotated Syntax of IMP

$$\begin{array}{rcl} Int & ::= & \ldots \text{ all integer numbers} \\ Bool & & \\ Name & & \\ Val & & \\ AExp & & \end{array}$$

$$\text{if } BExp \;\text{ then }\; Stmt \;\text{ else }\; St$$

$$\begin{array}{rcll} BExp & ::= & Bool & \\ & | & AExp \leq AExp & [seqstrict,\; extends \leq_{Int \times Int \to Bool}] \\ & | & \text{not } BExp & [strict,\; extends \neg_{Bool \to Bool}] \\ & | & BExp \text{ and } BExp & [strict(1)] \end{array}$$

$$\begin{cases} \text{if } k \text{ then } k_1 \text{ else } k_2 \rightleftharpoons k \curvearrowright \text{if } \square \text{ then } k_1 \text{ else } k_2 \\ \text{if true then } k_1 \text{ else } k_2 \longrightarrow k_1 \\ \text{if false then } k_1 \text{ else } k_2 \longrightarrow k_2 \end{cases}$$

K Configuration and Semantics of IMP

$$\begin{array}{rcl} KResult & ::= & Val \\ K & ::= & KResult \mid \text{List}^{\cdot}_{\curvearrowright}[K] \\ Config & ::= & \llparenthesis K \rrparenthesis_k \mid \llparenthesis State \rrparenthesis_{state} \\ & | & Val \mid [\![K]\!] \mid \llparenthesis \text{Set}[Config] \rrparenthesis_\top \end{array}$$

$$[\![p]\!] = \llparenthesis \llparenthesis p \rrparenthesis_k \; \llparenthesis \emptyset \rrparenthesis_{state} \rrparenthesis_\top$$

$$\langle\!| \llparenthesis v \rrparenthesis_k |\!\rangle_\top \longrightarrow v$$

$$\llparenthesis \underset{\sigma[x]}{\underline{\;x\;}} \rangle\!\rangle_k \; \llparenthesis \sigma \rrparenthesis_{state}$$

$$\text{true and } b \longrightarrow b$$

$$\text{false and } b \longrightarrow \text{false}$$

$$\llparenthesis \underset{\cdot}{\underline{x := v}} \rangle\!\rangle_k \; \llparenthesis \underset{\sigma[v/x]}{\underline{\;\sigma\;}} \rrparenthesis_{state}$$

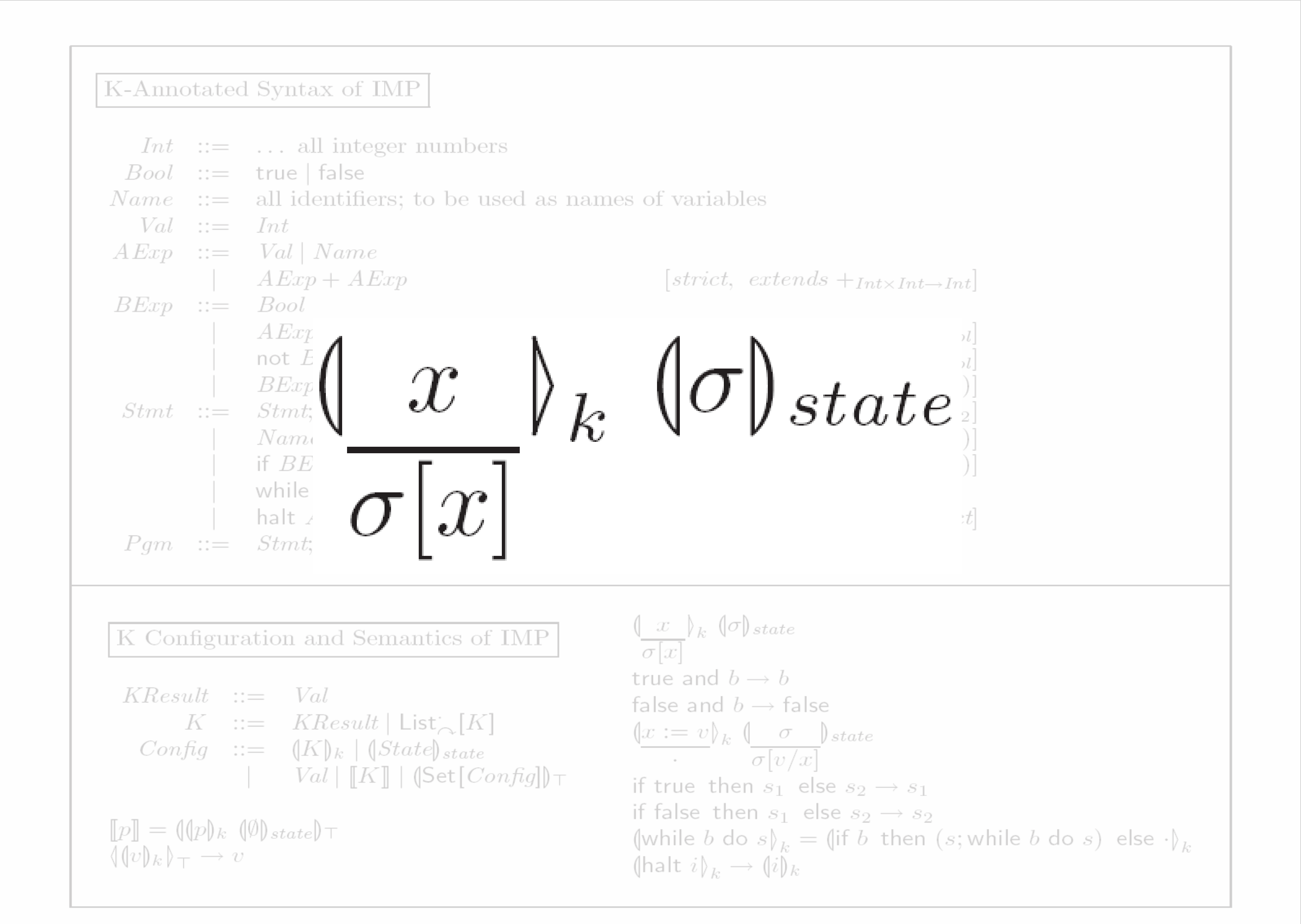
$$\text{if true then } s_1 \text{ else } s_2 \longrightarrow s_1$$

$$\text{if false then } s_1 \text{ else } s_2 \longrightarrow s_2$$

$$\llparenthesis \text{while } b \text{ do } s \rangle\!\rangle_k = \llparenthesis \text{if } b \text{ then } (s; \text{while } b \text{ do } s) \text{ else } \cdot \rangle\!\rangle_k$$

$$\llparenthesis \text{halt } i \rangle\!\rangle_k \longrightarrow \llparenthesis i \rrparenthesis_k$$

# K-CHALLENGE: Start with IMP



K-Annotated Syntax of IMP

$Int$ ::= . . . all integer numbers
$Bool$ ::= true | false
$Name$ ::= all identifiers; to be used as names of variables
$Val$ ::= $Int$
$AExp$ ::= $Val \mid Name$
| $AExp + AExp$ $[strict,\ extends\ +_{Int \times Int \to Int}]$
$BExp$ ::= $Bool$
| $AEx$ $l]$
| not $E$ $l]$
| $BExp$ $)]$
$Stmt$ ::= $Stmt;$ $2]$
| $Nam$ $)]$
| if $BE$ $)]$
| while
| halt $t]$
$Pgm$ ::= $Stmt;$

$$\llparenthesis \frac{x}{\sigma[x]} \rangle_k \ \llparenthesis \sigma \rrparenthesis_{state}$$

K Configuration and Semantics of IMP

$KResult$ ::= $Val$
$K$ ::= $KResult \mid \mathsf{List}^{\cdot}_{\curvearrowright}[K]$
$Config$ ::= $\llparenthesis K \rrparenthesis_k \mid \llparenthesis State \rrparenthesis_{state}$
| $Val \mid [\![K]\!] \mid \llparenthesis \mathsf{Set}[Config] \rrparenthesis_\top$

$[\![p]\!] = \llparenthesis \llparenthesis p \rrparenthesis_k \ \llparenthesis \emptyset \rrparenthesis_{state} \rrparenthesis_\top$
$\langle \llparenthesis v \rrparenthesis_k \rangle_\top \longrightarrow v$

$\llparenthesis \frac{x}{\sigma[x]} \rangle_k \ \llparenthesis \sigma \rrparenthesis_{state}$
true and $b \longrightarrow b$
false and $b \longrightarrow$ false
$\llparenthesis \frac{x := v}{\cdot} \rangle_k \ \llparenthesis \frac{\sigma}{\sigma[v/x]} \rrparenthesis_{state}$
if true then $s_1$ else $s_2 \longrightarrow s_1$
if false then $s_1$ else $s_2 \longrightarrow s_2$
$\llparenthesis$while $b$ do $s\rangle_k = \llparenthesis$if $b$ then $(s;$while $b$ do $s)$ else $\cdot\rangle_k$
$\llparenthesis$halt $i\rangle_k \longrightarrow \llparenthesis i \rrparenthesis_k$

# K-CHALLENGE: Add increment

$$AExp ::= \ldots \mid \texttt{++}\,Name$$

$$\left(\!\left|\underset{i}{\underline{\texttt{++}\,x}}\right\rangle_k \;\left(\!\left|\underset{\sigma[i/x]}{\underline{\quad\sigma\quad}}\right)\!\right)_{state} \qquad \text{where } i = \sigma[x] + 1$$

# K-CHALLENGE: Add output

$$Stmt ::= \ldots \mid \mathsf{output\_} \quad [strict] \mid \mathsf{halt}$$
$$Config ::= \ldots \mid \mathsf{List}[Val] \mid (\!|\mathsf{List}[Val]|\!)_{output}$$

$$[\![s]\!] = (\!|(\!|s|\!)_k\ (\!|\cdot|\!)_{state}\ (\!|\cdot|\!)_{output}|\!)_\top$$
$$\langle\!|(\!|\cdot|\!)_k\ (\!|vl|\!)_{output}|\!\rangle_\top = vl$$
$$(\!|\mathsf{halt}|\!\rangle_k \longrightarrow (\!|\cdot|\!)_k$$

$$(\!|\underset{\cdot}{\underline{\mathsf{output}\ v}}|\!\rangle_k\ \ \langle\!|\underset{v}{\underline{\ \cdot\ }}|\!)_{output}$$

# K-CHALLENGE: Add λ-expressions First, a substitution-based definition

$$Val ::= \ldots \mid \lambda Name.Exp$$

$$Exp ::= \ldots \mid Exp\,Exp \quad [strict]$$

$$\llparenthesis \underset{e[v/x]}{\underline{(\lambda x.e)\,v}} \rrparenthesis_k \quad \text{where } x \in Name,\ e \in Exp,\ v \in Val$$

# K-CHALLENGE: Add λ-expressions
# A closure-based definition

$$\mathit{Config} ::= (\!|K|\!)_k \mid (\!|\mathit{Env}|\!)_{env} \mid (\!|\mathit{Store}|\!)_{store} \mid (\!|\mathsf{List}[\mathit{Val}]|\!)_{output} \mid (\!|\mathsf{Set}[\mathit{Config}]|\!)_{\top}$$

$$[\![e]\!] = (\!|(\!|e|\!)_k\ (\!|\cdot|\!)_{env}\ (\!|\cdot|\!)_{store}\ (\!|\cdot|\!)_{output}|\!)_{\top}$$

$$(\!|\underset{\sigma[\rho[x]]}{\underline{\quad x \quad}}\rangle_k\ (\!|\rho|\!)_{env}\ (\!|\sigma|\!)_{store}$$

$$(\!|\underset{\cdot}{\underline{x := v}}\rangle_k\ (\!|\rho|\!)_{env}\ (\!|\underset{\sigma[v/\rho[x]]}{\underline{\quad \sigma \quad}}|\!)_{store}$$

$$(\!|\underset{i}{\underline{\texttt{++}\, x}}\rangle_k\ (\!|\rho|\!)_{env}\ (\!|\underset{\sigma[i/\rho[x]]}{\underline{\quad \sigma \quad}}|\!)_{store} \quad \text{where } i \text{ is } \sigma[\rho[x]] + 1$$

$$\mathit{Val} ::= \ldots \mid \mathit{closure}(\mathit{Name}, \mathit{Exp}, \mathit{Env})$$
$$\mathit{Exp} ::= \ldots \mid \lambda \mathit{Name}.\mathit{Exp} \mid \mathit{Exp}\ \mathit{Exp}\ [\mathit{strict}]$$

$$(\!|\underset{\mathit{closure}(x, e, \rho)}{\underline{\quad \lambda x.e \quad}}\rangle_k\ (\!|\rho|\!)_{env}$$

$$(\!|\underset{e \curvearrowright \mathit{restore}(\rho')}{\underline{\mathit{closure}(x, e, \rho)\ v}}\rangle_k\ (\!|\underset{\rho[l/x]}{\underline{\quad \rho' \quad}}|\!)_{env}\ (\!|\underset{\sigma[v/l]}{\underline{\quad \sigma \quad}}|\!)_{store} \quad \text{where } l \text{ is a fresh location}$$

# K-CHALLENGE: Add recursion

$$Exp ::= \ldots \mid \mu Name.Exp$$

$$\llparenthesis \mu x.e \rrparenthesis_k = \llparenthesis (\lambda x.e)\,(\mu x.e) \rrparenthesis_k$$

# K-CHALLENGE: referencing, dereferencing, addressing, location assignment

$$Val ::= \ldots \mid Loc$$
$$Exp ::= \ldots \mid \mathsf{ref}\ Exp\ [strict] \mid *\ Exp\ [strict] \mid \&\ Name$$
$$Stmt := \ldots \mid Exp := Exp\ [strict]$$

$$(\!|\frac{\mathsf{ref}\ v}{l}\rangle_k \ (\!|\frac{\sigma}{\sigma[v/l]}|\!)_{store} \quad \text{where } l \text{ is a fresh location}$$

$$(\!|\frac{*\,l}{\sigma[l]}\rangle_k \ (\!|\sigma|\!)_{store}$$

$$(\!|\frac{\&\ x}{\rho[x]}\rangle_k \ (\!|\rho|\!)_{env}$$

$$(\!|\frac{l := v}{\cdot}\rangle_k \ (\!|\frac{\sigma}{\sigma[v/l]}|\!)_{store}$$

# K-CHALLENGE: Add CALL/CC

$$
\begin{aligned}
&Exp ::= \ldots \mid \mathsf{callcc}\, Exp\ [strict] \\
&Val ::= \ldots \mid cc(K, Env)
\end{aligned}
$$

$$\llparenthesis \underset{v\ cc(k, \rho)}{\underline{\mathsf{callcc}\, v\ }} \curvearrowright k \rrparenthesis_{k}\ \ \llparenthesis \rho \rrparenthesis_{env}$$

$$\llparenthesis \underset{v \curvearrowright k}{\underline{cc(k, \rho)\, v \curvearrowright \_}} \rrparenthesis_{k}\ \ \llparenthesis \underset{\rho}{\underline{\_}} \rrparenthesis_{env}$$

# K-CHALLENGE: Add nondeterminism

$$Exp ::= \ldots \mid \mathsf{randomBool}$$

$$\mathsf{randomBool} \rightarrow \mathsf{true}$$

$$\mathsf{randomBool} \rightarrow \mathsf{false}$$

# K-CHALLENGE: Add aspects

$$Stmt ::= \ldots \mid \mathsf{aspect}\ Stmt$$
$$Config ::= \ldots \mid (\!|K|\!)_{aspect}$$

$$[\![s]\!] = (\!|(\!|s|\!)_{k}\ (\!|\cdot|\!)_{env}\ (\!|\cdot|\!)_{store}\ (\!|\cdot|\!)_{output}\ (\!|\cdot|\!)_{aspect}|\!)_{\top}$$

$$(\!|\frac{\mathsf{aspect}\ s}{\cdot}\rangle_{k}\ (\!|\frac{\_}{s}|\!)_{aspect}$$

$$(\!|\frac{\lambda x.e}{closure(x,(s \curvearrowright e),\rho)}\rangle_{k}\ (\!|\rho|\!)_{env}\ (\!|s|\!)_{aspect}$$

# K-CHALLENGE: Concurrency with threads and lock synchronization

$$Stmt ::= \ldots \mid \mathsf{spawn}\ Stmt \mid \mathsf{acquire}\ Exp\ [strict] \mid \mathsf{release}\ Exp\ [strict]$$

$$Config ::= \ldots \mid \llparenthesis \mathsf{Set}[Val \times Nat] \rrparenthesis_{holds} \mid \llparenthesis \mathsf{Set}[Config] \rrparenthesis_{thread} \mid \llparenthesis \mathsf{Set}[Val] \rrparenthesis_{busy}$$

$$[\![ s ]\!] = \llparenthesis \llparenthesis \llparenthesis s \rrparenthesis_k\ \llparenthesis \cdot \rrparenthesis_{env}\ \llparenthesis \cdot \rrparenthesis_{holds} \rrparenthesis_{thread}\ \llparenthesis \cdot \rrparenthesis_{store}\ \llparenthesis \cdot \rrparenthesis_{output}\ \llparenthesis \cdot \rrparenthesis_{aspect}\ \llparenthesis \cdot \rrparenthesis_{busy} \rrparenthesis_{\top}$$

$$\llparenthesis \llparenthesis \_ \rrparenthesis_{store}\ \llparenthesis vl \rrparenthesis_{output}\ \llparenthesis \_ \rrparenthesis_{aspect}\ \llparenthesis \_ \rrparenthesis_{busy} \rrparenthesis_{\top} = vl$$

$$\llparenthesis \frac{\mathsf{spawn}\ s}{\cdot} \rangle_k\ \llparenthesis \rho \rrparenthesis_{env}\ \frac{\cdot}{\llparenthesis \llparenthesis s \rrparenthesis_k\ \llparenthesis \rho \rrparenthesis_{env}\ \llparenthesis \cdot \rrparenthesis_{holds} \rrparenthesis_{thread}}$$

$$\frac{\langle \llparenthesis \cdot \rrparenthesis_k\ \llparenthesis lc \rrparenthesis_{holds} \rangle_{thread}}{\cdot}\ \llparenthesis \frac{ls}{ls - lc} \rrparenthesis_{busy}$$

$$\llparenthesis \frac{\mathsf{acquire}\ v}{\cdot} \rangle_k\ \langle (v, \frac{n}{s(n)}) \rangle_{holds} \qquad \llparenthesis \frac{\mathsf{release}\ v}{\cdot} \rangle_k\ \langle (v, \frac{s(n)}{n}) \rangle_{holds}$$

$$\llparenthesis \frac{\mathsf{acquire}\ v}{\cdot} \rangle_k\ \langle \frac{\cdot}{(v, 0)} \rangle_{holds}\ \llparenthesis \frac{ls}{ls\ v} \rrparenthesis_{busy}\ \ \texttt{when}\ \ v \notin ls \qquad \llparenthesis \frac{\mathsf{release}\ v}{\cdot} \rangle_k\ \langle \frac{(v, 0)}{\cdot} \rangle_{holds}\ \langle \frac{v}{\cdot} \rangle_{busy}$$

# K-CHALLENGE: Concurrency Rendez-vous synchronization

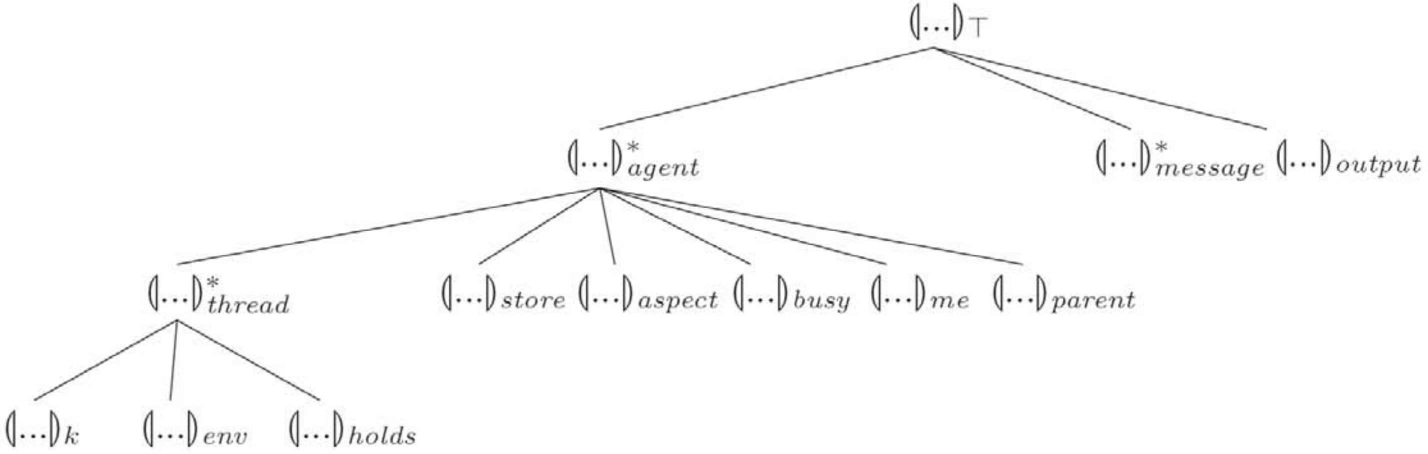
$$Stmt ::= \ldots \mid \mathsf{rv}\ Exp\ [strict]$$

$$\langle\!|\, \underset{\cdot}{\underline{\mathsf{rv}\, v}} \,|\!\rangle_k \ \langle\!|\, \underset{\cdot}{\underline{\mathsf{rv}\, v}} \,|\!\rangle_k$$

# K-CHALLENGE: Concurrency Distributed Agents with Message Comm.

$\mathit{Agent} ::=$ agent identifiers or names
$\mathit{Val} ::= \ldots \mid \mathit{Agent}$
$\mathit{Exp} ::= \ldots \mid \textsf{new-agent}\ \mathit{Stmt} \mid \textsf{receive-from}\ \mathit{Exp}\ [\mathit{strict}] \mid \textsf{receive} \mid \textsf{me} \mid \textsf{parent}$
$\mathit{Stmt} ::= \ldots \mid \textsf{send-asynch}\ \mathit{Exp}\ \mathit{Exp}\ [\mathit{strict}] \mid \textsf{send-synch}\ \mathit{Exp}\ \mathit{Exp}\ [\mathit{strict}]$

$\mathit{Config} ::= \ldots \mid \llparenthesis \textsf{Set}[\mathit{Config}] \rrparenthesis_{\mathit{agent}} \mid \llparenthesis \mathit{Agent}, \mathit{Agent}, \mathit{Val} \rrparenthesis_{\mathit{message}}$
$[\![ s ]\!] = \llparenthesis \llparenthesis \llparenthesis \llparenthesis s \rrparenthesis_k\ \llparenthesis \cdot \rrparenthesis_{\mathit{env}}\ \llparenthesis \cdot \rrparenthesis_{\mathit{holds}} \rrparenthesis_{\mathit{thread}}\ \llparenthesis \cdot \rrparenthesis_{\mathit{store}}\ \ \llparenthesis \cdot \rrparenthesis_{\mathit{aspect}}\ \llparenthesis \cdot \rrparenthesis_{\mathit{busy}}\ \llparenthesis n \rrparenthesis_{\mathit{me}}\ \llparenthesis n \rrparenthesis_{\mathit{parent}} \rrparenthesis_{\mathit{agent}}\ \llparenthesis \cdot \rrparenthesis_{\mathit{output}} \rrparenthesis_{\top}$ where $n \in \mathit{Agent}$ fresh
$\llparenthesis \llparenthesis vl \rrparenthesis_{\mathit{output}}\ M \rrparenthesis_{\top} \to vl$ when $M$ contains only messages (zero or more)



# K-CHALLENGE: Concurrency Distributed Agents with Message Comm.

$$\llparenthesis \underset{m}{\underline{\text{new-agent } s}} \rangle_k \; \llparenthesis n \rrparenthesis_{me} \; \underset{\llparenthesis \llparenthesis \llparenthesis s \rrparenthesis_k \; \llparenthesis \cdot \rrparenthesis_{env} \; \llparenthesis \cdot \rrparenthesis_{holds} \rrparenthesis_{thread} \; \llparenthesis \cdot \rrparenthesis_{store} \; \llparenthesis \cdot \rrparenthesis_{aspect} \; \llparenthesis \cdot \rrparenthesis_{busy} \; \llparenthesis m \rrparenthesis_{me} \; \llparenthesis n \rrparenthesis_{parent} \rrparenthesis_{agent}}{\underline{\qquad\qquad\qquad\qquad\cdot\qquad\qquad\qquad\qquad}}$$

$$\llparenthesis A \rrparenthesis_{agent} \rightarrow \cdot \quad \text{when } A \text{ contains no thread}$$

$$\llparenthesis \underset{n}{\underline{\text{me}}} \rangle_k \; \llparenthesis n \rrparenthesis_{me}$$

$$\llparenthesis \underset{n}{\underline{\text{parent}}} \rangle_k \; \llparenthesis n \rrparenthesis_{parent}$$

$$\llparenthesis \underset{\cdot}{\underline{\text{send-asynch } m \; v}} \rangle_k \; \llparenthesis n \rrparenthesis_{me} \; \underset{\llparenthesis n, m, v \rrparenthesis_{message}}{\underline{\qquad\qquad\cdot\qquad\qquad}}$$

$$\llparenthesis \underset{v}{\underline{\text{receive-from } n}} \rangle_k \; \llparenthesis m \rrparenthesis_{me} \; \underset{\cdot}{\underline{\llparenthesis n, m, v \rrparenthesis_{message}}}$$

$$\llparenthesis \underset{v}{\underline{\text{receive}}} \rangle_k \; \llparenthesis m \rrparenthesis_{me} \; \underset{\cdot}{\underline{\llparenthesis \_, m, v \rrparenthesis_{message}}}$$

$$\langle \llparenthesis \underset{\cdot}{\underline{\text{send-synch } m \; v}} \rangle_k \; \llparenthesis n \rrparenthesis_{me} \rangle_{agent} \; \langle \llparenthesis \underset{v}{\underline{\text{receive-from } n}} \rangle_k \; \llparenthesis m \rrparenthesis_{me} \rangle_{agent}$$

$$\llparenthesis \underset{\cdot}{\underline{\text{send-synch } m \; v}} \rangle_k \; \langle \llparenthesis \underset{v}{\underline{\text{receive}}} \rangle_k \; \llparenthesis m \rrparenthesis_{me} \rangle_{agent}$$

# K-CHALLENGE: Self-Generation of Code

$$Exp ::= \ldots \mid \textsf{quote}\ Exp \mid \textsf{unquote}\ Exp \mid \textsf{eval}\ Exp\ [strict]$$
$$Val ::= \ldots \mid code(K)$$
$$K ::= \ldots \mid quote(Nat, \textsf{List}[K]) \mid code(\textsf{List}[K]) \mid K \boxed{\curvearrowright} K\ \ [strict] \mid \boxed{Id}(\textsf{List}[K])\ \ [strict(2)] \mid K \boxed{,} K\ \ [strict]$$

$$(\!|\textsf{quote}(k)|\!)_k = (\!|quote(0,k)|\!)_k$$
$$quote(n, k_1 \curvearrowright k_2) = quote(n,k_1) \boxed{\curvearrowright} quote(n,k_2) \qquad code(k_1) \boxed{\curvearrowright} code(k_2) = code(k_1 \curvearrowright k_2)$$
$$quote(n, f(kl)) = \boxed{f}(quote(n,kl)) \text{ if } f \neq \textsf{quote}, \textsf{unquote} \qquad \boxed{f}(code(kl)) = code(f(kl))$$
$$quote(n, \textsf{quote}(k)) = \boxed{\textsf{quote}}(quote(s(n),k))$$
$$quote(0, \textsf{unquote}(k)) = k$$
$$quote(s(n), \textsf{unquote}(k)) = \boxed{\textsf{unquote}}(quote(n,k))$$

$$quote(n,(k,kl)) = quote(n,k) \boxed{,} quote(n,kl) \text{ if } kl \neq \cdot \qquad code(k) \boxed{,} code(kl) = code(k,kl)$$
$$quote(n,k) = code(k) \text{ if } k \in Val \cup Name \qquad \textsf{eval}\ code(k) = k$$

# Conclusion

- K: The Concurrent Rewrite Abstract Machine
- Attempts at maximizing the amount of concurrency in a formal semantic definition
  - Explicit sharing of data
  - Special support for lists and bags
  - Special representation of computations and tasks
- Next step: efficient implementation based on transactions

Stop here

# The Idea

- There is a major difference between
  - Sequential rewrite steps; and
  - Parallel rewrite steps
- Arbitrary degree of parallelism at each step: from minimal to maximal

# From Conditional to Unconditional

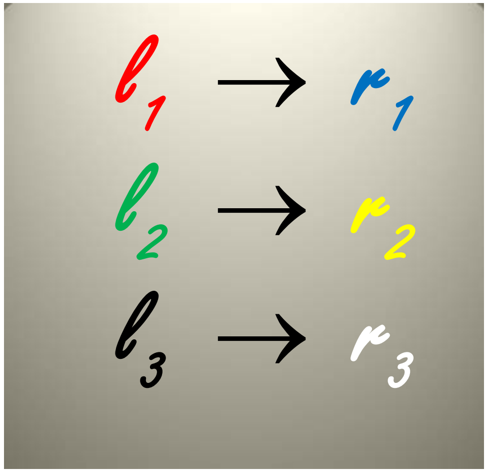
- Many transformations in the literature
  - 1996: Alouini, Kirchner

# Concurrent Rewriting

- Concurrent implementations of rewriting
  - 1990: Aida, Goguen, Meseguer
  - 1996: Alouini, Kirchner (also GC in this context)
- We want a different thing:
  - Concurrent rewriting as a formalism
  - Shared data allowed and specifiable

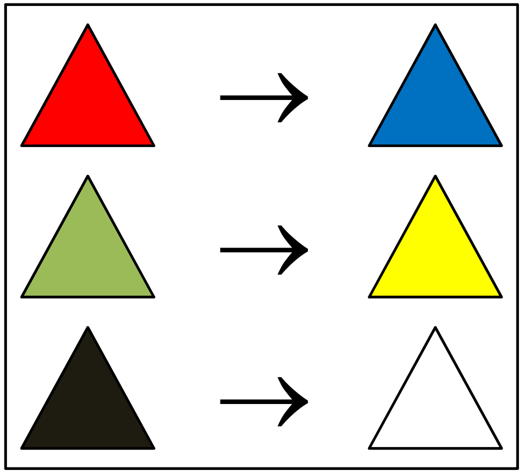
# A Rewrite System

Consider a three rule rewrite system



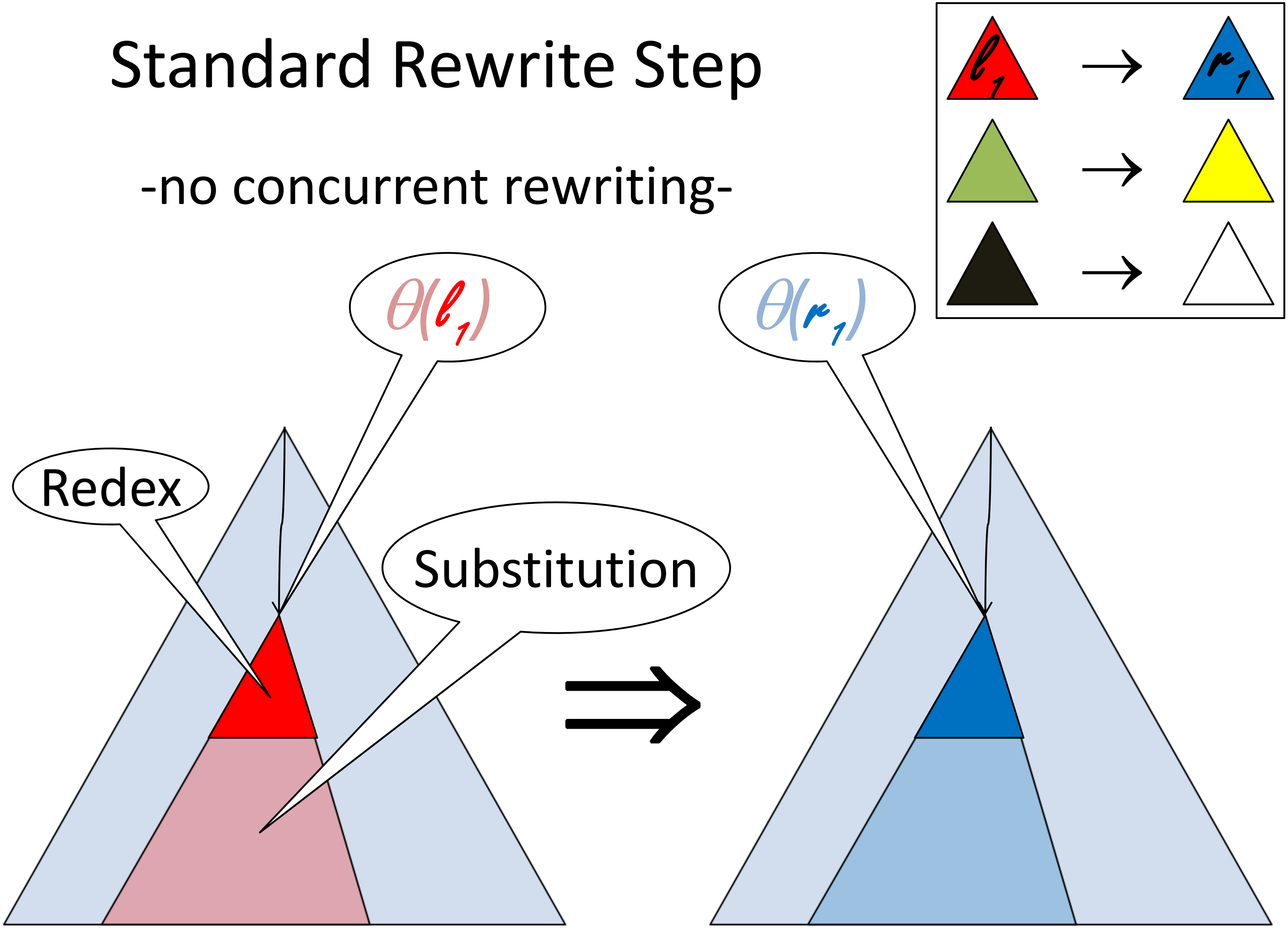
$$\ell_1 \to r_1$$
$$\ell_2 \to r_2$$
$$\ell_3 \to r_3$$

or graphically

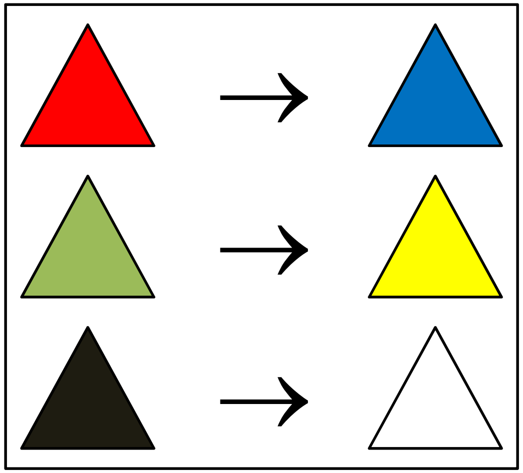


# Standard Rewrite Step

-no concurrent rewriting-

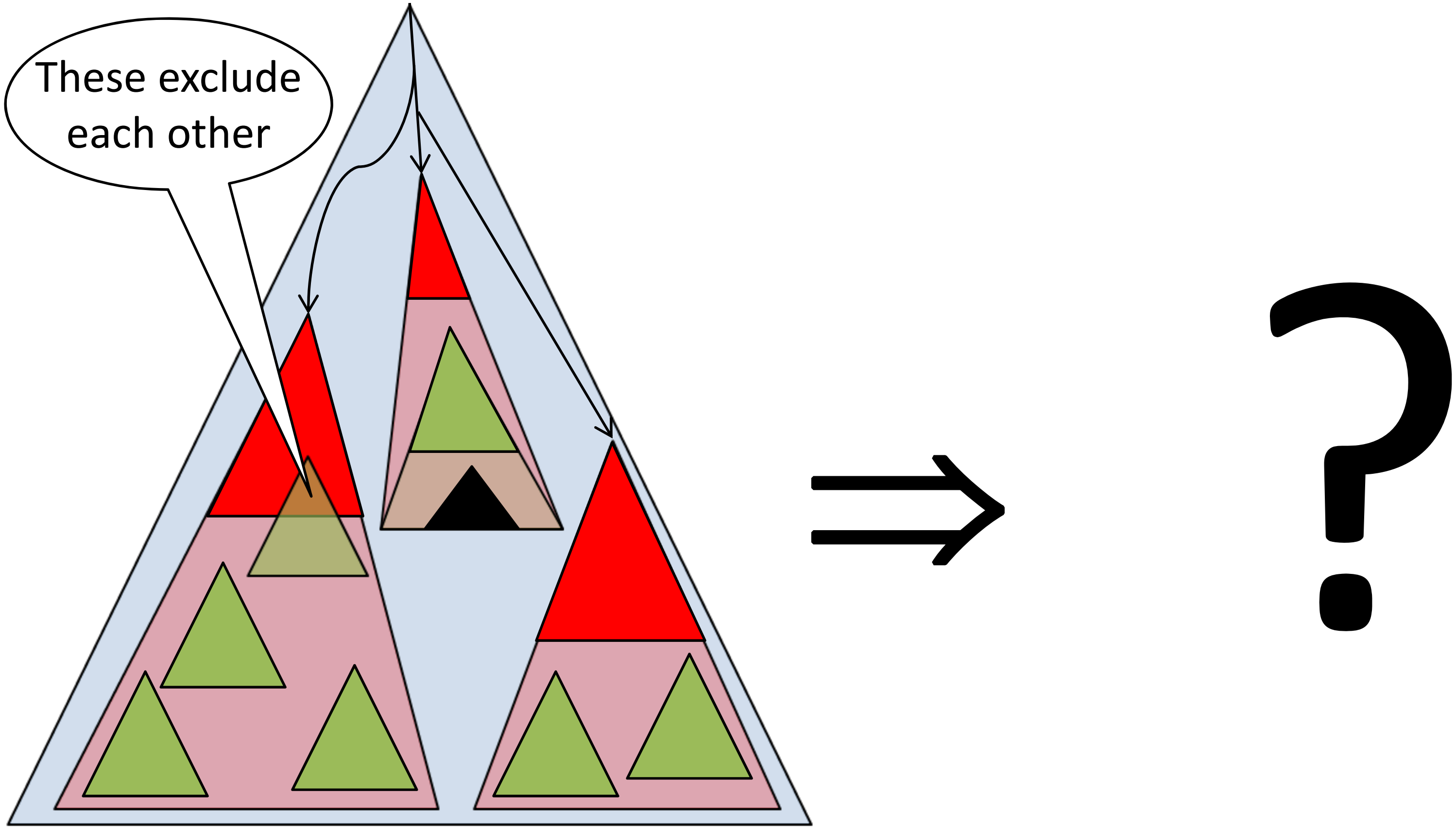


# Concurrent Rewrite Step in K



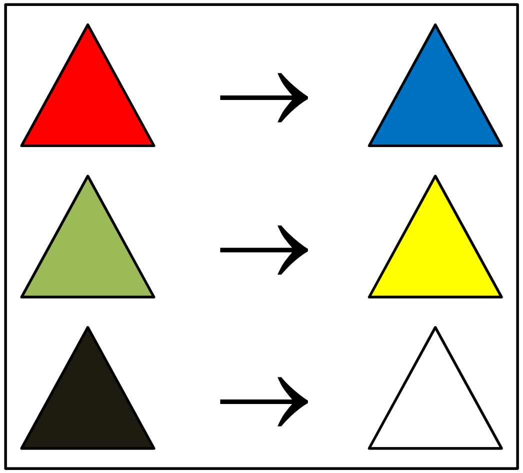
Q: How many rewrites can be applied concurrently on the term below?

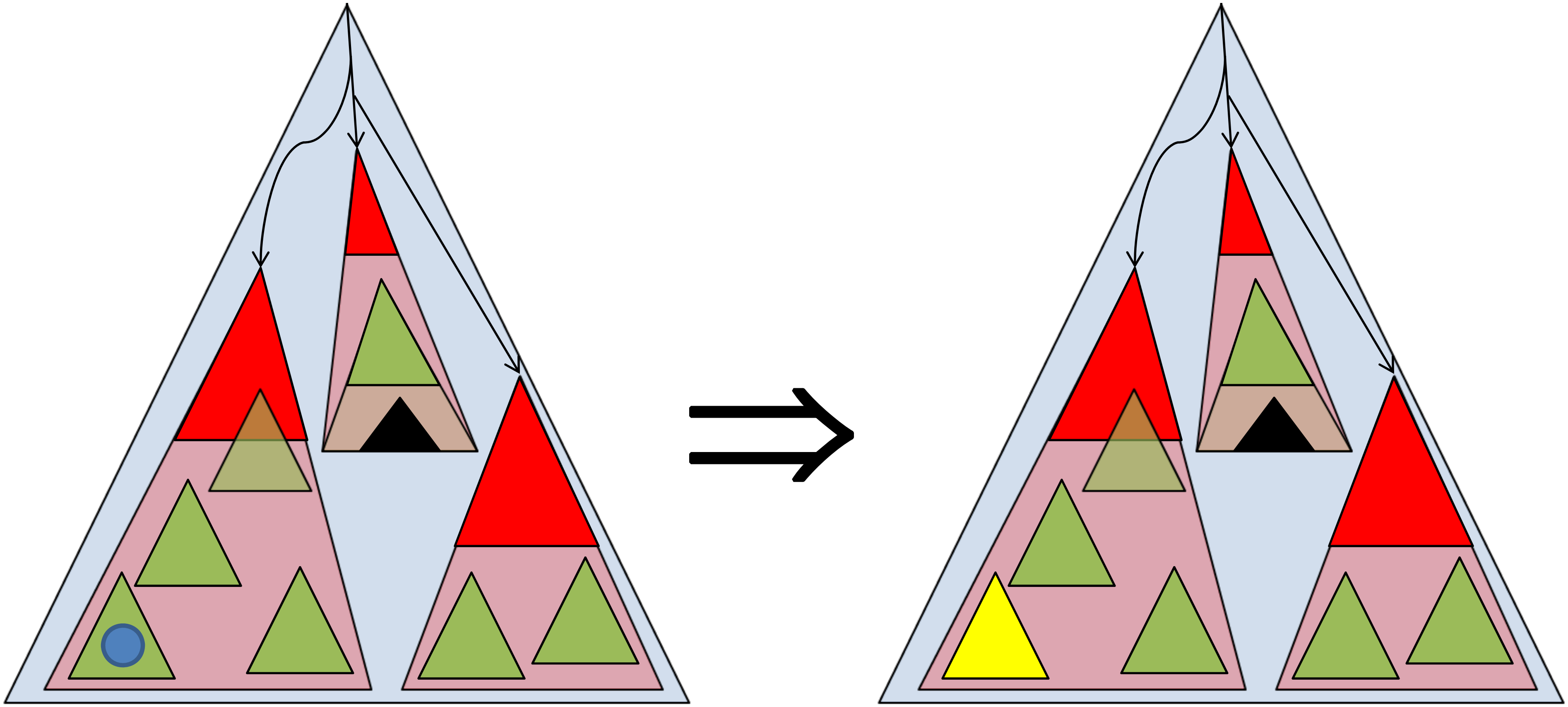
A: 0, 1, 2, …, 10 out of 11 redexes!



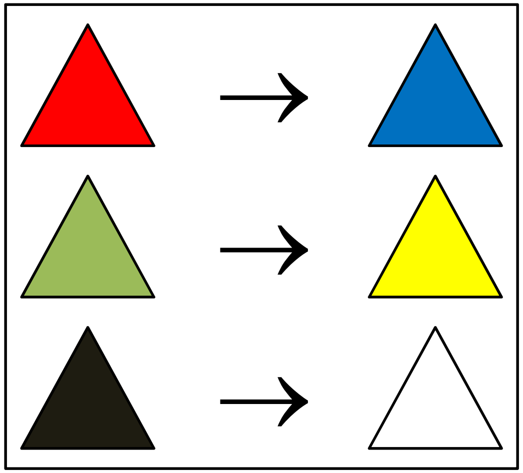
# Concurrent Rewrite Step in K

-1 rewrite- (standard rewriting)

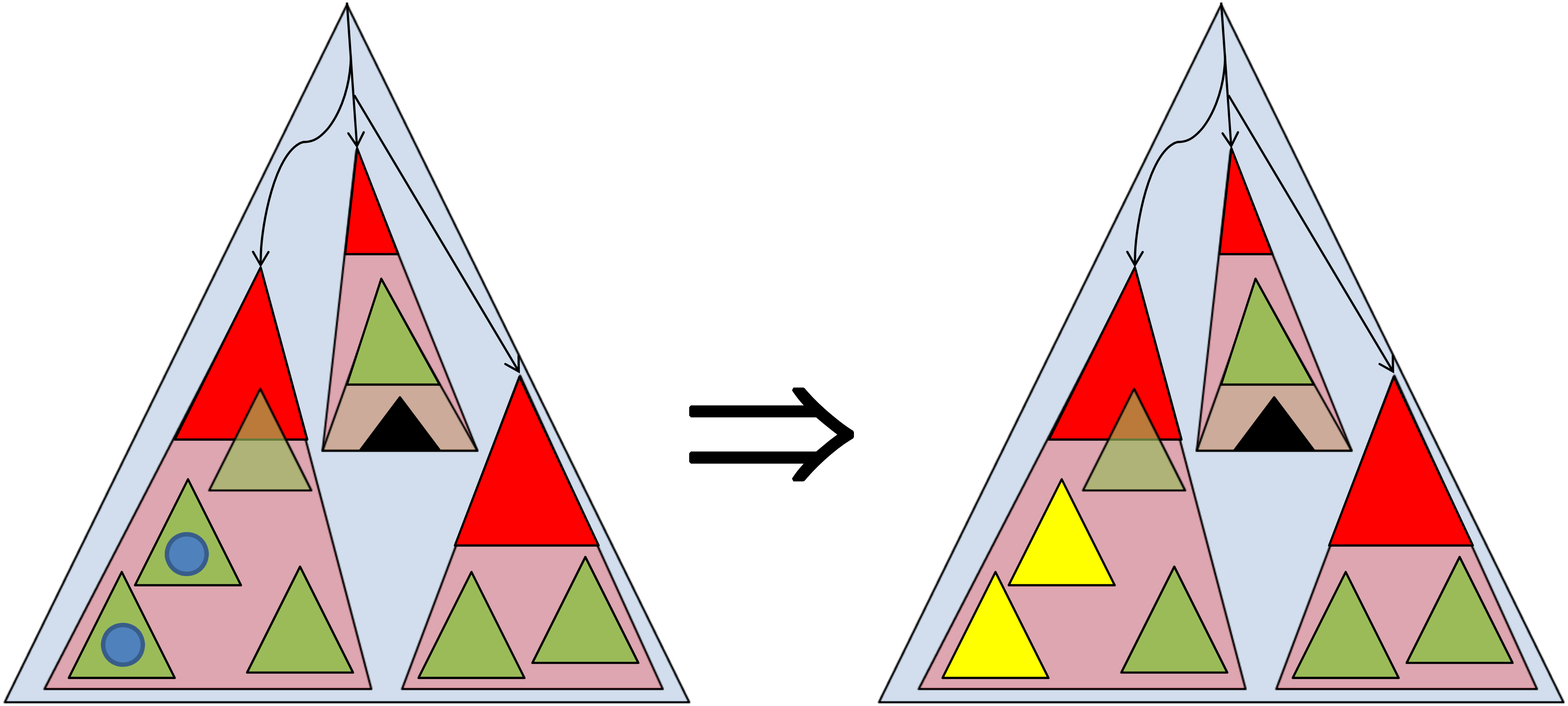




# Concurrent Rewrite Step in K

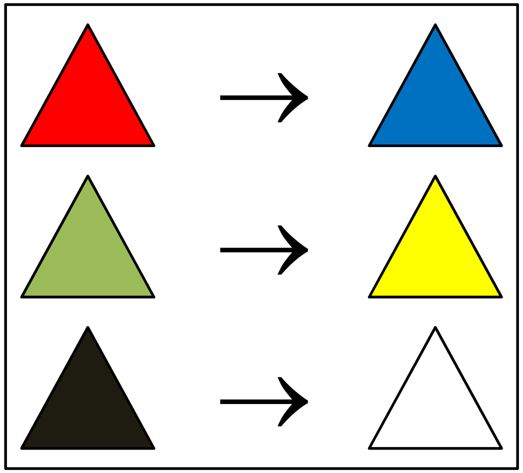


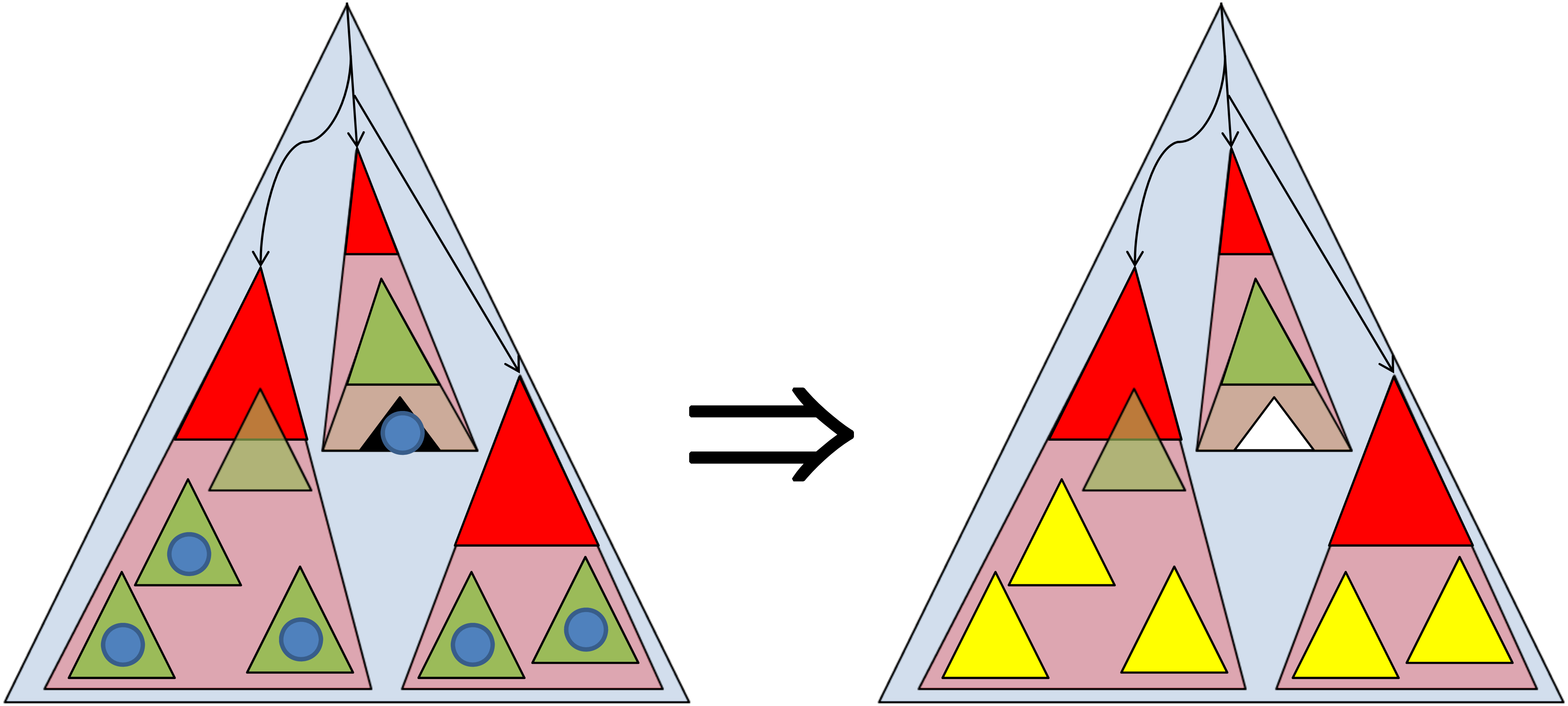
-2 concurrent rewrites-



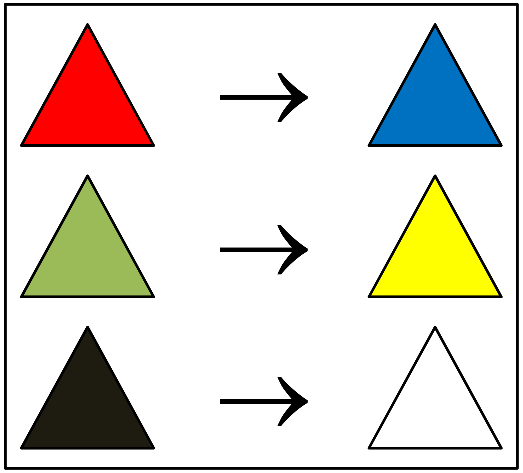
# Concurrent Rewrite Step in K

-6 concurrent rewrites-

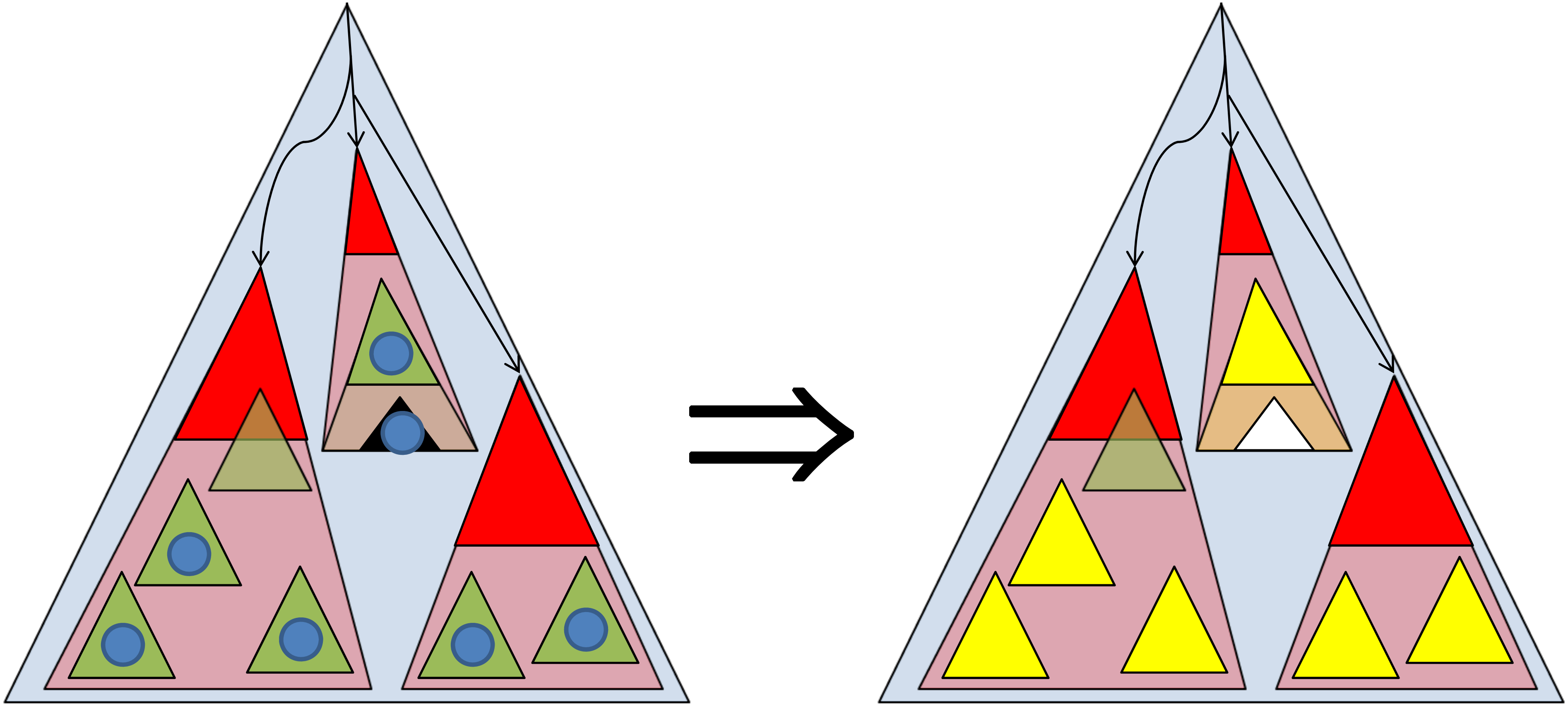




# Concurrent Rewrite Step in K

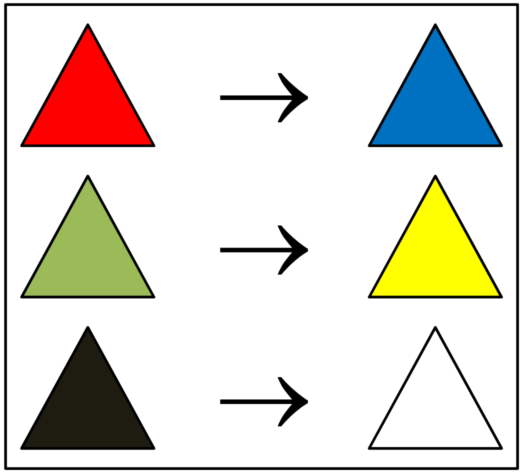


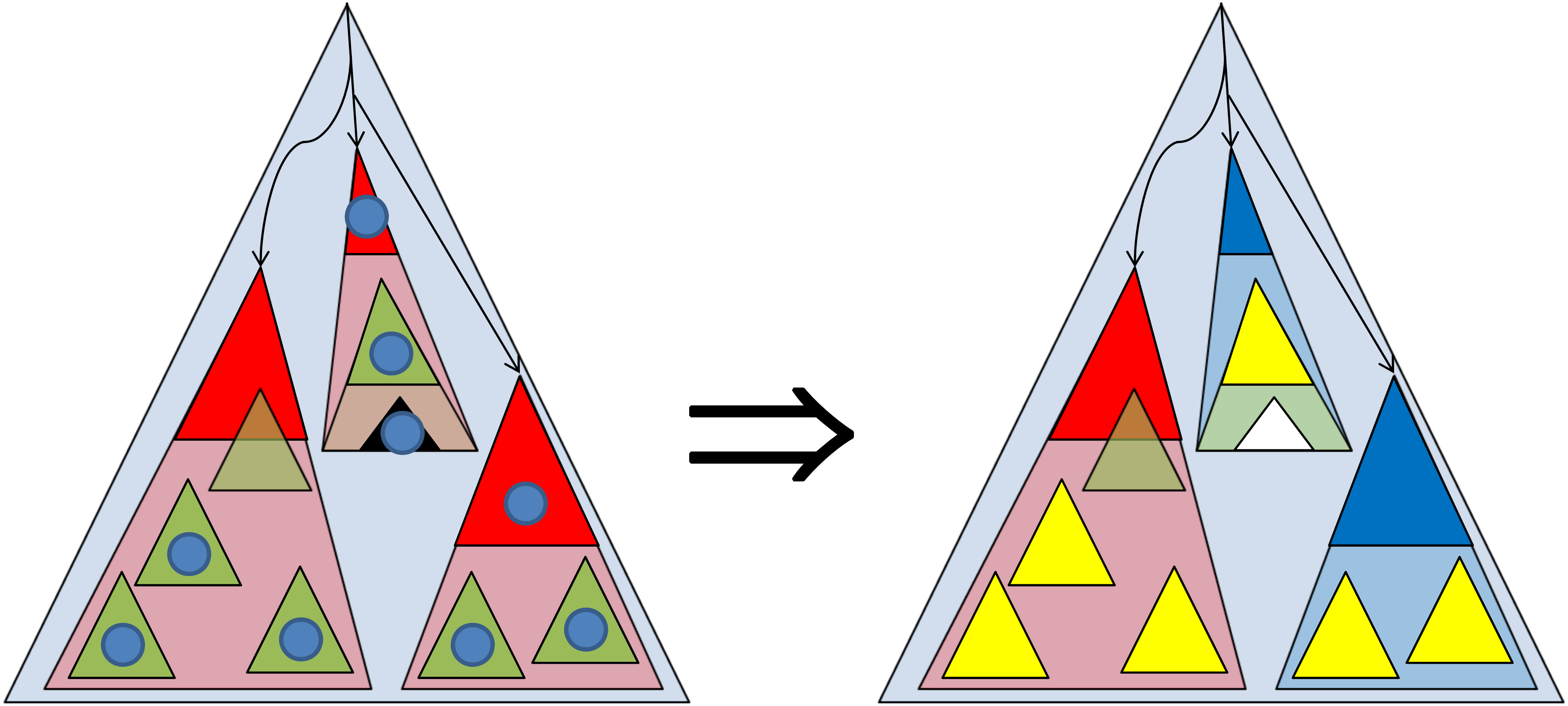
-7 concurrent rewrites-



# Concurrent Rewrite Step in K

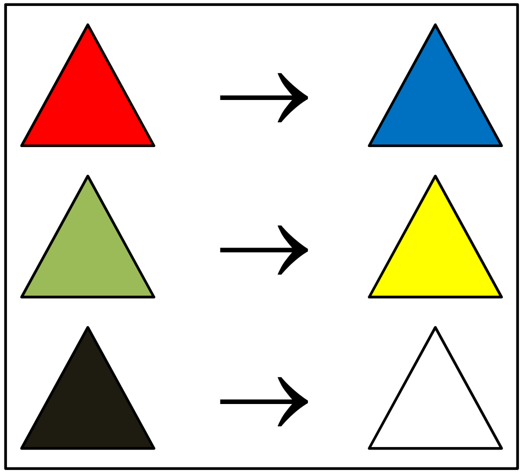
-9 concurrent rewrites-

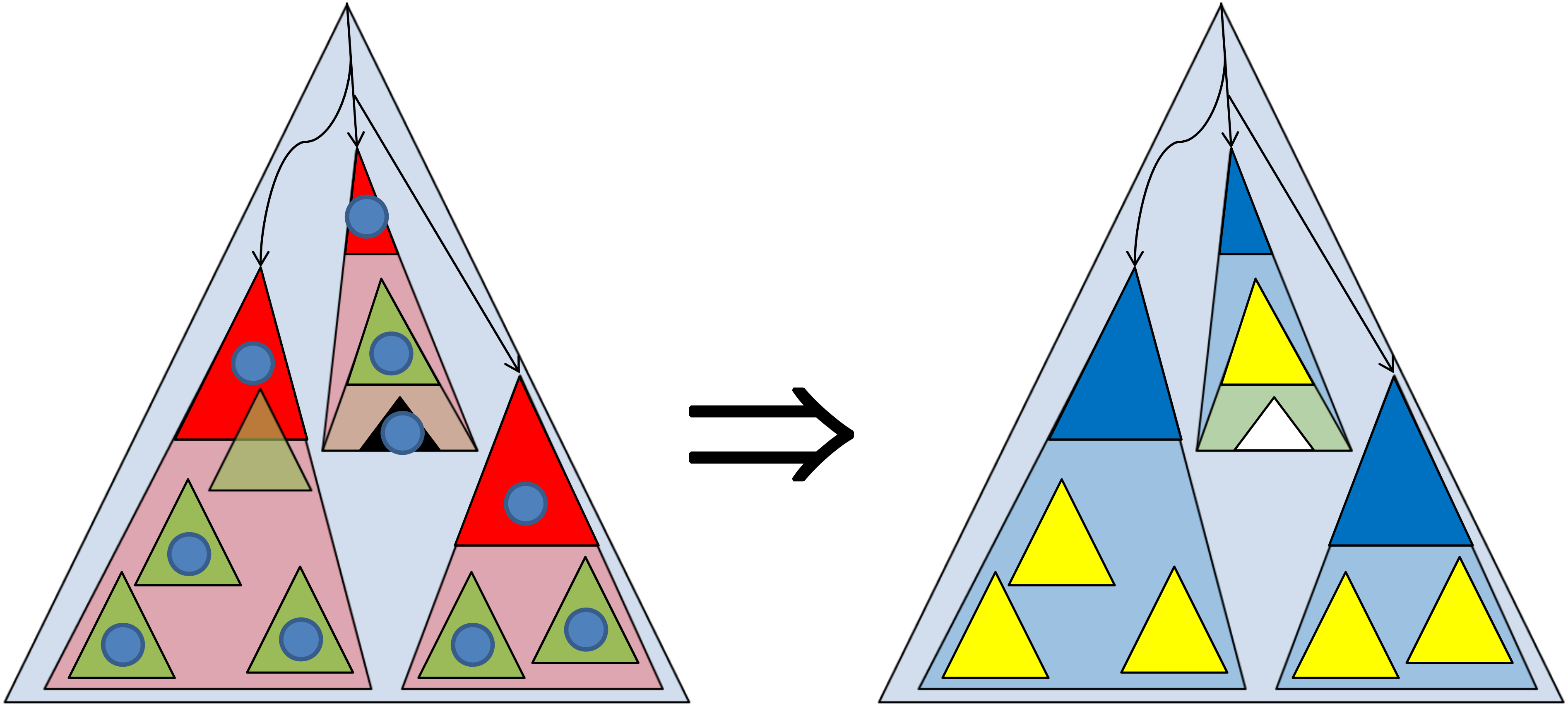




# Concurrent Rewrite Step in K

## -10 concurrent rewrites (ver 1)-





# Concurrent Rewrite Step in K

## -10 concurrent rewrites (ver 2)-