KRAM—Extended Report

Traian Florin Șerbănuță and Grigore Roșu

University of Illinois at Urbana-Champaign
{tserban2,grosu}@illinois.edu

Abstract

Term rewriting proved to be a simple, uniform and powerful computational paradigm. Rewrite rules independently match and apply anywhere, unconstrained by the context. Rewriting is particularly appealing for defining truly concurrent systems, since rewrite rules can apply in parallel. Unfortunately, there is an inherent impediment in using term rewriting for defining concurrent systems or programming languages: overlapping rewrite rules *cannot* proceed concurrently. This limitation enforces an interleaving semantics in situations where one may not want it. For example, two threads accessing different regions of memory need to interleave since the corresponding rewrite rules overlap on the memory subterm. Or two message receiving operations of two distributed agents need to interleave since the corresponding rewrite rules overlap on the message pool subterm.

This paper presents the concurrent rewrite abstract machine (KRAM), a generalization of term rewriting in which rules explicitly state what can be concurrently shared with other rules, like in graph rewriting. A parallel rewrite relation is defined and proved sound, complete and serializable with respect to conventional rewriting. The KRAM serves as the computational infrastructure of $\mathbb{K}$, an executable semantic framework in which programming languages, calculi, as well as type systems or formal analysis tools can be defined, making use of configurations, computations and rules. A series of examples are discussed, including a non-trivial higher-order multi-threaded distributed language; all examples were defined and executed using the K-Maude tool.

1 Introduction

Consider rewriting the term $h(f(a),0,1)$ using the following canonical term rewrite system, where h is a ternary operation, g is binary, f is unary, 0, 1, a, b are constants, and x, y are variables:

(1) $h(x,y,1) \Rightarrow h(g(x,x),y,0)$
(2) $h(x,0,y) \Rightarrow h(x,1,y)$
(3) $a \Rightarrow b$
(4) $f(x) \Rightarrow x$

The term $h(f(a),0,1)$ has a unique normal form, $h(g(b,b),1,0)$, which can be reached in a minimum of 4 rewrite steps, e.g., $h(f(a),0,1) \Rightarrow h(a,0,1) \Rightarrow h(b,0,1) \Rightarrow h(b,1,1) \Rightarrow$

$h(g(b,b),1,0)$. In spite of the fact that all four rule instances above overlap on the term $h(f(a),0,1)$, the concurrent rewrite abstract machine (KRAM) in this paper can achieve the same result in *one concurrent rewrite step.* No other existing term rewriting approach can rewrite $h(f(a),0,1)$ to $h(g(b,b),1,0)$ in one concurrent step.

Let us first discuss intuitively how and why the four rules above can apply concurrently on $h(f(a),0,1)$. First, note that rule (1) modifies the first and the third arguments of h regardless of the second argument, while rule (2) modifies the second argument of h regardless of its first and third arguments. Therefore, rules (1) and (2) share (without changing it) the top operator h and yield complementary changes on the original term, so they can safely apply their changes in parallel on term $h(f(a),0,1)$. Moreover, note that none of these rules cares that x is specifically bound or points to $f(a)$, or what happens with $f(a)$ during their application. Therefore, we can rewrite the $f(a)$ that x points to in parallel with the application of rules (1) and (2). Using a similar argument, rules (3) and (4) can apply in parallel on $f(a)$ to rewrite it to b. Thus, rules (1), (2), (3) and (4) can in principle apply in one parallel rewrite step on $h(f(a),0,1)$ and produce $h(g(b,b),1,0)$.

To formalize the above intuition, the KRAM adopts the rewrite rules proposed by the $\mathbb{K}$ framework [20], which explicitly mention what part of the matched term is read/write and what part is read-only. This is achieved by underlining the read/write parts, and writing the changes underneath the line. For example, the rewrite rules (1) and (2) above become the following $\mathbb{K}$-rules:

$$(1)\ h(\underset{g(x,x)}{\underline{\quad x \quad}},\_,\underset{0}{\underline{1}}) \qquad\qquad (2)\ h(\_,\underset{1}{\underline{0}},\_)$$

The parts of the term which are not underlined are shared (read-only). Variables which are not reused in a rule (i.e., occur only once) play a purely structural role; they are called "anonymous variables" and are often replaced by a generic "_" variable (each occurrence of "_" stands for a distinct variable, like in Prolog). Conventional rewrite rules are special $\mathbb{K}$-rules, where the entire term gets rewritten; the standard notation $l \Rightarrow r$ is then allowed as syntactic sugar. In fact, the ASCII notation for $\mathbb{K}$-rules in K-Maude [21], our implementation of $\mathbb{K}$ onto Maude [5], conservatively extends that of standard rewrite rules in Maude; for example, the $\mathbb{K}$-rule (1) above is `h(x=>g(x,x),_,1=>0)` in K-Maude.

Two or more $\mathbb{K}$-rules can apply concurrently if and only if their read/write parts do not directly overlap and a special acyclicity condition holds. By direct overlapping we mean overlapping of proper subterms, i.e., subterms which do not correspond to rule variables; overlapping under rule variables is proved safe (Section 3) and thus allowed. For example, rules (1) and (2) above can apply concurrently, because the read/write parts of each act under the variables of the other. The acyclicity condition was initially unexpected; its necessity appeared while proving the serializability of KRAM concurrent rewriting (Theorem 3). Consider, for example, the term $f(g(a),h(b))$ to be rewritten using the following two $\mathbb{K}$-rules (f, g, h, a, and b, are operation symbols, while x and y are variables):

$$f(g(\underset{x}{\underline{a}}),x) \qquad\qquad f(y,h(\underset{y}{\underline{b}}))$$

A blind concurrent application of these two rules on $f(g(a),h(b))$ yields $f(g(h(b)),h(g(a)))$. However, this concurrent rewrite step is *non-serializable*, since there is no way to order the application of the two rules on $f(g(a),h(b))$ to obtain $f(g(h(b)),h(g(a)))$. While non-serializable concurrent rewriting may eventually be desirable for defining complex concurrent systems, in this paper we consider only serializable concurrent rewriting and give an acyclicity criterion that ensures it. This is formally defined in Section 3. Informally, we have a cyclic relationship which prevents the two rules from being applied concurrently on $f(g(a),h(b))$: a gets rewritten to $h(b)$, then b gets rewritten to $g(a)$, and so on.

The examples above were deliberately artificial, to explain the problem that we are attempting to solve and its subtleties using a minimal setting. In Section 2 we discuss several less artificial examples, such as concurrent sorting, concurrent Dijkstra's all shortest paths, a simple concurrent imperative language, call-by-value lambda calculus, a type checker for simply typed calculus, and an executable variant of π-calculus. In Section 7 we define and discuss a non-trivial higher-order, multi-threaded and distributed language, called AGENT. All examples discussed in this paper are executable using K-Maude [21] and can be downloaded from [1] together with several larger language definitions. The reason we include so many examples in this paper and refer the reader to more is because we believe that they help convey the idea that the proposed concurrent rewrite abstract machine, in spite of its conceptual simplicity, is actually quite practical.

Our technical contributions are all grouped in Section 3. The main idea underlying KRAM rewriting is to lift the problem to a problem of graph rewriting, then use graph rewriting to perform the concurrent step, and then recover a term, i.e. the result of the concurrent rewrite step, from the resulting graph. While lifting term rewriting to graph rewriting is not a new idea (several existing works are discussed in Section 3), previous efforts focused on doing so for efficiency reasons, to avoid repeating rewrites on identical subterms of the term to rewrite. Our main purpose for reducing the problem to graph rewriting is to "borrow concurrency" from a domain where the issue has been extensively researched. Unfortunately, due to the desired capability of $\mathbb{K}$-rules to explicitly state what is shared and to allow concurrent rewrites under variables, conventional notions of term graph representations could not be used unchanged in the lifting process. Also, as already mentioned, a novel and unexpected acyclicity condition was necessary in order to show that the resulting graph can be reinterpreted as a term and the obtained parallel graph rewriting, when reinterpreted as KRAM rewriting, is sound, complete and serializable for conventional term rewriting (Theorems 3 and 4).

2 Motivating Examples

The motivation for KRAM came from the use of rewriting logic in general [16] and of $\mathbb{K}$ in particular [20] as a semantic framework for programming languages, more specifically from observing that the lack of sharing information in conventional rewrite rules limits the potential for concurrency and thus cannot faithfully capture the intended concurrent semantics of the defined calculi or languages. $\mathbb{K}$ was proposed in

2003 and has been extensively used in teaching and research since then, including for defining real languages such as Java, C, Scheme and Verilog; we refer the reader interested in $\mathbb{K}$ to [20] and the references there.

Until now, the semantics of $\mathbb{K}$ was reduced to standard term rewriting, ignoring the sharing information in rules. This paper is the first to address the actual concurrent semantics of $\mathbb{K}$. Even though the current K-Maude implementation of the $\mathbb{K}$ framework does not fully support the actual concurrent KRAM rewriting, the main result of this paper (Theorem 4) ensures that K-Maude is sound and complete for KRAM wrt normal forms; the only thing lost is the concurrency available in KRAM. Since this paper is neither about $\mathbb{K}$ nor about K-Maude, we refrain from discussing these in depth, limiting ourselves to on-the-fly brief explanations needed to understand the context for the KRAM rules. The reader interested in executing the examples in this paper is referred to [1]. After executing an example, one can use the "`-latex`" option of the tool to convert the unidimensional ASCII notation into an easier to visualize bidimensional Latex notation. In the subsequent examples, all the defined modules were actually cut-and-paste from the Latex output of the tool.[1]

Concurrent sorting. The following K-Maude module sorts a list of integers. The imported builtin PL-INT module defines the syntactic category (or sort) *Int* as well as operations on integers, such as $>_{Int}$. The module K is imported by almost all definitions; it defines the syntactic category K, which should include all syntax (note the simple production "$K ::= Int$"), and provides common semantic infrastructure such as lists, sets, maps, cells, etc.:

MODULE SORT IMPORTS PL-INT+K
  $K ::= Int$
  INITIAL CONFIGURATION:
    $\langle \cdot_{List} \rangle_{\mathsf{sortme}}$
  K RULES:
    $\langle \_ \underline{x} \_ \underline{y} \_ \rangle_{\mathsf{sortme}} \quad \text{when } x >_{Int} y$
    $ \underline{y} \quad \underline{x}$
END MODULE

$\mathbb{K}$ definitions are typically based on *cells*, the same way the chemical abstract machine (CHAM) [3] is based on molecules. $\mathbb{K}$'s cells are labeled and written as $\langle data \rangle_{\mathsf{label}}$; algebraically, they are uninterpreted unary operations taking a data argument, where the data is typically (but not always) organized as a list, a set or a map. The units (or identities) of these structures are typically denoted by a central dot "$\cdot$"; to disambiguate, one may append the corresponding sort name to "$\cdot$", as we did within the cell "$\langle \cdot_{List} \rangle_{\mathsf{sortme}}$" above. Most $\mathbb{K}$ definitions define an *initial configuration*, which consists of a potentially nested structure of initialized cells; this structure is used to

[1] We did, though, manually adjust the generated Latex a little for pagination purposes. Also, all the examples at [1], including those discussed here, use a generic substitution defined using the reflective capabilities of $\mathbb{K}$; since reflection is rather intricate and outside the scope of this paper, the examples in this paper importing a module SUBSTITUTION have been adjusted to use a custom substitution instead of the generic one.

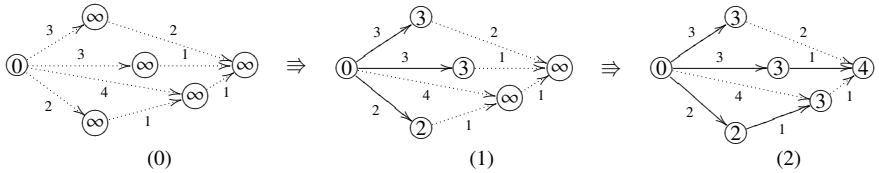

Figure 1: Dijkstra's all shortest paths derivation in two concurrent steps

compactly define all the cells, as well as to statically check and complete the subsequent $\mathbb{K}$-rules, as explained in detail in [20] and in part in Section 7. Rule completion is not needed for the simple examples in this section.

The module SORT above contains only one $\mathbb{K}$-rule, which states that any two unordered elements in the list cell can be swapped. The advantage of using KRAM here, as opposed to conventional term rewriting modulo associativity, is that multiple instances of this rule can apply concurrently, even ones whose two elements are interleaved. Let us show how one can use this rule to sort the list $3,8,5,7,4,1,2,6$ in three concurrent steps. We will mark how the numbers pair in the matching process by annotating the underline with indexed variables corresponding to each match. In the first concurrent step, the matching phase could mark for rewriting all positions, obtaining e.g. $\underline{3}_{x_1}, \underline{8}_{x_2}, \underline{5}_{x_3}, \underline{7}_{x_4}, \underline{4}_{y_3}, \underline{1}_{y_1}, \underline{2}_{y_2}, \underline{6}_{y_4}$. Upon applying the concurrent step, the list becomes $1,2,4,6,5,3,8,7$. In the second step, the matching phase can yield $1,2,\underline{4}_{x_1},6,5,\underline{3}_{y_1},\underline{8}_{x_2},\underline{7}_{y_2}$, inducing a second concurrent rewrite step, to $1,2,3,6,5,4,7,8$. Finally, there is only one possible rule instance left for matching, $1,2,3,\underline{6}_{x},5,\underline{4}_{y},7,8$, producing the sorted list.

Concurrent Dijkstra. The following module gives a one-rule KRAM for solving Dijkstra's all-shortest path problem:

MODULE DIJKSTRA IMPORTS PL-ID+NAT-INF
  $K ::= Id \mid Nat$
  $SetItem ::= Edge$
  $Edge ::= Id \xrightarrow{Nat} Id$
  INITIAL CONFIGURATION:
    $\langle \cdot Set \rangle_{\mathsf{graph}}\ \langle \cdot Map \rangle_{\mathsf{shortest}}$
  K RULES:
    $\langle x_1 \xrightarrow{w} x_2\ \_ \rangle_{\mathsf{graph}} \qquad \langle x_1 \mapsto c_1\ \_\ x_2 \mapsto \underline{\quad c_2 \quad} \rangle_{\mathsf{shortest}}$
    $w +_{Nat} c_1$
    when $w +_{Nat} c_1 <_{Nat} c_2$
END MODULE

The module PL-ID introduces identifiers as constants of sort *Id*, and NAT-INF introduces natural numbers with infinity. A graph is represented as a set of weighted edges $x_1 \xrightarrow{w} x_2$, saying that there is an edge from x_1 to x_2 of cost w. All shortest paths

are represented as a mapping, which is a set of bindings $x_i \mapsto c_i$; each such binding states that the shortest path from the root to node x_i is c_i. Common algebraic data-types, such as lists, sets and maps, are provided by the imported K module. By default, all these data-types include elements of sort *K*; if one wants more elements, then one has to add them explicitly. For example, we subsorted *Id* and *Nat* to *K*, so that we can have bindings of the form $Id \mapsto Nat$ in the $\langle\rangle_{\mathsf{shortest}}$ cell; also, we subsorted *Edge* to *SetItem*, so that we can have edges $x_1 \xrightarrow{w} x_2$ in the $\langle\rangle_{\mathsf{graph}}$ cell.

The initial term to rewrite should contain the graph in the $\langle\rangle_{\mathsf{graph}}$ cell, and a map mapping each node to ∞ in the $\langle\rangle_{\mathsf{shortest}}$ cell, except for the root node node, say a, which is mapped to 0. The rule above matches an edge $x_1 \xrightarrow{w} x_2$ in the graph, so that the current shortest path to x_2 is larger than the shortest path to x_1 plus w; if that is the case, the cost of the shortest path to x_2 is updated. We are only interested in the costs of the shortest paths here, not the shortest paths themselves. Those are easy to compute as well (e.g., storing x_1 next to the new cost of x_2), but we do not do it here. Note that everything is shared by the rule, except for the part it changes, the cost of x_2. Thus, many rules instances can apply in parallel, as far as they do not write the same shortest path costs. The graphical representation of a two-concurrent-step run of this KRAM is presented in Figure 1. Initially all graph edges are dotted while the nodes contain the initial minimal costs. As the rewriting proceeds, costs in the nodes are updated and the edges considered are depicted with full lines.

By Theorem 4, the concurrent rewrite steps produced by the KRAM above are serializable, so standard term rewriting analysis techniques can apply. For example, the corresponding term rewrite system terminates (the sum of the non-infinity shortest path costs decreases with the application of each rewrite rule) and is confluent (its critical pairs are joinable), so it admits unique normal forms. The normal forms give the shortest path costs, because any path computation can be mimicked with applications of the rule above. This may be one of the simplest implementations and proofs of correctness for Dijkstra's algorithm.

Multi-threaded IMP. The module in Figure 2 defines a simple multi-threaded imperative language, which is a fragment of the IMP++ language discussed in [20]. The language constructs in the categories *AExp* for arithmetic expressions, *BExp* for Boolean expressions, and *Stmt* for statements are self-explanatory; the statement construct `spawn(S)` spawns a new thread that executes `S` concurrently with the rest of the threads, all threads sharing the same state. Note that all syntactic categories are sunk into *K*, which is quite common in $\mathbb{K}$ definitions. Also, in this definition we need to explicitly define the results of computations (values in this case, but in general they can be anything, e.g., types when one defines a type checker); this is done by adding *Int* and *Bool* to the *KResult* category. *KResult* is defined as a subcategory of *K* in the imported builtin module K. *KResult* is not needed in "theoretical" definitions, but it is needed when one attempts to execute them using our tool, because of the way the strictness attributes (added in square brackets to the syntax) are implemented [21].

The strictness attributes are syntactic sugar for defining evaluation contexts. For example, addition is declared strict in its both arguments, while assignment is strict

MODULE MULTI-THREADED-IMP
IMPORTS K+PL-ID+PL-INT
$KResult ::= Int \mid Bool$
$K ::= AExp \mid BExp \mid Stmt$
$AExp ::= Id \mid Int \mid AExp + AExp$ [strict]
$BExp ::= Bool \mid AExp \leq AExp$ [strict]
$\mid$ `not` $BExp$ [strict]
$\mid BExp$ `and` $BExp$ [strict(1)]
$Stmt ::=$ `skip` $\mid Stmt$ `;` $Stmt$
$\mid Id$ `:=` $AExp$ [strict(2)]
$\mid$ `if` $BExp$ `then` $Stmt$ `else` $Stmt$ [strict(1)]
$\mid$ `while` $BExp$ `do` $Stmt$
$\mid$ `spawn(` $Stmt$ `)`
INITIAL CONFIGURATION:
$\langle \langle \cdot_K \rangle_{k*} \ \langle \cdot_{Map} \rangle_{state} \rangle_T$
K RULES:
$\langle \frac{X \ \_}{I} \rangle_k \ \langle \_ \ X \mapsto I \ \_ \rangle_{state}$
$I_1 + I_2 \Rightarrow I_1 +_{Int} I_2$
$I_1 \leq I_2 \Rightarrow I_1 \leq_{Int} I_2$
`not` $T \Rightarrow$ `not`$_{Bool}$ T
`true and` $B \Rightarrow B$
`false and` $\_ \Rightarrow$ `false`
`skip` $\Rightarrow \cdot$
$\langle \frac{X := I \ \_}{\cdot} \rangle_k \ \langle \_ \ X \mapsto \frac{\_}{I} \ \_ \rangle_{state}$
$S_1 \ ; \ S_2 \Rightarrow S_1 \curvearrowright S_2$
`if true then` S `else` $\_ \Rightarrow S$
`if false then` $\_$ `else` $S \Rightarrow S$
$\langle \frac{\texttt{while}\ B\ \texttt{do}\ S}{\texttt{if}\ B\ \texttt{then}\ S\ ;\ \texttt{while}\ B\ \texttt{do}\ S\ \texttt{else skip}} \rangle_k$
$\langle \frac{\texttt{spawn(}\ S\ \texttt{)} \ \_}{\cdot} \rangle_k \ \frac{\cdot}{\langle S \rangle_k}$
$\langle \cdot \rangle_k \Rightarrow \cdot$
END MODULE

Figure 2: $\mathbb{K}$ definition of multi-threaded IMP

only in its second argument. $\mathbb{K}$, and implicitly KRAM, are deliberately *context insensitive*, to increase their underlying potential for concurrency. Indeed, context sensitivity appears to inhibit concurrency because two rules changing each other's context cannot always safely proceed concurrently. $\mathbb{K}$ proposes a different mechanism to deal with evaluation contexts, in a context-insensitive manner. The sort K which extends the syntax is extended with a "task sequentialization" list structure, with constructor $\_ \curvearrowright \_$ (read "then"); for example, $k_1 \curvearrowright k_2$ means "first process k_1, *then* process k_2". With that, the strictness attributes of addition and assignment desugar into the following three bidirectional rules:

$$a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$$
$$a_1 + a_2 \rightleftharpoons a_2 \curvearrowright a_1 + \square$$
$$x := a \rightleftharpoons a \curvearrowright x := \square$$

The notation $\rightleftharpoons$ and terminology ("heating/cooling rules") is inspired from the CHAM [3]. Such rules allow us to "heat" (left-to-right application) a context by pulling redexes out and pushing them in front for processing, followed by "cooling" it down (right-to-left application) by plugging the result back into the context after processing. Such reversible rules are not directly executable, because they can lead to nontermination. For that reason, K-Maude translates them into left-to-right and right-to-left complementary variants, the former being applied when the redex is not a result and the later when the redex is a result. That is the reason for which we added *Int* and *Bool* to *KResult*. Like in CHAM, the heating/cooling rules are not meant to count as computational steps, but rather as structural rearrangements of the term so that computational rules apply. In fact, $\mathbb{K}$ allows two kinds of rules, structural and computational, the former including all the heating/cooling rules (possibly among others).

The configuration of this language has a top level cell $\langle\rangle_\top$ which holds inside a state cell (which holds a map) $\langle\rangle_{\mathsf{state}}$ and possibly multiple (as indicated by the "$*$" following the cell label) computation $\langle\rangle_{\mathsf{k}}$ cells; indeed, there will be one computation cell per thread. The heating rules will eventually sequentialize the computational task in each $\langle\rangle_{\mathsf{k}}$ cell, so that the redex where the next computational step can take place is at the top (or the left) of the cell $\langle\rangle_{\mathsf{k}}$. For that reason, many $\mathbb{K}$-rules match the top of the $\langle\rangle_{\mathsf{k}}$ cell. For example, the first $\mathbb{K}$-rule in Figure 2, which is the variable lookup rule, matches a program variable X at the top of a $\langle\rangle_{\mathsf{k}}$ cell and a binding $X \mapsto I$ in the $\langle\rangle_{\mathsf{state}}$ cell, and rewrites the variable X to its value I. When there is no ambiguity wrt the contents of a cell, one is not required to mention the particular list/set/map construct next to anonymous variables; for example, we did not mention the "$\curvearrowright$" construct between X and _ in the $\langle\rangle_{\mathsf{k}}$ cell. This is also supported by K-Maude.

The rules in Figure 2 are self-explanatory. `skip` and sequential composition are rewritten to the empty computation (identity of $\curvearrowright$) and the task sequentialization operation $\curvearrowright$. The rule for `spawn(S)` dissolves the spawning statement, at the same time adding a new computation cell with the spawned statement `S`. The last rule cleans up the configuration (removes empty cells).

The KRAM defined by the module in Figure 2 allows two or more threads to proceed concurrently, provided that they do not have a write-write or a read-write con-

flict on the same variable in the state. Indeed, if two threads lookup the same variable, then two different instances of the first $\mathbb{K}$-rule can be applied in parallel even though they share the $\langle\rangle_{\mathsf{state}}$ cell and the binding of the variable to its value. Also, two threads accessing (read or write) different variables can also proceed concurrently, because even though they share the $\langle\rangle_{\mathsf{state}}$ cell, each of the two rules applies its changes under the (anonymous) variables of the other. However, if two threads attempt to access the same location and at least one of the accesses is a write, then there is a conflict (the two positions of the two rules overlap) so the two corresponding rules cannot proceed concurrently; interleaving is enforced. This is the desired behavior of a (sequentially consistent) multi-threaded language and was our original motivation for KRAM.

Call-by-value λ-calculus. λ-abstraction and the (free) variables are results, and the application construct is strict in both arguments (call-by-value). β-reduction is only applied at the top of the computation, to inhibit reductions inside λ-abstractions:

MODULE LAMBDA IMPORTS SUBSTITUTION

$K ::= K\ K$ [strict]

$KResult ::= Id \mid \lambda Id.K$

K RULES:

$$\langle\ \frac{(\lambda X.E)\ E'}{E\,[\,E'\,/\,X\,]}\ \_\rangle_{\mathsf{k}}$$

END MODULE

No initial configuration was necessary in the above because the cell $\langle\rangle_{\mathsf{k}}$ is already defined in the module K, which is imported by SUBSTITUTION. As mentioned in Footnote 1, the substitution can be defined generically (for any binders, not only λ, but needs more notation) using the reflective capabilities of $\mathbb{K}$. For simplicity, in this paper we are assuming custom substitutions whenever needed.

The reader interested in $\mathbb{K}$ reflection is referred to [1], where one can also find several other variants of λ-calculus. Appendix D presents some generic reflection mechanisms, as they are written in K-Maude.

Type checker for simply-typed λ-calculus. One can give a disarmingly simple KRAM for type checking simply-typed λ-calculus:

MODULE SIMPLY-TYPED IMPORTS SUBSTITUTION

$KResult ::= Type$

$K ::= Id \mid \lambda Id{:}Type\,.\,K \mid K\ K$

$Type ::= \mathtt{type} \mid Type \rightarrow Type$

K RULES:

$(T \rightarrow T')\ T \Rightarrow T'$

$\lambda X{:}T\,.\,E \Rightarrow T \rightarrow E\,[\,T\,/\,X\,]$

END MODULE

Since *KResult* is a syntactic subcategory (or subsort) of K and since K-Maude inherits Maude's algebraic style allowing operations defined on subsorts to also be applied on supersorts, the KRAM above effectively allows mixing types and original syntax until, eventually, a result type is produced (assume the original expression closed). For simplicity we only considered one basic type, `type`, but one can easily add more as well as operations on them. For example, if one adds builtin integers by subsorting *Int* to K, then one needs to add an additional type `int` and a rule "$I \Rightarrow$`int`". The allowed mutilation of syntax and types may admittedly appear unorthodox at first sight, but note that the simple two rule KRAM above terminates and is confluent (critical pairs are joinable), so by virtue of Theorem 4 it indeed gives a correct type checker for simply-typed λ-calculus. Moreover, it may run in a sublinear number of concurrent rewrite steps.

Executable π-calculus. The K-Maude module below contains a $\mathbb{K}$ definition for a simple executable variant of the π-calculus.

MODULE EXECUTABLE-PI IMPORTS SUBSTITUTION

$$\begin{array}{lll} \mathit{Proc} ::= !\mathit{Proc} & & \mathit{Action} ::= \overline{\mathit{Id}}\langle \mathit{Id} \rangle \\ \quad | \ \mathit{Action}.\mathit{Proc} & & \quad | \ \mathit{Id}(\mathit{Id}) \\ \quad | \ \langle \mathsf{Bag}[\langle \mathsf{Bag}[\mathit{Proc}] \rangle_{\mathsf{sum}}] \rangle_{\mathsf{par}} & & \end{array}$$

K RULES:

$$\langle \frac{\_ \ \overline{C}\langle X \rangle.P}{P} \rangle_{\mathsf{sum}} \ \langle \frac{\_ \ C(Y).Q}{Q\,[\,X/\,Y\,]} \rangle_{\mathsf{sum}}$$

$$\langle \frac{\_ \ \overline{C}\langle X \rangle.P}{P} \rangle_{\mathsf{sum}} \ \langle \frac{\_ \ !C(Y).Q}{\cdot} \rangle_{\mathsf{sum}} \ \frac{\cdot}{\langle Q\,[\,X/\,Y\,] \rangle_{\mathsf{sum}}}$$

$$\langle R \ \langle Q \ (\nu X)P \rangle_{\mathsf{sum}} \rangle_{\mathsf{par}} \Rightarrow (\nu Y)\langle R \ \langle Q \ P\,[\,Y/\,X\,] \rangle_{\mathsf{sum}} \rangle_{\mathsf{par}} \quad \text{where Y is fresh}$$

$$\langle \langle P \rangle_{\mathsf{par}} \rangle_{\mathsf{sum}} \Rightarrow P$$

END MODULE

The syntax defined above is quite similar to the original syntax of the π-calculus [17]. However, similar to the approach using CHAM [4], to enhance the parallel communication inside processes, we use bags to represent the choice operator (the $\langle\rangle_{\mathsf{sum}}$ cell) and the parallel composition (the $\langle\rangle_{\mathsf{par}}$ cell). $\mathbb{K}$ provides a special syntactic category construct $\mathsf{Bag}[\mathrm{S}]$ for bags (or multisets) of elements of sort S (and similar ones for lists, sets, maps). Using that, note that processes can be built as controlled nested cells: a $\langle\rangle_{\mathsf{par}}$ cell holds a bag of $\langle\rangle_{\mathsf{sum}}$ cells, each containing a bag of processes to be chosen amongst. This convention does not alter the expressivity, since any of the cells could contain only one element. It is standard to assume guarded choice; we do it, too: each process in a $\langle\rangle_{\mathsf{sum}}$ cell must start with an action. The 0 process is represented by $\langle\langle\cdot\rangle_{\mathsf{sum}}\rangle_{\mathsf{par}}$. For executability, we follow Pict [18] and only allow replication for input expressions. According to [18], this does not limit the formal power of the calculus.

The KRAM above only contains four $\mathbb{K}$-rules. The first two are for communication: the first is standard (note that the non-communicating processes are discarded from the two $\langle\rangle_{\mathsf{sum}}$ cells), while the second defines replication triggered by input. The third rule defines scope extrusion by pushing the ν binder up. The fourth and final rule "releases" a bag of parallel-composed processes once they have reached the top of a sum cell.

3 KRAM Term Rewriting—Intuition

Rewriting logic (RL) [14] was introduced by Meseguer as a unified framework for concurrency. It generalizes both equational logic and term rewriting, and, more importantly, it organizes their combination into a formal logic, with complete deduction and initial models. Even though rewriting logic does not explicitly define a concurrent rewrite step relation, as we do in KRAM, it is in fact implicit in its deduction rules and it is not difficult to isolate it. Even though there are theoretical encodings of (limited) graph rewriting into rewriting logic [15], those encodings make intensive use of equations that need to be applied in both directions, so they are not practical. It is fair to say that rewriting logic was not conceived to deal with subterm sharing, or using subterm sharing to enhance concurrency. Consider again the four-rule KRAM discussed at the beginning of Section 1, this time naming the anonymous variables that appear in its $\mathbb{K}$-rules:

$$\text{(1) } h(\underline{\quad x \quad}_{g(x,x)}, y, \underline{1}_{0}) \qquad \text{(2) } h(x, \underline{0}_{1}, y) \qquad \text{(3) } a \to b \qquad \text{(4) } f(x) \to x$$

Flattening these $\mathbb{K}$-rules into RL rewrite rules, in RL one can apply either rules (1), (3), and (4), or rules (2), (3), and (4) concurrently on term $h(f(a),0,1)$, to obtain either $h(g(b,b),0,0)$ or $h(b,1,1)$, respectively. However, executing both rules (1) and (2) in parallel is impossible with the deduction rules of RL because "the same object cannot be shared by two simultaneous rewrites" [15], i.e., rule instances are not allowed to overlap. However, as seen in the concurrent IMP example in Section 2, being able to apply rules like (1) and (2) concurrently is crucial for faithfully capturing concurrent semantics of programming languages.

3.1 $\mathbb{K}$-Rules and KRAM

$\mathbb{K}$-rules describe how a term can be transformed into another term by altering some of its parts. They share the idea of match-and-replace of standard term rewriting; however, each $\mathbb{K}$-rule identifies a read-only pattern, the local context of the rule. This pattern is used to glue together read-write patterns, that is, subparts to be rewritten. Moreover, through its variables, it also provides information which can be used and shared by the read-write patterns. To some extent, the read-only pattern plays here the same role played by interfaces in graph rewriting [8].

To focus on the core of concurrent rewriting, in this section only we make the following three simplifying assumptions: (1) all $\mathbb{K}$-rules are unconditional; (2) all $\mathbb{K}$-rules are left linear; and (3) there are no lists, sets, bags, or maps involved. $\mathbb{K}$-rules

are typically unconditional and, when conditional, they have only very simple conditions anyway, which can be regarded as side conditions (as opposed to premises) that can be checked within their mathematical domain, without recursively invoking the KRAM rewriting. Also, (2) can be reduced to (1) by checking that two terms are equal; only syntactic equality is considered in $\mathbb{K}$-rules (as opposed to provability), which again can be checked easily without recursively invoking KRAM rewriting. (3) is the most subtle, because it may seem that one needs to extend KRAM to work "modulo" list or multiset axioms. However, that is not the case, because working modulo such axioms actually inhibits concurrency: indeed, having to restructure the term to rewrite in order for the rule to match is not only expensive, but may also be in conflict with other rules attempting to concurrently apply. We are adopting the approach proposed in [20], in which it is the rules that change in order to match and not the term to rewrite. Thus, we believe that our simplifying assumptions are acceptable.

A signature Σ is a pair (S, F) where S is a set of *sorts* and F is a set of operations $f : w \to s$, where f is an operation symbol, $w \in S^*$ is its arity, and $s \in S$ is its result sort. If w is the empty word ϵ then f is a constant. T_Σ is the universe of (ground) terms over Σ and $T_\Sigma(X)$ is that of Σ-terms with variables from the S-sorted set X. Given term $t \in T_\Sigma(X)$, let $vars(t)$ be the variables from X appearing in t. Given an ordered set of variables, $\mathcal{W} = \{\Box_1, \ldots, \Box_n\}$, named *context variables*, or *holes*, a *$\mathcal{W}$-context over* $\Sigma(X)$ (assume that $X \cap \mathcal{W} = \emptyset$) is a term $C \in T_\Sigma(X \cup \mathcal{W})$ in which each variable in $\mathcal{W}$ occurs once. The instantiation of a $\mathcal{W}$-context C with an n-tuple $\overline{t} = (t_1, \ldots, t_n)$, written $C[\overline{t}]$ or $C[t_1, \ldots, t_n]$, is the term $C[t_1/\Box_1, \ldots, t_n/\Box_n]$. One can regard $\overline{t}$ as a substitution $\overline{t} : \mathcal{W} \to T_\Sigma(X)$, defined by $\overline{t}(\Box_i) = t_i$, in which case $C[\overline{t}] = \overline{t}(C)$.

Definition 1. *A $\mathbb{K}$-**rule** $\rho : (\forall X)\ k[\, L \Rightarrow R\,]$ over a signature $\Sigma = (S, F)$ is a tuple (X, k, L, R), where:*

- *X is an S-sorted set, called the **variables** of the rule ρ;*
- *k is a $\mathcal{W}$-context over $\Sigma(X)$, called the **rule pattern**, where $\mathcal{W}$ are the **holes** of k; k can be thought of as the "read-only" part or the "local" context of ρ;*
- *$L, R : \mathcal{W} \to T_\Sigma(X)$ associate to each hole in $\mathcal{W}$ the **original term** and its **replacement term**, respectively; L, R can be thought of as the "read/write" part of ρ.*

We may write $(\forall X)\ k[\, \frac{l_1}{r_1}, \ldots, \frac{l_n}{r_n}\,]$ instead of $(\forall X)\ k[\, L \Rightarrow R\,]$ whenever $\mathcal{W} = \{\Box_1, \cdots, \Box_n\}$ and $L(\Box_i) = l_i$ and $R(\Box_i) = r_i$; this way, the holes are implicit and need not be mentioned.

*A set of K rules is called a $\mathbb{K}$-**system** or a **KRAM**.*

The variables in $\mathcal{W}$ are only used to formally identify the positions in k where rewriting takes place; in practice we typically use the compact notation above, that is, underline the to-be-rewritten subterms in place and write their replacement underneath. When the set of variables X is clear, it can be omitted.

Let us discuss how this definition captures the visual intuition by formally describing the rules from our running example at the beginning of this section. For

each of the four rules, the corresponding elements of a $\mathbb{K}$-rule are described in below:

	1	2	3	4
X	$\{x, y\}$	$\{x\}$	$\emptyset$	$\{x\}$
$\mathscr{W}$	$\{\Box_1, \Box_2\}$	$\{\Box\}$	$\{\Box\}$	$\{\Box\}$
p	$h(\Box_1, y, \Box_2)$	$h(x, \Box, y)$	$\Box$	$\Box$
L	$\Box_1 \mapsto x\,; \Box_2 \mapsto 1$	$\Box \mapsto 0$	$\Box \mapsto a$	$\Box \mapsto f(x)$
R	$\Box_1 \mapsto g(x, x)\,; \Box_2 \mapsto 0$	$\Box \mapsto 1$	$\Box \mapsto b$	$\Box \mapsto x$

Given a $\mathbb{K}$-rule $\rho : (\forall X)\ k[\,\mathrm{L} \Rightarrow \mathrm{R}\,]$, its associated 0-*sharing* $\mathbb{K}$-*rule* is $\rho_0 : (\forall X)\ \Box[\,\frac{\mathrm{L}(k)}{\mathrm{R}(k)}\,]$, that is a rule specifying the same transformation but without sharing anything. It is relatively easy to see that one can associate to any 0-sharing K rewrite rule a regular rewrite rule $(\forall X)\mathrm{L}(k) \Rightarrow \mathrm{R}(k)$. This is to account for the fact that, when applied in a non-concurrent fashion, K rules must obey the standard rewriting semantics. A $\mathbb{K}$ rule is *proper* if its read-only pattern k is a proper term.

In the subsequent sections we formalize KRAM term rewriting through an embedding into graph rewriting theory. The reasons for our choice are: (1) (term) graph rewriting [2, 10, 19] was shown to be sound and complete for term rewriting, which we want to preserve for KRAM; (2) the intuition that the pattern k of a $\mathbb{K}$-rule is meant to be "shared" with competing concurrent rule instances is conceptually captured by the notion of interface graphs of graph rewrite rules in the DPO (double-pushout) algebraic approach to graph rewriting [7, 8]; and (3) the results in the DPO theory of graph rewriting showing that if graph rule instances only overlap on the interface graphs, then they can be concurrently applied and the obtained rewrite step is serializable [9, 11, 13], which is also desirable semantics for KRAM.

However, although the theory of graph rewriting has early on shown the potential for parallelism with sharing of context, the existing term-graph rewriting approaches aim at efficiency: rewrite common subterms only once. More specifically, they do not attempt to use the context-sharing information for enhancing the potential for concurrency, as we want to do in KRAM. Consequently, the concurrency achieved by current term-graph rewriting approaches is no better than that of rewriting logic [14].

As our interests fall at the convergence of term-graph rewriting (for being sound and complete w.r.t. term rewriting) and the DPO approach to graph rewriting (for concurrency with sharing of context), the subsequent graph embedding of KRAM rewriting can be seen as an extension (enhancing the concurrency, while conserving soundness and completeness) of the jungle hypergraph rewriting [6, 10] incarnation of term-graph rewriting.

4 Background: Graphs and (Term-)Graph rewriting

Before formalizing our embedding of $\mathbb{K}$-rules into graph rules, we briefly recall the needed notions from the theory of graph grammars and graph transformations. We refer the interested reader to Corradini et al. [7] for a comprehensive survey of the

graph rewriting concepts used in this paper. As already mentioned, we prefer to use the double-pushout (DPO) approach [7].

4.1 Graph rewriting

Assuming fixed sets $\mathscr{L}_\mathrm{V}$ and $\mathscr{L}_\mathrm{E}$ for node and for edge labels, respectively, a *graph* G *over labels* $(\mathscr{L}_\mathrm{V}, \mathscr{L}_\mathrm{E})$ is a tuple $\mathrm{G} = \langle \mathrm{V}, \mathrm{E}, \mathrm{source}, \mathrm{target}, \mathrm{lv}, \mathrm{le} \rangle$, where V is the set of *vertices* (or *nodes*), E is a set of *edges*, $\mathrm{source}, \mathrm{target} : \mathrm{E} \to \mathrm{V}$ are the *source* and the *target* functions, and $\mathrm{lv} : \mathrm{V} \to \mathscr{L}_\mathrm{V}$ and $\mathrm{le} : \mathrm{E} \to \mathscr{L}_\mathrm{E}$ are the node and the edge *labeling functions*, respectively. We will use $\mathrm{V_G}$, $\mathrm{E_G}$, $\mathrm{source_G}$, ..., to refer to the corresponding components of the tuple describing a graph G. A *graph morphism* $f : \mathrm{G} \to \mathrm{G}'$ is a pair $f = \langle f_\mathrm{V} : \mathrm{V_G} \to \mathrm{V_{G'}}, f_\mathrm{E} : \mathrm{E_G} \to \mathrm{E_{G'}} \rangle$ of functions preserving sources, targets, and labels. Let $\mathbf{Graph}(\mathscr{L}_\mathbf{V}, \mathscr{L}_\mathbf{E})$ denote the category of graphs over labels $(\mathscr{L}_\mathrm{V}, \mathscr{L}_\mathrm{E})$. Given graph G, let $\prec_\mathrm{G} \subseteq \mathrm{V} \times \mathrm{V}$ be its *path relation*: $v_1 \prec_\mathrm{G} v_2$ iff there is a path from v_1 to v_2 in G. G is *cyclic* iff there is some $v \in \mathrm{V_G}$ s.t. $v \prec_\mathrm{G} v$. Given $v \in \mathrm{V_G}$, let $\mathrm{G}\restriction_v$ be the subgraph of G (forwardly) reachable from v.

A *graph rewrite rule* $p : (\mathrm{L} \xleftarrow{l} \mathrm{K} \xrightarrow{r} \mathrm{R})$, where p is its name, is a pair of graph morphisms $l : \mathrm{K} \to \mathrm{L}$ and $r : \mathrm{K} \to \mathrm{R}$, where l is injective. The graphs L, K, and R are called the *left-hand-side* (lhs), the *interface*, and the *right-hand-side* (rhs) of p, respectively.

The graph rewriting process can be intuitively described as follows. Given a graph G and a match of L into G satisfying some *gluing conditions* (discussed below), we can rewrite G into H in two steps: (1) delete from G the part from L that does not belong to K, obtaining a context C—the gluing conditions must ensure C is still a graph; (2) embed R into C by gluing it along the instance of K in C. Formally, given a graph G, a graph rule $p : (\mathrm{L} \xleftarrow{l} \mathrm{K} \xrightarrow{r} \mathrm{R})$, and a *match* $m : \mathrm{L} \to \mathrm{G}$, a *direct derivation from* G *to* H *using* p *(based on* m*)* exists iff the diagram below can be constructed,

$$\begin{array}{ccccc} \mathrm{L} & \xleftarrow{\ l\ } & \mathrm{K} & \xrightarrow{\ r\ } & \mathrm{R} \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle \overline{m}} & & \downarrow{\scriptstyle m^*} \\ \mathrm{G} & \xleftarrow{\ l^*\ } & \mathrm{C} & \xrightarrow{\ r^*\ } & \mathrm{H} \end{array}$$

where both squares are pushouts in the category of graphs. In this case, C is called the *context* graph, and we write $\mathrm{G} \overset{p,m}{\Longrightarrow} \mathrm{H}$ or $\mathrm{G} \overset{p}{\Rightarrow} \mathrm{H}$. As usual with pushouts, whenever l or r is an inclusion, the corresponding l^* or r^* can be chosen to also be an inclusion.

A direct derivation $\mathrm{G} \overset{p,m}{\Longrightarrow} \mathrm{H}$ exists iff the following *gluing conditions* hold [8]: *(Dangling condition)* no edge in $\mathrm{E_G} \setminus m_\mathrm{E}(\mathrm{E_L})$ is incident to any node in $m_\mathrm{V}(\mathrm{V_L} \setminus l_\mathrm{V}(\mathrm{V_K}))$; and *(Identification condition)* there are no $x, y \in \mathrm{V_L} \cup \mathrm{E_L}$ with $x \neq y$, $m(x) = m(y)$ and $x, y \notin l(\mathrm{V_K} \cup \mathrm{E_K})$. If it exists, H is unique up to graph isomorphism. The gluing conditions say that whenever a transformation deletes a node, it should also delete all its edges (dangling condition), and that a match is only allowed to identify elements coming from K (identification condition).

Given a family of graph-rewrite rules $p_i : (\mathrm{L}_i \xleftarrow{l_i} \mathrm{K}_i \xrightarrow{r_i} \mathrm{R}_i)$, $i = \overline{1,n}$, not necessarily distinct, their *composed graph-rewrite rule*, denoted as $p_1 + \cdots + p_n$, is a rule

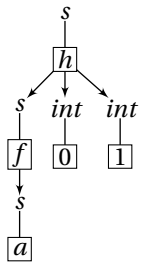

Figure 3: Jungle representation of $h(f(a),0,1)$

$p : (\mathrm{L} \xleftarrow{l} \mathrm{K} \xrightarrow{r} \mathrm{R})$ where L, K, and R are the direct sums of the corresponding components from $(p_i)_{i=\overline{1,n}}$ and, similarly, l and r are the canonical morphisms induced by $(l_i)_{i=\overline{1,n}}$ and $(r_i)_{i=\overline{1,n}}$, respectively. Given a graph G, matches $(m_i : \mathrm{L}_i \to \mathrm{G})_{i=\overline{1,n}}$ induce a combined match $m : \mathrm{L} \to \mathrm{G}$ defined as the unique arrow amalgamating all particular matches from the universality property of the direct sum. Matches $(m_i : \mathrm{L}_i \to \mathrm{G})_{i=\overline{1,n}}$ have the *parallel independence* property iff for all $1 \le i < j < n$, $m_i(\mathrm{L}_i) \cap m_j(\mathrm{L}_j) \subseteq m_i(\mathrm{K}_i) \cap m_j(\mathrm{K}_j)$. If $(m_i : \mathrm{L}_i \to \mathrm{G})_{i=\overline{1,n}}$ have the parallel independence property and each m_i satisfies the gluing conditions for rule p_i, then the combined match m satisfies the gluing conditions for the composed rule $p_1 + \cdots + p_n$, and thus there exists a graph H such that $\mathrm{G} \xRightarrow{p_1+\cdots+p_n,m} \mathrm{H}$. Moreover, this derivation is serializable, i.e., $\mathrm{G} \xRightarrow{p_1+\cdots+p_{n-1},m'} \mathrm{H}_{n-1} \xRightarrow{p_n} \mathrm{H}$, where m' is the composition of $(m_i)_{i=\overline{1,n-1}}$ [11, Theorem 7.3] (recasting prior results [9, 13]).

4.2 Jungle Evaluation

The jungle term-graph rewriting [10, 12] approach we build upon uses (directed) hypergraphs to encode terms and rules. A *hypergraph* $\mathrm{G} = (\mathrm{V}, \mathrm{E}, \mathrm{source}, \mathrm{target}, \mathrm{lv}, \mathrm{le})$ over labels $(\mathscr{L}_\mathrm{V}, \mathscr{L}_\mathrm{E})$ has basically the same structure as a graph; however, each edge is allowed to have (ordered) multiple sources and targets, that is, the source and target mappings now have as range V^*, the set of strings over V. For a hypergraph G, $\mathit{indegree}_\mathrm{G}(v)$/$\mathit{outdegree}_\mathrm{G}(v)$ denote the number of occurrences of a node v in the target/source strings of all edges in G.

Given a signature $\Sigma = (\mathrm{S}, \mathrm{F})$, a *jungle* is a hypergraph over (S, F) satisfying that: (1) each edge is compatible with its arity, i.e., for each $e \in \mathrm{E}$ such that $\mathrm{le}(e) = f : s_1 \ldots s_k \to s$, it must be that $\mathrm{lv}^*(\mathrm{source}(e)) = s$ and $\mathrm{lv}^*(\mathrm{target}(e)) = s_1 \ldots s_k$; (2) $\mathit{outdegree}(v) \le 1$ for any $v \in \mathrm{V}$, that is, each node can be the source of at most one edge; and (3) G is acyclic.

A jungle represents a term as an acyclic hypergraph whose nodes are labeled by sort names, and whose edges are labeled by names of operations in the signature; Figure 3 depicts the jungle representation of term $h(f(a),0,1)$. Constants are edges without any target. Variables are represented as nodes which are not sources of any edge. Let VAR_G denote the variables of G; we have that $\mathrm{VAR}_\mathrm{G} = \{v \in \mathrm{V}_\mathrm{G} \mid$

$outdegree_G(v) = 0\}$. Non-linear terms are represented by identifying the nodes corresponding to the same variable. There could be multiple possible representations of the same term as a jungle, as identical subterms can be identified (or not) in the jungle representation.

The term represented by some node in a jungle is obtained by descending along hyperedges and collecting the hyperedge labels. Let G be a jungle. Then

$$term_G(v) = \begin{cases} v & \text{if } v \in \mathrm{VAR}_G \\ \mathrm{le}(e)(term^*_G(\mathrm{target}(e)) & \text{otherwise, where } \{e\} = \mathrm{source}^{-1}(v) \end{cases}$$

A *root* of a jungle is a node v such that $indegree(v) = 0$. Let ROOT_G denote the set of roots of G. Given a term t (with variables), a *variable-collapsed* tree representing t is a jungle G with a single root $root_G$ and which only identifies variable nodes, that is, for all $v \in \mathrm{V}$, $indegree(v) > 1$ implies that $v \in \mathrm{VAR}_G$.

A term rewrite rule $left \to right$ is encoded as a jungle evaluation rule $\mathrm{L} \hookleftarrow \mathrm{K} \xrightarrow{r} \mathrm{R}$ in the following way:

L is a variable-collapsed tree corresponding to *left*.

K is obtained from L by removing the hyperedge corresponding to the top operation of *left* (that is, $\mathrm{source}^{-1}(root_L)$).

R is obtained from K as follows: if *right* is a variable (i.e., the rule is collapsing, then the $root_L$ is identified with *right*; otherwise, R is the disjoint union of K and a variable collapsing tree R′ corresponding to *right*, where $root_{R'}$ is identified with $root_L$ and each variable of R′ is identified with its counterpart from VAR_L.

- $\mathrm{L} \hookleftarrow \mathrm{K}$ and $\xrightarrow{r} \mathrm{R}$ are inclusions with the exception that r maps $root_L$ to *right* if *right* is a variable.

Jungle evaluation is done according to the DPO graph rewriting approach presented above (although here we are talking about hypergraphs, the results above carry through). In particular, note that the gluing conditions are satisfied for each matching morphism m in a jungle G. The dangling condition holds because $\mathrm{V_K} = \mathrm{V_L}$ (so there could be no dangling edge. The identification condition could be violated only if the edge e representing the top operation of l, i.e., $\{e\} = \mathrm{source}^{-1}(root_L)$ was identified with another edge e'. However, since the source of an edge is unique, this would lead to $m(root_L) = m(\mathrm{source}(e'))$, which contradicts with the fact that G (being a jungle) is acyclic. The (hyper)graph rewriting step obtained upon applying an evaluation rule to a jungle is called an *evaluation step*. Since $\mathrm{V_K} = \mathrm{V_L}$ we have that $\mathrm{V_C} = \mathrm{V_G}$, and thus for each $v \in \mathrm{V_G}$, its correspondent in H is $r^*(v)$.

Among the many interesting results relating term rewriting with jungle evaluation, we will build our results on the ones presented below.

Theorem 1. *Let p be an evaluation rule for a rewrite rule ρ, and let G be a jungle.*

1. *Evaluation steps preserve jungles, i.e., if $\mathrm{G} \overset{p}{\Rightarrow} \mathrm{H}$ then H is a jungle;*

2. *If* $G \overset{p}{\Rightarrow} H$, *then for each* $v \in V_G$ $term_G(v) \overset{\rho^n}{\Longrightarrow} term_H(r^*(v))$

3. *If* ρ *is left-linear and* $term_G(v) \overset{\rho}{\Rightarrow} t'$ *for some* $v \in V_G$, *then there exists* H *such that* $G \overset{p}{\Rightarrow} H$.

Proof. (1) and (2) are proved by Hoffmann and Plump [12][Theorems 5.4 and 5.5].
(3) follows from [12][Lemma 6.2] and [19][Theorem 4.8]. □

The induced rewriting relation

Theorem 1 actually shows that jungle evaluation is both *sound and complete* for one step of term rewriting using left linear rules.

Corollary 1. *If* G *is a variable-collapsed tree and* ρ *is left-linear, then* $term_G(root_G) \overset{\rho}{\Rightarrow} t'$ *iff there exists a jungle* G' *such that* $G \overset{p}{\Rightarrow} G'$ *and* $term_{G'}(r^*(root_G)) = t'$.

Indeed, let $\mathscr{R}$ be a left-linear rewrite system, and let $\overline{\mathscr{R}}$ be the system containing the evaluation rules corresponding to the rewrite rules in $\mathscr{R}$. Let $\Rightarrow^1_{\overline{\mathscr{R}}}$ be the relation defined on Σ-terms by $t \Rightarrow^1_{\overline{\mathscr{R}}} t'$ iff $G \overset{p}{\Rightarrow} H$, where G is a variable-collapsed tree, $term_G(root_G) = t$, $p \in \overline{\mathscr{R}}$, and $term_H(r^*(root_G)) = t'$. Then,

Corollary 2. $\Rightarrow^1_{\overline{\mathscr{R}}} = \Longrightarrow^1_{\mathscr{R}}$.

4.3 The bipartite graph representation of jungles

Although inspired from jungle evaluation, and relying of the results presented above, our graph rewriting approach for capturing KRAM rewriting will not use hypergraph jungles, but rather an extension of their equivalent (bipartite) graph representation.

We call a *graph representation of hypergraph* $G = (V_G, E_G, source_G, target_G, lv_G, le_G)$ *over labels* $(\mathscr{L}_V, \mathscr{L}_E)$ any graph isomorphic with the bipartite graph $G' = (V_{G'}, E_{G'}, source_{G'}, target_{G'}, lv_{G'}, le_{G'})$ over labels $(\mathscr{L}_V \cup \mathscr{L}_E, \mathit{Int})$, defined by

- $V_{G'} = V_G \cup E_G$;
- $E_{G'} = \bigcup_{e \in E_G} \{(e,i) \mid \text{ if } |source_G(e)| < i \leq 0 \text{ or } 0 < i \leq |target_G(e)|$;
- $source_{G'}((e,i)) = v$, if $i \leq 0$ and v is the $(-i+1)$th element in $source_G(e)$, and $source_{G'}((e,i)) = e$, if $i > 0$;
- $target_{G'}((e,i)) = e$, if $i \leq 0$, and $target_{G'}((e,i)) = v$, if $i > 0$ and v is the ith element of $target_G(e)$;
- $lv_{G'} = lv_G \cup le_G$;
- $le((e,i)) = i$

Conversely, to any bipartite labeled graph such that the two partitions V_1 and V_2 have labels in disjoint sets L_1 and L_2, respectively, we can associate a hypergraph over (L_1, L_2), or one over (L_2, L_1), respectively, depending whether V_1 are chosen to be nodes and V_2 edges, or the converse.

G is a *graph jungle over* Σ if it is the graph representation of some jungle over Σ. Graph jungles can be characterized as follows:

Proposition 1. *Given a signature* $\Sigma = (S, F)$*, a graph* G *over* $(S \cup F, Int)$ *is a graph jungle over* Σ *iff:*

0. G *is bipartite, partitions given by nodes with labels in* S*—**sort nodes**—, and* F*—**operation nodes**—;*

1. every operation node labeled by $f : s_1 \cdots s_n \to s$ *is*

   (i) the target of exactly one edge, labeled with 0 *and having its source labeled with* s*, and*

   (ii) the source of n *edges having distinct labels in* $\{1, \cdots, n\}$*, such that* $\mathrm{lv}(\mathrm{target}(e)) = s_{\mathrm{le}(e)}$ *for each such edge* e*;*

2. every sort node has at most one outward edge; and

3. G *is acyclic.*

For example, the bipartite graph representation of the jungle in Figure 3 (associated to the term $h(f(a), 0, 1)$) is represented by graph G in Figure 4. To avoid cluttering, and since there is no danger of confusion, we choose to omit the label 0 when representing the graphs.

The above definitions and results carry on, but must be adjusted to address the fact that hypergraph edges are translated into operation nodes in addition to the edges. Let us quickly revise the definitions and results.

The variables of a graph jungle are sort nodes without outward edges: $\mathrm{VAR_G} = \{v \in \mathrm{V_G} \mid \mathrm{lv}(v) \in \mathrm{S} \text{ and } \mathit{outdegree}_{\mathrm{G}}(v) = 0\}$. $\mathit{term}_{\mathrm{G}}(v)$ is defined on(ly) on sort nodes by:

$$\mathit{term}_{\mathrm{G}}(v_s) = \begin{cases} v_s, \text{ if } v_s \in \mathrm{VAR_G} \\ \sigma(t_1, \ldots, t_n), \text{ if } \{v_e\} = \mathrm{target}(\mathrm{source}^{-1}(v_s)), \mathrm{le}(v_e) = \sigma : s_1 \ldots s_n \to s, \text{ and} \\ \qquad t_i = \mathit{term}_{\mathrm{G}}(\mathrm{target}(e)) \text{ where } \mathrm{source}(e) = v_e \text{ and } \mathrm{le}(e) = i \end{cases}$$

The notions of root and variable-collapsing tree do not change. For the graph representation of evaluation rules, the only thing that needs to be adjusted is that if the rule is non-collapsing, then K is now obtained from L by removing the operation node linked to root of L and all its adjacent edges. Again, the gluing conditions are satisfied for any matching (graph) morphism into a graph jungle G. The argument for the identification condition carries on. The dangling condition is also satisfied by the fact that in any graph representing a jungle (including L and G), any operation node with label $\sigma : s_1 \ldots s_n \to s$ has exactly $n+1$ adjacent edges, which are all present in L. Evaluations steps being performed according to the DPO approach now in the context of concrete graphs, Theorem 1 can be recast as follows:

Theorem 2. *Let p be a graph evaluation rule for a rewrite rule ρ, and let G be a graph jungle.*

1. *Evaluation steps preserve graph jungles, i.e., if $G \overset{p}{\Rightarrow} H$ then H is a graph jungle;*
2. *If $G \overset{p}{\Rightarrow} H$, then for each $v \in V_G$ $term_G(v) \overset{\rho^n}{\Longrightarrow} term_H(r^*(v))$*
3. *If ρ is left-linear and $term_G(v) \overset{\rho}{\Rightarrow} t'$ for some $v \in V_G$, then there exists H such that $G \overset{p}{\Rightarrow} H$.*

5 KRAM graph rewriting

KRAM graph rewriting uses the same mechanisms and intuitions of jungle rewriting, but relaxes the definitions of both graph jungles and graph evaluation rules to increase the potential for concurrency in the case of context sharing.

The relaxation at the level of rules is that, similarly to the original definition of jungle rules [10], instead of practically removing the entire left-hand-side of an evaluation rule during the evaluation step (by sectioning the root of L from the rest in K), *KRAM graph rewrite rules* allow more of the local context (precisely, the k part of a $\mathbb{K}$ rule) to be preserved by a rule, and thus potentially allow other rules to share it for parallel rewriting. However, departing from the definition of jungle rules, we relax the requirement that the order between the nodes of K and variables of R should be the same as in L, to allow rules such as reading or writing the value of a variable from a store.

KRAM term-graphs are closely related to the graph jungles—they actually coincide for ground terms. The difference is that the KRAM term-graph representation allows certain variables (the anonymous and the pattern-hole variables) to be omitted from the graph. By reducing the number of nodes that need to be shared (i.e., by not forcing these variable nodes to be shared in the interface graph), this "partiality" allows terms at those positions to be concurrently rewritten by other rules.

5.1 KRAM Term-Graphs

The top-half of Figure 4 shows the KRAM term-graphs involved in the graph representations of the $\mathbb{K}$-rules (1)–(4) of our running example. For example the representation of variable x can be observed as the (singleton) graph R for rule (4), the constants a and b as graphs L and R from rule (3), and the term $f(x)$ as graph L in rule (4); all these KRAM term-graphs are also graph jungles. The bottom-half of Figure 4 shows the KRAM term-graphs involved in the graph transformation which uses all four rules combined to rewrite the graph representation of $h(f(a),0,1)$ (graph G) to one that can be used to retrieve $h(g(b,b),1,0)$ (graph H).

The novel aspect of our representation is that, unlike the graph jungles, the KRAM term-graphs are *partial*: they do not require each operation node to have outward edges for all sorts in its arity. This partiality plays a key role in "abstracting away" the

anonymous variables and the holes of the pattern. For example, the number of outward edges specified for the nodes labeled with h have all possible values between 3 (its normal arity) in graphs G and H, to 0, e.g., in graph K for rule (1). This flexibility is crucial for enhancing concurrency; only through it rules (1) and (2) can apply in parallel, as it allows the outward edge of h labeled with 1 to be rewritten by rule (1), while h is still shared with rule (2). This is achieved by relaxing the 1.(ii) property of Proposition 1 to allow partially specified operations. For self-containedness reasons, we write the entire definition, but follow the same structure as in Proposition 1.

Definition 2. *Given a signature* $\Sigma = (S, F)$*, a* ***KRAM*** Σ***-term-graph*** *is a graph* G *over labels* $(S \cup F, \{\epsilon\} \cup Nat)$ *satisfying the following:*

0. G *is bipartite, partitions given by nodes with labels in* S*—**sort nodes**—, and* F*—**operation nodes**—;*
1. *every operation node labeled by* $f : s_1 \cdots s_n \to s$ *is*
   (i) *the target of exactly one edge, labeled with* 0 *and having its source labeled with* s*, and*
   (ii) *the source of* ***at most*** n *edges having distinct labels in* $\{1, \cdots, n\}$*, such that* $\mathrm{lv}(\mathrm{target}(e)) = s_{\mathrm{le}(e)}$ *for each such edge* e*;*
2. *every sort node has at most one outward edge; and*
3. G *is acyclic.*

Let $\mathbf{KGraph}_\Sigma$ *denote the full subcategory of* $\mathbf{Graph}(\mathbf{S} \cup \mathbf{F}, \{\epsilon\} \cup Nat)$ *having KRAM* Σ*-term-graphs as objects.*

Note that any graph jungle is a KRAM term-graph. In the sequel, for notational simplicity KRAM term-graphs will be referred to as just term-graphs. Therefore, most of the definitions from graph jungles can be easily extended for term-graphs.

Given a set of anonymous variables $A \subseteq X$, an A-anonymizing variable-collapsed tree representing of a term $t \notin A$ with variables from X is obtained from a variable-collapsed tree representing t by removing the variable nodes corresponding to variables in A and their adjacent edges.

The root nodes of a term-graph G, ROOT_G are no different than for graph jungles; however, VAR_G now only captures the non-anonymous variables. To capture all variables, we need to additionally identify partially specified operation nodes.

Open, and variable nodes. Let G be a term-graph over $\Sigma = (S, F)$. The set OPEN_G of *open (or incomplete) operation nodes* of G, consists of the operation nodes whose outward edges are incompletely specified. Formally, $\mathrm{OPEN}_G = \{v \in \mathrm{lv}^{-1}(S) \mid |s^{-1}(v)| < \mathit{arity}(\mathrm{lv}(v))\}$. The set of *term variables* of G, TVARS_G consists from the variables of G and the positions of the unspecified outward edges for open operation nodes (which stand for anonymous variables). Formally, $\mathrm{TVARS}_G = \mathrm{VAR}_G \cup \{x_{v,i} \mid v \in \mathrm{OPEN}_G, 1 \leq i \leq \mathit{arity}(\mathrm{lv}(v)) \wedge i \notin \mathrm{le}(\mathrm{source}^{-1}(v))\}$.

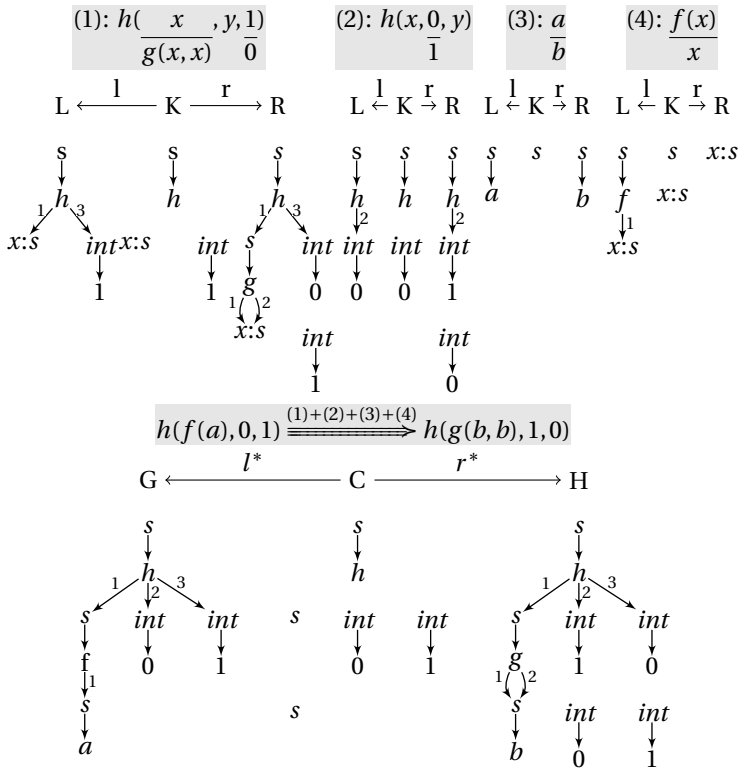

Figure 4: Graph representations for the $\mathbb{K}$ rules (1)–(4) from the motivating example and their concurrent application.

To account for the anonymous variables, the definition of *term* changes as follows:

$$term_{\mathrm{G}}(v_s) = \begin{cases} v_s, \text{ if } v_s \in \mathrm{VAR_G} \\ \sigma(t_1,\ldots,t_n), \text{ if } \{v_e\} = \mathrm{target}(\mathrm{source}^{-1}(v_s)), \mathrm{le}(v_e) = \sigma : s_1 \ldots s_n \to s, \text{ and} \\ t_i = subterm_{\mathrm{G}}(v_e, i) \text{ for any } 1 \le i \le n \end{cases}$$

where $subterm_{\mathrm{G}}$ is defined by on pairs of operation nodes with integers by

$$subterm_{\mathrm{G}}(v_e, i) = \begin{cases} x_{v_e,i}, \text{ if } x_{v_e,i} \in \mathrm{TVARS_G} \\ term_{\mathrm{G}}(\mathrm{target}(e)), \text{ if } \mathrm{source}(e) = v_e \text{ and } \mathrm{le}(e) = i \end{cases}$$

5.2 From $\mathbb{K}$-rules to graph rewrite rules

As we want KRAM graph rewriting to be a conservative extension of graph jungle evaluation, every 0-sharing $\mathbb{K}$ rule $(\forall \mathrm{X})\ \square[\ \frac{left}{right}\ ]$ is encoded as the graph jungle evaluation rule corresponding to the rewrite rule $left \to right$—see, for example the encodings of rules (3) and (4) in Figure 4. However, if the local context k is non-empty, then the rule is encoded so that the variable-collapsed tree representing k would not be modified by the rule. To be more precise, instead of obtaining K by removing the outgoing edge from the root of L, we will instead only remove the edges connecting the hole variables to their parent operations. Moreover, to further increase concurrency, the variables which appear in the read only pattern k but not in the left substitution are anonymized.

Let us discuss the representation of the $\mathbb{K}$-rule (1) in Figure 4, namely $h(\frac{x}{g(x,x)}, y, \underline{1})$. The left-hand-side is represented as a $\{y\}$-anonymized variable collapsed tree representing $h(x,y,1)$; variable y is anonymized as only appearing in the pattern k. The interface K is obtained from L by severing (through the removal of edges labeled by 1 and 3) the part of L representing the read-only pattern $h(\square_1, y, \square_2)$ (which is the $\{y, \square_1, \square_2\}$-anonymized variable collapsed tree representing $h(\square_1, y, \square_2)$) from the parts of L representing the left substitution (namely, x and 1). Thus, the l morphism from K to L is clearly an inclusion. R is obtained by taking the disjoint union between K and the variable-collapsed trees corresponding to terms $g(x,x)$ and 0 given by the right substitution, identifying the variables, and "gluing" them to the part representing the read-only pattern through edges from operation node h labeled 1 and 3, respectively. Similarly as for the l morphism, the morphism r can also be chosen to be an inclusion.

The graph rules in Figure 4 are obtained using the definition below. To avoid clutter, we do not depict node or edge names (except for variables). Also, the actual morphisms are not drawn (they are either inclusions or obvious collapsing morphisms).

Definition 3. *Let* $\rho : (\forall \mathrm{X})\ k[\ \mathrm{L} \Rightarrow \mathrm{R}\ ]$ *be a* $\mathbb{K}$ *rewrite rule.*

If ρ *is* 0*-sharing, then the* $\mathbb{K}$ *graph rewrite rules representing* ρ *coincide with the graph evaluation rules corresponding to the rewrite rule associated to* ρ.

Otherwise, a $\mathbb{K}$ ***graph rewrite rule*** *representing* ρ *is a graph rewrite rule* $(\mathrm{L}_\rho \xleftarrow{l_\rho} \mathrm{K}_\rho \xrightarrow{r_\rho} \mathrm{R}_\rho)$ *such that:*

L_ρ *is a* A*-anonymized variable collapsed tree representation of* $L(k)$*, where* $A = vars(k) \setminus vars(L)$ *are the anonymous variables of* ρ*;*

K_ρ. *Let* K_0 *be the subgraph of* L_ρ *which is a* A*-anonymized variable collapsed tree representing* k*; then* $K_\rho = (V_{K_\rho}, E_{K_\rho})$ *is given by* $V_{K_\rho} = V_{L_\rho}$ *and* $E_{K_\rho} = E_{L_\rho} \setminus \{e \in E_{L_\rho} \mid \text{source}(e) \in V_{K_0}$ *and* $\text{target}(e) \notin V_{K_0}\}$*.* l_ρ *is the inclusion morphism.*

R_ρ *Let* R_0 *be an* A*-anonymized variable collapsed tree representation of* $R(k)$ *containing* K_0 *as a subgraph. Then* R_ρ *is obtained as the pushout between the inclusions of* $K_0 \cup VAR_{R_0}$ *into* K_ρ *and* R_0*, respectively.*

The nodes from K_0 *will be called pattern nodes.*

Note that the edges removed from L_ρ to obtain K_ρ are those whose target corresponds to the hole variables of k.

Similarly to the graph jungle rules, the (basic) $\mathbb{K}$ graph rules defined above ensure that the gluing conditions are satisfied for any matching morphism. For the remainder of this section, let us fix G to be a term-graph, $\rho_i : (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$, $i = \overline{1,n}$ to be $\mathbb{K}$ graph-rewrite rules, and $m_i : L_i \to G$ to be parallel independent matches. Let $\rho : (L \xleftarrow{l} K \xrightarrow{r} R)$ be the composed rule of $\left(\rho_i\right)_{i=\overline{1,n}}$, and let $m : L \to G$ be the composition of the individual matches. It follows that m satisfies the gluing conditions for ρ, and thus (ρ, m) can be applied as a graph transformation. Let us now provide a concrete construction for the derivation of (ρ, m) in **Graph** which will be used in proving the subsequent results.

The pushout complement object of m and l can be defined in **Graph** as $C = G \setminus m(L \setminus K)$ where the difference is taken component-wise. That C is a graph is ensured by the gluing conditions. The standard construction of the pushout object H is to factor the disjoint union of C and R through the equivalence induced by the pushout morphism $\overline{m} : K \to C$ and r. We do this directly, by taking preference for elements in C, and thus choosing representatives from $\overline{m}(K)$ and by choosing as representatives variables for the equivalence classes induced by the parts of r belonging to collapsing rules.

Let us now state some facts given by the structure of $\mathbb{K}$ graph rewrite rules. Let $root_i$ identify the root of L_i in L. Since the lhs cannot be a variable, it follows that L_i has at least one edge and one operation node. K_i is a subgraph of L_i and l_i is the inclusion morphism; moreover K_i contains all nodes of L_i.

We have that $ROOT_L = \{root_i \mid i = \overline{1,n}\}$. Let now J be the set of indexes of collapsing rules. In the following, let i range over $\{1, \ldots, n\}$ and let j range over J.

We define H, together with $r^* : C \to H$ and $m^* : R \to H$, as follows:

- $V_H = (V_C \setminus \{m(root_j) \mid j \in J\}) \uplus (V_R \setminus V_K)$
- $r^*_V(v) = \begin{cases} v, & \text{if } v \neq m(root_j), \\ r^*_V(m(r(root_j))), & \text{if } v = m(root_j), \end{cases}$
- $m^*_V(v) = \begin{cases} v, & \text{if } v \notin V_K \\ r^*_V(m_V(v)), & \text{otherwise} \end{cases}$

- $\mathrm{E_H} = \mathrm{E_C} \uplus (\mathrm{E_R} \setminus \mathrm{E_K})$
- $r^*_{\mathrm{E}}(e) = e$ and $m^*_{\mathrm{E}}(e) = \begin{cases} e, \text{if } e \notin \mathrm{E_K} \\ m_{\mathrm{E}}(e), \text{otherwise} \end{cases}$
- $\mathrm{source_H}(e) = \begin{cases} r^*(\mathrm{source_C}(e)), \text{if } e \in \mathrm{E_C} \\ m^*_{\mathrm{V}}(\mathrm{source_R}(e)), \text{if } e \in \mathrm{E_R} \setminus \mathrm{E_K} \end{cases}$
- $\mathrm{target_H}(e) = \begin{cases} r^*_{\mathrm{V}}(\mathrm{target_C}(e)), \text{if } e \in \mathrm{E_C} \\ m^*_{\mathrm{V}}(\mathrm{target_R}(e)), \text{if } e \in \mathrm{E_R} \setminus \mathrm{E_K} \end{cases}$

Note that r^*_{V} is recursively defined. However, it is well defined, because G is acyclic and, since $r_{\mathrm{V}}(root_j) \in \mathrm{VAR_{L_j}}$, it must be that $\mathrm{G}\restriction_{m_{\mathrm{V}}(r_{\mathrm{V}}(root_j))}$ is a strict subgraph of $\mathrm{G}\restriction_{m_{\mathrm{V}}(root_j)}$, implying that the recursion should end because both G and J are finite. It can be easily verified that (H, r^*, m^*) is a pushout of $(\overline{m}, r)$.

Suppose G is a $\mathbb{K}$ graph representation of term t, i.e., that $\mathrm{ROOT_G} = \{root_{\mathrm{G}}\}$, $\mathrm{G} = \mathrm{G}\restriction_{root_{\mathrm{G}}}$, and $term_{\mathrm{G}}(root_{\mathrm{G}}) = t$. When applying a (composed, or not) $\mathbb{K}$ graph rewrite rule to graph G, $root_{\mathrm{G}}$ must be preserved in the context C, because K contains all nodes of L. Therefore, let us define the top of the obtained graph H as being $root_{\mathrm{H}} = r^*(root_{\mathrm{G}})$. Note that $root_{\mathrm{H}}$ might not be equal to $root_{\mathrm{G}}$, because $root_{\mathrm{G}}$ could be identified with a variable node by a collapsing rule; moreover, $root_{\mathrm{H}}$ might not be the only element of $\mathrm{ROOT_H}$, because of the potential "junk" left by the application of the rule. Nevertheless, the term $term_{\mathrm{H}}(root_{\mathrm{H}})$ would be the one to which $term_{\mathrm{G}}(root_{\mathrm{G}})$ was rewritten.

5.3 Applying $\mathbb{K}$ rules as graph rules

To show that $\mathbf{KGraph}_\Sigma$ admits similar constructions for (composed) $\mathbb{K}$ graph-rewrite rules as **Graph**, that is, that the graphs described above are in fact term-graphs, we need to strengthen the constraints on the matching morphisms.

Indeed, without further constraints, applying K graph rules on term-graphs can produce cyclic graphs. Take for example, the graph G in Figure 5(a), representing the term $f(h(b), h(b))$. Upon applying the K graph rule corresponding to $f(\underset{a}{\underline{x}}, h(\underset{x}{\underline{b}}))$, we obtain a cyclic graph depicted as graph H in Figure 5(b).

One could validly argue that this problem arose because graph G was not a tree; however, the example, depicted in Figure 5(b), shows that it is possible that after applying a composed $\mathbb{K}$ graph-rewrite rule on a completely non-collapsed term-graph using a match whose components satisfy the parallel independence property, the graph obtained (we are guaranteed to obtain one) may not be a term-graph. Consider $\mathbb{K}$-rules $f(g(\underset{x}{\underline{a}}), x)$ and $f(y, h(\underset{y}{\underline{b}}))$ discussed in Section 1, together with the term to rewrite $f(g(a), h(b))$. Upon formalizing terms as term-graphs and $\mathbb{K}$-rules as $\mathbb{K}$ graph rewrite rules, the result of applying the composed $\mathbb{K}$ graph rewrite rule on the graph representing $f(g(a), h(b))$ is the graph H in Figure 5(b), *which has a cycle* and thus it is not a term-graph.

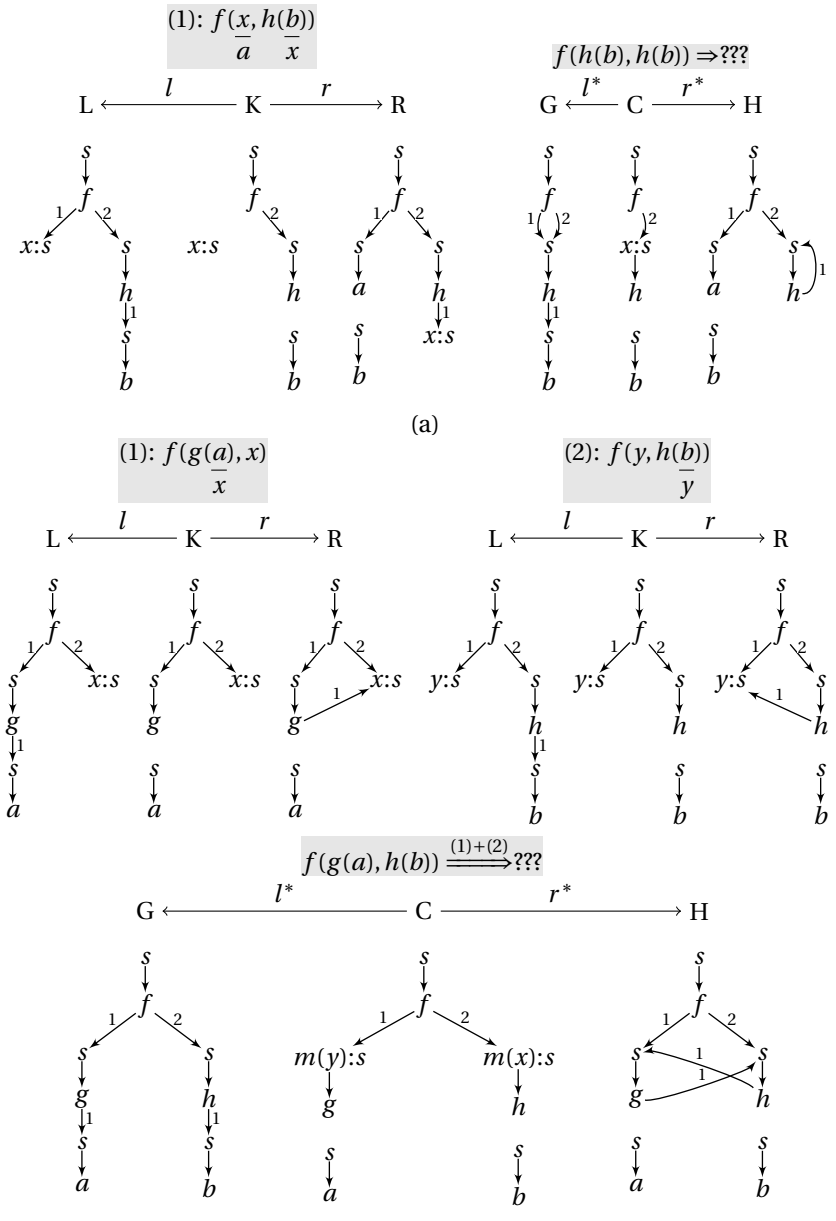

Figure 5: K graph rewriting can introduce cycles: (a) on a term-graph with sharing; (b) using parallel reductions.

The reason for the cycle being introduced in both examples from Figure 5 is that the matches overlap, allowing variable nodes to precede operation nodes in the path order of G, while r reorders the mapping of the variables to create a cycle. In jungle rewriting [10] this issue is prevented by imposing a statically checkable condition on the rules, namely that the path relation between the nodes preserved from L should not be changed by R. Formally, we say that a rule $\rho : (\mathrm{L} \xleftarrow{l} \mathrm{K} \xrightarrow{r} \mathrm{R})$ is *cycle free* if whenever $v \prec_\mathrm{R} x$ with $v \in \mathrm{V_K}$ and $x \in \mathrm{VAR_L} \cap \mathrm{V_K}$, it must be that $v \prec_\mathrm{L} x$. This condition is sufficient to prevent the introduction of cycles; however, we find it rather strong in our programming language context—in particular, this condition would disallow rules like the IMP rule for reading the value of a variable from the store (Figure 2). In what follows, we give a (semantical) condition on the matching morphism m rather than the rule which is sufficient to avoid the introduction of cycles.

Given a (composed) term-graph rewrite rule $\rho : (\mathrm{L} \xleftarrow{l} \mathrm{K} \xrightarrow{r} \mathrm{R})$, r induces on K a (partial) replacement order $\prec_r = r^{-1}(\prec_\mathrm{R})$, i.e., $v_1 \prec_r v_2$ in K iff $r(v_1) \prec_\mathrm{R} r(v_2)$ (there is a path from $r(v_1)$ to $r(v_2)$ in R). Moreover, given match m of p into G, m induces on K a (partial) matching order $\prec_m = l^{-1}(m^{-1}(\prec_\mathrm{G}))$, i.e., $v_1 \prec_r v_2$ in K iff $m(v_1) \prec_\mathrm{G} m(v_1)$ (l is an inclusion). Although both these (partial) orders are strict, their combination is not guaranteed to remain strict. We say that the match m is *cycle free* w.r.t. p if the transitive closure of $\prec_m \cup \prec_r$ is also a strict (partial) order.

Proposition 2. *(1) If any matching morphism for a* $\mathbb{K}$ *graph rewriting rule* ρ *is cycle free, then* ρ *is a jungle graph rewriting rule. (2) If* ρ *is a* $\mathbb{K}$ *graph rule,* G *is a term-graph,* $\mathrm{G} \overset{(\rho,m)}{\Longrightarrow} \mathrm{H}$*, and* m *is cycle free w.r.t.* ρ*, then* H *is acyclic.*

Proof. Let $\rho : (\mathrm{L} \xleftarrow{l} \mathrm{K} \xrightarrow{r} \mathrm{R})$ be a $\mathbb{K}$ graph rewriting rule.

(1) Suppose that there exist $v \in \mathrm{V_K}$ and $x \in \mathrm{VAR_L}$ such that $v \prec_\mathrm{R} x$ and $v \not\prec_\mathrm{L} x$. Let then G be the graph obtained from L by adding an edge e such that $\mathrm{source}(e) = x$ and $\mathrm{target}(e) = v$. G is still acyclic, because L is acyclic and because $v \not\prec_\mathrm{L} x$. Let $m : \mathrm{L} \to \mathrm{G}$ be the inclusion morphism. We have that m is not cycle free, since $v \prec_\mathrm{R} x$ implies that $v \prec_r x$ and $x \prec_\mathrm{G} v$ implies that $x \prec_m v$, contradiction.

(2) Proof by contradiction. Assume that H is not acyclic, and let $e_0, \ldots, e_n$ be a sequence of edges in H exhibiting a cycle. Let then $e_{\alpha_0}, \ldots, e_{\alpha_m}$ be a subsequence of the above sequence with the property that all its elements are edges in C and that the blocks of edges between them (including the one starting at e_{α_m} and wrapping over to e_{α_0} are alternating between C and $\mathrm{R} \setminus \mathrm{K}$. Then, if the edges between e_{α_i} and $e_{\alpha_{i+1}}$ are all in C, it must be that both $\mathrm{source}(e_{\alpha_i})$ and $\mathrm{target}(e_{\alpha_{i+1}})$ are in $\mathrm{V_K}$, and moreover, that $\mathrm{source}(e_{\alpha_i}) \prec_m \mathrm{target}(e_{\alpha_{i+1}})$. Similarly, if the edges between e_{α_i} and $e_{\alpha_{i+1}}$ are in $\mathrm{R} \setminus \mathrm{K}$, then both $\mathrm{target}(e_{\alpha_i})$ and $\mathrm{source}(e_{\alpha_{i+1}})$ are in $\mathrm{V_K}$ (which we already knew from the previous sentence) and that $\mathrm{target}(e_{\alpha_i}) \prec_r \mathrm{source}(e_{\alpha_{i+1}})$. But this precisely implies that m is not cycle free, contradiction. □

One might be tempted to think that if a morphism is not cycle free, then the resulting graph is bound to be cyclic. However, this is not the case, because when applying a composed rule, the composing rules not involved in the cyclicity condition might break the cycle, and thus produce a valid term-graph, as exhibited by the example in Figure 6. The graph to be rewritten is a representation of term

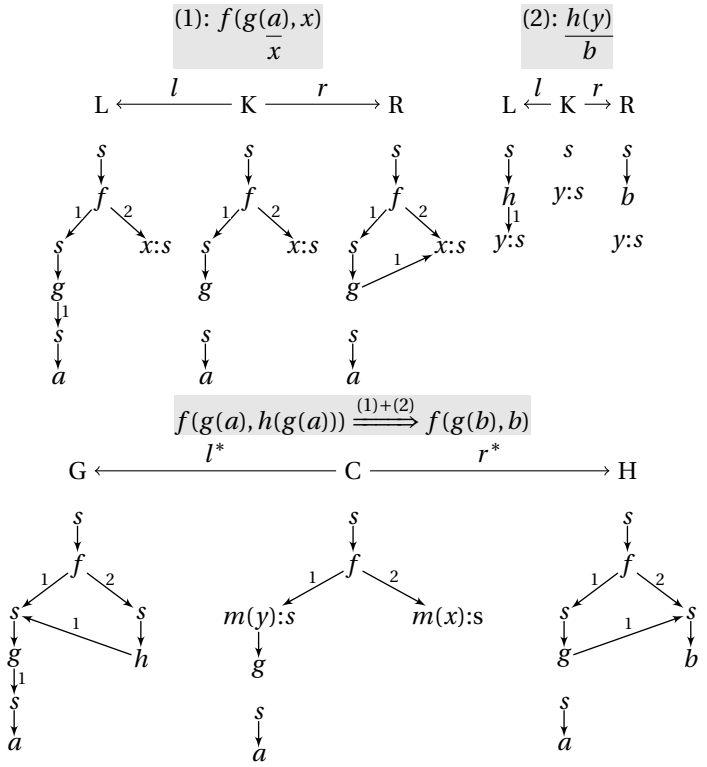

Figure 6: Non-cycle-free match producing non-cycling result

$f(g(a),h(g(a)))$ in which the two occurrences of the subterm $g(a)$ have been identified. The match of the composed rule (1) + (2) is not cycle-free because of rule (1). Moreover, if we would only apply rule (1), its application would lead to a cycle. However, when applying both rules together, the cycle is broken, and the resulting graph is indeed corresponding to the term $f(g(b),b)$ which can be obtained from the original term by regular term rewriting. Nevertheless, if the original graph is a tree, then cycle freeness of the matching morphism characterizes acyclicity of the resulting graph.

Proposition 3. *Let* G *be a tree term-graph.*

1. *If* ρ *is a simple* $\mathbb{K}$ *graph rule and m is a match for* ρ *into* G*, then m is cycle free.*
2. *If* ρ *is a composed* $\mathbb{K}$ *graph rule and* $G \overset{(\rho,m)}{\Longrightarrow} H$*, then* H *is acyclic iff m is cycle free w.r.t.* ρ.

Proof. Observation 1: Since G is a tree, $v_1 \prec_G v$ and $v_2 \prec_G v$ implies that either $v_1 \prec_G v_2$ or $v_2 \prec_G v_1$.

Observation 2: Assuming m is not cycle free, since both $\prec_m$ and $\prec_r$ are acyclic, it must be that the cycle is obtained by an alternating sequence $v_1 \prec_r x_1 \prec_m v_2 \ldots \prec_r x_{n-1} \prec_m v_n = v_1$, where x_i is a variable node and v_i is a pattern node for all $1 \leq i < n$.

(1) Let us show that is impossible to have $x \prec_m v$ where x is a variable node and v is a pattern node, whence m must be cycle free. Indeed, $x \prec_m v$ means that $m(x) \prec_{\mathrm{G}} m(v)$, which would lead to $x \prec_{\mathrm{L}} v$ (since $m(\mathrm{L})$ is a subtree of G), which is not possible, as x is a leaf in L.

(2) We only need to prove that if m is not cycle-free, then H has cycles, as the converse was proven in the general case by Proposition 2. Assume m is not cycle free, and consider a minimal sequence exhibiting a cycle as in Observation 2. We want to show that this sequence is also valid if we replace $\prec_m$ with $\prec_{\overline{m}} = \overline{m}^{-1}(\prec_{\mathrm{C}})$, which would necessarily lead to a cycle in H, as H is obtained as the pushout between C and R identifying K. We again reason by contradiction and assume that this is not the case, that is, there exists $1 \leq i < n$ such that $x_i \prec_m v_{i+1}$ but $x_i \not\prec_{\overline{m}} v_{i+1}$. However, this can only happen if an edge between $m(x_i)$ and $m(v_{i+1})$ in G is removed by another rule. Therefore, there must exist a pattern node v and a variable node x such that $x_i \prec_m v$, $v \prec_{\mathrm{L}} x$, $x \prec_m v_{i+1}$, and $v \not\prec_{\mathrm{K}} x$. From $v_{i+1} \prec_r x_{i+1}$ we deduce that $v_{i+1} \prec_r x_{i+1}$ are part of the same rule, and therefore there must be some $v' \in \mathrm{K}$ such that $v' \prec_{\mathrm{L}} v_{i+1}$ and $v' \prec_{\mathrm{L}} x_{i+1}$. Using Observation 1, $m(v) \prec_{\mathrm{G}} m(x) \prec_{\mathrm{G}} m(v_{i+1})$ and $m(v') \prec_{\mathrm{G}} m(v_{i+1})$ implies that either $m(v) \prec_{\mathrm{G}} m(v')$ or $m(v') \prec_{\mathrm{G}} m(v)$. Using the parallel independence condition we deduce that $m(v) \prec_{\mathrm{G}} m(v')$, whence $x_i \prec_m v \prec_m v' \prec_m x_{i+1} \prec_m v_{i+2}$. However, $x_i \prec_m v_{i+2}$ is in contradiction with our original assumption that the cycle was minimal. □

Next result shows that, under cycle-freeness conditions, $\mathbf{KGraph}_\Sigma$ is closed under (parallel) derivations using $\mathbb{K}$ graph rewrite rules.

Theorem 3. *Let* G, $(\rho_i)_{i=\overline{1,n}}$, $(m_i)_{i=\overline{1,n}}$, ρ, m, C, *and* H *be defined as above. If m is cycle-free w.r.t. p then the following hold:*

(Parallel) Derivation: $\mathrm{G} \overset{\rho,m}{\underset{\mathbf{KGraph}_\Sigma}{\Longrightarrow}} \mathrm{H}$;

Serialization: *There exist* $(\mathrm{G}_i)_{i=\overline{0,n}}$ *such that* $\mathrm{G}_0 = \mathrm{G}$, $\mathrm{G}_n = \mathrm{H}$, *and* $\mathrm{G}_{i-1} \overset{\rho_i}{\underset{\mathbf{KGraph}_\Sigma}{\Longrightarrow}} \mathrm{G}_i$ *for each* $1 \leq i \leq n$.

Proof. From the parallel independence condition, there exists a derivation $\mathrm{G} \overset{\rho,m}{\Longrightarrow} \mathrm{H}$ in **Graph**, and, H must be acyclic (Proposition 2). To prove the Derivation claim we only need to show that the graphs produced by the derivation, C and H, are indeed term-graphs.

Assuming that we have proved the Derivation claim, we can use the serializability result for the category of graphs iteratively, the first step being the following: From $\mathrm{G} \overset{p_1+\cdots+p_n,m}{\Longrightarrow} \mathrm{H}$ we deduce that $\mathrm{G} \overset{p_1+\cdots+p_{n-1},m'}{\Longrightarrow} \mathrm{H}' \overset{p_n}{\Longrightarrow} \mathrm{H}$, where m' is the composition of $(m_i)_{i=\overline{1,n-1}}$; however, by the derivation claim, H′ is also a term-graph, and, therefore, we can iterate to obtain the serialization result in $\mathbf{KGraph}_\Sigma$.

To prove the derivation part of the theorem, we only need to show that the graphs C and H defined above are term-graphs. First, let us show that C is a term-graph.

Conditions (0)—C is bipartite, (1.ii) at most n consistently labeled outward edges for each operation node, (2)—at most one outward edge for each sort node, and (3)—C is acyclic are obviously satisfied, since we only remove nodes and edges. For (1.i) we only need to notice that whenever $e \in \mathrm{E_L} \setminus \mathrm{E_K}$ such that source(e) is a sort node then target(e) $\in \mathrm{V_L} \setminus \mathrm{V_K}$ since it is the root operation node corresponding to a 0-sharing rule. Let $l^* : \mathrm{C} \to \mathrm{G}$ and $\overline{m} : \mathrm{K} \to \mathrm{C}$ be the morphisms completing the pushout diagram. We have that l^* is an inclusion and $\overline{m}$ is the restriction and co-restriction of m to K and C, respectively.

Let us now additionally verify that H is a term-graph.

(0)—H is bipartite. This is ensured by the fact that R is bipartite and r only identifies nodes of the same kind.

(1.i)—each operation node has exactly one inward edge. Proof by contradiction. Suppose there exists distinct edges e, e' in $\mathrm{E_H}$ such that $\mathrm{target_H}(e) = \mathrm{target_H}(e)$ and it is an operation node. Since $\top_i$ and $r_\mathrm{V}(\top_i)$ are sort nodes, we can assume, as above that $e \in \mathrm{E_C}$, $e' \in \mathrm{E_R} \setminus \mathrm{E_K}$, $\mathrm{target_R}(e') \in \mathrm{V_K}$, and $\mathrm{target_C}(e) = m_\mathrm{V}(\mathrm{target_R}(e'))$. However, $e' \in \mathrm{E_R} \setminus \mathrm{E_K}$, $\mathrm{target_R}(e') \in \mathrm{V_K}$, and $\mathrm{target_R}(e')$ operation node constitute a contradiction with the fact that R satisfies $(1.i)$, since there should be another edge in $\mathrm{E_K}$ with the same target as e'.

(1.ii)—each operation node's outward edges are consistent. Since both C and R are term-graphs, the labels of outward edges of operation sorts, as well as the labels of their targets must be consistent in H. To complete our proof we only need to additionally show that no duplicates are introduced by the merging. Proof by contradiction. Suppose there exists distinct edges e and e' in $\mathrm{E_H}$, such that $\mathrm{source_H}(e) = \mathrm{source_H}(e')$ is an operation node, and $\mathrm{le_H}(e) = \mathrm{le_H}(e')$. Then we can assume that $e' \in \mathrm{E_C}$, $e \in \mathrm{E_R} \setminus \mathrm{E_K}$, and $\mathrm{source_R}(e) \in \mathrm{V_K}$, inducing that $\mathrm{source_C}(e') = m_\mathrm{V}(\mathrm{source_R}(e))$. From $e \in \mathrm{E_R} \setminus \mathrm{E_K}$ and $\mathrm{source_R}(e) \in \mathrm{V_K}$ we infer that there exists i such that $e \in \mathrm{E_{R_i}} \setminus \mathrm{E_{K_i}}$ and $s_{\mathrm{R}_i}(e) \in \mathrm{V_K}$ is an operation node. Therefore, $x_{\mathrm{source_{R_i}}(e),\mathrm{le_{R_i}}(e)}$ cannot be a (term) variable of R_i, and therefore, it cannot be a term variable of L_i, as well. Moreover, since $\mathrm{source_{R_i}}(e) \in \mathrm{V_K}$, it must be that $\mathrm{source_{R_i}}(e) \in \mathrm{V_{L_i}}$, and hence there exists $e_i \in \mathrm{E_{L_i}}$ such that $\mathrm{source_{L_i}}(e_i) = \mathrm{source_{R_i}}(e)$ and $\mathrm{le_{L_i}}(e_i) = \mathrm{le_{R_i}}(e)$. But this implies that $e_i \in \mathrm{E_{L_i}} \setminus \mathrm{E_{K_i}}$, which contradicts with the fact that $e' \in \mathrm{E_C}$ (since e' has the same source and label).

(2)—each sort node has at most one outward edge. Proof by contradiction. Suppose there exist distinct edges e and e' in $\mathrm{E_H}$ such that $\mathrm{source_H}(e) = \mathrm{source_H}(e') = v$, and v is a sort node. We can then suppose (without loss of generality) that $e \in \mathrm{E_C}$ and $e' \in \mathrm{E_R} \setminus \mathrm{E_K}$. Then $\mathrm{source_R}(e') \in \mathrm{V_K}$ and $v = \mathrm{source_H}(e) = \mathrm{source_C}(\mathrm{E}) = m_\mathrm{V}(\mathrm{source_R}(e'))$. Reusing a previous argument, from $\mathrm{source}_r(e') \in \mathrm{V_K}$, $e' \in \mathrm{E_R} \setminus \mathrm{E_K}$ and $\mathrm{source_R}(e')$ sort node we deduce that $\mathrm{source_R}(r') \in \mathrm{ROOT_L}$. Therefore, there exists i such that $e' \in \mathrm{E_{R_i}} \setminus \mathrm{E_{K_i}}$ and $\mathrm{source_{R_i}}(e') = \top_i$. However, this implies that $\mathrm{source_C^{-1}}(m_\mathrm{V}(\mathrm{source_{R_i}}(e'))) = \emptyset$, which contradicts with $e \in \mathrm{E_C}$.

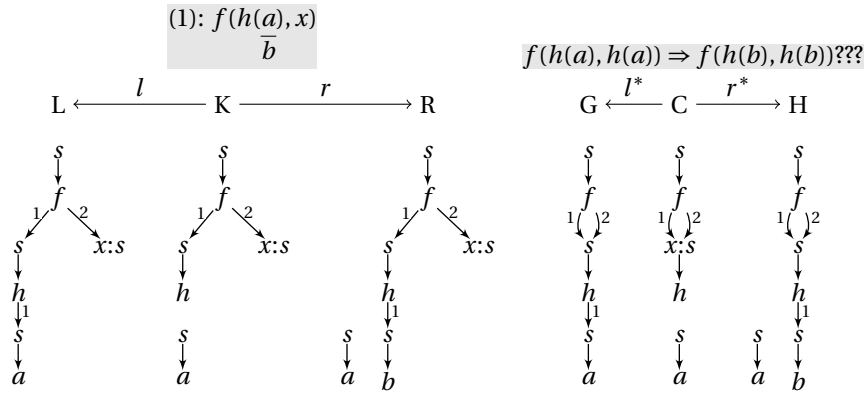

Figure 7: Subterm sharing might lead to unsound $\mathbb{K}$ graph rewriting.

(3)—H is acyclic. This is ensured by the hypothesis that m is cycle-free w.r.t. p. □

6 KRAM rewriting—Semantics

Theorem 3 allows us to capture the serializable fragment of KRAM concurrent rewriting as the relation $\Rrightarrow$ defined below:

Definition 4. *Let t be a Σ-term and let $\rho_1, \cdots, \rho_n$ be $\mathbb{K}$-rules (not necessarily distinct). Then $t \overset{\rho_1+\cdots+\rho_n}{\Longrightarrow} t'$ iff there is a term-graph H such that $\mathrm{G} \xrightarrow[\mathbf{KGraph}_\Sigma]{K2G(\rho_1)+\cdots+K2G(\rho_n)} \mathrm{H}$ and $term^{\mathrm{H}}(\top_{\mathrm{H}}) = t'$, where G is the tree term-graph representing t. We say that $t \Rrightarrow t'$ iff there is a (composed) $\mathbb{K}$-rule ρ such that $t \overset{\rho}{\Rrightarrow} t'$.*

We next show that the KRAM concurrent rewriting above is a conservative extension of the standard term rewriting relation.

6.1 Soundness and completeness w.r.t. term-rewriting

We can give a straightforward definition for what it means for a $\mathbb{K}$-rule to match a term: *one* $\mathbb{K}$-rule $\rho : (\forall \mathrm{X})\ k[\ \mathrm{L} \Rightarrow \mathrm{R}\ ]$ matches a term t using context C and substitution θ iff its corresponding rewrite rule $K2R(\rho) : (\forall \mathrm{X})\mathrm{L}(k) \rightarrow \mathrm{R}(k)$ matches t using the same C and θ, that is, iff $t = \mathrm{C}[\theta(\mathrm{L}(k))]$. This conforms to the intuition that, when applied sequentially, $\mathbb{K}$-rules behave exactly as their corresponding rewrite rules. We next show that the rewrite relation induced by $\mathbb{K}$-rules indeed captures the standard term rewrite relation. We will do that by reducing rewriting using $\mathbb{K}$ graph rules to rewriting using 0-sharing $\mathbb{K}$ graph rules, which, as we previously mentioned is actually an instance of jungle evaluation in the graph world. Then, we can use the soundness and completeness of jungle evaluation w.r.t. term rewriting to obtain that $\mathbb{K}$ term rewriting is sound and complete w.r.t regular term rewriting.

However, it turns out that, although preserving the term-graph structure (under cycle-freeness assumptions, $\mathbb{K}$ rewriting on graphs might not be sound w.r.t. term rewriting in the presence of subterm sharing. Consider the example in Figure 7. We want to apply rule $f(h(\underline{a}),x)$, corresponding to the regular rewrite rule $f(h(a),x) \to f(h(b),x)$, to the term $f(h(a),h(a))$. If we would represent $f(h(a),h(a))$ as a tree, then the $\mathbb{K}$ graph rewriting step would be sound, leading to a graph depicting $f(h(b),h(a))$; however, if we decide to collapse the tree representing $h(a)$ then we obtain $f(h(b),h(b))$, as depicted in Figure 7 which cannot be obtained through regular rewriting. The reason for this unsound rewriting is that part of the read-only patten of the rule is shared. To overcome this, we will restrict the read-only pattern of the rule to only match against a tree in the graph to be rewritten. We say that a match $m : \mathrm{L} \to \mathrm{G}$ of a $\mathbb{K}$ graph rewrite rule $\rho : (\mathrm{L} \xleftarrow{l} \mathrm{K} \xrightarrow{r} \mathrm{R})$ is *safe* if $m(\mathrm{K}\restriction_{root_\mathrm{L}})$ is a tree in G, that is, if $indegree_\mathrm{G}(m_\mathrm{V}(v)) = 1$ for any $v \in \mathrm{V}_{\mathrm{K}\restriction_{root_\mathrm{L}}} \setminus \{root_\mathrm{L}\}$. Note that, if G is a tree then all matching morphism on G are safe.

Proposition 4. *Let* ρ *be a proper* $\mathbb{K}$ *rewrite rule, let* ρ_0 *be its associated* 0*-sharing* $\mathbb{K}$ *rewrite rule, and let* m *be a cycle free safe matching morphism for* $K2G(\rho)$ *in* G*. Let* H *be such that* $\mathrm{G} \overset{K2G(\rho),m}{\underset{\mathbf{KGraph}_\Sigma}{\Longrightarrow}} \mathrm{H}$*, and let* H' *be such that* $\mathrm{G} \overset{K2G(\rho_0),m}{\underset{\mathbf{KGraph}_\Sigma}{\Longrightarrow}} \mathrm{H}'$*. Then for any* $v \in \mathrm{ROOT}_\mathrm{G}$*,* $term^\mathrm{H}(v) = term^\mathrm{H}(v)$*.*

Proof. First, cycle freeness ensures the existence of H; moreover, any 0-sharing $\mathbb{K}$-rule generates the graph representation of a jungle evaluation rule, and thus the existence of H' is ensured.

Second, since ρ is proper, neither ρ nor ρ_0 is collapsing, and therefore $\mathrm{ROOT}_\mathrm{G} \subseteq \mathrm{ROOT}_\mathrm{H}$ and $\mathrm{ROOT}_\mathrm{G} \subseteq \mathrm{ROOT}_{\mathrm{H}'}$, so the final claim is also defined.

Let $K2G(\rho) : (\mathrm{L}_\rho \xleftarrow{l_\rho} \mathrm{K}_\rho \xrightarrow{r_\rho} \mathrm{R}_\rho)$ and $K2G(\rho_0) : (\mathrm{L}_\rho \xleftarrow{l_{\rho_0}} \mathrm{K}_{\rho_0} \xrightarrow{r_{\rho_0}} \mathrm{R}_{\rho_0})$ be the complete descriptions of $K2G(\rho)$ and $K2G(\rho_0)$, and let C, C' be the corresponding context graphs obtained in the process of applying the rules to G.

We have that $\mathrm{C} = \mathrm{G} \setminus m(\mathrm{L}_\rho \setminus \mathrm{K}_\rho)$, whence $\mathrm{V_C} = \mathrm{V_G}$ and $\mathrm{E_C} = \mathrm{E_G} \setminus \{m(e_{\Box_i}) \mid e_{\Box_i} \in \mathrm{E}_{\mathrm{L}_\rho}, \mathrm{target}(e_{\Box_i}) \text{ corresponds to } \Box_i \in \mathscr{W}\}$. Also $\mathrm{C}' = \mathrm{G} \setminus m(\mathrm{L}_\rho \setminus \mathrm{K}_{\rho_0})$, whence $\mathrm{V}_{\mathrm{C}'} = \mathrm{V_G} \setminus m_\mathrm{V}(v_0)$ and $\mathrm{E}_{\mathrm{C}'} = \mathrm{E_G} \setminus (\mathrm{source}^{-1}(v_0) \cup \mathrm{target}^{-1}(v_0))$, where $v_0 = \mathrm{target}(\mathrm{source}^{-1}(root_{\mathrm{L}_\rho}))$.

H' is obtained by "gluing" on C' $\mathrm{R}_{\rho_0} \setminus \mathrm{K}_{\rho_0}$, that is the variable collapsed tree representation of $\mathrm{R}(k)$ in which the root and the variable nodes have been removed. This gluing is done by setting the source of the topmost edge to be $m_\mathrm{V}(root_\mathrm{L})$ and the target of any edge whose target is variable node x in R_{ρ_0} to be $m_\mathrm{V}(x)$.

H is obtained by "gluing" on C $\mathrm{R}_\rho \setminus \mathrm{K}_\rho$, that is the variable collapsed tree representation of $\{\mathrm{R}(\mathrm{H}ole) \mid \Box \in \mathscr{W}\}$ in which the variable nodes have been removed, and an edge $e'_{\Box_i}$ for each $\Box_i$ has been added having as target the node representing the root of $\mathrm{R}(\Box_i)$. The gluing is done by setting the target of any edge whose target is variable node x in R_ρ to be $m_\mathrm{V}(x)$ and by setting the source of $e'_{\Box_i}$ to be $\mathrm{source_G}(e_{\Box_i})$.

We can define a morphism $f : \mathrm{H} \to \mathrm{H}'$, as follows: For $\mathrm{C} \setminus m(\mathrm{K}_0)$ it is the identity: $f_\mathrm{V}(v) = v \; if \; v \in \mathrm{V_C} \setminus m(\mathrm{V}_{\mathrm{L}_\rho}) = \mathrm{V_G} \setminus m(\mathrm{V}_{\mathrm{L}_\rho})$. $f_\mathrm{E}(e) = e \; if \; e \in \mathrm{E_C} \setminus m(\mathrm{E}_{\mathrm{L}_\rho}) = \mathrm{E_G} \setminus m(\mathrm{E}_{\mathrm{L}_\rho})$. For the root of L_ρ and for its variables, it is also the identity: $f_\mathrm{V}(m(root_{\mathrm{L}_\rho})) = m(root_{\mathrm{L}_\rho})$; $f_\mathrm{V}(m(x)) = m(x)$. Now, for $\mathrm{K}_0 = \mathrm{K}_\rho \restriction_{root_{\mathrm{L}_\rho}}$, it yields the copy of K_0 in R:

$f_V(m_V(v)) = v$ for any $v \in V_{K_\rho\restriction_{root_{L_\rho}}}$, $v \neq root_{L_\rho}$ and $f_V(m_V(e)) = e$ for any $e \in E_{K_\rho\restriction_{root_{L_\rho}}}$. Finally, the mapping $R_\rho \setminus K_\rho$ it is already determined by the mapping of the elements coming from K_0, ad basically says that the variable collapsed trees corresponding to $R(\Box_i)$ are mapped to their corresponding (variable collapsed) subtrees coming from R_{ρ_0}.

It is relatively easy to verify that f is an injective morphism. Moreover the nodes and edges which are not in its image are part of the graph $m(K_0)$ (excluding $root_L$ and the topmost operation node as well as its adjacent edges), which, by being required to be a tree in G, has no incoming edge, and thus is not part of $\overline{H'} = H'\restriction_{ROOT_G}$. Hence, the restriction and co-restriction of f to $\overline{H} = H\restriction_{ROOT_G}$ and $\overline{H'}$, respectively, is a bijection, and, therefore for any $v \in ROOT_G$, $H\restriction_v$ is isomorphic with $H'\restriction_v$, whence $term^H(v) = term^{H'}(v)$. □

Since the $\mathbb{K}$ graph representation of a term t without anonymous variables is a graph jungle representing the same term, and since the $\mathbb{K}$ term-graph representation of a 0-sharing $\mathbb{K}$ rewrite rule is a graph jungle rule representing the rewrite rule associated to it, we can use the soundness and completeness of jungle rewriting w.r.t. standard term rewriting [6, 10] to prove the sequential soundness and completeness of $\mathbb{K}$ graph rewriting w.r.t. standard term rewriting, and, by combining that with Theorem 3, to prove the serializability result for KRAM concurrent rewriting.

Theorem 4. *Let $\rho, \rho_1, \ldots, \rho_n$ be $\mathbb{K}$ rules. The following hold:*

Completeness: *If $t \xRightarrow{K2R(\rho)} t'$ then $t \overset{\rho}{\Rightarrow} t'$.*

Soundness: *If $t \overset{\rho}{\Rightarrow} t'$ then $t \xRightarrow{K2R(\rho)^*} t'$.*

Serializability: *If $t \xRightarrow{\rho_1+\cdots+\rho_n} t'$, then there exists a sequence of terms $t_0, \cdots, t_n$, such that $t_0 = t$, $t_n = t'$, and $t_{i-1} \overset{\rho_i^*}{\Rightarrow} t_i$.*

Proof. Let G be the tree term-graph representation of t.

Completeness. From the completeness of jungle evaluation, we infer that there exists H such that $G \xRightarrow{m,K2G(\rho_0)} H$ and $term^H(m^*(root_G)) = t'$. Since G is a tree, m must be both cycle-free and safe for $K2G(\rho)$. From Proposition 4 we then infer that $G \xRightarrow{m,K2G(\rho)} H'$ and that $term^{H'}(m^*(root_G)) = t'$, whence $t \overset{\rho}{\Rightarrow} t'$.

Soundness. $t \overset{\rho}{\Rightarrow} t'$ implies that $G \xRightarrow{m,K2G(\rho)} H'$ such that $term^{H'}(m^*(root_G)) = t'$. Again, since G is a tree, m must be both cycle free and safe, whence, by Proposition 4, $G \xRightarrow{m,K2G(\rho_0)} H$ such that $term^H(m^*(root_G)) = t'$, and by the soundness of jungle evaluation, $t \xRightarrow{K2R(\rho)^*} t'$.

Serializability. $t \overset{\rho_1+\cdots+\rho_n}{\Longrightarrow} t'$ implies that $G \overset{K2G(\rho_1)+\ldots+K2G(\rho_n)}{\Longrightarrow} H$ such that $term^H(m^*(root_G)) = t'$. Applying Theorem 3, we deduce that there exist $G_0, G_1, \ldots, G_n$ such that $G_0 = G$, $G_n = H$, and $G_{i-1} K2G(\rho_1) G_i$. Since G is a tree and all rules satisfy the parallel independence property, we can deduce that the matching morphism for each of the steps is safe, and thus also cycle free. Therefore we can for each step apply Proposition 4, and the the soundness of jungle evaluation w.r.t. rewriting, to obtain the desired answer.

□

Therefore, KRAM concurrent rewriting is sound and complete for term rewriting, while providing a higher degree of concurrency in one step than existing approaches, be them either through term-graph rewriting or through term rewriting.

7 Case study: The AGENT Language

In this section we define AGENT, a multi-agent, multi-threaded, language which shows the easiness of defining concurrency features within $\mathbb{K}$ in the context of a non-trivial language.

The language starts with arithmetic expressions, then gradually grows in complexity by adding: functions; recursion; call with current continuation and abrupt termination; statements; input/output; multithreading with shared memory and synchronization; and agents with synchronous and asynchronous communication, and with broadcasting and barriers. To stress the independence among language features, as well as the modularity induced by the $\mathbb{K}$ framework, we introduce each feature in its separate module, as they were developed in the K-Maude tool. The modules are displayed using the LaTeX code produced by the tool, but removing the 'K RULES' keyword and massaging pagination to save space. Being solely interested in defining the semantics here, and not in parsing, we use a single syntactic category, K.

Expressions. The arithmetic and boolean expressions used in AGENT are defined in a module EXP similarly to the expressions used in IMP, except that they all share the same syntactic category K. EXP includes integers and booleans as primary values, and allows arithmetic expressions built from them using operators like addition, multiplication and division, as well as comparison operators and logical connectives. All operators are strict and act on values as their corresponding builtin operators.

Conditional. The conditional construct is strict in its first argument, expecting it to be evaluated to a boolean value, and then chooses one of the branches based on that value.

MODULE IF IMPORTS K+PL-BOOL

KResult ::= *Bool*

K ::= `if` *K* `then` *K* `else` *K* [strict(1)]

`if` true `then` *E* `else` _ ⇒ *E*

`if` false `then` _ `else` *E* ⇒ *E*

END MODULE

MODULE EXP IMPORTS K+PL-INT

$KResult ::= Bool$	$I_1 + I_2 \Rightarrow I_1 +_{Int} I_2$
$\mid Int$	$I_1 * I_2 \Rightarrow I_1 *_{Int} I_2$
$K ::= K + K$ [strict]	$I_1 \,/\, I_2 \Rightarrow I_1 /_{Int} I_2$
$\mid K * K$ [strict]	when $I_2 =/=_{Bool} 0$
$\mid K \,/\, K$ [strict]	$I_1 \leq I_2 \Rightarrow I_1 \leq_{Int} I_2$
$\mid K \leq K$ [seqstrict]	$V_1 == V_2 \Rightarrow V_1 ==_{Bool} V_2$
$\mid K == K$ [strict]	$\texttt{not}\ T \Rightarrow \texttt{not}_{Bool}\ T$
$\mid \texttt{not}\ K$ [strict]	$\texttt{true and}\ E \Rightarrow E$
$\mid K \texttt{ and } K$ [strict(1)]	$\texttt{false and}\ E \Rightarrow \texttt{false}$

END MODULE

Figure 8: AGENT: Arithmetic expressions module

λ-abstraction and β-substitution. As AGENT is a call-by-value language, we here use the LAMBDA module defined in Section 2.

Fixpoint recursion. AGENT allows recursion through the standard fix-point constructor 'μ_._', whose semantics is given as a by-need unrolling when it reaches the top of computation.

MODULE MU IMPORTS SUBSTITUTION

$K ::= \mu Id.K$	$\langle \dfrac{\mu X.E}{E\,[\,\mu X.E\,/\,X\,]}\ \_ \rangle_{\mathsf{k}}$

END MODULE

Call with current continuation. The CALLCC module extends the LAMBDA module (as it relies on the application construct) with a strict '`callcc`__' construct. The argument of callcc is expected to evaluate to a function to which the current continuation is passed as a value wrapped by '*cc*(_)'. If that value becomes the first argument of an application construct, then second argument of the application is passed to the original continuation.

MODULE CALLCC IMPORTS LAMBDA

	$\langle \dfrac{\texttt{callcc}\ V \curvearrowright K}{V\ cc(K)} \rangle_{\mathsf{k}}$
$K ::= \texttt{callcc}\ K$ [strict]	
$KResult ::= cc(K)$	$\langle \dfrac{cc(K)\ \ V \curvearrowright \_}{V \curvearrowright K} \rangle_{\mathsf{k}}$

END MODULE

Abrupt termination. Although callcc allows the encoding of most control-intensive constructs, including halt, we include a definition of halt not depending on any of the existing features.

MODULE HALT IMPORTS K

$K ::= \texttt{halt}\ K$ [strict]	$\langle \dfrac{\texttt{halt}\ V \curvearrowright \_}{V} \rangle_{\mathsf{k}}$

END MODULE

Sequential composition. Sequential composition is achieved through the construct '_;_' and its unit value '`skip`'. The semantics of '_;_' is that it evaluates the first argument, and then it discards its value, leaving the second argument to be evaluated. Note that we allow the first argument to evaluate to any value, and thus allow any expression to be used as a statement.

MODULE SEQ IMPORTS K

$K ::= K$; K [strict(1)]

KResult ::= `skip`

V ; $S \Rightarrow S$

END MODULE

Input/Output. As input and output transcend the computation structure, we need to specify the structure of the configuration required to give semantics to these constructs. The '`read`' expression evaluates to the first integer extracted from the list in the $\langle\rangle_{\mathsf{input}}$ cell, while the '`print_`' statement evaluates its argument, and then it appends it to the list in the $\langle\rangle_{\mathsf{output}}$ cell.

MODULE IO IMPORTS PL-INT+SEQ

$K ::= Int$

| `read`

| `print` K [strict]

INITIAL CONFIGURATION:

$\langle \cdot K \rangle_{\mathsf{k}}$ $\langle \cdot List \rangle_{\mathsf{in}}$ $\langle \cdot List \rangle_{\mathsf{out}}$

$\langle \frac{\texttt{read}\ \_}{I} \rangle_{\mathsf{k}}$ $\langle \frac{I\ \_}{\cdot} \rangle_{\mathsf{in}}$

$\langle \frac{\texttt{print}\ V\ \_}{\texttt{skip}} \rangle_{\mathsf{k}}$ $\langle \frac{\_\ \cdot}{V} \rangle_{\mathsf{out}}$

END MODULE

Loop statement. Although loops could be easily simulated using recursion, we here give it a direct (imperative) semantics, by unrolling it when on-top of the computation.

MODULE WHILE IMPORTS IF+SEQ

$K ::= \texttt{while}\ K\ \texttt{do}\ K$

$\langle \frac{\texttt{while}\ E\ \texttt{do}\ S}{\texttt{if}\ E\ \texttt{then}\ S\ \texttt{;}\ \texttt{while}\ E\ \texttt{do}\ S\ \texttt{else}\ \texttt{skip}} \rangle_{\mathsf{k}}$

END MODULE

Memory. To be able to write meaningful sequential programs we introduce memory. However, to keep in tone with our functional flavor, we prefer references instead of variable declarations. For that, we require that the configuration contains a $\langle\rangle_{\mathsf{mem}}$ cell containing a map (initially empty) of locations (naturals) to values, and a $\langle\rangle_{\mathsf{nextLoc}}$ cell containing the next available location (initially 0). The '`ref_`' expression evaluates its argument, and then it allocates it in the memory at the next available location, evaluating itself to that location. '`*_`' evaluates its argument and then dereferences it to the value it maps to in the memory. '`_:=_`' expects a dereferencing expression as its first argument, and updates the value in the memory mapped to by it with the value of the second argument. The evaluation of the location in the first argument is ensured through a context declaration.

MODULE REF IMPORTS SEQ

$K ::= Nat$ | `ref` K [strict] | $* K$ [strict] | $K := K$ [strict(2)]

INITIAL CONFIGURATION: $\langle \cdot K \rangle_{\mathsf{k}}\ \langle \cdot Map \rangle_{\mathsf{mem}}\ \langle 0 \rangle_{\mathsf{nextLoc}}$

CONTEXT: $* \square := \_$

$$\langle \frac{\texttt{ref}\ V}{N}\ \_ \rangle_{\mathsf{k}}\ \langle \_\ \frac{\cdot}{N \mapsto V}\ \_ \rangle_{\mathsf{mem}}\ \langle \frac{N}{\mathtt{s}_{Nat}\ N}\ \rangle_{\mathsf{nextLoc}}$$

$$\langle \frac{* N}{V}\ \_ \rangle_{\mathsf{k}}\ \langle \_\ N \mapsto V\ \_ \rangle_{\mathsf{mem}}$$

$$\langle \frac{* N := V}{\texttt{skip}}\ \_ \rangle_{\mathsf{k}}\ \langle \_\ N \mapsto \frac{\_}{V}\ \_ \rangle_{\mathsf{mem}}$$

END MODULE

Threads and synchronization. Let us now define a generic multi-threading module. Note that this module is independent of the previous ones; the only module required is SEQ, as we want our multi-threading statements to evaluate to '`skip`'. As we need to allow multiple execution threads synchronized using locks, our multi-threading minimal configuration is written to reflect that: the $\langle\rangle_{\mathsf{thread}}$ cell has multiplicity "zero-or-more" (indicated by the $*$ suffix) and must contain a computation $\langle\rangle_{\mathsf{k}}$ cell and a $\langle\rangle_{\mathsf{holds}}$ cell to account for the locks held; besides the multiple threads, the system must also contain a $\langle\rangle_{\mathsf{busy}}$ cell containing the acquired but not yet released locks. The semantics of locks is that any value can act as a lock, and that locks are re-entrant (this is why the $\langle\rangle_{\mathsf{holds}}$ cell contains a map from values to naturals). The '`spawn_`' statement creates a new thread containing the given argument as its initial computation and having default initial values for other potential cells within thread (e.g., the $\langle\rangle_{\mathsf{holds}}$ cell is initialized with the empty map, as specified by the configuration). When the computation of a thread is reduced to a value, the thread is dissolved and its resources are freed. Lock acquire is defined through two rules, depending whether the lock is already held by the thread (in which case its counter is increased), or it is available (in which case it is added to the $\langle\rangle_{\mathsf{busy}}$ cell and mapped to 0 in the $\langle\rangle_{\mathsf{holds}}$ cell). The rules for lock release mirror those for lock acquiring. Finally, '`rendezvous_`' evaluates its argument and then blocks until another thread requires a rendez-vous on the same value; then, the two threads advance together.

MODULE THREADS

IMPORTS PL-NAT+SEQ

$K ::= Nat$ | `spawn` K | `acquire` K [strict] | `release` K [strict] | `rendezvous` K [strict]

INITIAL CONFIGURATION:

$$\langle \langle \cdot K \rangle_{\mathsf{k}}\ \langle \cdot Map \rangle_{\mathsf{holds}} \rangle_{\mathsf{thread}*}\ \langle \cdot Set \rangle_{\mathsf{busy}}$$

$$\langle \_\ \langle \frac{\texttt{spawn}\ S}{\texttt{skip}}\ \_ \rangle_{\mathsf{k}}\ \_ \rangle_{\mathsf{thread}}\ \frac{\cdot}{\langle \_\ \langle S \rangle_{\mathsf{k}}\ \_ \rangle_{\mathsf{thread}}}$$

$$\frac{\langle \_ \ \langle V \rangle_{\mathsf{k}} \ \langle \mathit{Holds} \rangle_{\mathsf{holds}} \ \_ \rangle_{\mathsf{thread}}}{\cdot} \ \langle \frac{\mathit{Busy}}{\mathit{Busy} -_{\mathit{Set}} \mathtt{keys}\ \mathit{Holds}} \rangle_{\mathsf{busy}}$$

$$\langle \frac{\mathtt{acquire}\ V}{\mathtt{skip}} \ \_ \rangle_{\mathsf{k}} \ \langle \_ \ \frac{\cdot}{V \mapsto 0} \ \_ \rangle_{\mathsf{holds}} \ \langle \mathit{Busy} \ \frac{\cdot}{V} \rangle_{\mathsf{busy}}$$

when $\mathtt{not}_{\mathit{Bool}}\ V\ \mathtt{in}\ \mathit{Busy}$

$$\langle \frac{\mathtt{acquire}\ V}{\mathtt{skip}} \ \_ \rangle_{\mathsf{k}} \ \langle \_ \ V \mapsto \frac{N}{\mathtt{s}_{\mathit{Nat}}\ N} \ \_ \rangle_{\mathsf{holds}}$$

$$\langle \frac{\mathtt{release}\ V}{\mathtt{skip}} \ \_ \rangle_{\mathsf{k}} \ \langle \_ \ V \mapsto \frac{\mathtt{s}_{\mathit{Nat}}\ N}{N} \ \_ \rangle_{\mathsf{holds}}$$

$$\langle \frac{\mathtt{release}\ V}{\mathtt{skip}} \ \_ \rangle_{\mathsf{k}} \ \langle \_ \ \frac{V \mapsto 0}{\cdot} \ \_ \rangle_{\mathsf{holds}} \ \langle \_ \ \frac{V}{\cdot} \ \_ \rangle_{\mathsf{busy}}$$

$$\langle \frac{\mathtt{rendezvous}\ V}{\mathtt{skip}} \ \_ \rangle_{\mathsf{k}} \ \langle \frac{\mathtt{rendezvous}\ V}{\mathtt{skip}} \ \_ \rangle_{\mathsf{k}}$$

END MODULE

Communicating agents. The AGENTS module allows dynamic creation and termination of agents. Agents are self-aware and creator-aware, and communicate by means of asynchronous and synchronous message-send commands, by targeted and non-targeted receive expressions, and by broadcasting and global barriers. Given the complexity of interaction we want to achieve, the configuration required to give their semantics must contain several new cells. First, each agent is contained in an $\langle\rangle_{\mathsf{agent}}$ cell, and contains at least a $\langle\rangle_{\mathsf{control}}$ cell with a $\langle\rangle_{\mathsf{k}}$ cell within it, and $\langle\rangle_{\mathsf{me}}$ and $\langle\rangle_{\mathsf{parent}}$ cells providing identification information. The reason for the $\langle\rangle_{\mathsf{control}}$ cell is that we want to allow the control mechanism of a thread to be more complex, while still being able to completely stop an agent if needed. Additional cells are: $\langle\rangle_{\mathsf{nextAgent}}$—a counter for the next available agent, $\langle\rangle_{\mathsf{world}}$—a set containing the agents currently in the system, $\langle\rangle_{\mathsf{barrier}}$ and $\langle\rangle_{\mathsf{waiting}}$—containing the current status of the global barrier, and the agents waiting at it, respectively, and $\langle\rangle_{\mathsf{msgs}}$ which contains a bag of messages, each wrapped into a $\langle\rangle_{\mathsf{msg}}$ cell and containing cells describing the sender ($\langle\rangle_{\mathsf{from}}$), the set of recipients ($\langle\rangle_{\mathsf{to}}$), and the message itself ($\langle\rangle_{\mathsf{body}}$). Since this module introduces substantially more language constructs than the preceding ones, we next explain the rules inline within the definition.

MODULE AGENTS IMPORTS PL-NAT+SEQ

$K ::= \mathit{Nat} \mid \mathit{Bool} \mid \mathtt{newAgent}\ K \mid \mathtt{haltAgent} \mid \mathtt{me} \mid \mathtt{parent}$

$\mid \mathtt{receive} \mid \mathtt{rcvFrom}\ K\,[\mathrm{strict}] \mid \mathtt{send}\ K\ \mathtt{to}\ K\,[\mathrm{strict}]$

$\mid \mathtt{sendSynch}\ K\ \mathtt{to}\ K\,[\mathrm{strict}] \mid \mathtt{bcst}\ K[\mathrm{strict}] \mid \mathtt{barrier}$

INITIAL CONFIGURATION:

$$\left\langle \begin{array}{c} \langle\langle 0 \rangle_{\mathsf{me}} \ \langle -_{\mathit{Int}}\ 1 \rangle_{\mathsf{parent}} \ \langle\langle \cdot K \rangle_{\mathsf{k}} \rangle_{\mathsf{control}} \rangle_{\mathsf{agent}*} \\ \langle 1 \rangle_{\mathsf{nextAgent}} \ \langle 0 \rangle_{\mathsf{world}} \ \langle \mathsf{true} \rangle_{\mathsf{barrier}} \ \langle \cdot \mathit{Set} \rangle_{\mathsf{waiting}} \\ \langle\langle\langle \cdot K \rangle_{\mathsf{from}} \ \langle \cdot \mathit{Set} \rangle_{\mathsf{to}} \ \langle \cdot K \rangle_{\mathsf{body}} \rangle_{\mathsf{msg}*} \rangle_{\mathsf{msgs}} \end{array} \right\rangle$$

MODULE AGENT

IMPORTS EXP+IF+LAMBDA+MU+CALLCC

IMPORTS HALT+SEQ+IO+REF+WHILE+THREADS+AGENTS

$Bag ::= \texttt{run(}\ K\ \texttt{,}\ List\ \texttt{)}$

MACRO: $\texttt{run(}\ K\ \texttt{,}\ L\ \texttt{)} = \langle \_\ \langle K \rangle_{\mathsf{k}}\ \langle L \rangle_{\mathsf{in}}\ \_ \rangle_{\mathsf{T}}$

INITIAL CONFIGURATION:

$$\left\langle \begin{array}{c} \left\langle \begin{array}{c} \left\langle \left\langle \langle \cdot K \rangle_{\mathsf{k}}\ \langle \cdot Map \rangle_{\mathsf{holds}} \right\rangle_{\mathsf{thread}*} \right\rangle_{\mathsf{control}}\ \langle \cdot Set \rangle_{\mathsf{busy}} \\ \langle \cdot Map \rangle_{\mathsf{mem}}\ \langle 0 \rangle_{\mathsf{nextLoc}}\ \langle 0 \rangle_{\mathsf{me}}\ \langle -_{Int}\ 1 \rangle_{\mathsf{parent}} \end{array} \right\rangle_{\mathsf{agent}*} \\ \langle 1 \rangle_{\mathsf{nextAgent}}\ \langle 0 \rangle_{\mathsf{world}}\ \langle \mathsf{true} \rangle_{\mathsf{barrier}}\ \langle \cdot Set \rangle_{\mathsf{waiting}} \\ \left\langle \left\langle \langle \cdot K \rangle_{\mathsf{from}}\ \langle \cdot Set \rangle_{\mathsf{to}}\ \langle \cdot K \rangle_{\mathsf{body}} \right\rangle_{\mathsf{msg}*} \right\rangle_{\mathsf{msgs}}\ \left\langle \langle \cdot List \rangle_{\mathsf{in}}\ \langle \cdot List \rangle_{\mathsf{out}} \right\rangle_{\mathsf{I/O}} \end{array} \right\rangle_{\mathsf{T}}$$

END MODULE

Figure 9: Main module of the AGENT language definition.

The '`newAgent_`' expression creates a new agent, setting as its non-default values the computation cell (initialized with the given argument), the $\langle\rangle_{\mathsf{me}}$ cell (initialized as the next available agent id, which is incremented), and the $\langle\rangle_{\mathsf{parent}}$ cell (initialized as the id of the creating agent); additionally, the new agent id is registered in $\langle\rangle_{\mathsf{world}}$ and is returned as the value of '`newAgent_`'.

$$\left(\begin{array}{c} \langle \_\ \langle \underline{\texttt{newAgent}\ S} \rangle_{\mathsf{k}}\ \langle N_1 \rangle_{\mathsf{me}}\ \_ \rangle_{\mathsf{agent}}\ \langle \underline{\quad N_2 \quad} \rangle_{\mathsf{nextAgent}} \\ \underline{\qquad\qquad \cdot \qquad\qquad} \\ \langle \_\ \langle N_2 \rangle_{\mathsf{me}}\ \langle N_1 \rangle_{\mathsf{parent}}\ \langle S \rangle_{\mathsf{k}}\ \_ \rangle_{\mathsf{agent}} \end{array} \begin{array}{c} \underline{\mathsf{s}_{Nat}\ N_2} \\ \\ \langle \_\ \underline{\quad \cdot \quad} \rangle_{\mathsf{world}} \\ N_2 \end{array} \right)$$

When the control of an agent becomes empty, the agent is dissolved and unregistered from the $\langle\rangle_{\mathsf{world}}$ cell; hence '`haltAgent`' only needs to empty the contents of the $\langle\rangle_{\mathsf{control}}$ cell.

$$\frac{\langle \_\ \langle \cdot \rangle_{\mathsf{control}}\ \langle N \rangle_{\mathsf{me}}\ \_ \rangle_{\mathsf{agent}}\ \langle \_\ \underline{N} \rangle_{\mathsf{world}}}{\cdot}$$

$$\langle \_\ \langle \texttt{haltAgent}\ \_ \rangle_{\mathsf{k}}\ \_ \rangle_{\mathsf{control}} \Rightarrow \langle \cdot \rangle_{\mathsf{control}}$$

'`me`' and '`parent`' have the straightforward semantics, yielding the contents of the $\langle\rangle_{\mathsf{me}}$ and $\langle\rangle_{\mathsf{parent}}$ cells of the enclosing agent:

$$\frac{\langle \underline{\texttt{me}}\ \_ \rangle_{\mathsf{k}}\ \langle N \rangle_{\mathsf{me}}}{N} \qquad\qquad \frac{\langle \underline{\texttt{parent}}\ \_ \rangle_{\mathsf{k}}\ \langle N \rangle_{\mathsf{parent}}}{N}$$

An agent can send any results (including agent ids) to other agents, provided it knows their identity. To model asynchronous communication, $\langle\rangle_{\mathsf{msg}}$ cells hold the sender of the message, the intended set of receivers, and a message body:

$$\langle N_1\rangle_{\mathsf{me}}\ \langle \underset{\texttt{skip}}{\underline{\texttt{send}\ V\ \texttt{to}\ N_2}}\ \_\rangle_{\mathsf{k}}\ \underset{\langle\langle N_1\rangle_{\mathsf{from}}\ \langle N_2\rangle_{\mathsf{to}}\ \langle V\rangle_{\mathsf{body}}\rangle_{\mathsf{msg}}}{\underline{\cdot}}$$

An agent can request to receive a message from a certain agent, or from any agent, waiting until that happens. Upon receiving, the agent's id is removed from the $\langle\rangle_{\mathsf{to}}$ cell of the message:

$$\langle N\rangle_{\mathsf{me}}\ \langle \underset{V}{\underline{\texttt{receive}}}\ \_\rangle_{\mathsf{k}}\ \langle \_\ \langle \_\ \underset{\cdot}{\underline{N}}\ \_\rangle_{\mathsf{to}}\ \langle V\rangle_{\mathsf{body}}\ \_\rangle_{\mathsf{msg}}$$

$$\langle\langle N_2\rangle_{\mathsf{from}}\ \langle \_\ \underset{\cdot}{\underline{N_1}}\ \_\rangle_{\mathsf{to}}\ \langle V\rangle_{\mathsf{body}}\rangle_{\mathsf{msg}}\ \langle N_1\rangle_{\mathsf{me}}\ \langle \underset{V}{\underline{\texttt{rcvFrom}\ N_2}}\ \_\rangle_{\mathsf{k}}$$

A message can also be broadcast to all agents in the $\langle\rangle_{\mathsf{world}}$ cell:

$$\langle N\rangle_{\mathsf{me}}\ \langle \underset{\texttt{skip}}{\underline{\texttt{bcst}\ V}}\ \_\rangle_{\mathsf{k}}\ \langle W\rangle_{\mathsf{world}}\ \underset{\langle\langle N\rangle_{\mathsf{from}}\ \langle W\rangle_{\mathsf{to}}\ \langle V\rangle_{\mathsf{body}}\rangle_{\mathsf{msg}}}{\underline{\cdot}}$$

Once a message has no receivers, it can be removed:

$$\langle \_\ \langle\cdot\rangle_{\mathsf{to}}\ \_\rangle_{\mathsf{msg}} \Rightarrow \cdot$$

Agents can also communicate synchronously if the sender chooses so, in which case the sender and the receiver need to be matched together for the exchange to occur:

$$\left(\begin{array}{c}\langle \_\ \langle N_1\rangle_{\mathsf{me}}\ \langle \underset{\texttt{skip}}{\underline{\texttt{sendSynch}\ V\ \texttt{to}\ N_2}}\ \_\rangle_{\mathsf{k}}\ \_\rangle_{\mathsf{agent}} \\ \langle \_\ \langle N_2\rangle_{\mathsf{me}}\ \langle \underset{V}{\underline{\texttt{rcvFrom}\ N_1}}\ \_\rangle_{\mathsf{k}}\ \_\rangle_{\mathsf{agent}}\end{array}\right)$$

$$\langle \underset{\texttt{skip}}{\underline{\texttt{sendSynch}\ V\ \texttt{to}\ N_2}}\ \_\rangle_{\mathsf{k}}\ \langle \_\ \langle N_2\rangle_{\mathsf{me}}\ \langle \underset{V}{\underline{\texttt{receive}}}\ \_\rangle_{\mathsf{k}}\ \_\rangle_{\mathsf{agent}}$$

AGENT supports global synchronization by means of barriers. When an agent reaches a barrier and the barrier is on, the agent adds itself to the $\langle\rangle_{\mathsf{waiting}}$ cell:

$$\langle N\rangle_{\mathsf{me}}\ \langle \texttt{barrier}\ \_\rangle_{\mathsf{k}}\ \langle \mathsf{true}\rangle_{\mathsf{barrier}}\ \langle W\ \underset{N}{\underline{\cdot}}\rangle_{\mathsf{waiting}}$$

When all agents in $\langle\rangle_{\mathsf{world}}$ are waiting, the barrier is lifted:

$$\underset{\mathsf{false}}{\underline{\langle \mathsf{true}\rangle_{\mathsf{barrier}}}}\ \langle W\rangle_{\mathsf{waiting}}\ \langle W\rangle_{\mathsf{world}} \quad \text{when } W =/=_{Bool} \cdot$$

Then, agents unregister from the $\langle\rangle_{\mathsf{waiting}}$ cell and proceed:

$$\langle N\rangle_{\mathsf{me}}\ \langle \underset{\texttt{skip}}{\underline{\texttt{barrier}}}\ \_\rangle_{\mathsf{k}}\ \langle \mathsf{false}\rangle_{\mathsf{barrier}}\ \langle \_\ \underset{\cdot}{\underline{N}}\ \_\rangle_{\mathsf{waiting}}$$

Once all the agents proceeded, the barrier is reseted to true:

$$\underset{\mathsf{true}}{\underline{\langle \mathsf{false}\rangle_{\mathsf{barrier}}}}\ \langle\cdot\rangle_{\mathsf{waiting}}$$

END MODULE

KRAM, and specifically the existence of a read-only pattern on which rule instances can overlap, allows for example, that two agents send or receive messages simultaneously, while accessing the same $\langle\rangle_{\mathsf{msgs}}$ cell, including the case of multiple agents simultaneously reading the same broadcast message and removing themselves from the receivers set.

Putting them all together. Figure 9 contains the main module of the language, which loads all modules defined so far, and defines a running configuration consistent with previous configurations (i.e., extending the transitive closure of the subcell relation). For example, the $\langle\rangle_{\mathsf{control}}$ cell of an agent contains here a pool of threads, each with its own $\langle\rangle_{\mathsf{k}}$ cell, instead of just one $\langle\rangle_{\mathsf{k}}$ cell. However, thanks to context abstraction provided by $\mathbb{K}$ [20], this does not affect the applicability of the already defined rules.

Since this is the final module of the AGENT definition, we also define a '`run(_,_)`' macro, which given an initial computation and input list sets the contents of the $\langle\rangle_{\mathsf{k}}$ cell and the $\langle\rangle_{\mathsf{in}}$ cell with the provided values in the initial configuration of the system.

8 Conclusion

We introduced KRAM, the concurrent rewriting abstract machine. The distinguished aspect of the KRAM is that its rewrite rules explicitly state what portions of the term can be concurrently shared with other rules. This sharing information allows to increase the potential for concurrent rewriting, but it may also lead to inconsistencies if not used properly. We showed that, under reasonable conditions, KRAM rewriting is actually sound, complete, and serializable w.r.t. term rewriting. Several examples were discussed, indicating that the KRAM can be a viable framework to faithfully define truly concurrent calculi and languages.

References

[1] The K Framework. URL `http://k-framework.googlecode.com`.

[2] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In *PARLE'87*, volume 259 of *LNCS*, pages 141–158. Springer, 1987.

[3] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[4] G. Boudol. Some chemical abstract machines. In *REX School/Symp.*, volume 803 of *LNCS*, pages 92–123. Springer, 1993.

[5] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[6] A. Corradini and F. Rossi. Hyperedge replacement jungle rewriting for term-rewriting systems and programming. *Theoretical Computer Science*, 109(1&2): 7–48, 1993.

[7] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation: Basic concepts and double pushout approach. In *Handbook of graph grammars and computing by graph transformations*, volume 1, pages 163–246. World Scientific, 1997.

[8] H. Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer, 1978.

[9] H. Ehrig and H.-J. Kreowski. Parallelism of manipulations in multidimensional information structures. In *MFCS'76*, volume 45 of *LNCS*, pages 284–293. Springer, 1976.

[10] A. Habel, H.-J. Kreowski, and D. Plump. Jungle evaluation. In *ADT'87*, volume 332 of *LNCS*, pages 92–112. Springer, 1987.

[11] A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.

[12] B. Hoffmann and D. Plump. Implementing term rewriting by jungle evaluation. *ITA*, 25:445–472, 1991.

[13] H.-J. Kreowski. Transformations of derivation sequences in graph grammars. In *FCT'77*, pages 275–286, 1977.

[14] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[15] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *CONCUR '96*, volume 1119 of *LNCS*, pages 331–372. Springer, 1996.

[16] J. Meseguer and G. Roșu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[17] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.

[18] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[19] D. Plump. Term graph rewriting. In *Handbook of graph grammars and computing by graph transformation*, volume 2, pages 3–61. World Scientific, 1999.

[20] G. Roșu and T. F. Șerbănuță. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 2010. doi: 10.1016/j.jlap.2010.03.012. In press.

[21] T. F. Șerbănuță and G. Roșu. K-Maude: A rewriting based tool for semantics of programming languages. In *WRLA'10*, LNCS, To appear.

[22] M. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):334–368, 2002.

A Differences between Terms and Term-Graphs

Collapsing vs. non-collapsing. The graph representation of terms has many merits, among which the one of a more efficient representation stands out. Using graphs instead of trees, one could collapse identical subterms, transforming the tree into a "DAG". This not only improves on the size of the representation, but can also potentially speed up the rewriting process: a rewrite rule application in a shared subterm now stands for multiple applications of the same rule in the tree representation of the term. Moreover, for non-linear rules, maximum subterm sharing reduces variable equality comparison to simple pinter equality. For these reasons, even most of the current *term* rewrite engines (including, e.g., ASF+SDF [22] or Maude [5]) use maximum subterm sharing data structures to represent the terms to be rewritten.

This representation is indeed effective and efficient for executing deterministic systems. However, when analyzing executions it is often more important to capture all potential behaviors than to have an efficient execution. Therefore, in that case, it is preferable that subterms are not shared, to allow them to be rewritten separately, and thus be able to explore all potential executions.

Since the primary goal of the $\mathbb{K}$ framework is to capture all potential (concurrent) executions of a system, we here take the second approach, and only identify variables, therefore, representing ground terms as trees and thus allowing all possible concurrent behaviors. Nevertheless, the development presented here should work as-is for collapsed terms if one is more interested in execution than in analysis.

Term substitutions and term-graph morphisms The connection between terms and term-graphs carries on at the level of morphisms: a graph morphism induces a substitution between the term variables of the source graph to terms over the term variables of the destination graph.

Definition 5 (substitution induced by a term-graph morphism)**.** *Let h be a morphism between term-graphs* G *and* G′*. Then h induces a substitution* $\theta_h : \mathrm{TVARS_G} \to \mathrm{T_\Sigma(TVARS_{G'}}$*, defined by*

$$\theta_h(x) = \begin{cases} term^{G'}(h_V(x)), \text{ if } x \in \mathrm{VAR_G} \\ term^{G'}(v'), \text{ if } x = (v,i) \quad \text{and there exists an edge labeled} \\ \qquad\qquad\qquad\qquad\quad \text{with } i \text{ from } h_V(v) \text{ to } v' \\ (h_V(v), i), \text{ if } x = (v,i) \quad \text{otherwise} \end{cases}$$

Moreover, by applying this substitution on any term corresponding to a node v in the source graph, we obtain the term corresponding to $h(v)$ in the target graph.

Conversely, for any substitution $\theta : \mathrm{X} \to \mathrm{T_\Sigma(Y)}$, and any Σ term t such that $vars(t) = \mathrm{X}$, there exists a unique morphism $h_\theta : graph(t) \to graph(\theta(t))$ mapping node $\bar{\epsilon}$ to $\bar{\epsilon}$, and the substitution induced by this morphism is θ.

On left-linearity. Since the term to be rewritten is a tree, we cannot test term equality as node identity and therefore, we will disallow non-left-linear rules, but will allow side conditions for the matching requiring that two variables are equal, which are testable as tree isomorphism.

Definition 6. *The **linearization** of a term t (or a substitution $\theta : Y \to T_\Sigma(X)$) is defined as the term $\overline{t}$ (or $\overline{\theta}$) obtained by renaming each **duplicate** occurrence of a variable x in t (or in $(\theta(y))_y$) with $x^{(n)}$, where n is a fresh natural number. The X'-**linearization** of a term r (or a substitution θ) is defined as the term $\overline{t}^{X'}$ (or $\overline{\overline{theta}}^{X'}$) obtained by renaming each occurrence of a variable $x \in X'$ in t (or in $(\theta(y))_{y \in Y}$) with $x^{(n)}$, where n is a fresh natural number.*

*The **left-linearization** of a $\mathbb{K}$ rewrite rule $\rho : k[\underline{L}] \underline{\quad} \overline{R}$ is the rule $\overline{\rho} : k'[\underline{L'}] \underline{\quad} \overline{R}$, where $k' = \overline{k}$ is the linearization of p and $L' = \overline{L}^{vars(k)}$ is the linearization of L obtained after first renaming each variable also appearing in p.*

B $\mathbb{K}$ rewriting over lists and sets

This section was used in [20] to show how concurrency can be enhanced by handling matching modulo (ACU) axioms differently. We presented below for reader's convenience.

Take for example the following rules for reading/writing a variable in the state (obtained by translating the corresponding IMP++ rules into rewrite rules):

$$\langle x \curvearrowright k \rangle_{\mathsf{k}}\ \langle \sigma\ x \mapsto i \rangle_{\mathsf{state}} \to \langle i \curvearrowright k \rangle_{\mathsf{k}}\ \langle \sigma\ x \mapsto i \rangle_{\mathsf{state}} \text{ and}$$
$$\langle x = i' \ ; \curvearrowright k \rangle_{\mathsf{k}}\ \langle \sigma\ x \mapsto i \rangle_{\mathsf{state}} \to \langle k \rangle_{\mathsf{k}}\ \langle \sigma\ x \mapsto i' \rangle_{\mathsf{state}}$$

The $\langle\rangle_{\mathsf{k}}$ cell is a unary operation $\langle\_\rangle_{\mathsf{k}}$ that holds a $\curvearrowright$-separated list of tasks (i.e., $\curvearrowright$ is associative and has identity "$\cdot$"). The $\langle\rangle_{\mathsf{state}}$ cell holds a map, i.e. a set of bindings of names to integers, constructed with the associative and commutative (AC) concatenation operation "_ _", having identity "$\cdot$" The topmost concatenation operation "_ _" (used to put cells together) is also an AC operator with identity "$\cdot$". x, k, σ, i, and i' are variables, x standing for a name, k for the rest of the computation, σ for the remainder of the state, and i, i' for integers.

Consider a system containing only these rules, and let $\langle a \rangle_{\mathsf{k}}\ \langle a \rangle_{\mathsf{k}}\ \langle b = 3\ ; \rangle_{\mathsf{k}}\ \langle a \mapsto 1\ b \mapsto 2 \rangle_{\mathsf{state}}$ be the (ground) term to be rewritten, i.e., two threads $\langle a \rangle_{\mathsf{k}}$ whose tasks are to read a from the state, and a thread $\langle b = 3\ ; \rangle_{\mathsf{k}}$ updating the value of b. All threads could advance simultaneously: the first two by reading the value of a (since a is shared), and the third by updating the value of b (since the location of b is independent of that of a). However, this is impossible to achieve directly using the deduction rules of rewriting logic. One reason, addressed in this section, is that traditional matching modulo axioms requires that the term be rearranged to fit the pattern, and thus it limits concurrency where sharing is allowed: there is no way to re-arrange the term so that any two of the rule instances match simultaneously. To address that, we propose a special treatment for matching operators governed by axioms.

The main idea is to think of the matching process as *changing the rule to fit the term rather than changing the term to fit the rule*; in that sense, rewrite rules become rule schemata. Our goal is to modify the rule to match a concrete representation of the term. The reason for requiring adjustments to the rule is that while the term is concrete, and, for example, each list constructor has only two arguments, the rule is specified modulo axioms.

Assume a generic $\mathbb{K}$ rule $\rho : (\forall X)\ k[\ L \Rightarrow R\ ]$. The following two K rules precisely capture the intuition we have about reading (ρ_r) or writing (ρ_w) a variable in the state, respectively:

$$\rho_r : \langle \frac{x}{i} \curvearrowright \_ \rangle_k\ \langle \_\ x \mapsto i \rangle_{\mathsf{state}} \qquad \rho_w : \langle \frac{x = i'\ ;}{\cdot} \curvearrowright \_ \rangle_k\ \langle \_\ x \mapsto \frac{\_}{i'} \rangle_{\mathsf{state}}$$

ρ_r states that x is replaced with its corresponding value, while the rest of the context stays unchanged, allowing it to be shared with other rules. Similarly, ρ_w specifies that the value corresponding to x in the state should be updated to the new value i', and the assignment statement should be consumed (specified by replacing it with "$\cdot$", the unit for "$\curvearrowright$"), again leaving the rest of the context unchanged. If we want to identify the anonymous variables, these rules could be alternatively written as:

$$\rho_r : \langle \frac{x}{i} \curvearrowright k \rangle_k\ \langle \sigma\ x \mapsto i \rangle_{\mathsf{state}} \qquad \rho_w : \langle \frac{x = i'\ ;}{\cdot} \curvearrowright k \rangle_k\ \langle \sigma\ x \mapsto \frac{i}{i'} \rangle_{\mathsf{state}}$$

When formalizing these rules according to Definition 1, we obtain the following K rules:

$\rho_r : (\forall X_r)\ p_r[\ \frac{L_r}{R_r}\ ]$	$\rho_w : (\forall X_w)\ p_w[\ \frac{L_w}{R_w}\ ]$
$X_r = \{x, i, k, \sigma\}$	$X_w = \{x, i, i', k, \sigma\}$
$\mathscr{W}_r = \{\Box\}$	$\mathscr{W}_w = \{\Box_1, \Box_2\}$
$p_r = \langle \Box \curvearrowright k \rangle_k\ \langle \sigma\ x \mapsto i \rangle_{\mathsf{state}}$	$p_w = \langle \Box_1 \curvearrowright k \rangle_k\ \langle \sigma\ x \mapsto \Box_1 \rangle_{\mathsf{state}}$
$L_r(\Box) = x$	$L(\Box_1) = x = i'\ ;$ and $L(\Box_2) = i$
$R_r(\Box) = i$	$R(\Box_1) = \cdot$ and $R(\Box_2) = i'$

In what follows we describe a sequence of steps, which, if applied in order, generate all concrete instances of ρ needed to replace matching modulo (ACU) axioms by plain term matching. Before going into more technical details, let us briefly describe each step. First step deals with the unit axiom, generating additional rules to account for variables being matched to the unit of an operation. Second step prepares the terrain for dealing with associativity; each variable which could stand for a term topped in an associative operation is replaced by an arbitrary number of variables separated by that operation. In step 3 we deal with commutativity, by generating rule instances for all permutations of arguments of commutative operations. Finally, in step 4, we deal with the associativity axiom, properly parenthesizing all parts of the rule containing associative operations. Note that, although both steps 2 and 4 deal with associativity, steps 3 needs to be inserted between them to generate all permutations needed for the AC operations.

This generative process of generating all matching instances for rules serves only for a theoretical purpose, as it actually generates an infinite number of concrete rule instances. In a practical implementation of these ideas, we expect that the rule schema would dynamically be adjusted in the process of matching, creating concrete rule instances by-need.

1. Resolving Unit We assume that the concrete terms to be matched are always kept in normal form w.r.t. the unit axioms, that is, the unit † of an operation $\star$ cannot appear as an argument of that operation. This can be obtained by either reducing the term after each rewriting step, or by reducing it only once at the beginning, and ensuring that the rewrite steps preserve this property. Assuming this, to address matching modulo unit it is sufficient then that for each variable x of the same sort as † appearing as an argument of operation $\star$, say in a subterm $\star(x, p)$, we generate an additional rule in which $\star(x, p)$ is replaced by p (and similarly if x is the second argument of $\star$). Moreover, if x appears at the top of a replacement term R(□), then R(□) must be † in the additional generated rule. As a matching example, for the pattern $\langle \sigma\ x \mapsto v \rangle_{\mathsf{state}}$ we need to generate an additional pattern $\langle x \mapsto v \rangle_{\mathsf{state}}$ since σ could match $\cdot$, the unit of _ _.

2. Multiplying associative variables For simplicity, we assume that each sort has at most one associative operator defined on it; our definitions satisfy that—in fact, besides the K sort, which itself is a list, all other sorts with associative operators allowed by K are lists, bags, sets, and maps. Moreover, we will assume that all rules topped in an associative operator $\star$ of sort S have by default two (or only one if $\star$ is also commutative) anonymous variables of sort S at the top of the rule, one on each side of the read-only pattern, to account for the fact that the rule may match in a context. The associativity will be resolved in two steps. The first step, described here, is that for each rule containing a variable l of a sort S constructed with an associative operator $\_ \star \_$ and for each natural number $n \geq 2$, a rule in which l is replaced by $l_1 \star l_2 \star \cdots \star l_n$ must be added to the existing rules. Continuing our example above, the matching pattern instances associated to $\langle \sigma\ x \mapsto v \rangle_{\mathsf{state}}$ would now be (including the one from desugaring the unit axiom): $\langle x \mapsto v \rangle_{\mathsf{state}}$, $\langle \sigma\ x \mapsto v \rangle_{\mathsf{state}}$, $\langle \sigma_1\ \sigma_2\ x \mapsto v \rangle_{\mathsf{state}}$, and so on.

3. Resolving Commutativity For each occurrence of a subterm $\star(t_1, t_2)$ in a rule, with $\star$ being commutative, add (if it doesn't already exist) a rule in which $\star(t_1, t_2)$ is replaced by $\star(t_2, t_1)$, effectively generating all permutations for terms built with AC operators. The patterns above are enriched to the following patterns: $\langle x \mapsto v \rangle_{\mathsf{state}}$, $\langle \sigma\ x \mapsto v \rangle_{\mathsf{state}}$, $\langle x \mapsto v\ \sigma \rangle_{\mathsf{state}}$, $\langle \sigma_1\ \sigma_2\ x \mapsto v \rangle_{\mathsf{state}}$, $\langle \sigma_1\ x \mapsto v\ \sigma_2 \rangle_{\mathsf{state}}$, $\langle x \mapsto v\ \sigma_1\ \sigma_2 \rangle_{\mathsf{state}}$ (and the ones equivalent to them modulo renamings of the fresh variables), and so on.

Next step is only needed if associative operators are handled as ordinary binary operations when representing the term. If, for example, associative operations are represented as operations with a variable number of arguments this step may be skipped. However, we here prefer to keep this step in order to preserve the algebraic structure of the terms.

4. Resolving Associativity For each rule containing subterms of the form $t_1 \star t_2 \star \cdots \star t_n$ where $\star$ is an associative operator, generate rules containing all possible ways to put parentheses so that each occurrence of $\_ \star \_$ has only two arguments. Note that matching terms containing subterms built from new variables using only

$\star$ these rules need not be considered, as they will be equivalent with rules containing just one new variable instead of that subterm. Keeping this in mind, the following patterns are the final concrete patterns associated to the ones presented above: $\langle x \mapsto v \rangle_{\mathsf{state}}$, $\langle \sigma\ x \mapsto v \rangle_{\mathsf{state}}$, $\langle x \mapsto v\ \sigma \rangle_{\mathsf{state}}$, $\langle \sigma_1\ (\sigma_2\ x \mapsto v) \rangle_{\mathsf{state}}$, $\langle (\sigma_1\ x \mapsto v)\ \sigma_2 \rangle_{\mathsf{state}}$, $\langle \sigma_1\ (x \mapsto v\ \sigma_2) \rangle_{\mathsf{state}}$, and $\langle (x \mapsto v\ \sigma_1)\ \sigma_2 \rangle_{\mathsf{state}}$, and so on.

Note that, since now parts of the original variables might be grouped together with other parts of the matching pattern, parenthesizing makes virtually impossible to rewrite those variables, or even associative operators, unless the entire list is being rewritten. Therefore, we require that for each sort S containing an associative operation $\star$, and any variable l of sort S, whenever $\star$ or l appears at top in a term to be replaced, i.e., $t = \mathrm{L}[\Box]$, $\Box$ must not be an argument of $\star$ in the read-only pattern p. The restriction concerning l may indeed inhibit parallelism when rewriting list variables. However, this situation does not seem to be very common in practice; in particular, it does not appear in any of our current definitions using K. However, the restriction concerning $\star$ is not as big of a concern, as it can be satisfied in two ways. The first one is to push the hole $\Box$ up in the term as long as $\star$ operations are on top of it, and thus inhibit the parallelism. The second, is to push the hole down, by moving $\star$ into the pattern, and splitting $\Box$ into two new holes, requiring L to map them to the two arguments of $\star$, and updating R accordingly (including the possibility that R maps one of the holes to $\cdot$, while the other to $\mathrm{R}(\Box)$).

Example: Matching the IMP++ variable read/write rules Let us now show that interpreting K rules as rule schemata as described above allows multiple concurrent matching instances. Recall the two rules for reading/writing a variable in the store (in their K form):

$$\rho_r\colon \langle \underset{i}{\underline{x}} \curvearrowright k \rangle_{\mathsf{k}}\ \langle \sigma\ x \mapsto i \rangle_{\mathsf{state}} \quad \Big| \quad \rho_w\colon \langle \underset{\cdot}{\underline{x = i'\ ;}} \curvearrowright k \rangle_{\mathsf{k}}\ \langle \sigma\ x \mapsto \underset{i'}{\underline{i}} \rangle_{\mathsf{state}}$$

Note that both rules have the bag constructor at top, and thus are considered to have a bag variable, say b, at top, as well. Recall also the ground termed to be matched, this time parenthesized: $((\langle a \rangle_{\mathsf{k}}\ \langle a \rangle_{\mathsf{k}})\ \langle b = 3\ ; \rangle_{\mathsf{k}})\ \langle a \mapsto 1\ b \mapsto 2 \rangle_{\mathsf{state}}$. Then, the three concretizations of the 2 schema rules above which can match this term are: $\rho_{w,1}\colon (b_1\ \langle \underset{\cdot}{\underline{x = i'\ ;}} \rangle_{\mathsf{k}})\ \langle \sigma_1\ x \mapsto \underset{i'}{\underline{i}} \rangle_{\mathsf{state}}$, $\rho_{r,1}\colon ((\langle \underset{i}{\underline{x}} \rangle_{\mathsf{k}}\ b_1)\ b_2)\ \langle x \mapsto i\ \sigma_1 \rangle_{\mathsf{state}}$, and $\rho_{r,2}\colon ((b_1\ \langle \underset{i}{\underline{x}} \rangle_{\mathsf{k}})\ b_2)\ \langle x \mapsto i\ \sigma_1 \rangle_{\mathsf{state}}$.

C Definitions of $graph(t)$ and $term^{\mathrm{H}}(v)$

Figure 4 exhibits on the top the K graph representation of rules (1)–(4), from the motivating example presented at the beginning of the section, and at the bottom, the graph transformation using all four rules combined to rewrite the graph representation of $h(f(a),0,1)$ (graph G) to that of $h(g(b),1,0)$ (graph H). For example the representation of variable x can be observed as the (singleton) graph R for rule (4), the constants a and b as graphs L and R from rule (3), and the term $f(x)$ as graph L in

rule (4). Note that we take full advantage of not having to specify outward edges for all sorts in the signature; for example, the number of outward edges specified for the node labeled with h have all possible values between 3 (its normal arity) in graphs G and H, to 0, e.g., in graph K for rule (1).

This representation basically associates two nodes and two edges for each non-empty, non-variable position α in a term. The first node is an operation node (v_α), labelled by the operation symbol at position α in the term; e.g., in graph G from Figure 4, the node labeled by a is v_{11}, as it corresponds to position 11 in the term $h(f(a),0,1))$, while the node labeled by f is v_1. The second node is a sort node ($v_{\overline{\alpha}}$) labeled by the result sort of the operation symbol; to continue with the example above, $v_{\overline{11}}$ is the node labeled with s above v_{11}). The first edge (e_α) is an unlabeled edge linking the two nodes presented above, having as source the sort node and as target the operation node; that is, e_{11} is the edge between $v_{\overline{11}}$ and v_{11}. Supposing that $\alpha = \beta i$ (it is non-empty), then the second edge ($e_{\overline{\alpha}}$) links the operation node v_β (corresponding to the preceding position in the term) to $v_{\overline{\alpha}}$, and it is labeled by i the increment from position β to α; for our example, edge $e_{\overline{11}}$ is the edge labeled by 1 linking node v_1 (labeled by f) to node $v_{\overline{11}}$ (labeled by s). Empty positions (ϵ) are treated similarly as above, but, since they correspond to top symbols in the terms, and thus do not have any operation symbols above them, the edge $e_{\overline{\epsilon}}$ does not exist; returning to our example, in graph G, v_ϵ is labeled by h, $v_{\overline{\epsilon}}$ is the node labeled by s above v_ϵ, and e_ϵ is the unalbeled edge between them. Non-anonymous variables are assigned unique sort nodes, identified by their names and labeled by their sort, and for each position $\alpha = \beta i$ of a variable x, the edge $e_{\overline{\alpha}}$ is labeled by i and links the operation node v_β, corresponding to position β, to sort node x; for example, the graph L from rule (1) in Figure 4 contains the node x which is labeled by s which is linked to node v_ϵ (labeled by h) through edge $e_{\overline{1}}$, labeled by 1. Anonymous variables (as well as variables representing context holes) do not generate either nodes or edges, and they are the reasons for the operation nodes having incompletely specified outward edges according to their arity. Next definition summarizes this representation of terms as graphs.

Definition 7 (from terms to graphs)**.** *Given a* Σ*-term* t *and a set of "anonymous" variables* X*, the* ***term-graph associated to*** t***, omitting the variables from*** X*, written* $graph^{\mathrm{X}}(t)$ *is the* Σ*-term-graph* G*, defined for non-variable terms by:*

- $\mathrm{V} = \{\alpha, \overline{\alpha} \mid \alpha$ *position in* t*, and* $t{\restriction_\alpha} \notin vars(t)\} \cup vars(t) \setminus \mathrm{X}$*;*
- $\mathrm{E} = \mathrm{V} \setminus vars(t) \setminus \{\overline{\epsilon}\} \cup \{\overline{\alpha} \mid \alpha$ *position in* t*, and* $t{\restriction_\alpha} \in vars(t) \setminus \mathrm{X}\}$*;*
- $\mathrm{source}(\overline{\alpha i}) = \alpha$*, and* $\mathrm{source}(\alpha) = \overline{\alpha}$*;*
- $\mathrm{target}(\alpha) = \alpha$ *and* $\mathrm{target}(\overline{\alpha}) = \begin{cases} t{\restriction_\alpha}, \textit{ if } t{\restriction_\alpha} \in vars(t) \\ \overline{\alpha}, \textit{ otherwise} \end{cases}$*;*
- $\mathrm{lv}(\alpha) = symbol(t{\restriction_\alpha})$*,* $\mathrm{lv}(\overline{\alpha}) = sort(t{\restriction_\alpha})$*, and* $\mathrm{lv}(x) = sort(x)$*;*
- $\mathrm{le}(\overline{\alpha i}) = i$*, and* $\mathrm{le}(\alpha) = \epsilon$.

Given a variable, say x, $graph^{\mathrm{X}}(x) = \begin{cases} \{x : sort(x)\},\ \textit{if } x \notin \mathrm{X} \\ \{\epsilon : sort(x)\},\ \textit{if } x \in \mathrm{X} \end{cases}$.

When $\mathrm{X} = \emptyset$ *we simply write* $graph(t)$ *for* $graph^{\emptyset}(t)$.

Since the variables have unique ids, this representation, similar to that of term-graphs and jungles, shares multiple occurrences of the same variable.

Moreover, although a naming scheme for nodes and edges was required for formally defining the representation, node names do not need to be specified (except for variables) when depicting the graph associated to a term. Besides the fact that the graphical intuition allows them to be easily inferred, we can easily name a the node or an edge by finding its corresponding position in the term, which can be found by concatenating the labels of the edges on the path from the top of the graph to it; for example node labeled by 0 in graph R of rule (1) must be in position 3 in the term represented, and thus its name should be v_3, and the name of the edge from node labeled by g to node x (in the same graph) is $e_{\overline{11}}$, as the path from the top its target indicates position 11.

Although some nodes labeled with operation symbols might not have all outward edges according to their arity specified, we can still recover a term from any term-graph, by using (anonymous) variables to fill the unspecified arguments of an operation. Given a term-graph G and a sort node v, there exists a unique term (up to variable renaming) t associated to the subgraph $\mathrm{G}\restriction_v$, defined as follows:

Definition 8 (from nodes to terms)**.** *Let* G *be a term-graph over* $\Sigma = (\mathrm{S}, \mathrm{F})$. *The* S*-indexed function* $term^{\mathrm{G}}_s : \mathrm{lv}^{-1}(s) \to \mathrm{T}_{\Sigma,s}(\mathrm{TVARS}_(\mathrm{G}))$ *associates to each node labelled by s a* Σ*-term of sort s with variables from* $\mathrm{TVARS}_(\mathrm{G})$*, defined as follows:*

- $term_{\mathrm{G}}(v) = v$, *if* $v \in \mathrm{VAR}_{\mathrm{G}}$*;*
- $term_{\mathrm{G}}(v) = f(term^1_{s_1}(v'), \cdots term^n_{s_n}(v'))$, *if* $\mathrm{lv}(v) = s$, *e is the only edge s.t.* $s(e) = v$, $v' = t(e)$, $\mathrm{lv}(v') = f : s_1 \cdots s_n \to s$, *and* $term^i_s$ *is defined by*

$$term^i_s(v) = \begin{cases} term_s(v'),\ \textit{if } t(s^{-1}(v) \cap \mathrm{le}^{-1}(i)) = \{v'\} \\ x_{v,i},\ \textit{if } t(s^{-1}(v) \cap \mathrm{le}^{-1}(i)) = \emptyset \end{cases}$$

According to the definition above, and assuming the graphs in Figure 4 are constructed from terms using Definition 7, the term corresponding to the graph K of rule (3) is ϵ, where ϵ is a variable of sort s, while the term corresponding to the graph L of rule (2) is $h(x_{\epsilon,1}, 0, x_{\epsilon,3})$, where $x_{\epsilon,1}$ is a variable of sort s and $x_{\epsilon,3}$ is a variable of sort *int*.

D Defining reflective features in $\mathbb{K}$

D.1 Generic AST Visitor

MODULE K-VISITOR
  IMPORTS K-WRAPPERS+K
  K::= apply *KLabel* to *Set* in *List{K}*

| $K \curvearrowright K$ [strict]
| $K_{\square} K$ [strict]
| *unbox*(*K*) [strict]
KResult ::= $\boxed{List\{K\}}$
KLabel ::= $\boxed{KLabel}$

K RULES:
apply *A* to *Labels Label* in *Label* (*Kl*) ⇒ *A* (*Label* (*Kl*))
apply *A* to *Labels* in *Label* (*Kl*) ⇒ $\boxed{Label}$ (apply *A* to *Labels* in *Kl*) when $\texttt{not}_{Bool}$ *Label* in *Labels*
apply *A* to *Labels* in · ⇒ ▫
apply *A* to *Labels* in $K_1 \curvearrowright K_2$ ⇒ apply *A* to *Labels* in $K_1 \curvearrowright$ apply *A* to *Labels* in K_2 when $K_1 =/=_{Bool} \cdot \texttt{and}_{Bool} K_2 =/=_{Bool} \cdot$
apply *A* to *Labels* in $\cdot List\{K\}$ ⇒ $\boxed{\cdot List\{K\}}$
apply *A* to *Labels* in K_1 „ *NeKl* ⇒ apply *A* to *Labels* in $K_1 \square$ apply *A* to *Labels* in *NeKl*
CONTEXT: $\boxed{Label}$ (_ „ □ „ _)
$unbox(\boxed{K}) \Rightarrow K$
$\boxed{Label}$ ($\boxed{Kl}$) ⇒ $\boxed{Label\ (\ Kl\ )}$
$\boxed{Kl} \square \boxed{Kl'} \Rightarrow \boxed{Kl \text{ „ } Kl'}$
$\boxed{K_1} \curvearrowright \boxed{K_2} \Rightarrow \boxed{K_1 \curvearrowright K_2}$
END MODULE

D.2 Generic substitution

MODULE SUBSTITUTION
IMPORTS K-VISITOR+PL-INT+PL-ID
K ::= *Id* | *Nat*
| [*Id*] *K*
| *K* [*K* / *Id*]
| *K*[$K^{\square}$ / *Id*]
| $eval(K)[K^{\square} / Id]$ [strict(1)]
KLabel ::= [$K^{\square}$ / *Id*]
Id ::= id_{Nat}
INITIAL CONFIGURATION:
$\langle \cdot K \rangle_{\mathsf{k}}$ $\langle 0 \rangle_{\mathsf{nextId}}$

K RULES:
$\langle \frac{\mathrm{K}'\ [\ K\ /\ Y\ ]}{unbox(\mathrm{K}'[K^{\square} / Y])} \rangle_{\mathsf{k}}$
$[K^{\square} / Y]$ (*Y*) ⇒ $\boxed{K}$
$\langle \frac{[K^{\square} / Y]\ (\ [\ X\ ]\ \mathrm{K}'\ )}{\boxed{[\_]\_}\ (\ \boxed{id_N}_{\square} eval(\mathrm{K}'[id_N{}^{\square} / X])[K^{\square} / Y]\ )} \rangle_{\mathsf{k}}$ $\langle \frac{N}{\mathtt{s}_{Nat}\ N} \rangle_{\mathsf{nextId}}$
$eval(\boxed{\mathrm{K}'})[K^{\square} / Y] \Rightarrow \mathrm{K}'[K^{\square} / Y]$
MACRO: $\mathrm{K}'[K^{\square} / Y] =$ apply $[K^{\square} / Y]$ to [_] _ $get\mathrm{KL}abel(Y)$ in K′
END MODULE

D.3 Code generation

MODULE QUOTE-UNQUOTE
IMPORTS PL-NAT+K-VISITOR
K ::= `quote` K
| `unquote` K
| `lift` K [strict]
| `eval` K [strict]
| *quote*(K, *Nat*)
KLabel ::= $quote_{Nat}$

K RULES:
$\langle \frac{\texttt{quote}\ K}{quote(K,0)}\ \_ \rangle_{\texttt{k}}$
$quote_N$ (`quote` K) ⇒ `quote_` (*quote*(K, $\texttt{s}_{Nat}\ N$))
$quote_0$ (`unquote` K) ⇒ K
$quote_{\texttt{s}_{Nat}\ N}$ (`unquote` K) ⇒ `unquote_` (*quote*(K, N))
`lift` V ⇒ V
`eval` K ⇒ K
MACRO: *quote*(K, N) = apply $quote_N$ to `quote_` `unquote_` in K
END MODULE