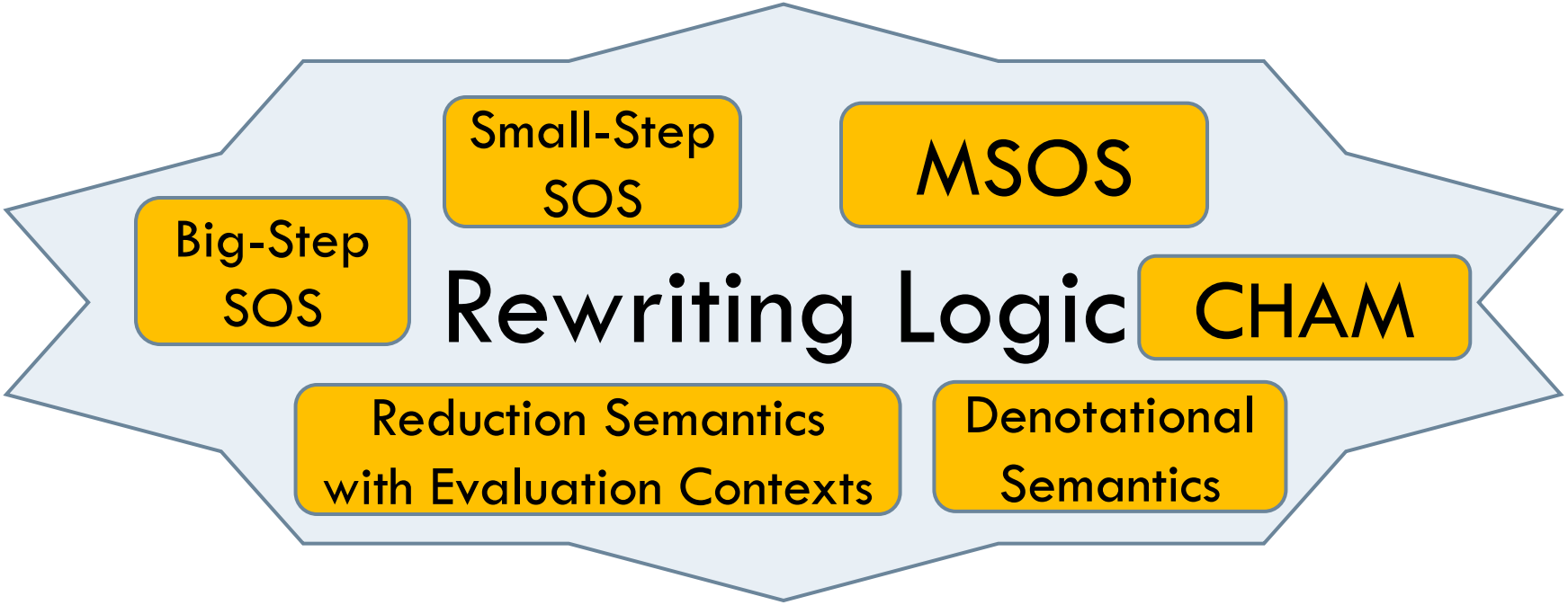
K
A REWRITE-BASED LANGUAGE DEFINITIONAL FRAMEWORK

Grigore Rosu

CS422 – Programming Language Design

Observation

- The programming language definitional approaches discussed so far, each with its advantages and disadvantages, have been *faithfully* represented in rewriting logic and Maude
  - By faithful representation we mean one which preserves everything the original had: computation granularity (step-for-step), modularity



Natural Questions

- ***(Q1)*** *Why not use rewriting logic instead as an ideal semantic framework for programming languages?*
  - Good question! ☺
- Well, rewriting logic is so general that it is does not tell us *how* to define languages in it
  - All the various styles we used so far in class and corresponding each to some well-established semantic approach stand as proof
- ***(Q2)*** *Can we then develop a particular "ideal" style within rewriting logic, namely one that has all the advantages of the other styles at the same time overcoming their limitations?*
  - That was precisely the motivation for the K framework
  - Whether K achieved it or not is, and will probably always be, open

The K Framework

- K started as a style within rewriting logic, but it got its own concurrent semantics to better directly capture the intended concurrent semantics of the defined programming languages
- K consists of two components:
  - The *K definitional technique* can be used within any rewriting logic setting and can be executed on any rewrite engine
  - The *K concurrent rewrite abstract machine*, or the *KRAM*, brings more concurrency to K definitions then their direct translation to rewriting logic gives, but it has no implementation to date; we only use it as a theoretical model for the time being (same as the claimed concurrency of the CHAM)

Relationship Between K and SOS

- K, like other styles, borrowed from SOS the ideas of "syntax-driven semantics" and of "configuration". Unlike SOS,
  - K is based on "rewriting" instead of "reduction", so permission to apply rules needs to be taken, instead of given; indeed, in rewriting rules apply wherever they match, with no contextual restrictions
  - K takes permission to apply rules by structural means (it does not use operator strategies as rewriting logic and Maude, because those are of limited use in K)
  - K rules have no premises (only side conditions)
  - In other words, the K computational model is simple and uniform:
    - Apply rules wherever they match, provided the side conditions hold

Relationship Between K and MSOS

- K borrowed from MSOS the ideas of labeling semantic items and of only mentioning in each rule those parts of the configuration which are relevant to that rule. Unlike in MSOS,
  - In K the syntactic and the semantic components are treated uniformly, the syntax being just another part of the configuration
  - In K all syntactic and semantic components are stored in units called cells, which can be arbitrarily nested and labeled; the labels themselves can also be rewritten in K
  - K's meaning for the non-mentioned parts of the configuration is "they can be concurrently changed by other rule instances"; MSOS, like SOS, is by its very nature an interleaving semantics, because each step ends up taking place at the top of the configuration

Relationship Between K and RSEC

- K borrowed from reduction semantics with evaluation contexts (RSEC) the basic idea of "evaluation context". Unlike RSEC,
  - K represents evaluation contexts flattened as sequences of computational tasks, as we did in CHAM (actually we borrowed that idea in CHAM from K, since the airlock was not powerful enough to correctly define the evaluation strategies of IMP's constructs)
  - K does not make any attempt to be faithful to syntax; in particular, it uniformly supports purely syntactic definitions based on substitutions, as well as implementation-like definitions of abstract machines based on environments, stacks, continuations, etc.
  - K prefers to use its nested-cell approach to define configurations, instead of treating the configurations as "syntax" in order to use the same syntactic mechanisms also for reading/writing semantic info.

Relationship Between K and CHAM

- K borrowed from the CHAM the ideas of representing configurations as possibly nested bags (or multisets) and of heating/cooling the syntax, but in a more general setting and without the chemical load. For example:
  - K's cells can contain not only bags, but also lists, sets and maps
  - K's rules can apply everywhere they match, not only in solutions; if one wants to limit the application of a rule to solutions only like in the CHAM, then one can simply mention the membrane in the rule
  - K does not use any airlock, because the airlock is unnecessary when one allows the full power of AC matching, like K does. The CHAM had the airlock for chemical intuitions and for technical concerns that multiset matching is not feasible. Today's advances in rewriting modulo AC make CHAM's technical concern a non-issue anymore

Relationship Between K and Denotational Semantics

- Even though K has not been inspired by denotational semantics, both are mathematically grounded. Moreover, it should be possible to associate a denotational semantics to any K definition as follows (nobody did it formally so far, though):
  - K, through its representation in rewriting logic, can be endowed with an initial model semantics, which can be regarded as "the" mathematical domain of interpretation for the language syntax
  - To achieve that, we need to isolate the syntax from the rest of the configuration and, instead, to interpret the syntax into the domain of functions from configurations to configurations
  - To define the function associated to each language construct, one would need to "run" the K semantics, operation which can be regarded as a fixed-point

K Definitions / K Systems

- K definitions, also called K systems, consist of:
  - *Configurations*
    - Nested and labeled associative or associative/commutative "soups", holding all necessary information: current computation, environments, stores, threads, etc.
  - *Computations*
    - Special list structures extending abstract syntax
  - *Rules*
    - Can be structural or computational
    - Structural rules allow for re-arrangements of the configuration, in particular of the computations (we call some of these structural rules heating/cooling rules, inspired from the CHAM)
    - Computational rules are those performing actual computational steps

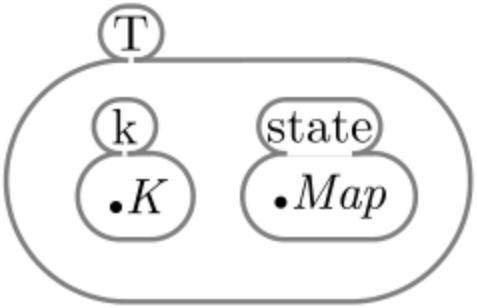
Configurations

- Nested and labeled cells holding any algebraic structure, including sets, multisets, lists, maps, etc.
  - For example, the configuration of IMP consists of a top-level cell holding a computation (explained shortly) cell and a state cell:

$$Configuration_{\text{IMP}} \equiv \langle \langle K \rangle_{\text{k}} \ \langle \mathbf{Map}\{Id \mapsto Int\} \rangle_{\text{state}} \rangle_{\text{T}}$$

  - Here is a concrete cell holding and empty computation and an empty state (dots, possibly qualified, are the units of lists, sets, maps). Both notations below are supported by our implementation of K (which compiles into Maude):

$$\langle \langle \cdot K \rangle_{\text{k}} \ \langle \cdot Map \rangle_{\text{state}} \rangle_{\text{T}}$$



Syntax of Configurations

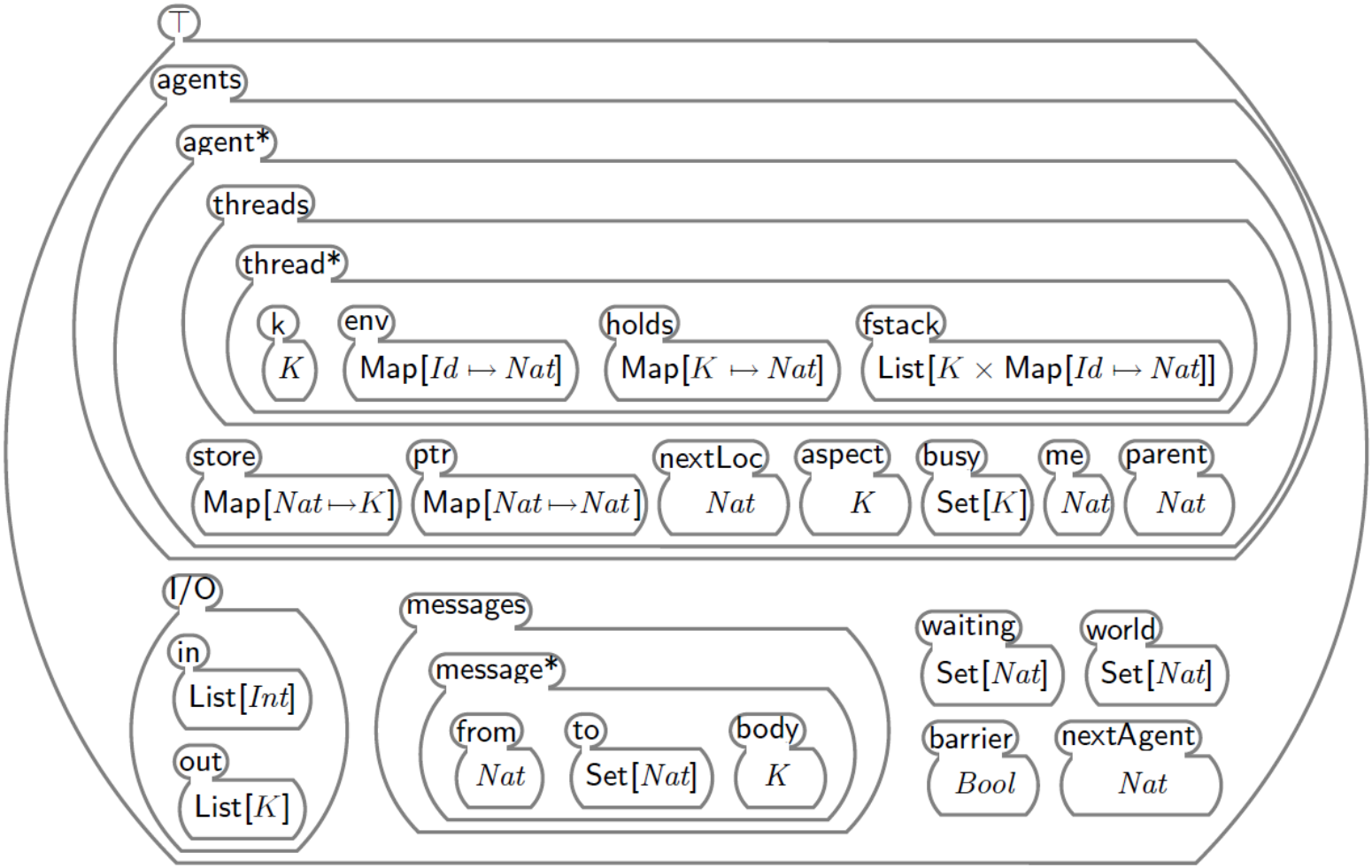
- K configurations obey the following general syntax:

$$
\begin{array}{rcl}
Cell & ::= & \langle CellContents \rangle_{CellLabel} \\
CellContents & ::= & Sort \mid \mathbf{Bag}\_\{Cell\} \\
CellLabel & ::= & CellName \mid CellName* \\
CellName & ::= & \top \mid \mathsf{k} \mid \_ \mid \mathsf{env} \mid \mathsf{store} \mid \ldots
\end{array}
$$

A * means that type of cell can appear multiple times

- Each cell has a label (possibly empty, as the whitespace cell name above indicates) and can contain anything, including a bag of other cells. Lists, sets, bags, and maps are assumed "builtin" and can be used whenever desired. They all have the dot "." as unit, which can be qualified with the corresponding sort name to avoid confusion if desired or needed

Configuration of CHALLENGE



Computations

- Computations are list terms of special "builtin" sort K, which have the following form (curved arrow reads "followed by" or "and then"):

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

- They extend the syntax of the language and of its evaluation contexts with the "followed by" construct
- Examples

$$a_1 \curvearrowright \Box + a_2$$
$$a_2 \curvearrowright a_1 + \Box$$
$$s_1 \curvearrowright s_2$$
$$b \curvearrowright \text{if } \Box \text{ then } s_1 \text{ else } s_2$$

K Rules

- So far, we only introduced special K syntax, namely syntax used for configurations and syntax used for computations
- K definitions, or K systems, consist of syntax as above plus a set of K rules that operate on this syntax by iteratively transforming terms until they cannot be rewritten anymore
- K rules can be
  - *Structural*, which have no computational meaning and whose role is to rearrange the term so that computational rules can apply; and
  - *Computational*, which define the computational steps that irreversibly modify, or evolve, the configuration

K Heating/Cooling Rules

- A special category of K structural rules is particularly common in K definitions, namely the *heating/cooling rules*
  - Typically reversible
  - Typically used to define evaluation strategies

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

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

$$\text{if } k \text{ then } k_1 \text{ else } k_2 \rightleftharpoons k \curvearrowright \text{if } \square \text{ then } k_1 \text{ else } k_2$$

Computational Classes

- Heating/cooling rules lead to classes of computations (equivalence classes if rules are reversible), for example:

$$
\begin{aligned}
& x * (y + 2) \\
& x \curvearrowright (\square * (y + 2)) \\
& x \curvearrowright (\square * (y \curvearrowright (\square + 2))) \\
& x \curvearrowright (\square * (2 \curvearrowright (y + \square))) \\
& (y + 2) \curvearrowright (x * \square) \\
& y \curvearrowright (\square + 2) \curvearrowright (x * \square) \\
& 2 \curvearrowright (y + \square) \curvearrowright (x * \square) \\
& x * (y \curvearrowright (\square + 2)) \\
& x * (2 \curvearrowright (y + \square))
\end{aligned}
$$

Strictness Notation

- We prefer to annotate syntax as follows:

$$AExp + AExp \qquad [strict]$$

$$\text{if } BExp \text{ then } Stmt \text{ else } Stmt \qquad [strict(1)]$$

- Which desugars into heating/cooling rules:

$$a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$$
$$a_1 + a_2 \rightleftharpoons a_2 \curvearrowright a_1 + \square$$
$$\text{if } k \text{ then } k_1 \text{ else } k_2 \rightleftharpoons k \curvearrowright \text{if } \square \text{ then } k_1 \text{ else } k_2$$

Example: K Annotated Syntax of IMP

Original language syntax	K Strictness
$AExp ::= Int$	
$\mid Id$	
$\mid AExp$ `+` $AExp$	$[strict]$
$\mid AExp$ `/` $AExp$	$[strict]$
$BExp ::= Bool$	
$\mid AExp$ `<=` $AExp$	$[seqstrict]$
$\mid$ `not` $BExp$	$[strict]$
$\mid BExp$ `and` $BExp$	$[strict(1)]$
$Stmt ::=$ `skip`	
$\mid Id$ `:=` $AExp$	$[strict(2)]$
$\mid Stmt$ `;` $Stmt$	
$\mid$ `if` $BExp$ `then` $Stmt$ `else` $Stmt$	$[strict(1)]$
$\mid$ `while` $BExp$ `do` $Stmt$	
$Pgm ::=$ `vars` **List**`{`Id`}` `;` $Stmt$	

K Rules in Their Full Generality

- The heating/cooling rules above are very particular
- In general, K rules can match a pattern and modify only parts of it:

$$p[\underset{r_1}{l_1}, \underset{r_2}{l_2}, \ldots, \underset{r_n}{l_n}]$$

- Example: the K semantics of variable assignment in IMP:

$$\langle \underset{\cdot}{x := i} \cdots \rangle_{\mathsf{k}} \ \langle \cdots x \mapsto \underset{i}{\_} \cdots \rangle_{\mathsf{state}}$$

- The _ and the … stand for "whatever"
  - The former is just an ordinary nameless variable (like in Prolog)
  - The latter used when the cell holds an associative or an associative and commutative "soup", case in which it also includes its top-level construct

Discussion on K Rules

- The notation for K rules generalizes usual notation for deduction rules
  - Consider a logic and associate it a signature adding syntax for the meta-logical terms: sorts $Theory$ and $Sequent$ for theories and sequents, operation

$$\_ \vdash \_ : Theory \times Sentence \rightarrow Sequent$$

  - Now K rules where the pattern p is taken to be empty and the number n of terms above the line is taken to be 1 and the sort of the term above the line is $Set[Sequent]$ while the sort of the term below the line is $Sequent$ become nothing but conventional deduction rules in the considered logic
  - When p is empty and n is 1, we prefer to use the conventional rewrite notation, with arrows ($\rightarrow$ or similar) instead of a horizontal line
- K rules are equivalent with (but more compact than) conventional rewrite rules when one is not interested in concurrency
- K rules are like transactions: modified parts are read-write, rest of the pattern is read-only; concurrent rules can share the read-only

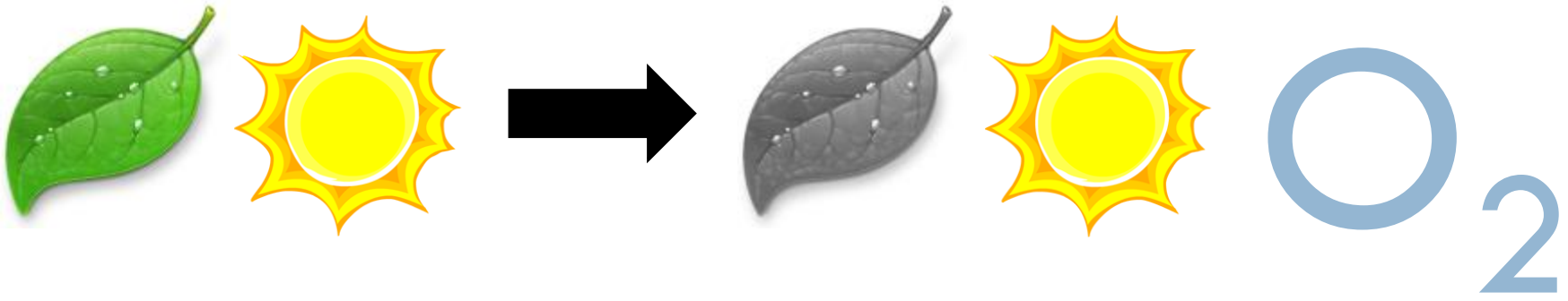
Complete K Semantics of IMP

Original language syntax	K Strictness	K Semantics
$AExp ::= Int$		
$\mid Id$		$\langle \underline{x} \cdots \rangle_{\text{k}} \langle \cdots x \mapsto i \cdots \rangle_{\text{state}}$ $\quad\; i$
$\mid AExp$ + $AExp$	$[strict]$	i_1 + $i_2 \rightarrow i_1 +_{Int} i_2$
$\mid AExp$ / $AExp$	$[strict]$	i_1 / $i_2 \rightarrow i_1 /_{Int} i_2$ where $i_2 \neq 0$
$BExp ::= Bool$		
$\mid AExp$ <= $AExp$	$[seqstrict]$	i_1 <= $i_2 \rightarrow i_1 \leq_{Int} i_2$
$\mid$ not $BExp$	$[strict]$	not $t \rightarrow \neg_{Bool} t$
$\mid BExp$ and $BExp$	$[strict(1)]$	$true$ and $b \rightarrow b$
		$false$ and $b \rightarrow$ false
$Stmt ::=$ skip		skip $\rightarrow \cdot$
$\mid Id$:= $AExp$	$[strict(2)]$	$\langle \underline{x := i} \cdots \rangle_{\text{k}} \langle \cdots x \mapsto \_ \cdots \rangle_{\text{state}}$ $\quad\; \cdot \qquad\qquad\; i$
$\mid Stmt$; $Stmt$		s_1 ; $s_2 \rightarrow s_1 \curvearrowright s_2$
$\mid$ if $BExp$ then $Stmt$ else $Stmt$	$[strict(1)]$	if $true$ then s_1 else $s_2 \rightarrow s_1$
		if $false$ then s_1 else $s_2 \rightarrow s_2$
$\mid$ while $BExp$ do $Stmt$		$\langle$ while b do s $\cdots \rangle_{\text{k}}$ if b then $(s$; while b do $s)$ else $\cdot$
$Pgm ::=$ vars List$\{Id\}$; $Stmt$		$\langle$ vars xl ; $s \rangle_{\text{k}} \langle \cdot \rangle_{\text{state}}$ $\quad\; s \qquad\quad\; xl \mapsto 0$

Concurrency in K

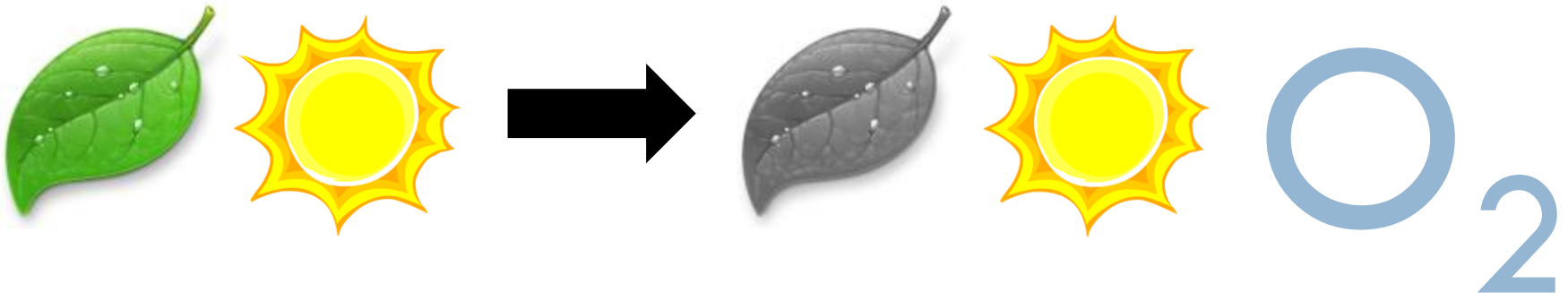
- The remaining slides are optional
- They explain why and how K systems achieve "more true concurrency" than other frameworks
- The slides are quite metaphorical; if interested in the formal details, then please check the paper
  - "An Overview of the K Semantic Framework" in Journal of Logic and Algebraic Programming, Volume 79, Issue 6, August 2010, Pages 397-434

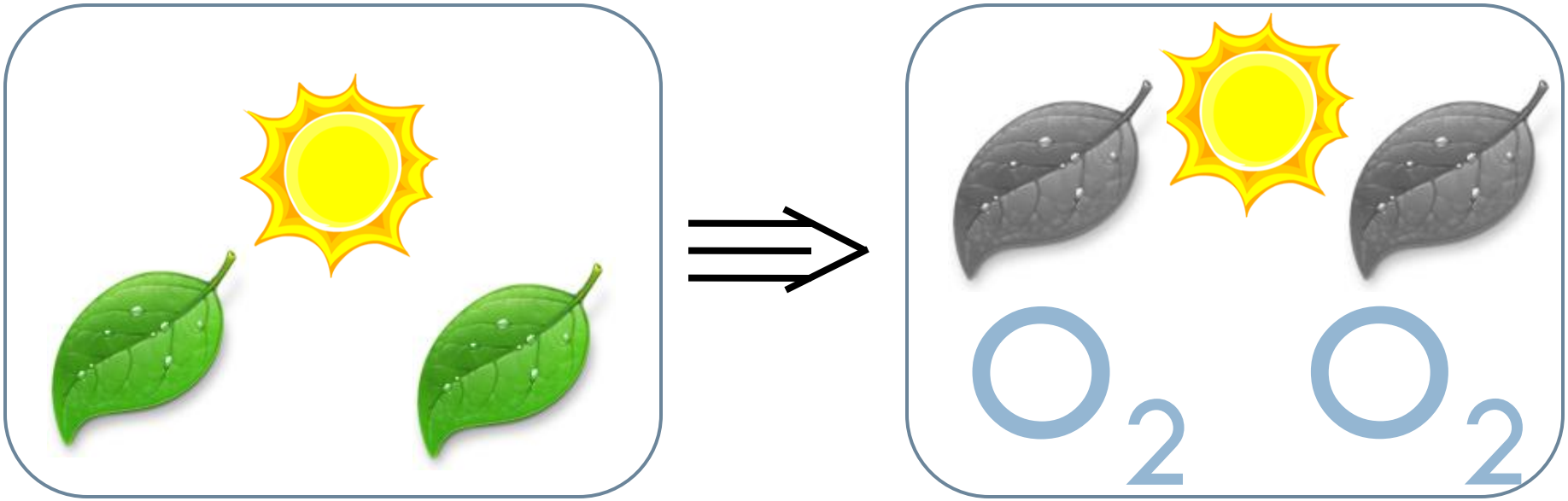
Why Explicit Data Sharing? Example: Resource Sharing



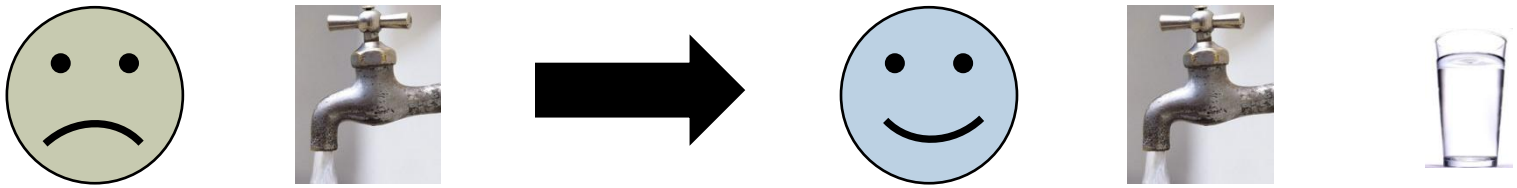
- We want photosynthesis to apply concurrently in spite of the fact that the sun is shared by all rule instances (that is, rules overlap!)

Why Explicit Data Sharing?
Example: Resource Sharing



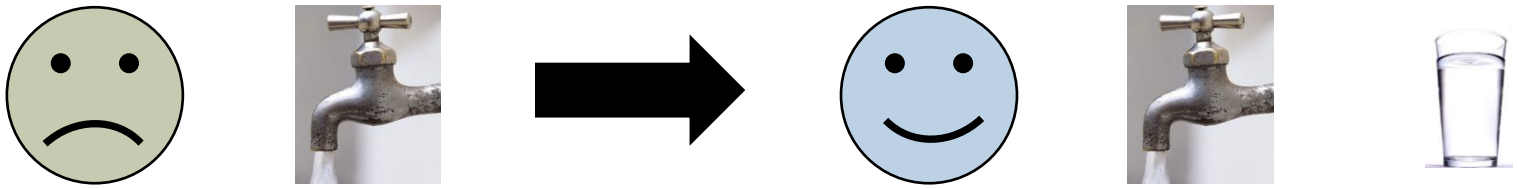


Why Explicit Data Sharing? Example: Mutual Exclusion

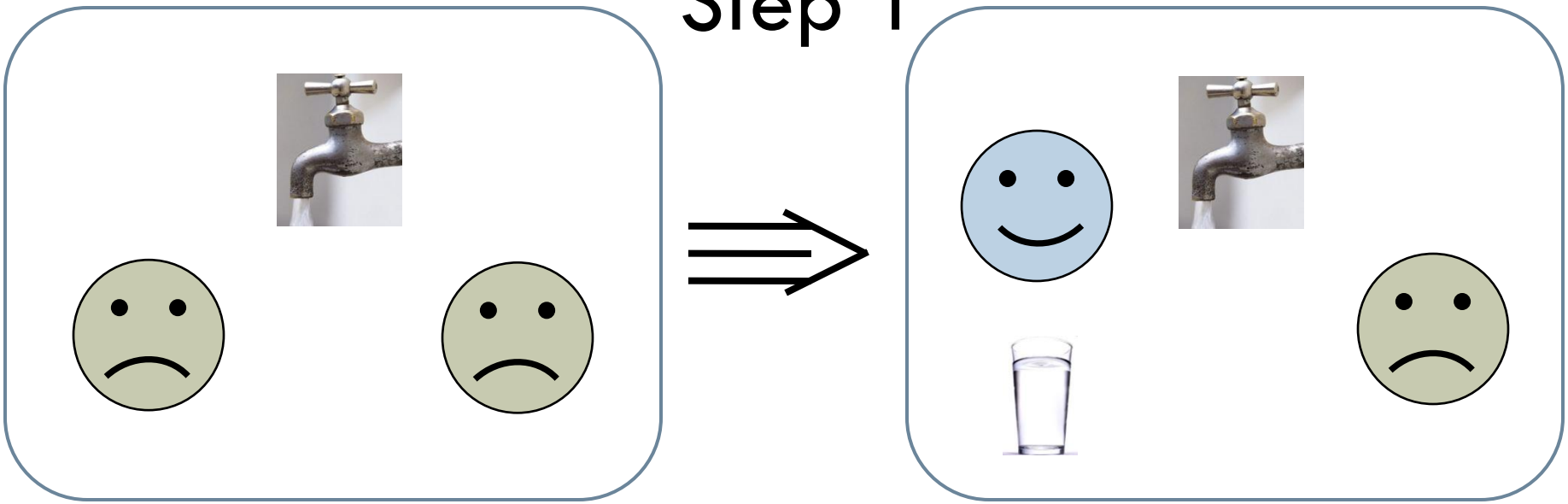


- Access to critical resource (water faucet here) cannot be concurrent, by design.
- Takes two steps to get two glasses of water, in spite of potential for concurrent execution

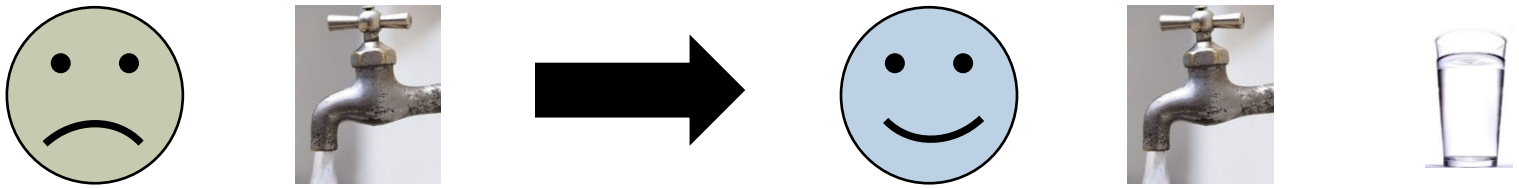
Why Explicit Data Sharing?
Example: Mutual Exclusion



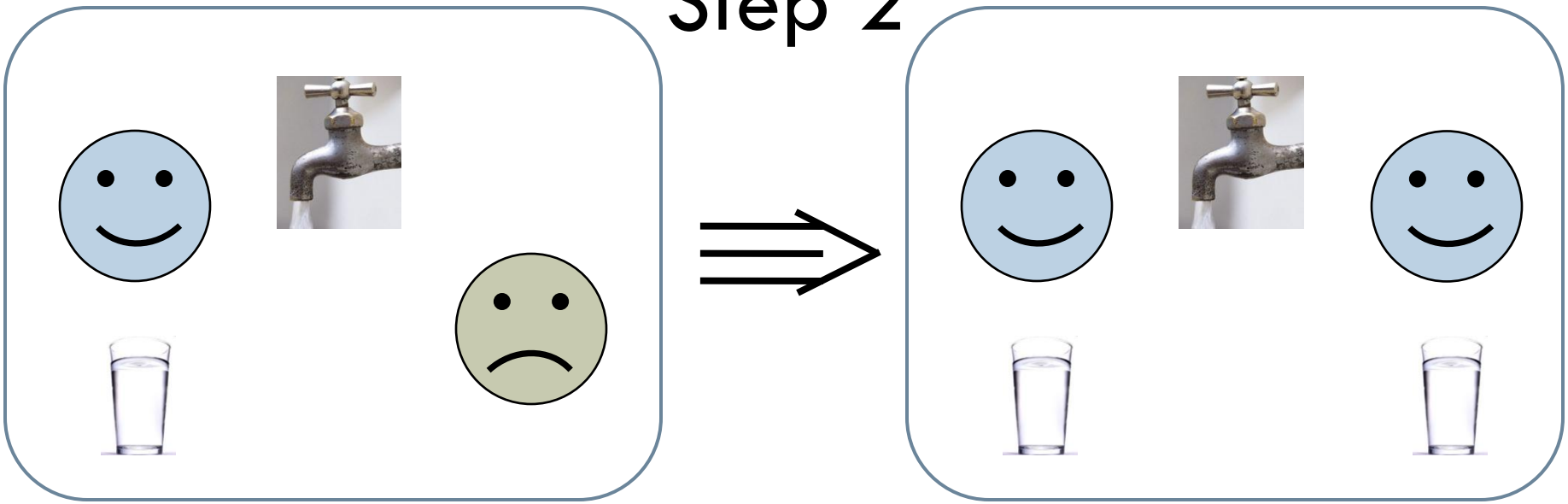
Step 1



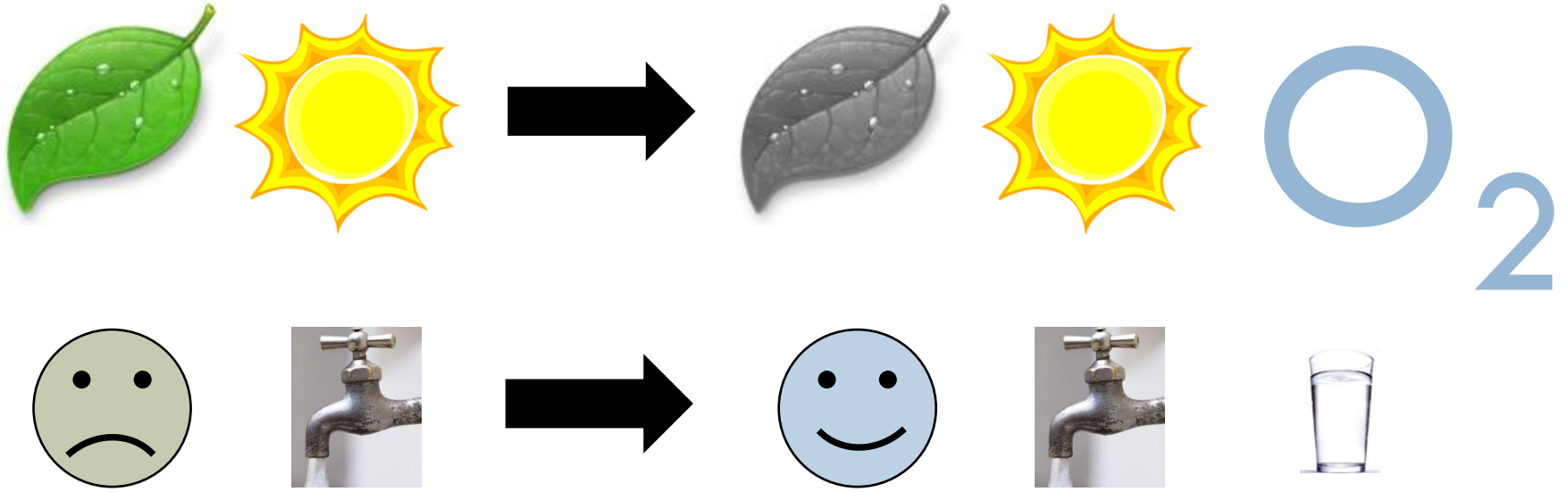
Why Explicit Data Sharing?
Example: Mutual Exclusion



Step 2

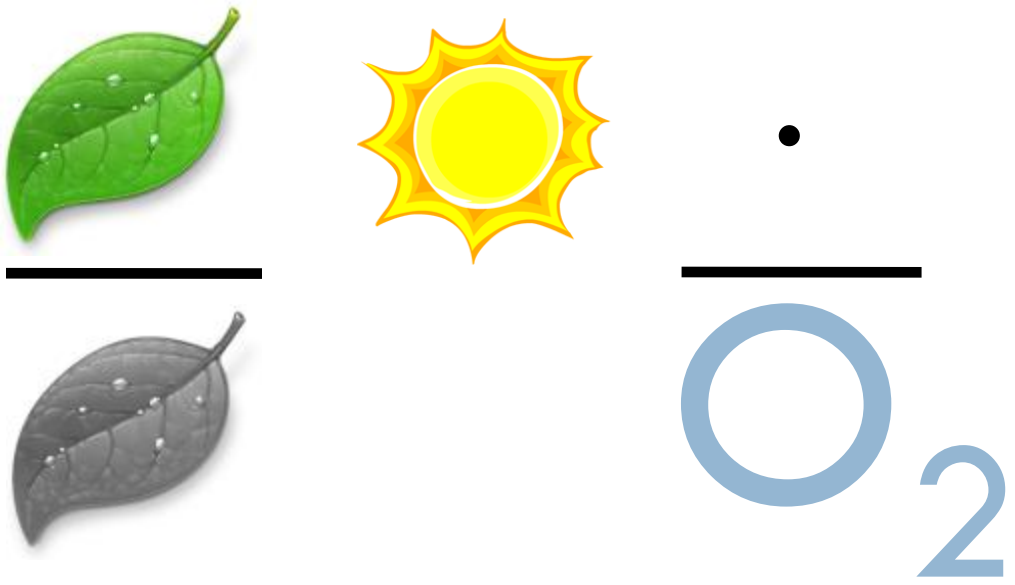


Conventional Rewrite Rules Are Not Expressive Enough for Concurrency



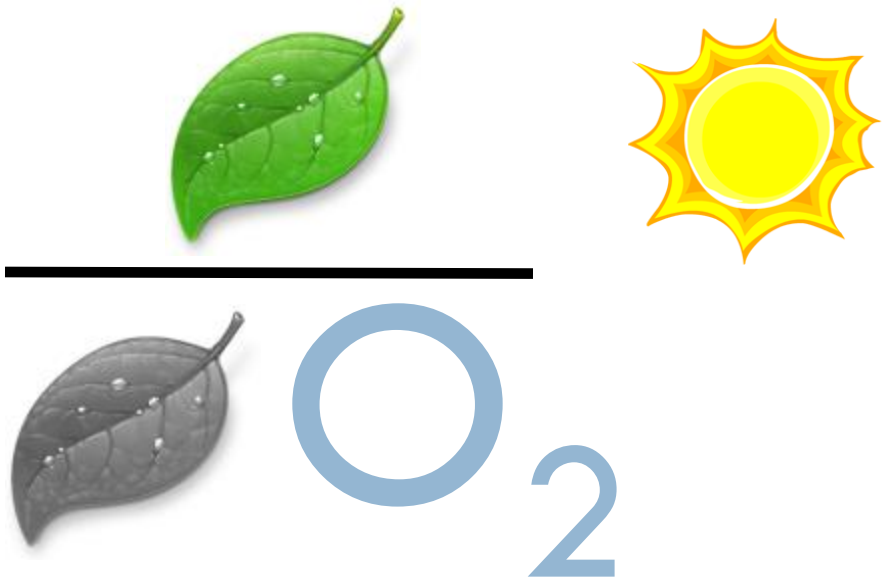
- As conventional rewrite rules, the two rules above are identical (leaf -> face, sun -> water, …)
- Yet, we want them to have totally different meaning wrt concurrency semantics!

Example of K Rule
Resource Sharing

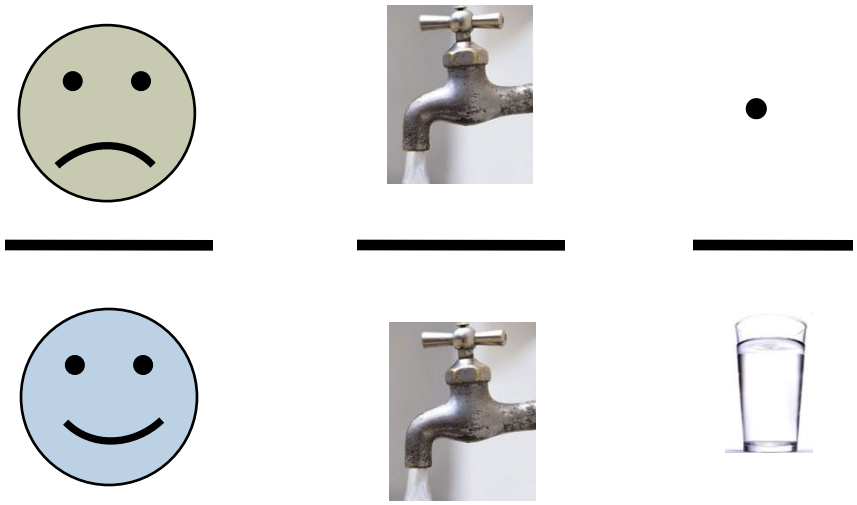


The dot "." is the unit of both bags and lists

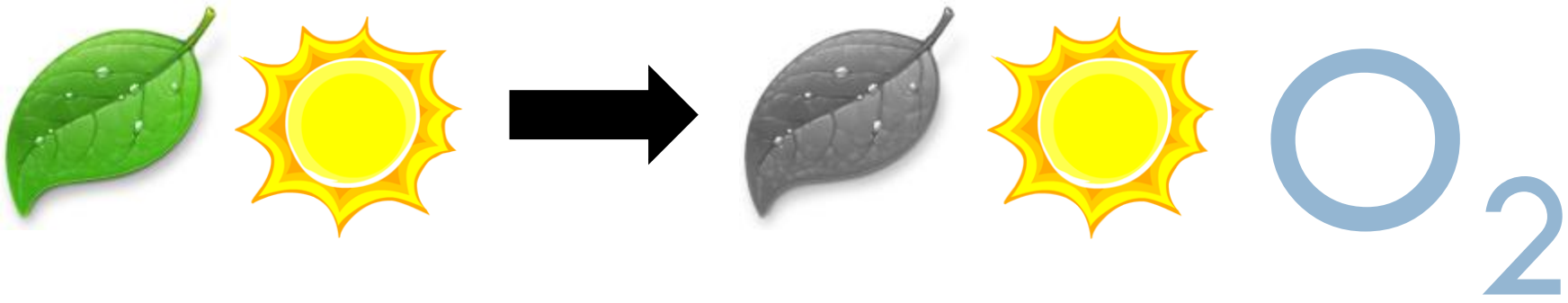
Example of K Rule
Resource Sharing – Alternative rule

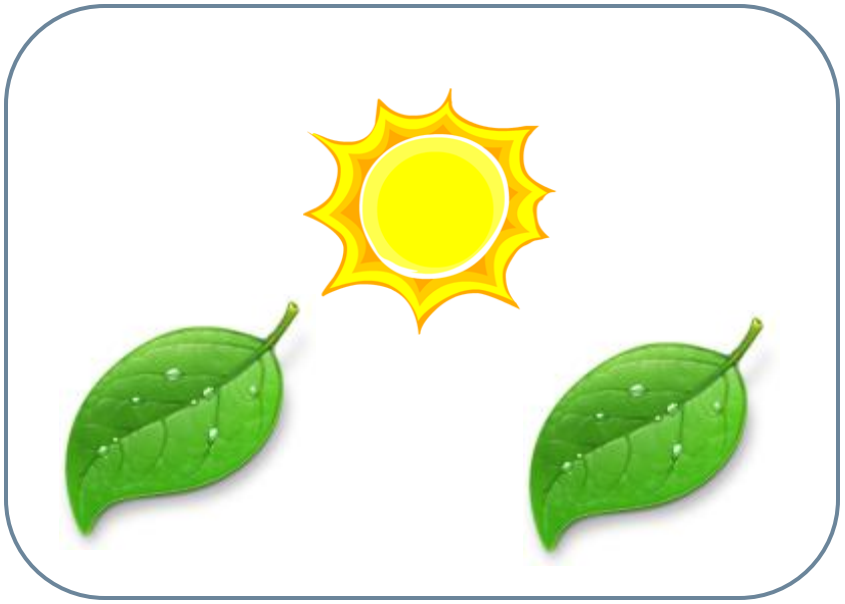


Example of K Rule
Mutual Exclusion



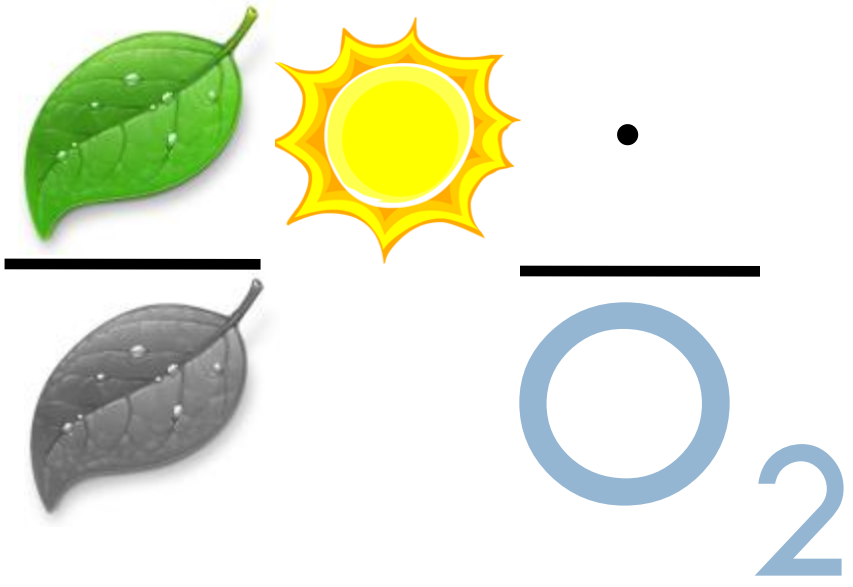
Rewriting Modulo … Insufficient





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



Desugared into a finite number of multiset equivalent rules

Special Support for Lists and Bags in K