submitted to the Journal of Logic and Algebraic Programming

An Overview of the K Semantic Framework

Grigore Roșu, Traian Florin Șerbănuță

Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N Goodwin, Urbana, IL 61801, USA

Abstract

K is 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*. Configurations organize the system/program state in units called cells, which are labeled and can be nested. Computations carry "computational meaning" as special nested list structures sequentializing computational tasks, such as fragments of program; in particular, computations extend the original language or calculus syntax. K (rewrite) rules generalize conventional rewrite rules by making it explicit which parts of the term they read-only, write-only, or do not care about. This distinction makes K a suitable framework for defining truly concurrent languages or calculi even in the presence of sharing. Since computations can be handled like any other terms in a rewriting environment, that is, they can be matched, moved from one place to another in the original term, modified, or even deleted, K is particularly suitable for defining control-intensive language features such as abrupt termination, exceptions or call/cc.

This paper gives an overview of the K framework: what it is, how it can be used, and where it has been used so far. It also proposes and discusses the K definition of Challenge, a programming language that aims at challenging and exposing the limitations of the various existing semantic frameworks.

1. Introduction

This paper is a gentle introduction to K, a rewriting-based semantic definitional framework. K was introduced by the first author in the lecture notes of a programming language design course at the University of Illinois at Urbana-Champaign (UIUC) in Fall 2003 [1], as a means to define executable concurrent languages in rewriting logic using the Maude executable specification system [2]. Since 2003, K has been used continuously in teaching programming languages at UIUC, in seminars in Spain and Romania, as well as in several research initiatives. A more formal description of K can be found in [3, 4].

The introduction and development of K was largely motivated by the observation that after more than 40 years of systematic research in programming language semantics, the following important (multi-)question remains largely open to the working programming language designer, and not only:

Is there any language definitional framework that, at the same time,

1. *Gives a strong intuition, even precise recipes, on how to define a language?*
2. *Same for language-related definitions, such as type checkers, type inferencers, abstract interpreters, safety policy or domain-specific checkers, etc.?*
3. *Can define arbitrarily complex language features, including, obviously, all those found in existing languages, capturing also their intended computational granularity?*
4. *Is modular, that is, adding new language features does not require to modify existing definitions of unrelated features? Modularity is crucial for scalability and reuse.*
5. *Supports non-determinism and concurrency, at any desired granularity?*
6. *Is generic, that is, not tied to any particular programming language or paradigm?*
7. *Is executable, so one can "test" language or formal analyzer definitions, as if one already had an interpreter or a compiler for one's language? Efficient executability of language definitions may even eliminate the need for interpreters or compilers.*

Email addresses: `grosu@illinois.edu` (Grigore Roșu), `tserban2@illinois.edu` (Traian Florin Șerbănuță)

8. *Has state-exploration capabilities, including exhaustive behavior analysis (e.g., finite-state model-checking), when one's language is non-deterministic or/and concurrent?*
9. *Has a corresponding initial-model (to allow inductive proofs) or axiomatic semantics (to allow Hoare-style proofs), so that one can formally reason about programs?*

The list above contains a *minimal* set of desirable features that an ideal language definitional framework should have. There are additional desirable, yet of a more subjective nature and thus harder to quantify, requirements of an ideal language definitional framework. For example, it should be simple and easy to understand, teach and use by mainstream enthusiastic language designers, not only by language experts —in particular, an ideal framework should not require its users to have advanced concepts of category theory, logics, or type theory, in order to use it. Also, it should have good data representation capabilities and should allow proofs of theorems about programming languages that are easy to comprehend. Additionally, a framework providing support for parsing programs directly in the desired language syntax may be desirable to one requiring the implementation of an additional, external to the definitional setting, parser.

The nine requirements above are nevertheless ambitious. Some proponents of existing language definitional frameworks may argue that their favorite framework has these properties; however, a careful analysis of existing language definitional frameworks reveals that they actually fail to satisfy some of these ideal features (detailed discussions on how the current approaches fail to satisfy the requirements above can be found in [3, 4] and to some extent in Section 2). Others may argue that their favorite framework has some of the properties above, the "important ones", declaring the other properties either "not interesting" or "something else". For example, one may say that what is important in one's framework is to get a dynamic semantics of a language, but its (model-theoretical) algebraic denotational semantics, proving properties about programs, model checking, etc., are "something else" and therefore are allowed to need a different "encoding" of the language. Our position is that an ideal language definitional framework should not compromise any of the nine requirements above.

Whether K satisfies all the requirements above or not is, and probably will always be, open. What we can mention with regards to this aspect, though, is that K was motivated and stimulated by the observation that the existing language definitional frameworks fail to fully satisfy these minimal requirements; consequently, K's design and development were conducted aiming *explicitly* to fulfill all nine requirements discussed above, promoting none of them at the expense of others.

The K framework consists of two components:

1. The *K concurrent rewrite abstract machine*, abbreviated KRAM and discussed in Section 4;
2. The *K technique*, discussed in Section 5.

Like in term rewriting, a K-system consists of a signature for building terms and of a set of rules for iteratively rewriting terms. Like in rewriting logic [5], K rules can be applied concurrently and unrestricted by context. The novelty of the KRAM (Section 4) is that its rules contain, besides expected information saying how the original term is modified (the *write data*), also information about what parts of the term are shared with other rules (the *read-only data*). This additional information allows K to be a truly concurrent definitional framework. The K concurrent rewrites associated to a K-system may require several interleaved rewrites in the rewrite logic theory straightforwardly (i.e., by forgetting the read-only information) associated to the K-system.

Even though the KRAM aims at maximizing the amount of concurrency that can be achieved in a rewriting setting, it does not tell *how* one can define a programming language or a calculus as a K-system. In particular, a bad K definition may partially or totally inhibit KRAM's potential for concurrency. The K technique discussed in Section 5 proposes an approach and supporting notation to make the use of the KRAM convenient when formally defining programming languages and calculi. Moreover, the K technique can also be and actually has already been intensively used as a technique to define languages and calculi as term rewrite systems or as rewrite logic theories. There is one important thing lost in the translation of K into rewriting logic though, namely the degree of true concurrency of the original K definition. Ignoring this true concurrency aspect, the relationship between "serializable" K and rewriting logic in general and Maude in particular is the same as that between any of the conventional semantic styles discussed in [6] and rewriting logic and Maude: the latter can be used to execute and analyze K definitions.

This paper is structured as follows. Section 2 discusses related programming language semantic approaches, their limitations, and how they relate to K. Section 3 discusses the K framework intuitively, by means of defining a simple imperative language and an extension of it; this section should give the reader quite a clear feel for what K is about and how it operates. Section 4 describes the K concurrent rewrite abstract machine (Kram) and is the most technical of this paper; however, it formalizes a relatively intuitive process, so the reader interested more in using K then in the technical details of its core concurrent rewriting machinery can safely skip it. Section 5 presents the K technique, explaining essentially how the Kram or other rewrite infrastructures can be used to define programming language semantics by means of nested-cell configurations, computations anf rewrite rules. Section 6 shows K at work: it introduces and shows how to give a K semantics to Challenge, a programming language containing varied features known to be problematic to define in some frameworks. Challenge was conceived as a means to provoke the various language definitional frameworks and to expose their limitations. Finally, Section 7 concludes the paper, lists some recent research based on K, and gives some future directions.

2. Existing Approaches and Their Limitations for Programming Language Semantics

Since K was motivated by the need for a semantic framework satisfying the requirements discussed in Section 1, it is important to understand the limitations of the existing approaches as semantic frameworks for programming languages. In this section we group the relevant existing approaches into two categories: structural operational semantics frameworks and rewriting frameworks. For each approach we list its main characteristics, its limitations, and its relationship with K.

2.1. Structural Operational Semantics (SOS) Frameworks

We here discuss the most common SOS approaches, namely big-step and small-step SOS, their modular variant MSOS, and reduction semantics with evaluation contexts.

Big-Step SOS. Introduced as natural semantics in [7], also named relational semantics[8], or evaluation semantics, big-step semantics is "the most denotational" of the operational semantics. One can view big-step definitions as definitions of functions interpreting each language construct in an appropriate domain. Big-step operational semantics can be easily and efficiently interpreted in any recursive, functional or logical framework. It is particularly useful for defining type systems.
Limitations: Due to its monolithic, single-step evaluation, it is hard to debug or trace big-step semantic definitions. If the program is wrong, no information is given about where the failure occurred. Divergence is not observable in the evaluation relation. It may be hard or impossible to model concurrent features. It is not modular, e.g., to add side effects to expressions, one must redefine the rules to allow expressions to evaluate to pairs (value-store). It is inconvenient (and non-modular) to define complex control statements.
Relationship to K: None.

Small-step SOS. Introduced by Plotkin in [9], also called transition semantics or reduction semantics, small-step semantics captures the notion of one computational step. Therefore, it stops at errors, pointing them out and it is easy to trace and debug. It gives interleaving semantics for concurrency.
Limitations: Like big-step, it is non-modular. It does not give a "true concurrency" semantics, that is, one has to choose a certain interleaving (no two rules can be applied on the same term at the same time), mainly because reduction is forced to occur only at the top. It is still hard to deal with control — one has to add corner cases (additional rules) to each statement to propagate control changing information. Each small step traverses the entire program to find the next redex; since the size of the program can grow unbounded, each small step may take unbounded resources in the worst case.
Relationship to K: None.

Modular SOS. Mosses introduced *modular SOS (MSOS)* [10, 11] to deal with the non-modularity of small-step and big-step SOS. The MSOS solution involves moving the non-syntactic state components as labels on transitions, plus a discipline of only selecting needed attributes from the states.
Limitations: Same as the ones above, except for modularity.
Relationship to K: Both MSOS and K make use of labeled information to achieve modularity. MSOS uses labels as record fields on the transition relation, while K uses labels as cell names in the configuration. In both MSOS and K one can use the labels in semantic rules to only refer to those items of interest.

Reduction semantics with evaluation contexts. Introduced by Felleisen and colleagues (see, e.g., [12]), the evaluation contexts style improves over small-step SOS in two ways: (1) it gives a more compact semantics to context-sensitive reduction, by using parsing to find the next redex rather than small-step rules; and (2) it provides the possibility of also modifying the context in which a reduction occurs, making it much easier to deal with control-intensive features. Additionally, one can also incorporate the configuration as part of the evaluation context, and thus have full access to semantic information "by need".
Limitations: It still only allows "interleaving semantics" for concurrency. It is too "rigid to syntax", in that it is hard or impossible to define semantics in which values are more than plain syntax; for example, one cannot give a syntactic semantics to a functional language based on closures for functions (instead of substitution), because one needs special, non-syntactic means to handle and recover environments (as we do in K, see Section 6). Although context-sensitive rewriting might seem to be easily implementable by rewriting, in fact one has to perform an amount of "parsing" work linear in the size of the program for each computational step, like in small-step SOS. However, one might obtain efficient implementations for restricted forms of context-reduction definitions by applying refocusing techniques [13].
Relationship to K: Both reduction semantics and K make use of evaluation contexts and can store or modify them. Reduction semantics relies on a mechanism to "split/plug" syntax into contexts, while K relies on a mechanism to "heat/cool" syntax into computations. The former is achieved by an implicit "advanced" parsing of syntax into a context and a redex, while the latter is achieved using rewrite rules.

2.2. Rewriting Frameworks

We here discuss some rewriting approaches which have been used or have the potential to be used for programming language semantics.

Term Rewriting and Rewriting Logic. Rewriting logic was proposed by Meseguer [5] as a logical formalism generalizing both equational logic and term rewriting, so it more suitable for language semantics than plain term rewriting. The rewriting logic semantics (RLS) project [14, 15] is an initiative aiming at using rewriting logic for various semantic aspects of programming languages; a substantial body of work on RLS, both of theoretical and practical nature, is discussed in [14, 15], which we do not repeat here. It is worth mentioning that all the various SOS semantic approaches in Section 2.1 (and others not discussed here) can be framed as particular definitional styles in rewriting logic and then executed using Maude [6].
Limitations: Rewriting logic is a general-purpose computational framework, which provides the infrastructure but *no hints on how* to define a language semantics as a rewrite system. Definitional styles need to be represented within rewriting as shown in [6], such as those in Section 2.1. Additionally, rule instances cannot overlap, which results in an enforcement of interleaving in some situations where one would like to have true concurrency. For example, two threads reading from a shared store have to interleave the store operations even if they act on different locations, because the corresponding rule instances overlap on the store; some hints are given in [16] on how to use equations in an ingenious way to still achieve theoretical true concurrency in the presence of sharing, but that approach is impractical for language semantics.
Relationship to K: Like rewriting logic, K is a computational framework that aims at maximizing the amount of concurrency that can be achieved by means of term rewriting. However, one K concurrent rewrite step may take several interleaved rewriting logic steps.

Graph Rewriting. Graph rewriting [17] extends the intuitions of term rewriting to arbitrary graphs, by developing a match-and-apply mechanism which can work directly on graphs.
Limitations: Graph rewriting is rather complex and we are not aware of works using it for programming language semantics. Part of the reason could be the notorious difficulty of graph rewriting to deal with structure copying and equality testing, operations which are crucial for programming language semantics and relatively easy to provide by term rewriting systems. Finally, in spite of decades of research, at our knowledge there are still no graph rewriting engines comparable in performance to term rewrite engines.
Relationship to K: The K rules, like those in graph-rewriting, have both read-only components, which could be shared among rule instances to maximize concurrency, and write-only components.

The Chemical Abstract Machine. Berry and Boudol's chemical abstract machine [18], or CHAM, is both a rewrite-based model of concurrency and a specific style of giving language semantics. The CHAM views a distributed state as a "solution" in which many "molecules" float, and understands concurrent transitions as "reactions" that can occur simultaneously in many points of the solution.

Limitations: While chemistry as a model of computation sounds attractive, technically speaking the CHAM is in fact a restricted case of rewriting logic, as noticed in [5]. Moreover, some of the chemical "intuitions", such as the airlock operation which imposes a particular chemical "discipline" to access molecules inside a solution, inhibit the potential for the now well-understood and efficient matching and rewriting *modulo associativity and commutativity.* In fact, at our knowledge, there is no competitive implementation of the CHAM. Although this solution-molecule paradigm seems to work for languages with simple state structure, it is not clear how one could represent the state for complex languages with threads, locks, environments, etc. Finally, CHAMs provide no mechanism to freeze the current molecular structure as a "value", and then to store or retrieve it, as we would need to define language features like call/cc. It would therefore seem hard to define complex control-intensive language features in CHAM.
Relationship to K: Like the CHAM, K also organizes the configuration of the program or system as a potentially nested structure of molecules (called cells and potentially labeled in K). Like in CHAM, the configuration is also rewritten until it "stabilizes". Unlike in CHAM, the K rules can match and write inside and across multiple cells in one parallel step. Also, unlike in CHAM (and in rewriting logic), K rewrite rules can apply concurrently even in cases when they overlap (read-only structure).

P-Systems. Păun's membrane systems (or P-systems) [19] are computing devices abstracted from the structure and the functioning of the living cell. In classical *transition P-systems*, the main ingredients of such a system are the *membrane structure*, in the compartments of which *multisets* of symbol-objects evolve according to given *evolution rules.* The rules are localized, associated with the membranes (hence, with the compartments), and they are used in a *nondeterministic maximally parallel* manner.
Limitations: When looked at from a programming language semantics perspective, the P-systems have a series of limitations that have been addressed in K, such as: (1) Lack of structure for non-membrane terms, which are plain constants; and (2) Strict membrane locality (even stricter than in the CHAM), which prevents rules to match and rewrite within multiple membranes at the same time. Regarding (1), programming languages often handle complex data-structures or make use of complex semantic structures, such as environments mapping variables to values, or closures holding code as well as environments, etc., which would be hard or impossible to properly encode using just constants. Regarding (2), strict membrane locality would require one to write many low-level rules and introduce and implement by local rules artificial "cross-membrane communication protocols". For example, the semantics of variable lookup involves acquiring the location of the variable from the environment cell (which holds a map), then the value at that location in the store cell (which holds another map), and finally the rewrite of the variable in the code cell into that value. All these require the introduction of encoding constants and rules in several membranes and many computation steps; compare that to the one step K rule in Section 6.
Relationship to K: K-systems share with P-systems the ideas of cell structures and the aim to maximize concurrency. However, K rules can span across multiple cells at a time, and the objects contained in a cell can have a rich algebraic structure (as opposed to being constants, as in P-systems). [20] presents a in-depth comparison between the two formalisms, showing how one can use K to model P-systems.

3. K Overview by Example

Here we briefly describe the K framework, what it offers and how it can be used. We use as concrete examples the IMP language, a very simple imperative language, and IMP++, an extension of IMP with: (1) increment to exhibit side-effects for expressions; (2) output; (3) halt, to show how K deals with abrupt termination; and (3) spawning of threads, to show how concurrency is handled. We define both an executable semantics and a type system for these languages. The type system is included mainly for demonstration purposes, to show that one can use the same definitional framework, K, to define both formal language semantics and language abstractions. The role of this section is threefold: first, it gives the reader a better understanding of the K framework before we proceed to define it rigorously in the remainder of the paper; second, it shows how K avoids the limitations of the various more conventional semantic approaches; and third, it shows that K is actually easy to use, in spite of the intricate K concurrent abstract machine technicalities discussed in Section 4 — indeed, users of the K framework need not be familiar with all those intricate details, the same way users of a concurrent programming language need not be aware of the underlying architecture of the concurrent machine running their programs. In fact, in this section we make no distinction between the K rewrite abstract machine and the K technique, referring to these collectively as "the K framework", or more simply just "K".

Programming languages, calculi, as well as type systems or formal analyzers can be defined in K by making use of special, potentially nested *(K) cell* structures, and *(K) (rewrite) rules*. There are two types of K rules: *computational rules*, which count as computational steps, and *structural rules*, which do not count as computational steps. The role of the structural rules is to rearrange the term so that the computational rules can apply. K rules are *unconditional* (they can be thought of as rule schemata and may have ordinary side conditions, though), and they are *context-insensitive*, so K rules apply concurrently as soon as they match, without any contextual delay or restrictions.

One sort has a special meaning in K, namely the sort K of *computations*. The intuition for terms of sort K is that they have computational contents, such as programs or fragments of programs have; indeed, computations extend the syntax of the original language. Computations have a list structure with "$\curvearrowright$" (read "followed by") concatenating two computations and "$\cdot$" the empty computation; the list structure captures the intuition of computation sequentialization. Computations give an elegant and uniform means to define and handle evaluation contexts [12] and/or continuations [21]. Indeed, a computation "$v \curvearrowright C$" can be thought of as "$C[v]$, that is, evaluation context C applied to v" or as "passing v to continuation C". Computations can be handled like any other terms in a rewriting environment, that is, they can be matched, moved from one place to another, modified, or even deleted. A term may contain an arbitrary number of computations, which can evolve concurrently; they can be thought of as execution threads. Rules corresponding to inherently sequential operations (such as lookup/assignment of variables in the same thread) must be designed with care, to ensure that they are applied only at the top of computations.

The distinctive feature of K compared to other term rewriting approaches in general and to rewriting logic in particular, is that K allows rewrite rules to apply *concurrently* even in cases when they overlap, provided that they do not change the overlapped portion of the term. This allows for *truly concurrent semantics* to programming languages and calculi. For example, two threads that read the same location of memory can do that concurrently, even though the corresponding rules overlap on the store location being read. The distinctive feature of K compared to other frameworks for true concurrency, like chemical abstract machines [18] or membrane systems [19], is that rewrite rules can match across and inside multiple cells and thus perform changes many places at the same time, in one concurrent step.

K achieves, in one uniform framework, the benefits of both the chemical abstract machines (or CHAMs) and reduction semantics with evaluation contexts, at the same time avoiding what might be called the "rigidity to chemistry" of the former and the "rigidity to syntax" of the latter. Any CHAM and any context reduction definition can be captured in K with minimal (in our view *zero*) representational distance. K can support concurrent language definitions with either an interleaving or a true concurrency semantics. Like the other semantic approaches that can be represented in rewriting logic [6], K can also be represented in rewriting logic and thus K definitions can be executed on existing rewrite engines, thus providing "interpreters for free" directly from formal language definitions; additionally, general-purpose formal analysis techniques and tools developed for rewriting logic, such as state space exploration for safety violations or model-checking, give us corresponding techniques and tools for the defined languages, at no additional development cost. Unlike the other operational semantic approaches whose representations in rewriting logic are *faithful*, in that the resulting rewriting logic theories are step-for-step equivalent with the original definitions, K cannot be captured faithfully by rewriting logic in any natural way; more precisely, the resulting rewriting logic theory may need more interleaved steps in order to capture one concurrent step in the original K definition.

3.1. K Semantics of IMP

Figure 1 shows the complete K definition of IMP, except for the configuration; the IMP configuration is explained separately below. The left column in Figure 1 gives the IMP syntax. The middle column contains special syntax K annotations, called strictness attributes, stating the evaluation strategy of some language constructs. Finally, the right column gives the semantic rules.

K makes intensive use of the algebraic CFG notation, to define configurations, in particular of list, set, multiset and map structures. For example, $\mathsf{List}[S]$ defines comma-separated lists of elements of type S, and could be expressed with the CFG rule $\mathsf{List}[S] ::= \cdot \mid S(,S)^*$, with the additional understanding that '$\_,\_$' is associative and $\cdot$ satisfies the unit axioms.

Like in the CHAM, program or system configurations in K are organized as potentially nested structures of *cells* (we call them cells instead of molecules to avoid confusion with terminology in CHAM and chemistry). However, unlike the CHAM which only provides multisets (or bags), K also provides list, set and map cells in addition to multiset (called bag) cells; K's cells may be labeled to distinguish them from each other. We use angle brackets as cell wrappers. The K configuration of IMP can be defined as:

$$\mathit{Configuration}_{\mathsf{IMP}} \equiv \langle\langle K\rangle_{\mathsf{k}}\ \langle \mathsf{Map}[\mathit{VarId} \mapsto \mathit{Int}]\rangle_{\mathsf{state}}\rangle_{\top}$$

In words, IMP configurations consist of a top cell $\langle ...\rangle_{\top}$ containing two other cells inside: a cell $\langle ...\rangle_{\mathsf{k}}$ which holds a term of sort K (terms of sort K are called computations and extend the original language syntax as explained in the next paragraph) and and a cell $\langle ...\rangle_{\mathsf{state}}$ which holds a map from variables to integers. For example, "$\langle\langle \mathtt{x := 1;\ y := x+1}\rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{state}}\rangle_{\top}$" is a configuration holding program "`x := 1; y := x+1`" and empty state, and "$\langle\langle \mathtt{x := 1;\ y := x+1}\rangle_{\mathsf{k}}\ \langle \mathtt{x} \mapsto 0, \mathtt{y} \mapsto 1\rangle_{\mathsf{state}}\rangle_{\top}$" is a configuration holding the same program and a state mapping `x` to 0 and `y` to 1. When we add threads (in IMP++), the configurations can hold multiple $\langle ...\rangle_{\mathsf{k}}$ cells (its bag structure allows that).

K provides special notational support for *computational structures*, or simply *computations*. Computations have the sort K, which is therefore builtin in the K framework; the intuition for terms of sort K is that they have computational contents, such as, for example, a program or a fragment of program has. Computations extend the original language/calculus/system syntax with special "$\curvearrowright$"-separated lists "$T_1 \curvearrowright T_2 \curvearrowright \cdots \curvearrowright T_n$" comprising *(computational) tasks*, thought of as having to be "processed" sequentially ("$\curvearrowright$" reads "followed by"). The identity of the "$\curvearrowright$" associative operator is "$\cdot$". Like in context reduction, K allows one to define evaluation contexts over the language syntax. However, unlike in context reduction, parsing does not play any crucial role in K, because K replaces the hard-to-implement split/plug operations of context reduction by plain, context-insensitive rewriting. Therefore, instead of defining evaluation contexts using context-free grammars and relying on splitting syntactic terms (via parsing) into evaluation contexts and redexes, in K we define evaluation contexts using special rewrite rules. For example, the evaluation contexts of sum, comparison and conditional in IMP can be defined as follows, by means of *structural rules* (recall that the sum "`+`" was non-deterministic and the comparison "`<=`" was sequential):

$$\begin{array}{rcl} a_1 \mathtt{+} a_2 & \rightleftharpoons & a_1 \curvearrowright \square \mathtt{+} a_2 \\ a_1 \mathtt{+} a_2 & \rightleftharpoons & a_2 \curvearrowright a_1 \mathtt{+} \square \\ a_1 \mathtt{<=} a_2 & \rightleftharpoons & a_1 \curvearrowright \square \mathtt{<=} a_2 \\ i_1 \mathtt{<=} a_2 & \rightleftharpoons & a_2 \curvearrowright i_1 \mathtt{+} \square \\ \mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 & \rightleftharpoons & b \curvearrowright \mathtt{if}\ \square\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \end{array}$$

The symbol $\rightleftharpoons$ stands for two structural rules, one left-to-right and another right-to-left.

The right-hand sides of the structural rules above contain, besides the task sequentialization operator "$\curvearrowright$", *freezer* operators containing "$\square$" in their names, such as "$\square$ + _", "_ + $\square$", etc. The first rule above says that in any expression of the form $a_1 \mathtt{+} a_2$, a_1 can be scheduled for processing while a_2 is being held for future processing. Since the rules above are bi-directional, they can be used at will to structurally re-arrange the computations for processing. Thus, when iteratively applied left-to-right they fulfill the role of *splitting* syntax into an evaluation context (the tail of the resulting sequence of computational tasks) and a redex (the head of the resulting sequence), and when applied right-to-left they fulfill the role of *plugging* syntax into context. Such structural rules are often called *heating/cooling rules* in K, because they are reminiscent of the CHAM heating/cooling rules; for example, $a_1 \mathtt{+} a_2$ is "heated" into $a_1 \curvearrowright \square \mathtt{+} a_2$, while $a_1 \curvearrowright \square \mathtt{+} a_2$ is "cooled" into $a_1 \mathtt{+} a_2$. A language definition can use structural rules not only for heating/cooling but also to give the semantics of some language constructs; this will be discussed later in this section.

To avoid writing obvious heating/cooling structural rules like the above, we prefer to use the *strictness attribute* syntax annotations in K, as shown in the middle column in Figures 1 and 2: "*strict*" means non-deterministically strict in all enlisted arguments (given by their positions) or by default in all arguments if none enlisted, and "*seqstrict*" is like *strict* but each argument is fully processed before moving to the next one (see the second structural rule of "`<=`" above).

The structural rules corresponding to strictness attributes (or the heating/cooling rules) decompose and eventually push the tasks that are ready for processing to the top (or the left) of the computation. Semantic rules then tell how to process the atomic tasks. The right column in Figure 1 shows the semantic K rules of IMP. To understand them, let us first discuss the important notion of a *K rule*, which is a strict generalization of the usual notion of a rewrite rule. To take full advantage of K's support for concurrency, K rules explicitly mention the parts of the term that they read, write, or don't care about. The underlined parts are those which are written by the rule; the term underneath the line is the new subterm replacing the one above the line.

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

Figure 1: K definition of IMP: syntax (left), annotations (middle) and semantics (right); $x \in VarId$, $xs \in \mathsf{Set}[VarId]$, $i, i_1, i_2 \in Int$, $t \in Bool$, $b \in BExp$, $s, s_1, s_2 \in Stmt$ (b, s, s_1, s_2 can also be in K)

All writes in a K rule are applied in *one parallel step*, and, with some reasonable restrictions discussed in Section 4 (that avoid read/write and write/write conflicts), writes in multiple K rule instances can also apply in parallel. The ellipses "..." represent the volatile part of the term, that is, that part that the current rule does not care about and, consequently, can be concurrently modified by other rules. The operations which are not underlined represent the read-only part of the term: they need to stay unchanged during the application of the rule. For example, the lookup rule in Figure 1 (first one) says that once program variable x reaches the top of the computation, it is replaced by the value to which it is mapped in the state, regardless of the remaining computation or the other mappings in the state. Similarly, the assignment rule says that once the assignment statement "$x \texttt{:=} i$" reaches the top of the computation, the value of x in the store is replaced by i and the statement dissolves; in K, "_" is a nameless variable of any sort and "$\cdot$" is the unit (or empty) computation (in practice, "$\cdot$" tends to be a polymorphic unit of most if not all list, set and multiset structures). The rule for variable declarations in Figure 1 (last one) expects an empty state and allocates and initializes with 0 all the declared variables; the dotted or dashed lines signifies that the rule is structural, which is discussed next.

K rules are split in two categories: *computational rules* and *structural rules*. Computational rules capture the intuition of computational steps in the execution of the defined system or language, while structural rules capture the intuition of structural rearrangement, rather than computational evolution, of the system. We use dashed or dotted lines in the structural rules to convey the idea that they are lighter-weight than the computational rules. Ordinary rewrite rules are a special case of K rules, when the entire term is replaced; in this case, we prefer to use the standard notation "$l \to r$" as syntactic sugar for computational rules and the notation "$l \rightharpoondown r$" or "$l \rightharpoonup r$" as syntactic sugar for structural rules. We have seen several structural rules at the beginning of this section, namely the heating/cooling rules corresponding to the strictness attributes. Figure 1 shows three more: "$s_1\ \texttt{;}\ s_2$" is rearranged as "$s_1 \curvearrowright s_2$", loops are unrolled when they reach the top of the computation (unconstrained unrolling would lead to undesirable non-termination), and declared variables are allocated in the state. There are no rigid requirements when rules should be computational versus structural and, in the latter case, when one should use "$l \rightharpoondown r$" or "$l \rightharpoonup r$" as syntactic sugar. We (subjectively) prefer to use structural rules for desugaring (like for sequential composition), loop unrolling and declarations, and we prefer to use "$\rightharpoonup$" when syntax is split into computational tasks and "$\rightharpoondown$" when computational tasks are put back into the original syntax.

Each K rule can be "desugared" into a standard term rewrite rule by combining all its changes into one top-level change. The relationship between K rules and conventional term rewriting and rewriting logic is discussed in Section 4. The main point is that the resulting conventional rewrite system associated to a

Original language syntax	K Strictness	K Semantics
$AExp \ ::= \ \ldots \mid \texttt{++}\, VarId$		$\langle \underset{i +_{Int} 1}{\underline{\texttt{++}\, x}} \ \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ x \mapsto \underset{i +_{Int} 1}{\underline{\ i\ }} \ \cdots\rangle_{\mathsf{state}}$
$Stmt ::= \ldots$		
$\mid \ \texttt{output}\,(AExp)$	$[strict]$	$\langle \underset{\cdot}{\underline{\texttt{output}\,(i)}} \ \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ \underset{i}{\underline{\cdot}}\rangle_{\mathsf{output}}$
$\mid \ \texttt{halt}$		$\langle \underset{\cdot}{\underline{\texttt{halt} \cdots}}\rangle_{\mathsf{k}}$
$\mid \ \texttt{spawn}\,(Stmt)$		$\langle \underset{\cdot}{\underline{\texttt{spawn}\,(s)}} \ \cdots\rangle_{\mathsf{k}}\ \underset{\langle s\rangle_{\mathsf{k}}}{\underline{\ \cdot\ }}$ $\langle\cdot\rangle_{\mathsf{k}} \rightharpoondown \cdot$

Figure 2: K definition of IMP++ (extends that of IMP in Figure 1, *without changing anything*)

K system lacks the potential for concurrency of the original K system.

3.2. K Semantics of IMP++

In spite of its simplicity, IMP++ suffices to reveal limitations in each of the conventional semantic approaches [6]; for example, big-step and small-step SOS as well as denotational semantics are heavily non-modular, modular SOS requires artificial syntactic extensions of the language in order to attain modularity, context reduction lacks modularity in some cases, the CHAM relies on a heavy airlock operation to match information in solution molecules, and all these approaches have limited or no support for true concurrency (the CHAM provides more support for true concurrency than the others, but it still unnecessarily enforces interleaving in some cases). In this section we show that K avoids these limitations.

Figure 2 shows how the K semantics of IMP can be seamlessly extended into a semantics for IMP++. To accommodate the output, a new cell needs to be added to the configuration:

$$Configuration_{\mathsf{IMP++}} \quad \equiv \quad \langle\langle K\rangle_{\mathsf{k}}\ \langle \mathsf{Map}[\,VarId \mapsto Int]\rangle_{\mathsf{state}}\ \langle \mathsf{List}[Int]\rangle_{\mathsf{output}}\rangle_{\top}$$

However, note that none of the existing IMP rules needs to change, because each of them only matches what it needs from the configuration. The construct `output` is strict and its rule adds the value of its argument to the end of the output buffer (matches and replaces the unit "$\cdot$" at the end of the buffer). The rule for `halt` dissolves the entire computation, and the rule for `spawn` creates a new $\langle \ldots\rangle_{\mathsf{k}}$ cell wrapping the spawned statement. The code in this new cell will be processed concurrently with the other threads. The last rule "cools" down a terminated thread by simply dissolving it; it is a structural rule because, again, we do not want it to count as a computation.

We conclude this section with a discussion on the concurrency of the K definition of IMP++. Since in K rule instances can share read-only data, various (actually all matching) instances of the look up rule can apply concurrently, in spite of the fact that they overlap on the state subterm. Similarly, since the rules for variable assignment and increment declare volatile everything else in the state except the mapping corresponding to the variable, multiple assignments, increments and reads of distinct variables can happen concurrently. However, if two threads want to write the same variable, or if one wants to write it while another wants to read it, then the two corresponding rules need to interleave, because the two rule instances are in a concurrency conflict. Note also that the rule for `output` matches and changes the end of the output cell; that means, in particular, that multiple outputs by various threads need to be interleaved for the same reason as above. On the other hand, the rule for `spawn` matches any empty top-level position and replaces it by the new thread, so threads can spawn threads concurrently. Similarly, multiple threads can be dissolved concurrently when they are done (last "cooling" structural rule). These desirable concurrency aspects IMP++ are possible to define formally thanks to the specific nature of the K rules. If instead we used standard rewrite rules instead of K rules, then many of the concurrent steps above would need to be interleaved because rewrite rule instances which overlap cannot be applied concurrently.

3.3. K Type System for IMP/IMP++

The K semantics of IMP/IMP++ discussed above can be used to execute even ill-typed IMP/IMP++ programs, which may be considered undesirable by some language designers. Indeed, one may want to

Original language syntax	K Strictness	K Semantics
$AExp ::= Int$		$i \rightarrow int$
$\mid\ VarId$		$\langle \underset{int}{\underline{x}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x\ \cdots\rangle_{\mathsf{vars}}$
$\mid\ AExp \texttt{ + } AExp$	$[strict]$	$int \texttt{ + } int \rightarrow int$
$\mid\ AExp \texttt{ / } AExp$	$[strict]$	$int \texttt{ / } int \rightarrow int$
$\mid\ \texttt{++}\ VarId$		$\langle \underset{int}{\underline{\texttt{++}\,x}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x\ \cdots\rangle_{\mathsf{vars}}$
$BExp ::= AExp \texttt{ <= } AExp$	$[strict]$	$int \texttt{ <= } int \rightarrow bool$
$\mid\ \texttt{not}\ BExp$	$[strict]$	$\texttt{not}\ bool \rightarrow bool$
$\mid\ BExp\ \texttt{and}\ BExp$	$[strict]$	$bool\ \texttt{and}\ bool \rightarrow bool$
$Stmt ::= \texttt{skip}$		$\texttt{skip} \rightarrow stmt$
$\mid\ VarId \texttt{ := } AExp$	$[strict(2)]$	$\langle \underset{stmt}{\underline{x \texttt{ := } int}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x\ \cdots\rangle_{\mathsf{vars}}$
$\mid\ Stmt \texttt{ ; } Stmt$	$[strict]$	$stmt \texttt{ ; } stmt \rightarrow stmt$
$\mid\ \texttt{if}\ BExp\ \texttt{then}\ Stmt\ \texttt{else}\ Stmt$	$[strict]$	$\texttt{if}\ bool\ \texttt{then}\ stmt\ \texttt{else}\ stmt \rightarrow stmt$
$\mid\ \texttt{while}\ BExp\ \texttt{do}\ Stmt$	$[strict]$	$\texttt{while}\ bool\ \texttt{do}\ stmt \rightarrow stmt$
$\mid\ \texttt{output}\ (AExp)$	$[strict]$	$\texttt{output}\ int \rightarrow stmt$
$\mid\ \texttt{halt}$		$\texttt{halt} \rightarrow stmt$
$\mid\ \texttt{spawn}\ (Stmt)$	$[strict]$	$\texttt{spawn}\ (stmt) \rightarrow stmt$
$Pgm ::= \texttt{vars}\ \mathsf{Set}[VarId] \texttt{ ; } Stmt$		$\langle \underset{s \curvearrowright pgm}{\underline{\texttt{vars}\ xl \texttt{ ; } s}}\rangle_{\mathsf{k}}\ \langle \underset{xl}{\underline{\cdot}}\rangle_{\mathsf{vars}}$
		$stmt \curvearrowright pgm \rightarrow pgm$

Figure 3: K type system for IMP++ (and IMP)

define a type checker for a desired typing policy, and then use it to discard as inappropriate programs that do not obey the desired typing policy. In this section we show how to define a type system for IMP/IMP++ using the very same K framework. The type system is defined like an (executable) semantics of the language, but one in the more abstract domain of types rather than in the concrete domain of integer and boolean values. The technique is general and has been used to define more complex type systems, such as higher-order polymorphic ones [22].

The typing policy that we want to enforce on IMP/IMP++ programs is easy: all variables in a program have by default integer type and must be declared, arithmetic/boolean operations are applied only on expressions of corresponding types, etc. Since programs and fragments of programs are now going to be rewritten into their types, we need to add to computations some basic types. Also, in addition to the computation to be typed, configurations must also hold the declared variables. Thus, we define the following (the "..." in the definition of K includes all the default syntax of computations, such as the original language syntax, "$\curvearrowright$", freezers, etc.):

$$\begin{array}{rcl} K & ::= & ... \mid int \mid bool \mid stmt \mid pgm \\ Configuration^{Type}_{\mathsf{IMP++}} & \equiv & \langle\langle K\rangle_{\mathsf{k}}\ \langle \mathsf{Set}[VarId]\rangle_{\mathsf{vars}}\rangle_{\top} \end{array}$$

Figure 3 shows the IMP/IMP++ type system as a K system over such configurations. Constants reduce to their types, and types are propagated through each language construct in a straightforward manner. Note that almost each language construct is strict now, because we want to type all its arguments in almost all cases in order to apply the typing policy of the construct. Two constructs make exception, namely the increment and the assignment. The typing policy of these constructs is that they take precisely a variable and not something that types to an integer. If we defined, e.g., the assignment strict and with rule $int \texttt{ := } int$, then our type system would allow ill-formed programs like "`x+y := 0`". Note how we defined the typing policy of programs "$\texttt{vars}\ xl \texttt{ ; } s$": the declared variables xs are stored into the $\langle ...\rangle_{\mathsf{vars}}$ cell (which is expected to initially empty) and the statement is scheduled for typing (using a structural rule), placing a "reminder" in the computation that the pgm type is eventually expected; once/if the statement is correctly typed, the type pgm is generated.

4. The K Concurrent Rewrite Abstract Machine

Rewriting logic deduction is very useful in capturing concurrent computations as they occur in distributed systems, that is *without* sharing of data. Indeed, if each process has its own state and communication of data is realized through message passing, then the rules of the system need to only apply on a process state or to match a process together with a message which can only be received by one process. However, this is not the case when modelling parallelism with sharing of data. In this case, it is conceivable that one would like to allow situations like the one-writer/multiple-readers pattern, in which multiple rules would be allowed to read the same parts of the term to be rewritten, provided that they do not modify them. Moreover, since rules might involve reading and writing parts of the data at the same time, a need for making the rules even more local naturally appears, to allow multiple rules to "share" the same context. Take for example a rewrite rules for reading/writing a variable in the store:

$$\langle x \curvearrowright k\rangle_{\mathsf{k}}\ \langle s_1, x \mapsto v, s_2\rangle_{\mathsf{store}} \ \rightarrow \langle v \curvearrowright k\rangle_{\mathsf{k}}\ \langle s_1, X \mapsto v, s_2\rangle_{\mathsf{store}} \ \text{ and}$$
$$\langle x\!:=\!v' \curvearrowright k\rangle_{\mathsf{k}}\ \langle s_1, x \mapsto v, s_2\rangle_{\mathsf{store}} \ \rightarrow \langle k\rangle_{\mathsf{k}}\ \langle s_1, x \mapsto v', s_2\rangle_{\mathsf{store}}$$

Here, the k cell is a unary operation $\langle\_\rangle_{\mathsf{k}}$ that "holds" a '$\curvearrowright$'-separated list of tasks (like a computation stack, or a first-order continuation; it satisfies the axioms of lists, i.e., '$\curvearrowright$' is associative and has identity), and the *store* cell holds a comma-separated list of bindings of names to values, while the topmost concatenation operation '$\_\_$' satisfies the (multi-) set axioms, i.e., associativity, commutativity and identity. x, k, s_1, s_2, v, and v' are variables, x standing for a name, k for the rest of the computation, s_1 and s_2 for the parts of store before and after the desired mapping, and v, v' for values. Consider a system containing only this rule, and an initial (ground) term like $\langle a\rangle_{\mathsf{k}}\ \langle a\rangle_{\mathsf{k}}\ \langle b\!:=\!3\rangle_{\mathsf{k}}\ \langle a \mapsto 1, b \mapsto 2\rangle_{\mathsf{store}}$, i.e., two "threads" $\langle a\rangle_{\mathsf{k}}$ whose tasks are to read 'a' from the store, and a thread $\langle b\!:=\!3\rangle_{\mathsf{k}}$ updating the value of 'b'. It might seem reasonable that all threads could advance simultaneously: first two by reading the value of a in the store (since a is shared in the store), and the third updating the value of b (since the location of b is independent of that of a). However, this would be impossible using the deduction rules presented above. One reason is that traditional AC-matching, which requires that the term be rearranged to fit the pattern, limits concurrency where sharing is allowed: there is no way to re-arrange the term so that any two of the rule instances match. Another, more fundamental reason, is that the top set constructor operation, and the store itself need to be shared for the rules to apply.

The K-rules (Section 4.2) introduced below allow such behaviors by making the change local to the exact position in which the update must be applied.

4.1. Preliminaries

A signature Σ is a tuple (S, F) such that S is a set of *sorts* and F is a set of tuples $f : w \rightarrow s$ where f is an operation symbol, $w \in S^*$ is its arity, and $s \in S$ is its result sort. Note that if w is the empty word ϵ, then f is a constant. The universe of terms T_Σ associated to a signature Σ contains all the terms which can be formed by iteratively applying symbols in F over existing terms (using constants as basic terms, to initiate the process), matching the arity of the symbol being applied with the result sorts of the terms it is being applied to. Given an S-sorted set of variables $\mathcal{X}$, the universe of terms $T_\Sigma(\mathcal{X})$ with operation symbols from F and variables from $\mathcal{X}$ consists of all terms in $T_{\Sigma(\mathcal{X})}$, where $\Sigma(\mathcal{X})$ is the signature obtained by adding the variables in $\mathcal{X}$ as constants to Σ, each to its corresponding sort. Given a term $t \in T_\Sigma(\mathcal{X})$, let *vars*$(t)$ be the variables form $\mathcal{X}$ appearing in t.

Substitutions.. A substitution is a mapping yielding terms (possibly with variables) for variables. Any substitution $\psi : \mathcal{X} \rightarrow T_\Sigma(\mathcal{Y})$ naturally extends to terms, yielding a homonymous mapping $\psi : T_\Sigma(\mathcal{X}) \rightarrow T_\Sigma(\mathcal{Y})$. A bijection $s : \mathcal{X} \rightarrow \mathcal{Y}$ naturally extends to a substitution $\psi : \mathcal{X} \rightarrow T_\Sigma(\mathcal{Y})$, termed as a *renaming of variables from* $\mathcal{X}$, defined by $\psi(x) = s(x)$ for each $x \in \mathcal{X}$.

For any substitution $\psi : \mathcal{X} \rightarrow T_\Sigma(\mathcal{Y})$, let *vars*$(\psi)$ be the set of variables in $\mathcal{Y}$ used by ψ, that is: $\mathit{vars}(\psi) = \bigcup_{x\in\mathcal{X}} \mathit{vars}(\psi(x))$. When $\mathcal{X}$ is small, the application of substitution $\psi : \{x_1, \ldots, x_n\} \rightarrow T_\Sigma(\mathcal{Y})$ to term t can be written as $t[\psi(x_1)/x_1, \ldots, \psi(x_n)/x_n] ::= \psi(t)$. This notation allows to use substitutions by need, without formally defining them.

Two substitutions $\psi : \mathcal{X} \rightarrow T_\Sigma(\mathcal{Y})$ and $\psi' : \mathcal{Y} \rightarrow T_\Sigma(\mathcal{Z})$ can be composed to a substitution $\psi' \circ \psi : \mathcal{X} \rightarrow T_\Sigma(\mathcal{Z})$, naturally defined as the composition between the extension of ψ' to $T_\Sigma(\mathcal{Y})$ and ψ.

Given a substitution $\psi : \mathcal{X} \rightarrow T_\Sigma(\mathcal{X})$, we can iteratively define the n^{th} power of ψ for any $n \geq 0$, $\psi^n : \mathcal{X} \rightarrow T_\Sigma(\mathcal{X})$ by $\psi^0(x) = x$ for any $x \in \mathcal{X}$, and $\psi^{n+1} = \psi \circ \psi^n$. If there exists some $n \geq 0$ such that $\mathit{vars}(\psi^n) = \emptyset$, then we say that ψ^n is a *fix-point* for ψ, and denote it as ψ^*.

Contexts.. Given an ordered set of variables, $\mathscr{W} = \{\square_1, \dots, \square_n\}$, named *context variables*, or *holes*, a $\mathscr{W}$*-context over* $\Sigma(\mathcal{X})$ (assume that $\mathcal{X} \cap \mathscr{W} = \emptyset$) is a term $C \in T_\Sigma(\mathcal{X} \cup \mathscr{W})$ which is linear in $\mathscr{W}$ (i.e., each hole appears only once). The set of all $\mathscr{W}$-contexts over $\Sigma(\mathcal{X})$ is written as $\mathit{Cxt}_\Sigma(\mathcal{X}, \mathscr{W})$. The instantiation of context $C \in \mathit{Cxt}_\Sigma(\mathcal{X}, \mathscr{W})$ with an n-tuple $\bar{t} = (t_1, \dots, t_n)$, written $C[\bar{t}]$ or $C[t_1, \dots, t_n]$, is the term $C[t_1/\square_1, \dots, t_n/\square_n]$. One can alternatively regard $\bar{t}$ as a substitution $\bar{t} : \mathscr{W} \to T_\Sigma(X)$, defined by $\bar{t}(\square_i) = t_i$, in which case $C[\bar{t}] = \bar{t}(C)$.

Term rewriting.. A rewrite system over a term universe T_Σ consists of rewrite rules, which can be locally matched and applied at different positions in a Σ-term to gradually transform it. For simplicity, we will consider in this paper only *unconditional* rewrite rules. A Σ*-rewrite rule* is a triple $(\mathcal{X}, l, r)$, written $(\forall \mathcal{X})\ l \to r$, where $\mathcal{X}$ is a set of variables and l and r are $T_\Sigma(\mathcal{X})$-terms, named the *left-hand-side (lhs)* and the *right-hand-side (rhs)* of the rule, respectively. A rewrite rule ϱ: $(\forall \mathcal{X})\ l \to r$ *matches* a Σ-term t using Σ-context C and substitution θ, iff $t = C[\theta[l]]$. If that is the case, then the term t *can be rewritten* according to the rule ϱ to $C[\theta[r]]$. A (Σ-)*rewrite-system* $\mathcal{R} = (\Sigma, R)$ is a set R of Σ-rewrite rules.

Rewriting Logic.. Due to the locality of rewrite rules, it follows naturally that multiple rules can apply in parallel. Rewriting logic [5, 23] exploits this, by allowing sideways and nested (i.e. rewriting-under-variables) parallelism in rule application. For the purpose of this paper, given a Σ rewrite system R, let $R \vdash t \Rrightarrow t'$ signify that t' can be obtain from t' in *one* parallel step, according to the rewriting logic deduction; moreover, let $R \vdash t \Rrightarrow^* t'$ signify that t' can be obtain from t' in zero, one, or more parallel steps. Although rewriting logic concurrent transitions yields a high degree of concurrency, the application of the rules enforces that the matching instances of rules applying concurrently do not overlap. This is a reasonable restriction if one assumes that the *entire* left-hand-side of a rule is matched and replaced by the right-hand-side. However, there are situations, such as concurrent systems with shared resources, which require and support a higher degree of concurrency than allowed by rewriting logic.

4.2. K-rules and K-systems

K-rules are schemas describing how a term can be transformed in another term, by altering some of its parts. They share the idea of match-and-replace of standard term rewriting, but each rule identifies an read-only pattern, the local context of the rule, which is used to glue together write-only patterns, that is, subparts to be rewritten, and to provide information which can be used and shared by them. To some extent, the read-only pattern plays here the same role played by interfaces in graph rewriting [17].

Similar in spirit with the distinction between equations and rules in rewriting logic, K rules can have two flavors, *structural* (which rearange the term without altering the flow of computation) and *computational* (which advance the state of the system). However, as in rewriting logic, this distinction is purely semantical and the concurrent machine needs not be aware of it. Therefore, we will not distinguish between structural and computational rules in this section.

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

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

We may write $(\forall \mathcal{X})\ p[\ \frac{l_1}{r_1}, \dots, \frac{l_n}{r_n}\ ]$ *instead of* $(\forall \mathcal{X})\ p[\ \frac{L}{R}\ ]$ *whenever* $\mathcal{W} = \{\square_1, \cdots, \square_n\}$ *and* $L(\square_i) = l_i$ *and* $R(\square_i) = r_i$*; this way, the holes are implicit and need not be mentioned.*

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

The variables in $\mathcal{W}$ are only used to formally identify the positions in p where rewriting takes place, which is particularly important in proofs. 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 $\mathcal{X}$ can be inferred from the context, it can be omitted. Moreover, a variable which does not appear in R, and it is thus only used in the matching phase, is termed an *anonymous variable.* Similarly to other languages (e.g., Prolog), we take the liberty of denoting anonymous variables by the '_' symbol.

For example, the following two K-rules precisely capture the intuition we have about reading (ρ_r) or writing (ρ_w) a variable in the store, respectively:

$$\rho_r\colon\ \langle \underset{v}{\underline{x}} \curvearrowright \_\rangle_{\mathsf{k}}\ \langle \_, x \mapsto v,\ \_\rangle_{\mathsf{store}} \quad\Big|\quad \rho_w\colon\ \langle \underset{\cdot}{\underline{x := v'}} \curvearrowright \_\rangle_{\mathsf{k}}\ \langle \_, x \mapsto \underset{v'}{\underline{\_}}, \_\rangle_{\mathsf{store}}$$

which, if we want to identify the anonymous variables, could be alternatively written as:

$$\rho_r\colon\ \langle \underset{v}{\underline{x}} \curvearrowright k\rangle_{\mathsf{k}}\ \langle s_1, x \mapsto v, s_2\rangle_{\mathsf{store}} \quad\Big|\quad \rho_w\colon\ \langle \underset{\cdot}{\underline{x := v'}} \curvearrowright k\rangle_{\mathsf{k}}\ \langle s_1, x \mapsto \underset{v'}{\underline{v}}, s_2\rangle_{\mathsf{store}}$$

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 (anonymous) value corresponding to x in the store should be updated to the new value v', 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. When formalizing these rules according to Definition 1, we obtain the following K-rules:

$\rho_r\colon\ (\forall \mathcal{X}_r)\ p_r[\ \dfrac{L_r}{R_r}\ ]$	$\rho_w\colon\ (\forall \mathcal{X}_w)\ p_w[\ \dfrac{L_w}{R_w}\ ]$
$\mathcal{X}_r = \{x, v, k, s_1, s_2\}$	$\mathcal{X}_w = \{x, v, v', k, s_1, s_2\}$
$\mathcal{W}_r = \{\square\}$	$\mathcal{W}_w = \{\square_1, \square_2\}$
$p_r = \langle \square \curvearrowright k\rangle_{\mathsf{k}}\ \langle s_1, x \mapsto v, s_2\rangle_{\mathsf{store}}$	$p_w = \langle \square_1 \curvearrowright k\rangle_{\mathsf{k}}\ \langle s_1, x \mapsto \square_1, s_2\rangle_{\mathsf{store}}$
$L_r(\square) = x$	$L(\square_1) = X := v'$ and $L(\square_2) = v$
$R_r(\square) = v$	$R(\square_1) = \cdot$ and $R(\square_2) = v'$

4.2.1. Relation to rewrite rules

In this section we analyze the relation between rewrite rules and K-rules, showing that K-rules are a conservative extension of standard rewrite rules, increasing the locality and potential for sharing.

To each rewrite rule $\varrho\colon (\forall \mathcal{X})\ l \to r$ we can naturally associate the K-rule $\mathit{R2K}(\varrho) = (\forall \mathcal{X})\ \underset{r}{\underline{l}}$, that is, a rule in which the pattern is just a hole $\square$ and L and R map $\square$ to l and r, respectively. Conversely, one can naturally associate to each K-rule $\rho\colon (\forall \mathcal{X})\ p[\ \dfrac{L}{R}\ ]$ the rewrite rule

$$\mathit{K2R}(\rho) = (\forall \mathcal{X})\ L(p) \to R(p).$$

Through this mapping, K-rules faithfully capture conventional rewrite rules, since $\mathit{K2R}(\mathit{R2K}(\varrho)) = \varrho$, for any rewrite rule ϱ. Note, however, that the opposite does not hold, that is, K-rules are strictly more informative than their corresponding rewrite rules, because the latter lose their "locality" of the changes; in particular, it is impossible to recover the K-rule from the obtained rewrite rule.

How much to share?. Another possible transformation from rewrite rules to K-rules is one that would enforce maximal data-sharing. Let us formally define such a transformation, say $\mathit{R2K}_{max}$. Given a rewrite rule $\varrho\colon (\forall \mathcal{X})\ l \to r$, the *maximally sharing K-rule* associated to ϱ is $\mathit{R2K}_{max}(\varrho) = (\forall \mathcal{X})\ p[\ \dfrac{L}{R}\ ]$ satisfying that for any other K-rule $\rho' : (\forall \mathcal{X})\ p'[\ \dfrac{L'}{R'}\ ]$ such that $\mathit{K2R}(\rho') = \varrho$, p' is a specialization of p, that is, there exists a substitution $\theta : \mathit{vars}(p) \to T_\Sigma(\mathcal{X} \cup \mathit{vars}(p')$ such that $p' = \theta(p)$.

It might seem that maximal sharing is always desirable; however maximal sharing might not always model the intended behavior. Consider for example the rewrite rule $a{*} \to b{*}$, where a, b, and $*$ are constants of some sort s and '$\_\_ : ss \to s$' is an (associative and commutative) operation composing elements of sort s. If the desired behavior is that $*$ is a catalyst allowing the transformation from a to b to happen, and therefore we want it to be potentially read by other rules, then the corresponding K-rule would be $\underset{b}{\underline{a}}{*}$; as seen shortly, such a rule allows a term $aa{*}$ to rewrite in one concurrent step to $bb{*}$. However, if $*$ is to be regarded as a token ensuring mutual exclusion, then the corresponding K-rule could be $\underset{b*}{\underline{a*}}$; as seen shortly, such rules cannot be applied concurrently on a term $aa{*}$, requiring, due to the overlapping of the token '$*$', two interleaved steps.

4.3. K-Transactions

In the previous section we claimed that K-rules can achieve more locality than usual rewrite-rules. Indeed, by allowing rules to share their read-only pattern, parallel rewriting using K-rules can capture more concurrent computations than conventional rewriting logic. In this section we will formalize instances of (multiple) K-rules applying to the same term by means of (concurrent) *K-transactions.* Intuitively, a K-transaction describes how a term is matched by one or multiple rules, and provides a mechanism to retrieve the term obtained upon applying all rules concurrently.

One could give a straight-forward definition for what it means for a K-rule to match a term: *one* K-rule matches a term if and only if its corresponding rewrite rule matches it.

Definition 2. *A K rule $(\forall\mathcal{X})\ p[\underset{R}{\underline{L}}]$* ***matches*** *a Σ-term t using context C and substitution θ if and only if its corresponding rewrite rule matches term t using the same context C and substitution θ, that is, if $t = C[\theta(L(p))]$.*

Hence, when analyzed in isolation, a K rule is no more special than its corresponding rewrite rule. Only when trying to apply multiple rules in parallel we can observe the benefits of the K-rules.

Intuition: matching as coloring.. As described above, the intuition in a K rule $(\forall\mathcal{X})\ p[\underset{R}{\underline{L}}]$ is that p represents a read-only pattern, which can be shared with other rules, while the terms given by L should be regarded as write-only, because no two rules should simultaneously modify the same part, and no rule should read it while another rule writes it .

We next give a visual intuition for combining multiple matches of K rules. Assume that the term to be rewritten is initially uncolored (or black), that is, all its positions are available to all rules. Whenever a K-rule matches, it colors the matched part of the term using two colors, green for the read-only pattern, which can be shared, and red for the write-only parts, so that they would not be touched by any other rule. When combining multiple matches concurrently, the following natural coloring policies apply:

1. Uncolored, or black, can be colored in any color by any K-rule;
2. No rule is allowed to color (green or red) a position which is already red;
3. Once a position is green, it cannot be colored in red by any K-rule.

In short, "red cannot be repainted and green can only be repainted green."

The first policy says that unconstrained parts of the term can be safely matched, in order to be rewritten, by any K-rules. The second policy says that a part of a term which is being written by some rule cannot be read or written by any other rule. Finally, the third policy says that a part of a term that is being read by some rule can be read but not written by other rules.

Analyzing this coloring through the resemblance between K rules and graph-rewriting rules, one can notice that the policy imposed by the above coloring rules, is in direct correspondence with the notion of parallel independence [17] of rules applications in graph rewriting, in the sense that the rule instances are allowed to overlap on their patterns, but not on anything else.

Let us next present a simple instance of using this coloring, using underlining to represent red, and boxing to represent green. Let Σ be a mono-sorted signature with f as a binary operation, g, h, and i as unary operation, and a and b as constants, and consider the following K-system on Σ:

$$\rho_1\colon\ (\forall x,y)\ f(\underset{h(x)}{\underline{g(x)}},y) \qquad \rho_2\colon\ (\forall x,y)\ f(x,\underset{h(y)}{\underline{g(y)}}) \qquad \rho_3\colon\ (\forall x)\ i(\underset{x}{\underline{i(x)}})$$

Let $h(f(g(h(i(i(a)))),g(b)))$ be the term to be rewritten. Rule ρ_2 matches and yields the coloring $h(\boxed{f}(g(h(i(i(a)))),\underline{g}(b)))$. Rule ρ_1 also matches since the colors are consistent, and colors the other g symbol in red: $h(\boxed{f}(\underline{g}(h(i(i(a)))),\underline{g}(b)))$. Finally, rule ρ_3 can be matched on the remaining uncolored positions, yielding the final coloring $h(\boxed{f}(\underline{g}(h(\boxed{i}(\underline{i}(a)))),\underline{g}(b)))$. Since all matches succeeded, we can apply all three rules simultaneously, obtaining the term $h(f(h(h(i(a))),h(b)))$.

The above coloring discipline, together with the possibility of applying all matched rules simultaneously represents the visual presentation of a (concurrent) *transaction.* A K-transaction corresponds to the concurrent application of multiple K-rules. Note the alternation of the colors in the matched term above: black, green, red, black, green, red, black, ... This alternation, together with the fact that these patterns could be arbitrarily nested, suggests that, in order to capture algebraically this visual intuition, we need a way to identify the sub-parts of the term according to their color, as well as a mechanism to put all parts together. This leads to the following rather technical definition for transactions:

Definition 3. *A **K-transaction on Σ-term** t is a tuple $(\mathcal{X}, \mathcal{P}, \mathcal{W}, C, \pi, L, R, \theta)$, where*

- *$\mathcal{X}$ are the **matching variables**;*
- *$\mathcal{P}$ are the **pattern variables**, i.e., place-holders for read-only patterns;*
- *$\mathcal{W}$ are the **write-only variables**, place-holders for replaced sub-patterns.*
- *$C \in Cxt_\Sigma(\mathcal{X}, \mathcal{P})$ is the top context;*
- *$\pi : \mathcal{P} \to Cxt_\Sigma(\mathcal{X}, \mathcal{W})$ gives the read-only patterns;*
- *$L, R : \mathcal{W} \to T_\Sigma(\mathcal{X})$ specify what is updated, and the corresponding updates;*
- *$\theta : \mathcal{X} \to T_\Sigma(\mathcal{P})$ is a substitution;*

such that the following restrictions hold:

1. *variables in $\mathcal{X}$, $\mathcal{P}$, and $\mathcal{W}$ are all distinct and each appears at most once in the set of terms given by C, π, and L; and*
2. *there exist fix points $(\theta + \pi + L)^*, (\theta + \pi + R)^* : X \cup \mathcal{W} \cup \mathcal{P} \to T_\Sigma$, satisfying $(\theta + \pi + L)^*(C) = t$.*

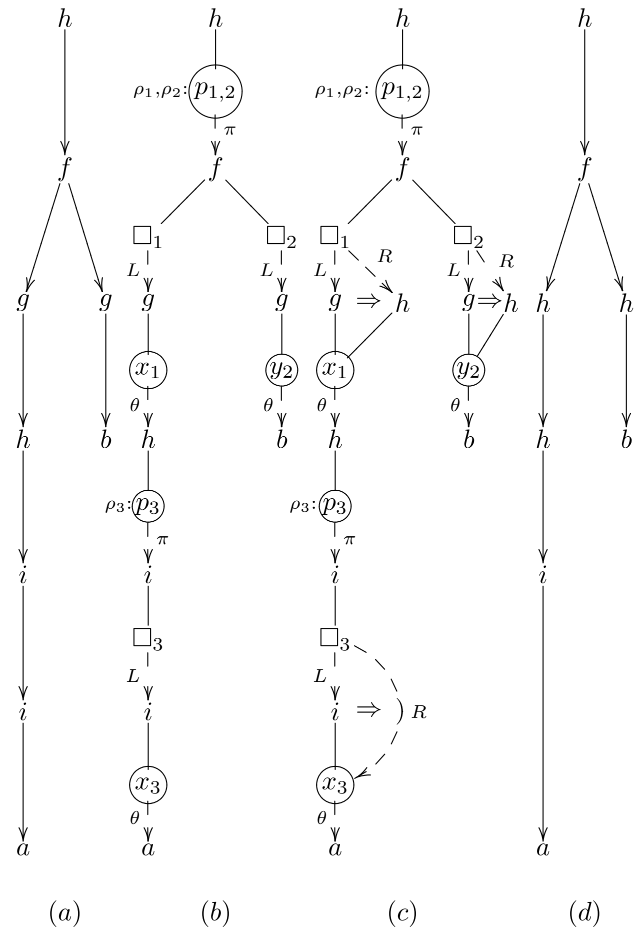

Figure 4: K transaction—visual representation: (a) original term; (b) parallel matching of rules ρ_1, ρ_2, ρ_3; (c) transaction corresponding to parallel application of rules ρ_1, ρ_2, ρ_3; (d) resulting term after one K concurrent rewrite step.

This definition relates easily to the visual intuition presented above: C will be the top black part, π gives the green parts, L gives the red parts, and θ gives the remaining black parts, while R specifies how the red parts should be updated. Finally, the fixpoint substitutions put all the parts together.

Note that the variables in $\mathcal{X}$ are required to appear only once in any of the terms generated by the transaction components, while in the K-rules they were not restricted in any way. The reason for enforcing left-linearity in the transaction is to allow maximum of flexibility for parallel evolution: even if two (or more) parts of the term may be required to be equal for the matching process, they would be handled by distinct variables, though they might correspond to equal subterms. This approach is different, but arguably more flexible than the one used by rewriting logic, which requires that the equal subterms identified by the non-linear occurrence of the same variable in the lhs of a rule should also be equal after the concurrent step is applied. This forces them to either not change, or to change by applying the same rules in the same way, thus limiting the potential concurrency, and basically enforcing an implementation through subterm-sharing.

Let us now formally define the transaction associated to the colored term presented above. The sets of variables are: $\mathcal{X} = \{x_1, y_2, x_3\}$, $\mathcal{P} = \{p_{1,2}, p_3\}$, and $\mathcal{W} = \{\Box_1, \Box_2, \Box_3\}$. The top context is $h(p_{1,2})$, and $\pi(p_{1,2}) = f(\Box_1, \Box_2)$. Then, $L(\Box_1) = g(x_1)$, and $R(\Box_1) = h(x_1)$, while $\theta(x_1) = h(p_3)$. Moreover, $L(\Box_2) = g(y_2)$, and $R(\Box_2) = h(y_2)$, while $\theta(y_2) = b$. Finally, $\pi(p_3) = i(\Box_3)$, $L(\Box_3) = i(x_3)$, $R(\Box_3) = x_3$, and $\theta(x_3) = a$. This transaction can be visualized in Figure 4. The visual representation in Figure 4(c) completely describes the transaction. The circled and boxed (holes) nodes correspond to the variables from $\mathcal{X}$, $\mathcal{P}$, and $\mathcal{W}$, and are interposed between the operation symbols of the original term, splitting it into parts. The singly-dashed lines connecting the variables with the subterms below are the substitutions π, L, and θ, for the original term, and additionally R for the resulting term. Finally, the double-arrows are used as an additional indication that the patterns specified by L are to be replaced with those specified by R. Figure 4(b) completely describes the matching part of the transaction, and, together with R it is enough to recover the original transaction.

From K-rules to K-transactions. To any K-rule ρ matching a term t using context C and substitution θ, we can associate a K-transaction on t. Let $\rho\colon\ (\forall\mathcal{X})\ p[\ \frac{L}{R}\ ]$ be a K-rule.

To the matching pattern $L(p)$, we associate the term $\mathit{linear}(L(p))$, which is a linear version of $L(p)$ that "tags" with natural numbers duplicated occurrences of the variables of p.

The *linearization of* ρ is a K-rule $\mathit{linear}(\rho)\colon\ (\forall\mathcal{X}')\ p[\ \frac{L'}{R}\ ]$, where $\mathcal{X}' = \mathit{vars}(\mathit{linear}(L(p)))$, and p' and L' are such that $L'(p') = \mathit{linear}(L(p))$, $L'\restriction_{\mathcal{X}} = L$, and $p'\restriction_{\mathcal{X}} = p$. A Σ-term t matched by ρ using context C and substitution θ, i.e., $t = \theta(L(p))$, yields a K-transaction $T_{\rho,C,\theta} = (\mathcal{X}_\rho, \mathcal{P}_\rho, \mathcal{W}_\rho, C_\rho, \pi_\rho, L_\rho, R_\rho, \theta_\rho)$, defined using $\mathit{linear}(\rho)$, as follows:

- $\mathcal{X}_\rho = \mathcal{X}'$, $\mathcal{W}_\rho = \mathcal{W}$, the set of hole-variables of p, and $\mathcal{P}_\rho = \{r\}$,
- $C_\rho = C[r]$, and $\pi_\rho(r) = p'$,
- $L_\rho = L'$ and $R_\rho = R$,
- $\theta_\rho(x') = \theta(x'\restriction_{\mathcal{X}})$, for any $x' \in \mathcal{X}'$,

where $(\forall\mathcal{X}')\ p'[\ \frac{L'}{R}\ ]$ is the linearization of ρ.

The above K-transaction associated to a K-rule instance is indeed well-defined: First, $(\theta_\rho + \pi_\rho + L_\rho)^3$ and $(\theta_\rho + \pi_\rho + R_\rho)^3$ are already fixpoints: since there is no nesting, it is enough to apply each substitution once (hence the number 3) to obtain a ground term. Moreover, $(\theta_\rho + \pi_\rho + L_\rho)^3(C_\rho) = C[(\theta_\rho + \pi_\rho + L_\rho)^3(r)] = C[\theta(L'(p)\restriction_{\mathcal{X}})] = C[\theta(\mathit{linear}(L(p))\restriction_{X})] = C[\theta(L(p))] = t$. Note that, since the linearization of a rule does not erase variables, but rather tags duplicates with numbers to distinguish them, there is no need for a distinct version of R'.

4.4. *Transaction composition.*

The same intuition of matching as coloring can be used when composing transactions. A straight forward intuition for transaction compositions is the following: since each transaction is a composition of rule instances, one can compose two or more transactions iff it can compose in a new transaction all K-rule instances corresponding to them. Let us below present a brief overview of the composition process; the complete exposition can be found in [3].

The term on which the transaction is defined is annotated to a *transaction term* which basically resembles the coloring of the terms from Section 4.3, but is more elaborate to allow the transaction to be recovered. This is achieved by "annotating" the operation symbols of the original term to encode the elements of the transaction, without altering the underlying structure of the term. Therefore, the transaction term associated to a transaction could be seen as the "compressed" version of the visual representation presented in Figure 4(b). Transaction terms are instrumental in defining transaction composition, because they are easier to compose, due to their identical structure, and their composition can be used to retrieve the composed transaction.

Given a transaction, one can use the simple coloring in Section 4.3 to mark the patterns given by π and the to-be-replaced parts given by L. What is missing to be able to apply the transaction is a correspondence between the to-be-replaced parts and their replacements from R (given in the transaction through the write-only variables $\mathcal{W}$), and how the variable subterms of L are re-configured by R (given in the transaction by the variables in $\mathcal{X}$).

To fix this shortcoming, transaction terms extend coloring terms by annotating them with variables from $\mathcal{W}$ and $\mathcal{X}$: the variables from $\mathcal{W}$ will be subscripted to red/underlined positions, while variables from $\mathcal{X}$ will be superscripted to any operation symbol. For example, the transaction terms corresponding to matching instances of the K-rules in our running example will be: $h(\boxed{f}(\underline{g}_{\square_1}(h^{x_1}(i(i(a)))), g(b)))$, $h(\boxed{f}(g(h(i(i(a)))), \underline{g}_{\square_2}(b^{y_2})))$, and $h(f(g(h(\boxed{i}(\underline{i}_{\square_3}(a^{x_3})))), g(b)))$.

Transaction composability is then defined as the existence of a "supremum" term which can combine (without conflicts) all the annotations of the corresponding transaction terms. For our example, the supremum term would be $h(\boxed{f}(\underline{g}_{\square_1}(h^{x_1}(\boxed{i}(\underline{i}_{\square_3}(a^{x_3})))), \underline{g}_{\square_2}(b^{y_2})))$, which together with the corresponding R-substitutions, suffices to completely describe the composed transaction.

Let $\sum_{i\in I} T_i$ denote the composition of the set of transactions $\{T_i\}_{i\in I}$ (assuming they are composable).

4.5. K deduction

The K deduction system consists of one deduction rule for deriving concurrent rewriting steps. Let $\mathcal{K}$ be a K-system. The concurrent transition of a term using $\mathcal{K}$ is defined as the application of a transaction obtained as composition of instances of rules in $\mathcal{K}$.

$$\frac{\cdot}{\mathcal{K} \vdash (\pi_T + \theta_T + L_T)^*(C_T) \Rrightarrow (\pi_T + \theta_T + L_T)^*(C_T)}, \quad \begin{array}{l}\text{where } T = \sum_{i\in I} T_i \text{, such that}\\ \text{each } T_i \text{ is of the form } T_{\rho,C,\theta}\text{, with } \rho \in \mathcal{K}.\end{array}$$

The multi-step rewriting relation is obtained as the reflexive and transitive closure of the one-step rewrite relation.

$$\textbf{Refexivity: } \frac{\cdot}{\mathcal{K} \vdash t \Rrightarrow^* t'} \quad \textbf{Transitivity: } \frac{\mathcal{K} \vdash t \Rrightarrow t',\ \mathcal{K} \vdash t' \Rrightarrow^* t''}{\mathcal{K} \vdash t \Rrightarrow^* t''}$$

The trivial translation map *R2K* from rewrite rules to K-rules is extended to term rewrite systems, yielding K-systems obtained by replacing each rule ϱ by *R2K*(ϱ). Similarly, map *K2R* is extended on K-systems to reduce them to regular term rewrite systems.

Capturing Rewriting Logic Computations.. The first part of the following result states that K-deduction conservatively extends rewriting logic deduction. The second part states that one may need multiple rewriting logic steps in the reduct rewrite system to simulate a parallel step deducible in a K-system; nevertheless, their transitive closures coincide. Since rewriting logic has the serializability property, this second part implies that K deduction is also serializable.

Theorem 1 ([3])**.** *Let $\mathcal{R}$ be a term rewrite system. If $\mathcal{R} \vdash t \longrightarrow t'$ then $R2K(\mathcal{R}) \vdash t \Rrightarrow t'$. If additionally $\mathcal{R}$ is left-linear, then the converse also holds.*

Let $\mathcal{K}$ be a K-system. If $\mathcal{K} \vdash t \Rrightarrow t'$ then $K2R(\mathcal{K}) \vdash t \longrightarrow^* t'$. Moreover, $K2R(\mathcal{K}) \vdash t \longrightarrow t'$ implies $\mathcal{K} \vdash t \Rrightarrow t'$. Hence, $\mathcal{K} \vdash t \Rrightarrow^* t'$ iff $K2R(\mathcal{K}) \vdash t \longrightarrow^* t'$.

4.6. Lists, sets and maps

In pursuing th goal of maximizing concurrency, we chose not to deal with lists and sets as nesting of constructs governed by axioms (such as associativity, commutativity, and identity) as done in standard term rewriting. Instead of using an associative binary operator to represent lists, we abstract lists as an operator with a variable number of arguments. For any sort S, the sort $\mathsf{List}^{\dagger}_{\star}[S]$ contains the $\star$-separated lists of elements os sort S, with list identity †. If unspecified, by default $\star$ is '_, _' and † is '$\cdot$'.

The specificity of rules involving lists comes then from the fact they can contain "special" list variables, which can stand for multiple arguments of an operation symbol. With this intuition in mind, matching a K rule containing a list variable as an argument of a list constructor is done by considering the variable as being a sequence of 0, one or more variables of the list element sort, and the transaction corresponding to this matching instance will use those variables instead of the meta list variable. Moreover, any rule which is topped in a list constructor will be considered to have default list variables in both sides, to allow the rule to match in context.

When a list variable occurring in a K rule is neither rewritten, nor used by the R substitution, the variable can be considered as anonymous. However, to distinguish the anonymous list variables from the rest, and to capture the intuition that a list variable stands for zero, one, or more elements, we will use '$\cdots$' to replace an anonymous list variable and the list constructors connecting it.

Example: Concurrent sorting in K. Consider the specification of comma-separated associative lists over a sort S of comparable elements. Let $\langle\_\rangle_{\mathsf{sort}}$ be a construct wrapping a lists of elements of sort S. Then, the following K-rule, quantified by x and y—variables of sort S,

$$\langle \cdots \underset{y}{\underline{x}} \cdots \underset{x}{\underline{y}} \cdots \rangle_{\mathsf{sort}} \text{ when } x > y$$

suffices to sort any list wrapped by sort. Since we know that the constructor for lists is in this case '_, _', the above rule stands for the "desugared" rule: $\langle \_,\_(L_b, \underset{y}{\underline{x}}, L_m, \underset{x}{\underline{y}}, L_e)\rangle_{\mathsf{sort}}$ when $x > y$

Let us show how one can use this rule to concurrently sort a lists of numbers: say we want to sort the list $3, 8, 5, 7, 4, 1, 2, 6$. We will mark how the numbers pair in the matching process by annotating the

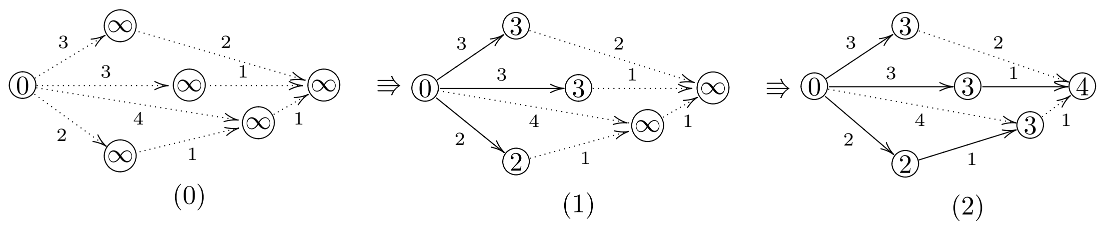

Figure 5: Concurrent Dijkstra derivation in 3 steps

underline with indexed variables corresponding to each match. Since the ellipses stand for anonymous variables, they can match anything, including positions already marked for rewriting. Therefore, in the first concurrent step the matching phase could mark for rewriting all positions obtaining something like $\underset{x_1}{\underline{3}}, \underset{x_2}{\underline{8}}, \underset{x_3}{\underline{5}}, \underset{x_4}{\underline{7}}, \underset{y_3}{\underline{4}}, \underset{y_1}{\underline{1}}, \underset{y_2}{\underline{2}}, \underset{y_4}{\underline{6}}$. Upon applying the concurrent step, the list would look as follows: $1, 2, 4, 6, 5, 3, 8, 7$. In the second round, the matching phase could mark the list to $1, 2, \underset{x_1}{\underline{4}}, 6, 5, \underset{y_1}{\underline{3}}, \underset{x_2}{\underline{8}}, \underset{y_2}{\underline{7}}$, inducing the one step transition to $1, 2, 3, 6, 5, 4, 7, 8$. Finally, there is only one possible rule instance left for matching, $1, 2, 3, \underset{x}{\underline{6}}, 5, \underset{y}{\underline{4}}, 7, 8$, leading in the third step to the sorted list $1, 2, 3, 4, 5, 6, 7, 8$.

Sets, Bags, and Maps (which can be regarded as sets of pairs) are handled similarly to the lists, with the provision that the matching process must now account for the possible permutations of the variables appearing in the set, as the semantics of sets assumes not only associativity, but also the commutativity of the operation constructing the set. For any sort S, the comma-separated set with elements from S and identity '$\cdot$' is denoted by $\mathsf{Set}[S]$. Similarly, $\mathsf{Bag}[S]$ denotes the sort of bags (i.e., multi-sets) of elements of a sort S, while $\mathsf{Map}[S_1 \mapsto S_2]$ denotes the sort of maps with keys from sort S_1 and values from sort S_2.

Example: Dijkstra's Algorithm in K. Consider a graph being expressed a sort *Node* containing node names defined as constants, as a sort *Edge* constructed with the following operator: '$\_ \xrightarrow{\_} \_ : \mathit{Node} \times \mathit{Nat} \times \mathit{Node}$'. Let $\langle \_ \rangle_{\mathsf{graph}}$ be a constructor wrapping a set of edges, and $\langle \_ \rangle_{\mathsf{shortest}}$ be a constructor wrapping a map from nodes to Nat^{∞}, that is naturals plus infinity. Assuming the initial term contains the set of nodes specifying the graph in the *graph* cell, and a map mapping each node to ∞ in the *shortest* cell, except for a special node, say a, which is mapped to 0, the following K rule suffices to compute all shortest paths from a to the other nodes of the graph.

$$\langle \cdots\ x \mapsto c_x\ \cdots\ y \mapsto \underset{t + c_x}{\underline{c_y}}\ \cdots \rangle_{\mathsf{shortest}}\ \langle \cdots\ x \xrightarrow{t} y\ \cdots \rangle_{\mathsf{graph}},\ \text{when } t + c_x < c_y$$

Since $\mathsf{shortest}$ holds a map, which has the set semantics, matching this rule is equivalent with matching any of the following two rules using only the list semantics:

$$\langle \cdots\ x \mapsto c_x\ \cdots\ y \mapsto \underset{t + c_x}{\underline{c_y}}\ \cdots \rangle_{\mathsf{shortest}}\ \langle \cdots\ x \xrightarrow{t} y\ \cdots \rangle_{\mathsf{graph}},\ \text{when } t + c_x < c_y$$

$$\langle \cdots\ y \mapsto \underset{t + c_x}{\underline{c_y}}\ \cdots\ x \mapsto c_x\ \cdots \rangle_{\mathsf{shortest}}\ \langle \cdots\ x \xrightarrow{t} y\ \cdots \rangle_{\mathsf{graph}},\ \text{when } t + c_x < c_y$$

The graphical representation of a run of this algorithm is presented in Figure 5. Initially all graph edges are dotted while the nodes contain the initial minimal costs. As the algorithm proceeds, costs in the nodes are updated and the edges considered are depicted with full lines.

5. The K Technique

Q/A

Q: *What are these Question/Answer boxes in this section?*
A: Each subsection in this section introduces an important component of the K technique, such as the its configurations, computations, or semantic rules. Each Q/A box captures the essence of the corresponding subsection *from a user perspective.* They will ease the understanding of how the various components fit together.

Like term rewriting and rewriting logic, the K concurrent rewrite abstract machine (KRAM) discussed in Section 4 can be used in various ways in various applications; in other words, the KRAM itself does not tell us *how* to define a programming language or calculus as a K system. In this section we present the *K technique*, which consists of a series of guidelines and notations that turn the KRAM or even plain term rewriting into an effective framework for defining programming languages or calculi. The development of the K technique has been driven by practical needs, and it is the result of our efforts to define various programming languages, paradigms, and calculi as rewrite or K systems. We would like to make two important observations before we proceed:

1. The K technique is *flexible* and *open-ended*. Our current guidelines and notations are convenient enough to define the range of languages, features and calculi that we considered so far. Some readers may, however, prefer different or new notations. As an analogy, recall that there are no rigid rules for how to write a configuration in SOS [9]: one may use the angle-bracket notation $\langle code, state, ...\rangle$, or the square bracket notation $[code, state, ...]$, or even the simple tuple notation $(code, state, ...)$; also, one may use a different (from comma) symbol to separate the various configuration ingredients and, even further, one could use writing conventions (such as the "state" or "exception" conventions in [8]) to simplify the writing of SOS definitions. Even though we believe that our notational conventions discussed in this section should be sufficient for any definitional task, we still encourage our reader to feel free to change our notations or propose new ones if needed to better fit one's needs or style. Nevertheless, our current prototype implementations of K rely on our current notation as described in this section; therefore, to use our tools one needs to obey our notation.
2. The K technique yields a *semantic definitional style*. As an analogy, no matter what notations one uses for configurations and other ingredients in SOS definitions (see item above), or even whether one uses rewriting logic or any other computational framework to represent and execute SOS definitions or not, SOS still remains SOS, with all its advantages and limitations; the same holds true for any other definitional style. Similarly, we expect that the K technique can be represented or implemented in various back-end computational frameworks. We prefer KRAM because we believe that it gives us the maximum of concurrency one can hope for in K definitions. However, if one is not sensitive to this true concurrency aspect or if one prefers a certain computational framework over anything else, then one can very well use the K technique in that framework. Indeed, the same way the various conventional language definitional styles become *definitional methodologies or styles* within rewriting logic as shown in [6], the K technique can also be cast as a definitional methodology or style within other computational frameworks. In [3, 24] we show how this can be done for rewriting logic and Maude, for example.

5.1. K Configurations: Nested Cell Structures

Q/A

Q: *Do I need to define a configuration for my language?*
A: No, but it is strongly recommended to define one whenever your language is non-trivial. Even if you define no configuration, you still need to define the cells used later on in the semantic rules; otherwise the rules will not parse.
Q: *How can I define a configuration?*
A: All you need is to define a potentially *nested-cell* structure like in Figure 6, which is a cell term over the simple cell grammar described below. By defining the configuration you shoot three rabbits with one stone:

- You implicitly define all the needed cells, which is required anyway;
- You have a better understanding of all the semantic ingredients that you need for your subsequent semantics as well as their role; and
- You have the possibility to reuse existing semantic rules that were conceived for more abstract configurations, via a process named *context transforming*.

In K definitions, the programming language, calculus or system configuration is represented as a potentially *nested cell* structure. This is similar is spirit to how configurations are represented in chemical abstract machines (CHAMs, see [18]) or in membrane systems (P-systems, see [19]), except that K's cells can hold more varied data and are not restricted to certain means to communicate with their environment. The various cells in a K configuration hold the infrastructure needed to process the remaining computation, including the computation itself; cells can hold, for example, computations (these are discussed in depth in Section 5.2), environments, heaps or stores, remaining input, output, analysis results, resources held, bookkeeping information, and so on. The number and type of cells that appear in a configuration is not fixed and is typically different from definition to definition. K assumes and makes intensive use of the entire range of structures allowed by algebraic CFGs, such as lists, sets, multisets and maps.

Formally, the K configurations have the following simple, nested-cell structure:

$$\begin{array}{rcll}
\mathit{Cell} & ::= & \langle \mathit{CellContents} \rangle_{\mathit{CellLabel}} & \\
\mathit{CellContents} & ::= & \mathit{Sort} \mid \mathsf{Bag}_{\textvisiblespace}[\mathit{Cell}] & \\
\mathit{CellLabel} & ::= & \mathit{CellName} \mid \mathit{CellName}{*} & \\
\mathit{CellName} & ::= & \top \mid \mathsf{k} \mid \textvisiblespace \mid \mathsf{env} \mid \mathsf{store} \mid \ldots & \text{(language specific cell names; first two are common)}
\end{array}$$

where *Sort* can be *any sort name*, including arbitrary list ($\mathsf{List}[\mathit{Sort}]$), set ($\mathsf{Set}[\mathit{Sort}]$), bag ($\mathsf{Bag}[\mathit{Sort}]$) or map ($\mathsf{Map}[\mathit{Sort}_1 \mapsto \mathit{Sort}_2]$) sorts. Many K definitions share the cell labels $\top$ (which stays for "*top*") and k (which stays for "*computation*"). They are built-in in our implementation of K in Maude [24], so one needs not declare them in each language definition. The white-space or "invisible" label "$\textvisiblespace$" may be preferred as an alternative to $\top$ and/or k, particularly when there is a need for only one cell type, like in the definitions of CCS and Pi calculi. The cells with starred labels say that there could be multiple instances, or clones, of that cell. This multiplicity information is optional[1], but can be useful for context transformers (see Section 5.4).

We have seen so far three K configurations, one for IMP, one for IMP++ and one for their their type system; we recall all three of them below:

$$\begin{array}{rcl}
\mathit{Configuration}_{\mathsf{IMP}} & \equiv & \langle\langle K\rangle_{\mathsf{k}}\ \langle \mathsf{Map}[\mathit{VarId} \mapsto \mathit{Int}]\rangle_{\mathsf{state}}\rangle_{\top} \\
\mathit{Configuration}_{\mathsf{IMP{+}{+}}} & \equiv & \langle\langle K\rangle_{\mathsf{k}}\ \langle \mathsf{Map}[\mathit{VarId} \mapsto \mathit{Int}]\rangle_{\mathsf{state}}\ \langle \mathsf{List}[\mathit{Int}]\rangle_{\mathsf{output}}\rangle_{\top} \\
\mathit{Configuration}^{\mathit{Type}}_{\mathsf{IMP{+}{+}}} & \equiv & \langle\langle K\rangle_{\mathsf{k}}\ \langle \mathsf{Set}[\mathit{VarId}]\rangle_{\mathsf{vars}}\rangle_{\top}
\end{array}$$

Notice that they all obey the general cell grammar above, that is, they are nested cell structures; the bottom cells only contain a sort and no other cells. As a more complex example, below is the K configuration of Challenge (see Section 6), a toy language conceived to challenge and expose the limitations the various language definitional frameworks:

$$\begin{array}{rcl}
\mathit{Configuration}_{\mathsf{Challenge}} & \equiv & \langle \mathit{Agents}_{\mathsf{Challenge}}\ \langle \mathsf{List}[\mathit{Int}]\rangle_{\mathsf{output}}\ \mathit{Messages}_{\mathsf{Challenge}}\ \langle \mathit{AgentId}\rangle_{\mathsf{nextAgent}}\rangle_{\top} \\
\mathit{Agents}_{\mathsf{Challenge}} & \equiv & \left\langle \begin{array}{l} \mathit{Threads}_{\mathsf{Challenge}}\ \langle \mathsf{Map}[\mathit{Nat} \mapsto \mathit{Val}]\rangle_{\mathsf{store}}\ \langle \mathit{Nat}\rangle_{\mathsf{nextLoc}} \\ \langle K\rangle_{\mathsf{aspect}}\ \langle \mathsf{Set}[\mathit{Val}]\rangle_{\mathsf{busy}}\ \langle \mathit{AgentId}\rangle_{\mathsf{me}}\ \langle \mathit{AgentId}\rangle_{\mathsf{parent}} \end{array} \right\rangle_{\mathsf{agent}*} \\
\mathit{Threads}_{\mathsf{Challenge}} & \equiv & \langle\langle K\rangle_{\mathsf{k}}\ \langle \mathsf{Map}[\mathit{VarId} \mapsto \mathit{Nat}]\rangle_{\mathsf{env}}\ \langle \mathsf{Map}[\mathit{Val} \mapsto \mathit{Nat}]\rangle_{\mathsf{holds}}\rangle_{\mathsf{thread}*} \\
\mathit{Messages}_{\mathsf{Challenge}} & \equiv & \langle\langle\langle \mathit{AgentId}\rangle_{\mathsf{sender}}\ \langle \mathit{AgentId}\rangle_{\mathsf{receiver}}\ \langle \mathit{Val}\rangle_{\mathsf{val}}\rangle_{\mathsf{message}*}\rangle_{\mathsf{messages}}
\end{array}$$

To make it more readable, we introduced some intuitive "macros" above, namely $\mathit{Agents}_{\mathsf{Challenge}}$, $\mathit{Threads}_{\mathsf{Challenge}}$, and $\mathit{Messages}_{\mathsf{Challenge}}$. Figure 6 shows a graphical representation of this configuration, which was generated automatically by the K2Latex component of our current implementation of K in Maude [24]. Note that the Challenge configurations have four levels of cell-nesting and several cells labels are starred, meaning that there can be multiple instances of those cells. For example, the top cell may contain multiple agent cells; each agent may contain, besides information like a local store, aspect, busy resources (used as locks for thread synchronization), etc., an arbitrary number of thread cells; each thread contains a local computation, a local environment and a number of resources (resources can be acquired multiple times by the same thread, so a map is needed). As one may expect, real life language definitions tend to employ rather complex configurations.

The advantage of representing configurations as nested cell-structures is that, like in MSOS [11], subsequent rules only need to mention those configuration items that are needed for those particular rules, as opposed to having to mention the entire configuration, whether needed or not, like in conventional SOS. We can add or remove items from a configuration as we like, only impacting the rules that use those particular configuration items. Rules that do not need the changed configurations items do not need to be touched. This is an important aspect of K, which significantly contributes to its modularity.

[1] Note, in particular, that we omitted it for the k label in the IMP++ configuration (IMP++ is multi-threaded).

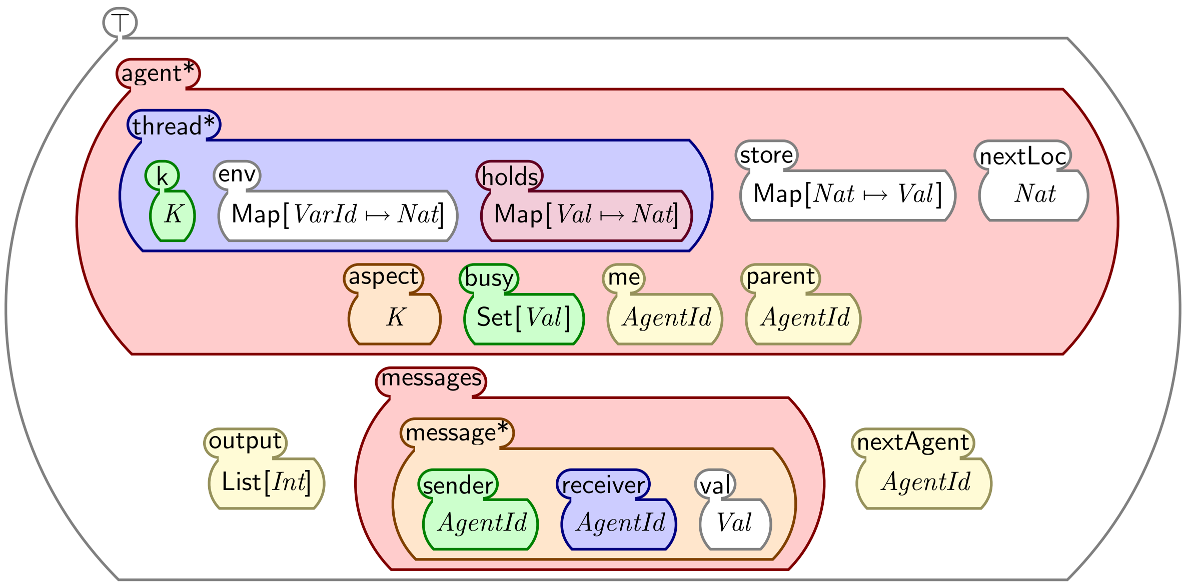

Figure 6: The configuration of the Challenge language.

Defining a configuration for a K semantics of a language, calculus or system is an optional step, in that it suffices to only define the desirable cell syntax so that configurations like the desired one parse as ordinary cell terms. That indeed provides all the necessary infrastructure to give the semantic K rules. However, providing a specific configuration term is useful in practice for at least two reasons. First, the configuration can serve as an intuitive skeleton for writing the subsequent semantic rules, one which can be consulted to quickly find out, for example, what kind of cells are available and where they can be found. Second, the configuration structure is the basis for *context transforming* (see Section 5.4), which gives more modularity to K rules by allowing them to be reusable in language extensions that require changes in the structure of the configuration.

5.2. K Computations: $\curvearrowright$-Separated Nested Lists of Tasks

Q/A

Q: *What are K computations?*
A: Computations are an intrinsic part of the K framework. They extend abstract syntax with a special nested-list structure and can be thought of as sequences of fragments of program that need to be processed sequentially.
Q: *Do I need to define computations myself?*
A: What is required is to define an abstract syntax of your language (discussed below) and desired evaluation strategies for the language constructs (discussed in Section 5.3.1), which need to be defined no matter what semantic framework you prefer. By doing so, you implicitly define the basic K computational infrastructure. In many cases you do not need to define any other computation constructs.
Q: *Do I need to understand in depth what computations are in order to use K?*
A: Not really. If you follow a purely syntactic definitional style mimicking reduction semantics with evaluation contexts [12] in K, then the only computations that that you will ever see in your rules are abstract syntax terms.
Q: *What is the benefit of using more complex (than abstract syntax) computations?*
A: K at its full strength. Many complex languages are very hard or impossible to define purely syntactically, while they admit elegant and natural definitions using proper K computations. For example, the Challenge language in Section 6.

K takes a very abstract view of language syntax and, in theory, it is not concerned *at all* with parsing aspects[2]. More precisely, in K there is only one top-level sort[3] associated to all the language syntax, called K and staying for *computational structures* or *computations*, and terms t of sort K have the abstract syntax tree (AST) representation $l(t_1, ..., t_n)$, where l is some K *label* and t_1,...,t_n are terms of sort K, extended with the list (infix) construct "$\curvearrowright$", read "followed by" or "and then"; for example, if t_1, t_2, ..., t_n are computations then $t_1 \curvearrowright t_2 \curvearrowright \cdots \curvearrowright t_n$ is also a computation, namely the one *sequentializing* t_1, t_2, ..., t_n. All the original language constructs, including constants and program variables, as well as all the freezers (discussed below), are regarded as labels. For notational convenience, we continue to write K-terms using the original syntax instead of the harder to read AST notation. Formally, computations are defined as follows:

$$\begin{array}{rcl} K & ::= & \mathit{KLabel}(\mathsf{List}[K]) \mid \mathsf{List}_{\curvearrowright}[K] \\ \mathit{KLabel} & ::= & \text{(one per language construct, plus auxiliary ones as needed)} \end{array}$$

The first construct scheme for K abstractly captures any programming language syntax as an AST, provided that one adds one *KLabel* for each language construct. For example, in the case of the IMP language, we add to *KLabel* all the following labels corresponding to the IMP syntax:

$$\begin{array}{rcl} \mathit{KLabel}_{\mathsf{IMP}} & ::= & \mathit{Int} \mid \mathit{VarId} \mid \_\texttt{+}\_ \mid \_\texttt{/}\_ \mid \_\texttt{<=}\_ \mid \texttt{not}\_ \mid \_\texttt{and}\_ \mid \texttt{skip} \mid \_\texttt{:=}\_ \mid \_\texttt{;}\_ \\ & \mid & \texttt{if}\_\texttt{then}\_\texttt{else}\_ \mid \texttt{while}\_\texttt{do}\_ \mid \texttt{vars}\_\texttt{;}\_ \end{array}$$

We recommend the use of the *mix-fix notation* for labels, like in the above labels corresponding to the IMP language; the mix-fix notation was introduced by the OBJ language [25] and followed by many other akin languages, where underscores in the name of an operation mark the places of its arguments. In addition to the language syntax, *KLabel* may include additional labels for semantic reasons; for example, labels corresponding to semantic domain values which may have not been automatically included in the syntax of the language, such as the *Bool* domain in the case of IMP. We take the liberty to call K *constants* those labels intended to take an empty list of arguments (e.g., the constants in the original syntax: skip, true, 1, 2, etc.).

It is convenient in many K definitions to distinguish syntactically between proper computations and computations which are finished. A similar phenomenon is common and well-accepted in the other definitional styles, which distinguish between proper expressions and values, for example. To make this distinction smooth, we add the *KResult* syntactic sub-category of K which is constructed using corresponding labels (all labels in *KResultLabel* are also in *KLabel*):

$$\begin{array}{rcl} \mathit{KResult} & ::= & \mathit{KResultLabel}(\mathsf{List}[K]) \\ \mathit{KResultLabel} & ::= & \text{(one per construct of terminated computations, e.g., values, results, etc.)} \end{array}$$

Among the labels in *KResultLabel* one may have certain language constants, such as true, 0, 1, etc., but also labels that correspond to non-constant terms, for example $\lambda\_.\_$; indeed, in some λ-calculi, λ-abstractions $\lambda x.e$ (or $\lambda\_.\_(x, e)$ in AST form), are values (or finished computations).

We take the liberty to write language or calculus syntax either in AST form, like in "$\lambda\_.\_(x, e)$" and "if_ then_ else_(b, s_1, s_2)", or in more readable mixfix form, "$\lambda x.e$" and "if b then s_1 else s_2". In our Maude implementation of K [24], thanks to Maude's builtin support for mixfix notation and corresponding parsing capabilities, we actually write programs using the mixfix notation. Even though theoretically unnecessary, this is actually very convenient in practice, because it makes language definitions more readable and, consequently, less error-prone. Additionally, programs in the defined languages can be regarded as terms the way they are, without any intermediate AST representation for them. In other implementations of K, one may need to use an explicit parser or to get used to reading syntax in AST representation. Either way, from here on we assume that programs, or fragment of programs, parse as computations in K.

The second construct scheme for K allows one to sequentialize computational tasks. Intuitively, $k_1 \curvearrowright k_2$ says "process k_1 then k_2". How this is used and what is the exact meaning of "process" is left open and depends upon the particular definition. For example, in a concrete semantic language definition it can mean "evaluate k_1 then k_2", while in a type inferencer definition it can mean "type and accumulate

[2]In practice, like in all other language semantics frameworks, some parser is always assumed or effectively used as a front-end to K to parse and transform the language syntax into its abstract K syntax.

[3]Technically, one can define more than one top-level computation sort; however, so far we have not found any major uses for that, so for simplicity we prefer to keep only one computation sort for now.

type constraints in k_1 then do the same for k_2", etc. The following are examples of computations making use of the $\mathsf{List}_{\curvearrowright}[K]$ structure of K (we use parentheses for disambiguation):

$$\begin{array}{l}
(\mathsf{if\ true\ then}\ \cdot\ \mathsf{else}\ \cdot) \curvearrowright \mathsf{while\ false\ do}\ \cdot \\
a_1 \curvearrowright \square + a_2 \\
a_2 \curvearrowright a_1 + \square \\
a_3 \curvearrowright (a_1 + a_2) + \square \\
a_3 \curvearrowright (a_1 \curvearrowright \square + a_2) + \square \\
b \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \\
b \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ (s \curvearrowright \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else}\ \cdot
\end{array}$$

The "$\cdot$" in the first and last computations above is the unit of K (given by $\mathsf{List}_{\curvearrowright}[K]$). Note that $\curvearrowright$-separated lists of computations can be nested. Most importantly note that, unlike in evaluation contexts, $\square$ is not a "hole" in K, but rather part of a *KLabel*; the *KLabel*s involving $\square$ above are

$$\mathit{KLabel} \ ::= \ \ldots \mid \_ + \square \mid \square + \_ \mid \mathsf{if}\ \square\ \mathsf{then}\_\ \mathsf{else}\_$$

The $\square$ carries the "plug here" intuition; e.g., one may think of "$a_1 \curvearrowright \square + a_2$" as "process a_1, then plug its result in the hole in $\square + a_2$". The user of K is not expected to declare these special labels. We assume them whenever needed. In our implementation of K in Maude [24], all these are generated automatically as constants of sort *KLabel* after a simple analysis of the language syntax.

Freezers. To distinguish the labels containing $\square$ in their name from the labels that encode the syntax of the language under consideration, we call the former *freezers*. The role of the freezers is therefore to store the enclosing computations for future processing. One can freeze computations at will in K, using freezers like the ones above, or even by defining new freezers. In complex K definitions, one may need many computation freezers, making definitions look heavy and hard to read if one makes poor choices for freezer names. Therefore, we adopt the following *freezer naming convention*, respected by all the freezers above:

> If a computation can be seen as $c[k, x_1, ..., x_n]$ for some multi-context c and a freezer is introduced to freeze everything except k, then the name of the freezer is "$c[\square, \_, ..., \_]$".

Additionally, to increase readability, we take the freedom to generalize the adopted mixfix notation in K and "plug" the remaining computations in the freezer, that is, we write $c[\square, k_1, ..., k_n]$ instead of $c[\square, \_, ..., \_](k_1, ..., k_n)$. For instance, if $\_@\_$ is some binary operation and if, for some reason, in contexts of the form $(e_1@e_2)@(e_3@e_4)$ one wishes to freeze e_1, e_3 and e_4 (in order to, e.g., process e_2), then, when there is no confusion, one may write $(e_1@\square)@(e_3@e_4)$ instead of $((\_@\square)@(\_@\_))(e_1, e_3, e_4)$. This convention is particularly useful when one wants to follow a reduction semantics with evaluation contexts style in K, because one can mechanically associate such a freezer to each context-defining production. For example, the freezer $(\_@\square)@(\_@\_)$ above would be associated to a production of the form "$\mathit{Cxt} ::= (\mathit{Exp}@\mathit{Cxt})@(\mathit{Exp}@\mathit{Exp})$".

5.3. K Rules: Computational and Structural

Q/A

Q: *How are the K rules different from conventional rewrite rules?*
A: The K framework builds upon the K concurrent rewrite abstract machine (Kram); how the Kram rules differ from standard rules is explained in Section 4.
Q: *What do I lose if I think of K rules as sugared variants of standard rules?*
A: Not much if you are *not* interested in *true concurrency*.
Q: *Does that mean that I can execute K definitions on any rewrite engine?*
A: Yes. However, it is desirable to use a rewrite engine with support at least for associative matching. In fact, our current implementation of K [24] desugars the K rules into ordinary rules and equations anyway.

The K technique aims, among other things, at maximizing the potential for concurrency in the defined languages. Similarly, as discussed in Section 4, the concurrent rewrite abstract machine (Kram) also aims at maximizing the potential for concurrency, but for rewriting. Therefore, it is natural that the rewrite-based K framework employs the KRAM as a rewriting infrastructure.

Recall from Section 4 that KRAM provides two kinds of rules, computational and structural:

Computational rules	*Structural rules*
$p[\underset{r_1}{\underline{l_1}}, \underset{r_2}{\underline{l_2}}, ..., \underset{r_n}{\underline{l_n}}]$	$p[\underset{r_1}{\underline{l_1}}, \underset{r_2}{\underline{l_2}}, ..., \underset{r_n}{\underline{l_n}}]$

They both consist of a local context, or pattern, p, with some of its subterms underlined and rewritten to corresponding subterms underneath the line. The idea is that the underlined subterms represent the "write-only" part of the rule, while the operations in p which are not underlined represent the "read-only" part of the rule and can be shared by concurrent rule instances. The difference between computational and structural rules is that rewrite steps using the latter do not count as computational steps, their role being to rearrange the structure of the term to rewrite so that computational rules can match and apply. In general, there are no rigid requirements on when a K semantic rule should be computational versus structural. While in most cases the distinction between the two is quite natural, there are situations where one needs to subjectively choose one or the other; for example, we chose the rule for variable declarations in the IMP semantics in Figure 1 to be structural, but we believe that some language designers may prefer it to be computational.

Recall also from Section 4 that we prefer to use the conventional rewrite rule notations "$l \rightarrow r$" and "$l \rightharpoondown r$" for computational and structural K rules, respectively, when $p = \square$ (that is, when there is only one write-only part, namely the entire pattern, and no read-only part). There is not much to say about K rules in addition to what has already been said in Sections 3 and 4. We would like to only elaborate a bit the heating/cooling rules and their corresponding strictness attributes.

5.3.1. Heating/Cooling Structural Rules

Q/A

Q: *What is the role of the heating/cooling rules?*
A: These are K's mechanism to define evaluation strategies of language constructs. They allow you to decompose fragments of programs into sequences of smaller computations, and to compose smaller computations back into fragments of programs.
Q: *Do I need to define such heating/cooling rules myself?*
A: Most likely no. It usually suffices to define *strictness attributes*, as discussed below; these are equivalent to defining evaluation contexts in reduction semantics. Strictness attributes serve as a notational convenience for defining obvious heating/cooling structural rules.

After defining the desired language syntax so that programs or fragments of programs become terms of sort K, called computations, the very first step towards giving a K semantics is to define the evaluation strategies or strictness of the various language constructs by means of heating/cooling rules, or more conveniently, by means of the special attributes described shortly. The heating/cooling rules allow us to regard computations many different, but completely equivalent ways. For example, "$a_1 + a_2$" in IMP may be regarded also as "$a_1 \curvearrowright \square + a_2$", with the intuition "schedule a_1 for processing and *freeze* a_2 in freezer $\square + \_$", but also as "$a_2 \curvearrowright a_1 + \square$" (recall from Section 3.1 that, in IMP, addition is intended to be non-deterministic). As discussed in Section 5.2, freezer are nothing but special labels whose role is to store computations for future processing.

Heating/cooling structural rules tell how to "pass in front" of the computation fragments of program that need to be processed, and also how to "plug their results back" once processed. In most language definitions, *all* such rules can be extracted automatically from K strictness operator attributes as explained below; Figure 1 shows several examples of strictness attributes. For example, the *strict* attribute of $\_ + \_$ is equivalent to the following two heating/cooling pairs of rules in K (a_1 and a_2 range over computations in K):

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

The symbol "$\rightleftharpoons$" is borrowed from the chemical abstract machine (CHAM) [18], as a shorthand for combinations of a heating rule ("$\rightharpoonup$") and a cooling rule ("$\leftharpoondown$"). Indeed, one can think of the first rule above as follows: to process $a_1 + a_2$, let us first "heat" a_1, applying the rule from left to right; once a_1 is processed (using other rules in the semantics) producing some result, place that result back into context via a "cooling" step, applying the rule from right to left. However, it is important to realize that these heating/cooling equations can be applied at any moment and in any direction, because they are regarded not as computational steps but as structural rearrangements. For example, one can use the heating/cooling rules for "_ + _" above to pick and pass in front either a_1 or a_2, then rewrite it one step only using semantic rules (defined later in this section), then plug it back into the sum, then pick and pass in front either a_1 or a_2 again and rewrite it one step only, and so on, thus obtaining the desired non-deterministic operational semantics of _ + _.

The general idea to define a certain evaluation context, say $c[\Box, N_1, ..., N_n]$, where N_1, ..., N_n are the various syntactic categories involved (or non-terminals in the CFG of the language), is to define a *KLabel* freezer $c[\Box, \_, ..., \_]$ like discussed in Section 5.2 together with a heating/cooling rule pair "$c[k, k_1, ..., k_n] \rightleftharpoons k \curvearrowright c[\Box, k_1, ..., k_n]$".

One should be aware that in K "$\Box$" is nothing but a symbol that we prefer to use as part of label names. In particular, "$\Box$" is *not* a computation (recall that in reduction semantics with evaluation contexts "$\Box$" is a special context, called a "hole"). For example, a hasty reader may think that K's approach to strictness is unsound, because one can "prove" wrong correspondences as follows:

$$\begin{array}{rcll} a_1 + a_2 & \rightharpoonup & a_1 \curvearrowright \Box + a_2 & \text{(by the first rule above applied left-to-right)} \\ & \rightharpoonup & a_1 \curvearrowright a_2 \curvearrowright \Box + \Box & \text{(by the second rule above applied left-to-right)} \\ & \leftharpoondown & a_1 \curvearrowright a_2 + \Box & \text{(by the first rule above applied right-to-left)} \\ & \leftharpoondown & a_2 + a_1 & \text{(by the second rule above applied right-to-left)} \end{array}$$

What is wrong in the above "proof" is that one cannot apply the second rule in the second step above, because $\Box + a_2$ is nothing but a convenient way to write the frozen computation $\Box +\_ \,(a_2)$. One may say that there is no problem with the above, because _ + _ is intended to be commutative anyway; unfortunately, the same could be proved for any non-deterministic construct, for example for a division operation, "/", if that was to be included in our language. Since the heating/cooling rules are thought of as structural rearrangements, so that computational steps take place *modulo* them, then it would certainly be wrong to have both "a_1/a_2" and "a_2/a_1" in the same computational class. One of K's most subtle technical aspects, which fortunately is transparent to users, is to find the right (i.e., as weak as possible) restrictions on the applications of heating/cooling equations, so that each computational equivalence class contains no more than one fragment of program. The idea is to only allow heating and/or cooling of operator arguments that are proper syntactic computations (i.e., terms over the original syntax, i.e., different from "$\cdot$" and containing no "$\curvearrowright$"). With that, for example, the computation equivalence class of the expression $x * (y + 2)$ in the context of a language definition with non-deterministically strict binary $+$ and $*$, consists of the terms:

$$\begin{array}{l} x * (y + 2) \\ x \curvearrowright (\Box * (y + 2)) \\ x \curvearrowright (\Box * (y \curvearrowright (\Box + 2))) \\ x \curvearrowright (\Box * (2 \curvearrowright (y + \Box))) \\ (y + 2) \curvearrowright (x * \Box) \\ y \curvearrowright (\Box + 2) \curvearrowright (x * \Box) \\ 2 \curvearrowright (y + \Box) \curvearrowright (x * \Box) \\ x * (y \curvearrowright (\Box + 2)) \\ x * (2 \curvearrowright (y + \Box)) \end{array}$$

Note that there is only one syntactic computation in the computation class above, namely the original expression itself. This is a crucial desired property of K.

5.3.2. Strictness Attributes

In K definitions, one typically defines zero, one, or more heating/cooling rules per language construct, depending on its intended evaluation/processing strategy. These rules tend to be straightforward and boring to write, so in K we prefer a higher-level and more compact and intuitive approach: we annotate the language syntax with *strictness attributes*. A language construct annotated as *strict*, such as for example the "_ + _" in Figure 1, is automatically associated a heating/cooling pair of rules as above for each of its subexpressions. If an operator is intended to be strict in only some of its arguments, then the positions of the strict arguments are listed as arguments of the *strict* attribute. For example, note that the strictness attribute of $\mathsf{if\_\ then\_\ else\_}$ in Figure 1 is *strict*(1); that means that a heating/cooling equation is added only for the first subexpression of the conditional, namely the equation "$\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \rightleftharpoons b \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$".

The two pairs of heating/cooling rules corresponding to the strictness attribute *strict* of $\_+\_$ above did not enforce any particular order in which the two subexpressions were processed. It is often the case that one wants a deterministic order in which the strict arguments of a language construct are processed, typically from left to right. Such an example is the relational operator `_<=_` in Figure 1, which was declared the strictness attribute *seqstrict*, saying that its subexpressions are processed deterministically, from left to right. The attribute *seqstrict* requires the definition of the syntactic category of result computations *KResult*, as discussed in Section 5.2, and it can be desugared automatically as follows: generate a heating/cooling pair of rules for each argument like in the case of *strict*, but requiring that all its previous arguments are in *KResult*. For example, the *seqstrict* attribute of $\_\leq\_$ desugars into (a_1, a_2 range over K and r_1 over *KResult*):

$$\begin{gathered} a_1 \leq a_2 \rightleftharpoons a_1 \curvearrowright \square \leq a_2 \\ r_1 \leq a_2 \rightleftharpoons a_2 \curvearrowright r_1 \leq \square \end{gathered}$$

Like the *strict* attribute, *seqstrict* can also take a list of numbers as argument and then the heating/cooling rules are generated so that the corresponding arguments are processed in that order.

Our most general strictness declaration in K, also supported by our current implementation [24], is to declare a certain syntactic context (a derived term) strict or sequentially strict in a certain list of arguments. For example, in the K definition of Challenge in Section 6, we declare the context "$*\,k_1$=k_2" to be *strict*(k_1), with the meaning that the assignment statement applied to a pointer needs to first evaluate the pointer expression before other semantic rules can apply.

5.4. Context Transforming

We next introduce one of the most advanced feature of K, the *context transforming*, which gives K an additional degree of modularity. The process of context transforming is concerned with automatically modifying existing K rules according to the cell structure defined by the desired configuration of a target language. The benefit of context transforming is that it allows us to define semantic rules more abstractly, without worrying about the particular details of the concrete final language configuration. This way, it implicitly enhances the modularity and reuse of language definitions: existing rules do not need to change as the configuration of the languages changes to accommodate additional language features, and language features defined generically once and for all can be reused across different languages with different configuration structures.

Defining a configuration (see Section 5.1) is therefore a necessary step in order to make use of K's context transforming. Assuming that the various cell-labels forming the configuration are distinct, then one can use the structure of the configuration to *automatically* transform abstract rule contexts/patterns, i.e., ones that do not obey the intended cell-structure of the configuration, into concrete ones that are well-formed within the current configuration structure. This rule context transforming process can be thought of as being applied statically, before the K-system is executed.

Consider, for example, the K semantic rule for the output statement in IMP++ (see Figure 2):

$$\langle \frac{\texttt{output}\,(i)}{\cdot} \cdots \rangle_{\mathsf{k}} \; \langle \cdots \frac{\cdot}{i} \rangle_{\mathsf{output}}$$

This rule says exactly what one wants the semantics of the output statement to be and as abstractly and compactly as possible: if "`output` (i)" is the next computational task, then append i to the end of the output buffer and dissolve the output statement. This rule perfectly matched the configuration structure of IMP++, because the IMP++ configuration structure was very simple: a top level cell containing all the other cells inside as simple, non-nested cells. Consider now defining a more complex language, like the Challenge language in Section 6 whose configuration was shown in Figure 6. The particular cell arrangement in the Challenge configuration makes the rule above directly inapplicable; to be precise, even though the rule context still parses as a *CellContents*-term, it will never match/apply when used in the context of the Challenge configuration.

Context transforming is about automatic adaptation of K rules like above to new configurations. Indeed, note that there is only one way to bring the cells $\langle ... \rangle_{\mathsf{k}}$ and $\langle ... \rangle_{\mathsf{output}}$ mentioned in the rule above together: to wrap the $\langle ... \rangle_{\mathsf{k}}$ cell within the two additional cells declared in the Challenge configuration, namely to transform the rule into the following one:

$$\langle\cdots\ \langle\cdots\ \langle\underset{\cdot}{\underline{\texttt{output}\,(i)}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{thread}}\ \cdots\rangle_{\mathsf{agent}}\ \langle\cdots\ \underset{i}{\underline{\cdot}}\rangle_{\mathsf{output}}$$

Thus, context transforming can be defined as the process of customising the paths to the various cells used in a rule according to the configuration of the target language. As part of this customisation process, volatile variables are used for the remaining parts of the introduced cells, so that other rule instances concerned with those parts of the cells can apply concurrently with the transformed rule.

5.4.1. The Locality Principle

The example rule for output above was rather simple, in that there was no confusion on how to complete the paths to the refered cells. Consider instead an abstract K rule for pointer dereferencing:

$$\langle\underset{v}{\underline{*\,l}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ l\mapsto v\ \cdots\rangle_{\mathsf{store}}$$

This says that if dereferencing of location l is the next computational task and if value v is stored at location l, then $*\,l$ rewrites to v. The configuration of Challenge considers the cells $\langle ... \rangle_{\mathsf{k}}$ and $\langle ... \rangle_{\mathsf{store}}$ at different levels in the structure, so a context transforming operation is necessary to adapt this abstract rule to Challenge. However, without care, there are two ways to do it:

$$\langle\cdots\ \langle\underset{v}{\underline{*\,l}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{thread}}\ \langle\cdots\ l\mapsto v\ \cdots\rangle_{\mathsf{store}}$$

$$\langle\cdots\ \langle\cdots\ \langle\underset{v}{\underline{*\,l}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{thread}}\ \cdots\rangle_{\mathsf{agent}}\ \langle\cdots\ \langle\cdots\ l\mapsto v\ \cdots\rangle_{\mathsf{store}}\ \cdots\rangle_{\mathsf{agent}}$$

The first Challenge-concrete rule above says that the thread containing the dereferencing and the store are part of the same agent, while the second rule says that they are in different agents (why we are allowed to multiply the agent cells is explained shortly). Even though we obviously meant the first one, both these rules are in fact valid concrete rules according to the configuration of Challenge.

To avoid such conflicts, context transforming relies on the *locality principle*: rules are transformed in a way that makes them as local as possible, or, in other words, in way that the resulting rule context matches in concrete configuration cells as deeply as possible. The locality principle therefore rules out the second rule transformation above, because it is less local than the former.

If, for some reason (which makes no sense for Challenge) one means a non-local transformation of a rule context, then one should add more cell-structure to the abstract rule for disambiguation. For example, if one really meant the second, non-local context transforming of the dereferencing rule above, then one should have written the abstract rule, for example, as follows:

$$\langle\cdots\ \langle\underset{v}{\underline{*\,l}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{agent}}\ \langle\cdots\ l\mapsto v\ \cdots\rangle_{\mathsf{store}}$$

Now there is only one way to context transform this abstract rule to fit the configuration of Challenge, namely like in the second Challenge-concrete rule above. Indeed, the $\langle ... \rangle_{\mathsf{store}}$ cell can only by within an $\langle ... \rangle_{\mathsf{agent}}$ cell and the $\langle ... \rangle_{\mathsf{k}}$ cell inside the declared $\langle ... \rangle_{\mathsf{agent}}$ cell can only be inside an intermediate $\langle ... \rangle_{\mathsf{thread}}$ cell. Therefore, context transforming applies recursively at all levels in the rule context.

Let us next consider one more example showing the locality principle at work, namely the abstract rule for variable lookup in languages with direct access to variable addresses (thus, variables are bount to their addresses in the environment and their addresses to their values in the store):

$$\langle\underset{v}{\underline{x}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x\mapsto l\ \cdots\rangle_{\mathsf{env}}\ \langle\cdots\ l\mapsto v\ \cdots\rangle_{\mathsf{store}}$$

The locality principle says that there is only one way to transform this rule in the context of the Challenge configuration, namely into the following rule:

$$\langle \cdots\ \langle \underset{v}{\underline{x}}\ \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ x \mapsto l\ \cdots\rangle_{\mathsf{env}}\ \cdots\rangle_{\mathsf{thread}}\ \langle \cdots\ l \mapsto v\ \cdots\rangle_{\mathsf{store}}$$

Without locality, the three cells in the abstract rule above could be included in two or even in three agents; when the first two cells are in the same agent, they could also appear in different threads.

5.4.2. The Cell-Cloning Principle

There are K rules in which one wants to refer to two or more cells having the *same label.* An artificial example was shown above, where more than one agent cell was needed. A more natural rule involving two cells with the same label would be one for thread communication or synchronization, in which the two threads are directly involved in the said action. For example, consider adding a rendezvous synchronization mechanism to IMP++ whose intended semantics is the following: a thread whose next computational task is a rendezvous barrier statement "`rv` v" blocks until another thread also reaches an identical "`rv` v" statement, and, in that case, both threads unblock and continue their execution. The following K rule captures this desired behavior of rendezvous synchronization:

$$\langle \underset{\cdot}{\underline{\texttt{rv}\ v}}\ \cdots\rangle_{\mathsf{k}}\ \langle \underset{\cdot}{\underline{\texttt{rv}\ v}}\ \cdots\rangle_{\mathsf{k}}$$

Since this K rule captures the essence of the intended rendezvous synchronization, we would like to reuse it unchanged in language definitions which are more complex than IMP++, such as the Challenge language in 6. Unfortunately, this rule will never match/apply as is on Challenge configurations, because two $\langle ...\rangle_{\mathsf{k}}$ cells can never appear next to each other. A context transforming operation is therefore necessary, but it is not immediately clear how the rule context should be changed. The *cell-cloning principle* applies when abstract rules refer to two or more cells with the same name, and it states that context transforming should be consistent with the cell cloning, or multiplicity, information provided as part of the configuration definition; this can be done using starred labels, as explained in Section 5.1. Note that, for example, the Challenge configuration in Figure 6 declares both the agent and the thread cells clonable. Thus, using the cell-cloning principle in combination with the locality principle, the abstract rule above is transformed into the following Challenge-concrete rule:

$$\langle \cdots\ \langle \underset{\cdot}{\underline{\texttt{rv}\ v}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{thread}}\ \langle \cdots\ \langle \underset{\cdot}{\underline{\texttt{rv}\ v}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{thread}}$$

The cell-cloning principle can therefore only be applied when one defines a configuration for one's language and, moreover, when one also provides the desired cell-cloning information (by means of starred labels). However, in our experience with defining languages in K, it is actually quite useful to spend the time and add the cell-cloning information to one's configuration; one not only gets the convenience and modularity that comes with context transforming for free, but also a better insight on how one's language configurations look when programs are executed and thus, implicitly, a better understanding of one's language semantics.

6. The Challenge Language Definition

This section contains the complete definition of Challenge, a non-trivial experimental programming language meant to challenge existing language definitional frameworks, and to show the benefits one gets by using the K framework to define it.

6.1. Syntax

The syntax of the Challenge language consists of two categories, expressions and statements. Starting with regular arithmetic expressions, the language gradually grows in complexity by adding expressions with side effects, pointers, (recursive) functions, highly non-trivial control-flow operations, such as call/cc, non-determinism, multi-threading with shared memory synchronization, agents and inter-agent communication, and code generation.

$$\begin{array}{rl}
\mathit{Exp} ::= & \#\mathit{Bool} \mid \#\mathit{Float} \mid \#\mathit{Int} \mid \#\mathit{Name} \\
& \mid \mathit{Exp} + \mathit{Exp}\ [\mathit{strict}] \\
& \mid \mathit{Exp} * \mathit{Exp}\ [\mathit{strict}] \\
& \mid \mathit{Exp} \leq \mathit{Exp}\ [\mathit{seqstrict}] \\
& \mid \texttt{not}\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid \mathit{Exp}\ \texttt{and}\ \mathit{Exp}\ [\mathit{strict}(1)] \\
& \mid \mathit{Stmt}\ ;\ \mathit{Exp}\ [\mathit{renameTo}\ \_ \curvearrowright \_] \\
& \mid ++\ \#\mathit{Name}\ [\mathit{renameTo}\ \texttt{inc}] \\
& \mid \&(\ \#\mathit{Name}\ ) \\
& \mid \texttt{ref}\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid *\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid \lambda\, \#\mathit{Name}\ .\ \mathit{Exp} \\
& \mid \mu\, \#\mathit{Name}\ .\ \mathit{Exp} \\
& \mid \mathit{Exp}\ \mathit{Exp}\ [\mathit{renameTo}\ \texttt{apply}\ \mathit{strict}] \\
& \mid \texttt{callcc}\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid \texttt{randomBool} \\
& \mid \texttt{new-agent}\ \mathit{Stmt} \\
& \mid \texttt{me} \\
& \mid \texttt{parent} \\
& \mid \texttt{receive} \\
& \mid \texttt{receive-from}\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid \texttt{quote}\ \mathit{Exp} \\
& \mid \texttt{unquote}\ \mathit{Exp} \\
& \mid \texttt{eval}\ \mathit{Exp}\ [\mathit{strict}]
\end{array}$$

$$\begin{array}{rl}
\mathit{Stmt} ::= & \\
& \mid \mathit{Stmt}\ ;\ \mathit{Stmt}\ [\mathit{renameTo}\ \_ \curvearrowright \_] \\
& \mid \{\texttt{vars}\ \mathsf{Set}[\#\mathit{Name}]\ ;\ \mathit{Stmt}\} \\
& \mid \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}\ [\mathit{strict}(1)] \\
& \mid \texttt{while}\ \mathit{Exp}\ \texttt{do}\ \mathit{Stmt} \\
& \mid \texttt{output}\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid \mathit{Exp} := \mathit{Exp}\ [\mathit{strict}(2)] \\
& \mid \texttt{aspect}\ \mathit{Stmt} \\
& \mid \texttt{spawn}\ \mathit{Stmt} \\
& \mid \texttt{acquire}\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid \texttt{release}\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid \texttt{rv}\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid \texttt{send-asynch}\ \mathit{Exp}\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid \texttt{send-synch}\ \mathit{Exp}\ \mathit{Exp}\ [\mathit{strict}] \\
& \mid \texttt{halt}
\end{array}$$

Besides specifying the grammar for the language, the K syntax also allows it to be annotated with attributes carrying semantic meaning. Strictness attributes (*strict, seqstrict*), specify that (some) arguments need to be evaluated before giving semantics to the entire construct, and the whether the order of their evaluation matters (*seqstrict*). For example, '+' is strict, requiring that both of its arguments be evaluated (regardless of order) before their sum can be evaluated; on the other hand, since the semantics for the 'and' operator will be shortcut, the operator is declared strict only in the first argument. Renaming attributes (*renameTo*) declare that certain constructs will be renamed to existing or new constructs for the semantics part. For example, function application is renamed to the prefix construct 'apply', while sequential composition is renamed to the K task sequencing construct.

6.2. Semantics

We will only give here rules for constructs having a different behavior than the one already presented for IMP + +.

Reading from the store. When a name is matched at the top of computation, it can replaced by value V, provided that the mapping of X to location L can be matched in the environment and the mapping of L to V can be matched in the store:

$$\langle \frac{x}{v}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x \mapsto l\ \cdots\rangle_{\mathsf{env}}\ \langle\cdots\ l \mapsto v\ \cdots\rangle_{\mathsf{store}}$$

Statement Block with variable declaration. When variables are declared, new locations are allocated in the environment. We use the 'nextLoc' cell to hold the next available location; also, we use `Rho[X <- L]` to map X to L in Rho. Once the executions of the statements in the block is completed, the old environment must be resumed.

$$\langle \frac{\{\texttt{vars}\ xs;\ s\}}{s \curvearrowright \rho}\ \cdots\rangle_{\mathsf{k}}\ \langle \frac{\rho}{\rho\,[\,xs \leftarrow n..n + |xs| - 1\,]}\rangle_{\mathsf{env}}\ \langle \frac{n}{n +\ \texttt{|xs|}}\rangle_{\mathsf{nextLoc}}$$

Side effects: increment. $\langle \frac{\mathsf{inc}(\,x\,)}{i +\ 1}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x \mapsto l\ \cdots\rangle_{\mathsf{env}}\ \langle\cdots\ l \mapsto \frac{i}{i +\ 1}\ \cdots\rangle_{\mathsf{store}}$

References. Obtaining the reference location of a name:

$$\langle \underset{l}{\underline{\texttt{\&(}\,x\,\texttt{)}}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x\mapsto l\ \cdots\rangle_{\mathsf{env}}$$

Dereferencing a location:

$$\langle \underset{v}{\underline{\texttt{*}\ l}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ l\mapsto v\ \cdots\rangle_{\mathsf{store}}$$

Evaluating an expression and returning a reference to it:

$$\langle \underset{n}{\underline{\texttt{ref}\ v}}\ \cdots\rangle_{\mathsf{k}}\ \langle \underset{\sigma\,\texttt{[}\,n\leftarrow v\,\texttt{]}}{\underline{\ \sigma\ }}\rangle_{\mathsf{store}}\ \langle \underset{n\,\texttt{+}\ \texttt{1}}{\underline{\ n\ }}\rangle_{\mathsf{nextLoc}}$$

Assignment. The right hand side of an assignment is evaluated to a value, specified by 'strict(2)' in the operator declaration, while the left-hand-side is evaluated to a l-value, here either a Name, or the dereferencing of a location (the 'strict(K)' in the K-context declaration).

$$\langle \underset{\cdot}{\underline{x\ \texttt{:=}\ v}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x\mapsto l\ \cdots\rangle_{\mathsf{env}}\ \langle \underset{\sigma\,\texttt{[}\,l\leftarrow v\,\texttt{]}}{\underline{\ \sigma\ }}\rangle_{\mathsf{store}}$$

CONTEXT: $\texttt{*}\ k\ \texttt{:=}\ k'\ \ [\texttt{strict}(k)]$

$$\langle \underset{\cdot}{\underline{\texttt{*}\ l\ \texttt{:=}\ v}}\ \cdots\rangle_{\mathsf{k}}\ \langle \underset{\sigma\,\texttt{[}\,l\leftarrow v\,\texttt{]}}{\underline{\ \sigma\ }}\rangle_{\mathsf{store}}$$

Function and aspects. Since our language allows usage of pointer and references, it is more convenient to work with environments and to represent functions as closures. In addition, the Challenge language allows the specification of aspects, whose semantics is that they affect each function defined after the '`aspect`' declaration statement to execute the statement provided as its argument prior to execution their body, in the environment of the function. This is achieved by using a special cell to hold the current aspect, and by including the computation contained in that cell in the closure of each function being evaluated.

$$\langle \underset{\cdot}{\underline{\texttt{aspect}\ s}}\ \cdots\rangle_{\mathsf{k}}\ \langle \underset{s}{\underline{\_}}\rangle_{\mathsf{aspect}}$$

$$\langle \underset{\texttt{closure}\,\texttt{(}\,x\,\texttt{,}\,s\curvearrowright e\,\texttt{,}\ \rho\,\texttt{)}}{\underline{\lambda\,x\,.\ \ e}}\ \cdots\rangle_{\mathsf{k}}\ \langle\rho\rangle_{\mathsf{env}}\ \langle s\rangle_{\mathsf{aspect}}$$

Function application. Upon function call, the evaluated arguments are bound to the formal parameters, then the body of the function is executed in the environment saved by the closure, and finally the calling environment must restored.

$$\langle \underset{e\curvearrowright\varrho}{\underline{\texttt{apply(}\,\texttt{closure}\,\texttt{(}\,x\,\texttt{,}\,e\,\texttt{,}\ \rho\,\texttt{)}\,\texttt{,}\,v\,\texttt{)}}}\ \cdots\rangle_{\mathsf{k}}\ \langle \underset{\rho\,\texttt{[}\,x\leftarrow n\,\texttt{]}}{\underline{\ \varrho\ }}\rangle_{\mathsf{env}}\ \langle\cdots\ \underset{n\,\mapsto v}{\underline{\ \cdot\ }}\ \cdots\rangle_{\mathsf{store}}\ \langle \underset{n\,\texttt{+}\ \texttt{1}}{\underline{\ n\ }}\rangle_{\mathsf{nextLoc}}$$

Restoring the original environment, when the body of the function is completely evaluated:

$$\langle v\curvearrowright \underset{\cdot}{\underline{\rho}}\ \cdots\rangle_{\mathsf{k}}\ \langle \underset{\rho}{\underline{\varrho}}\rangle_{\mathsf{env}}$$

Recursion. The semantics of $\mu X.E$ is given by evaluating the expression E in the environment/store where X is bound to a $\mu X.E$.

$$\langle \underset{e\curvearrowright\rho}{\underline{\mu\,x\,.\ \ e}}\ \cdots\rangle_{\mathsf{k}}\ \langle \underset{\rho\,\texttt{[}\,x\leftarrow n\,\texttt{]}}{\underline{\ \rho\ }}\rangle_{\mathsf{env}}\ \langle\cdots\ \underset{n\,\mapsto\mu\,x\,.\ \ e}{\underline{\ \cdot\ }}\ \cdots\rangle_{\mathsf{store}}\ \langle \underset{n\,\texttt{+}\ \texttt{1}}{\underline{\ n\ }}\rangle_{\mathsf{nextLoc}}$$

Call with current continuation (call/cc). The semantics of call/cc is that of packaging the reminder of the computation as a value and passing it to the function passed as argument. When the thus packaged computation is applied on a value, the entire computation at tha time is replaced by the value being put on top of the packed computation. Note that the environment needs to be packed together with the computation, same as for function closures.

$$\langle \underset{\mathsf{apply}(\,v\,,\,\mathtt{cc}\,(\,k\,,\,\rho\,)\,)}{\underline{\mathtt{callcc}\ \ v}} \curvearrowright k\rangle_{\mathsf{k}}\ \langle\rho\rangle_{\mathsf{env}}$$

$$\langle \underset{k\,[[\,v\,]]}{\underline{\mathsf{apply}(\,\mathtt{cc}\,(\,k\,,\,\rho\,)\,,\,v\,) \curvearrowright \_}}\rangle_{\mathsf{k}}\ \langle\underset{\rho}{\underline{\_}}\rangle_{\mathsf{env}}$$

Sequential non-determinism. 'randomBool' non-deterministically evaluates to either 'true' or 'false'.

$$\langle\underset{\mathtt{true}}{\underline{\mathtt{randomBool}}}\ \cdots\rangle_{\mathsf{k}} \qquad\qquad \langle\underset{\mathtt{false}}{\underline{\mathtt{randomBool}}}\ \cdots\rangle_{\mathsf{k}}$$

Threads. Threads are characterized by independent control flow, but shared memory. In our definition, a 'thread' cell is used to group together all (computation, environment, acquired resources) cells related to a thread The '`spawn`' command creates a new thread which is initialized with the statement passed as argument as its computation, and the environment of the thread creating it.

$$\langle\cdots\ \ \langle\underset{\cdot}{\underline{\mathtt{spawn}\ \ s}}\ \ \cdots\rangle_{\mathsf{k}}\ \langle\rho\rangle_{\mathsf{env}}\ \ \cdots\rangle_{\mathsf{thread}}\ \underset{\langle\langle s\rangle_{\mathsf{k}}\ \langle\rho\rangle_{\mathsf{env}}\ \langle\cdot\rangle_{\mathsf{holds}}\rangle_{\mathsf{thread}}}{\underline{\qquad\cdot\qquad}}$$

When the computation of a thread has completed, it can be dissolved and its held resources be released. 'busy' is a shared cell holding the names of the resources acquired by any of the threads.

$$\underset{\cdot}{\underline{\langle\cdots\ \ \langle\cdot\rangle_{\mathsf{k}}\ \langle holds\rangle_{\mathsf{holds}}\ \ \cdots\rangle_{\mathsf{thread}}}}\ \langle\underset{busy\ \mathtt{-'}\ \ \mathsf{keys}(\,holds\,)}{\underline{\qquad busy\qquad}}\rangle_{\mathsf{busy}}$$

Thread synchronization. To attain mutual exclusion, threads can be synchronized by means of locks. In this language, one can lock on any value. A lock can only be acquired it it is not busy. Same thread can acquire same lock multiple times, therefore multiplicities for each lock are maintained in the 'holds' cell.

$$\langle\underset{\cdot}{\underline{\mathtt{acquire}\ \ v}}\ \ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ \ v \mapsto \underset{\mathtt{s}\ \ n}{\underline{\ n\ }}\ \ \cdots\rangle_{\mathsf{holds}}$$

$$\langle\underset{\cdot}{\underline{\mathtt{acquire}\ \ v}}\ \ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ \ \underset{v\mapsto\,0}{\underline{\quad\cdot\quad}}\ \ \cdots\rangle_{\mathsf{holds}}\ \langle busy\ \&\ \ \underset{v}{\underline{\cdot}}\rangle_{\mathsf{busy}}\ \text{when}\ \mathtt{not}\ \ v\ \mathtt{in'}\ \ busy$$

$$\langle\underset{\cdot}{\underline{\mathtt{release}\ \ v}}\ \ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ \ v \mapsto \underset{n}{\underline{\mathtt{s}\ \ n}}\ \ \cdots\rangle_{\mathsf{holds}}$$

$$\langle\underset{\cdot}{\underline{\mathtt{release}\ \ v}}\ \ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ \ \underset{\cdot}{\underline{v\mapsto 0}}\ \ \cdots\rangle_{\mathsf{holds}}\ \langle\cdots\ \ \underset{\cdot}{\underline{v}}\ \ \cdots\rangle_{\mathsf{busy}}$$

Rendez-vous synchronization. Two threads can only pass together a barrier specified by a lock V. This is harder to achieve in all definitional frameworks we know of, because it requires the ability to access and reduce redexes of two computational units simultaneously.

$$\langle\underset{\cdot}{\underline{\mathtt{rv}\ \ v}}\ \ \cdots\rangle_{\mathsf{k}}\ \langle\underset{\cdot}{\underline{\mathtt{rv}\ \ v}}\ \ \cdots\rangle_{\mathsf{k}}$$

Agents. An agent is here a collection of threads, grouped in an 'agent' cell identified by an id held by the 'me' cell, and holding in the 'parent' cell a reference id to its creating agent. 'nextAgent' is used for providing fresh ids for agents.

$$\begin{aligned} &Agent ::= \\ &\qquad\qquad \mid \text{agent}(\,Nat\,) \end{aligned}$$

$\langle\underset{\mathsf{agent}(\,n\,)}{\underline{\mathtt{new\text{-}agent}\ \ s}}\ \cdots\rangle_{\mathsf{k}}\ \langle me\rangle_{\mathsf{me}}\ \langle\mathsf{agent}(\ \underset{n+\ 1}{\underline{\ \ n\ \ }}\ )\rangle_{\mathsf{nextAgent}}\ \underset{NewAgent}{\underline{\qquad\cdot\qquad}}$, where *NewAgent* stands for:

$$\langle\langle\langle s\rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{env}}\ \langle\cdot\rangle_{\mathsf{holds}}\rangle_{\mathsf{thread}}\ \langle\cdot\rangle_{\mathsf{busy}}\ \langle \mathtt{agent(}\,n\,\mathtt{)}\rangle_{\mathsf{me}}\ \langle me\rangle_{\mathsf{parent}}\ \langle\cdot\rangle_{\mathsf{store}}\ \langle 0\rangle_{\mathsf{nextLoc}}\ \langle\cdot\rangle_{\mathsf{aspect}}\rangle_{\mathsf{agent}}$$

When all threads inside an agent have completed, the agent can be dissolved.

$$\langle\langle\_\rangle_{\mathsf{store}}\ \langle\_\rangle_{\mathsf{nextLoc}}\ \langle\_\rangle_{\mathsf{aspect}}\ \langle\_\rangle_{\mathsf{busy}}\ \langle\_\rangle_{\mathsf{me}}\ \langle\_\rangle_{\mathsf{parent}}\rangle_{\mathsf{agent}} \longrightarrow \cdot$$

An agent can send any value (including agents ids) to other agents (provided it knows their id). To model asynchronous communication, each value sent is wrapped in a 'message' cell identifying both the sender and the intended receiver.

$$\langle\underset{a}{\underline{\mathtt{me}}}\ \cdots\rangle_{\mathsf{k}}\ \langle a\rangle_{\mathsf{me}}$$

$$\langle\underset{a}{\underline{\mathtt{parent}}}\ \cdots\rangle_{\mathsf{k}}\ \langle a\rangle_{\mathsf{parent}}$$

$$\langle\underset{\cdot}{\underline{\mathtt{send\text{-}asynch}\ \ a\,v}}\ \cdots\rangle_{\mathsf{k}}\ \langle me\rangle_{\mathsf{me}}\ \underset{\langle\mathtt{[}\,me\,\mathtt{,}\,a\,\mathtt{,}\,v\,\mathtt{]}\rangle_{\mathsf{message}}}{\underline{\qquad\cdot\qquad}}$$

An agent can request to receive a message from a certain agent, or from any agent.

$$\langle\underset{v}{\underline{\mathtt{receive\text{-}from}\ \ a}}\ \cdots\rangle_{\mathsf{k}}\ \langle me\rangle_{\mathsf{me}}\ \underset{\cdot}{\underline{\langle\mathtt{[}\,a\,\mathtt{,}\,me\,\mathtt{,}\,v\,\mathtt{]}\rangle_{\mathsf{message}}}}$$

$$\langle\underset{v}{\underline{\mathtt{receive}}}\ \cdots\rangle_{\mathsf{k}}\ \langle me\rangle_{\mathsf{me}}\ \underset{\cdot}{\underline{\langle\mathtt{[}\,\_\,\mathtt{,}\,me\,\mathtt{,}\,v\,\mathtt{]}\rangle_{\mathsf{message}}}}$$

The message can be sent synchronously, in which case, two agents need to matched together for the exchange to occur.

$$\langle\cdots\ \langle\underset{\cdot}{\underline{\mathtt{send\text{-}synch}\ \ a\,v}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{agent}}\ \langle\cdots\ \langle\underset{v}{\underline{\mathtt{receive}}}\ \cdots\rangle_{\mathsf{k}}\ \langle a\rangle_{\mathsf{me}}\ \cdots\rangle_{\mathsf{agent}}$$

$$\langle\cdots\ \langle\underset{\cdot}{\underline{\mathtt{send\text{-}synch}\ \ a\,v}}\ \cdots\rangle_{\mathsf{k}}\ \langle me\rangle_{\mathsf{me}}\ \cdots\rangle_{\mathsf{agent}}\ \langle\cdots\ \langle\underset{v}{\underline{\mathtt{receive\text{-}from}\ \ me}}\ \cdots\rangle_{\mathsf{k}}\ \langle a\rangle_{\mathsf{me}}\ \cdots\rangle_{\mathsf{agent}}$$

Abrupt termination. Execution of '`halt`' in one of the threads of an agent dissolves that agent.

$$\underset{\cdot}{\underline{\langle\cdots\ \langle\mathtt{halt}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{agent}}}}$$

Code Generation. The equations below implement the quote/unquote mechanism of runtime code generation. First, the syntax of computations is extended to include "boxed" versions of the K arrow and K list constructors, as well as the helping operator quote, which tracks the level of nesting of quoted expressions. Similarly, boxed versions are introduced for all K labels (ad thus for all language constructs), and a new Label code is added to hold values of type code

$$\begin{aligned}
&\mathit{KProper} ::= \ldots\\
&\qquad\qquad \mid K \boxed{,} K\ \text{[strict]}\\
&\qquad\qquad \mid K \boxed{\curvearrowright} K\ \text{[strict]}\\
&\qquad\qquad \mid \text{quote}(\,\mathit{Nat}\,,\mathit{List\{K\}}\,)\\
&\mathit{KProperLabel} ::= \ldots\\
&\qquad\qquad\qquad \mid \boxed{\sim}\\
&\qquad\qquad\qquad \mid \boxed{\mathit{KLabel}}\ [\mathit{strict}]\\
&\mathit{KResultLabel} ::= \ldots\\
&\qquad\qquad\qquad \mid \text{code}
\end{aligned}$$

Initially, a quoted expression starts at the quoting level 0, and acts as a morphism on all K constructors, transforming them into their boxed version, except for the quote/unquote labels. Each of this boxed items is supposed to evaluate to a code fragment; once this happens, the fragments are glued together to a bigger code fragment.

$$\langle\ \underset{\texttt{quote}(\,0\,,\,k\,)}{\underline{\texttt{quote}\ \ k}}\ \ \cdots\rangle_{\mathsf{k}}$$

$$\texttt{quote}(\,n\,,\,k_1 \curvearrowright k_2\,) \rightleftharpoons \texttt{quote}(\,n\,,\,k_1\,)\,\boxed{\curvearrowright}\,\texttt{quote}(\,n\,,\,k_2\,)$$

$$\texttt{code}\,(\,k_1\,)\,\boxed{\curvearrowright}\,\texttt{code}\,(\,k_2\,) \rightleftharpoons \texttt{code}\,(\,k_1 \curvearrowright k_2\,)$$

$$\texttt{quote}(\,n\,,\,label\,(\,ks\,)\,) \rightleftharpoons \boxed{label}\,(\,\texttt{quote}(\,n\,,\,ks\,)\,)\ \text{when}\ label \neq\ \texttt{quote}\ \ \textit{and}\ label \neq\ \texttt{unquote}$$

$$\boxed{label}\,(\,\texttt{code}\,(\,ks\,)\,) \rightleftharpoons \texttt{code}\,(\,label\,(\,ks\,)\,)$$

$$\texttt{quote}(\,n\,,\,\sim(\,kl\,)\,) \rightleftharpoons \boxed{\sim}\,(\,\texttt{quote}(\,n\,,\,kl\,)\,)$$

$$\boxed{\sim}\,(\,\texttt{code}\,(\,kl\,)\,) \rightleftharpoons \texttt{code}\,(\,\sim(\,kl\,)\,)$$

$$\texttt{quote}(\,\texttt{s}\ \ n\,,\,\texttt{unquote}\ \ k\,) \rightleftharpoons \boxed{\texttt{unquote}\ }\,(\,\texttt{quote}(\,n\,,\,k\,)\,)$$

$$\texttt{quote}(\,n\,,\,k\,,\,ks\,) \rightleftharpoons \texttt{quote}(\,n\,,\,k\,)\,\boxed{,}\,\texttt{quote}(\,n\,,\,ks\,)$$

$$\texttt{code}\,(\,k\,)\,\boxed{,}\,\texttt{code}\,(\,ks\,) \rightleftharpoons \texttt{code}\,(\,k\,,\,ks\,)$$

Quoting basic K constructs directly yields code

$$\texttt{quote}(\,n\,,\,\cdot\,) \rightleftharpoons \texttt{code}\,(\,\cdot\,)$$

$$\texttt{quote}(\,n\,,\,v\,) \rightleftharpoons \texttt{code}\,(\,v\,)$$

$$\texttt{quote}(\,n\,,\,x\,) \rightleftharpoons \texttt{code}\,(\,x\,)$$

Nesting of quotes and semantics of unquoting.

$$\texttt{quote}(\,n\,,\,\texttt{quote}\ \ k\,) \rightleftharpoons \boxed{\texttt{quote}\ }\,(\,\texttt{quote}(\,\texttt{s}\ \ n\,,\,k\,)\,)$$

$$\texttt{quote}(\,0\,,\,\texttt{unquote}\ \ k\,) \rightleftharpoons k$$

`Eval` expects a value of type code and inserts that code as next tasks of the current computation

$$\texttt{eval}\ \ \texttt{code}\,(\,k\,) \rightleftharpoons k$$

7. Conclusion, Current Work and Future Work

We presented the K semantic framework, consisting of a general purpose concurrent abstract rewriting machine, called KRAM, and a technique specialized for defining concurrent programming languages or systems. We presented compelling arguments that K brings together all the advantages of the existing language definitional frameworks, adds true concurrency to them, and avoids their limitations.

Although introduced relatively recently, K has already generated a consistent body of research projects and publications. Even from its incipient stages, K aimed at scalability: to define and analyze real-life programming languages. For example, a comprehensive definition of Java 1.4 in Maude was specified following the K technique and used to derive a state-of-art competitive model-checker for Java [26]. More recently, the same definition was adapted and used to verify security-related properties for Java programs [27, 28]. Besides analyzing Java programs, the K technique was also used to define a complete static semantics for *C* which can support checking pluggable domain specific policies, such as units of measurement [29], which can be used to analyze real C programs. Additionally, [30] uses K to define a symbolic semantics for pointer allocation in a C-like languages, aiming to runtime-verify memory safety. A K semantic definition of Scheme R5RS is given in [31], and one for the Beta language in [32].

There are also several tools and techniques based on K. For example, [24] describes the K-Maude prototype, a Maude implementation of the K framework which fully supports the K definitional style as portraited here. A module system for K aiming at maximizing modularity and code-reuse is discussed in [33]. The potential for efficient executability of K definitions has been first empirically noted in [34]. An experimental object-oriented programming language is defined using K in [35–37], together with several formal analyses and optimizations based on it. The K framework is compared in [20] with P-systems [19] and shown that it can be systematically used to define, execute and analyze P-systems. Matching logic is a new axiomatic semantics extending the benefits of both Hoare logic and separation logics, which fundamentally relies on the K framework [38, 39]. As shown in [22], K can also be effectively used to define type inferencers and prove their soundness w.r.t. the language semantics.

Future Work. Theoretically, we plan (1) to provide a stand-along model theory for K specifications and (2) to analyze in depth the relationships between K and other definitional frameworks. Practically, we intend to automatically generate very efficient and correct-by-construction interpretors for programming languages from their K semantics.

References

[1] G. Rosu, CS322, Fall 2003 - Programming Language Design: Lecture Notes, Tech. Rep. UIUCDCS-R-2003-2897, Department of Computer Science, University of Illinois at Urbana-Champaign, lecture notes of a course taught at UIUC (December 2003).

[2] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, C. Talcott, All About Maude, A High-Performance Logical Framework, Vol. 4350 of Lecture Notes in Computer Science, Springer, 2007.

[3] G. Rosu, Programmming Languages—A Rewriting Approach—, draft, accesible online at: http://fsl.cs.uiuc.edu/pub/pl.pdf.

[4] G. Rosu, K: A rewriting-based framework for computations – preliminary version, Tech. Rep. Department of Computer Science UIUCDCS-R-2007-2926 and College of Engineering UILU-ENG-2007-1827, University of Illinois at Urbana-Champaign (2007).

[5] J. Meseguer, Conditioned rewriting logic as a united model of concurrency, Theoretical Computer Science 96 (1) (1992) 73–155.

[6] T. F. Serbanuta, G. Rosu, J. Meseguer, A rewriting logic approach to operational semantics, Information and Computation 207 (2009) 305–340.

[7] G. Kahn, Natural semantics, in: F.-J. Brandenburg, G. Vidal-Naquet, M. Wirsing (Eds.), STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings, Vol. 247 of Lecture Notes in Computer Science, Springer, 1987, pp. 22–39.

[8] R. Milner, M. Tofte, R. Harper, D. Macqueen, The Definition of Standard ML (Revised), MIT Press, Cambridge, MA, USA, 1997.

[9] G. D. Plotkin, A structural approach to operational semantics, Journal of Logic and Algebraic Programming 60-61 (2004) 17–139, original version: University of Aarhus Technical Report DAIMI FN-19, 1981.

[10] P. D. Mosses, Pragmatics of modular SOS, in: H. Kirchner, C. Ringeissen (Eds.), Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings, Vol. 2422 of Lecture Notes in Computer Science, Springer, 2002, pp. 21–40.

[11] P. D. Mosses, Modular structural operational semantics, Journal of Logic and Algebraic Programming 60-61 (2004) 195–228.

[12] A. K. Wright, M. Felleisen, A syntactic approach to type soundness, Information and Computation 115 (1) (1994) 38–94.

[13] O. Danvy, L. R. Nielsen, Refocusing in reduction semantics, RS RS-04-26, BRICS, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, this report supersedes BRICS report RS-02-04. A preliminary version appears in the informal proceedings of the *Second International Workshop on Rule-Based Programming*, RULE 2001, Electronic Notes in Theoretical Computer Science, Vol. 59.4. (November 2004).

[14] J. Meseguer, G. Rosu, Rewriting logic semantics: From language specifications to formal analysis tools, in: D. A. Basin, M. Rusinowitch (Eds.), Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings, Vol. 3097 of Lecture Notes in Computer Science, Springer, 2004, pp. 1–44.

[15] J. Meseguer, G. Rosu, The rewriting logic semantics project, Theoretical Computer Science 373 (3) (2007) 213–237.

[16] J. Meseguer, Rewriting logic as a semantic framework for concurrency: a progress report, in: U. Montanari, V. Sassone (Eds.), CONCUR, Vol. 1119 of Lecture Notes in Computer Science, Springer, 1996, pp. 331–372.

[17] H. Ehrig, Introduction to the algebraic theory of graph grammars (a survey), in: V. Claus, H. Ehrig, G. Rozenberg (Eds.), Graph-Grammars and Their Application to Computer Science and Biology, Vol. 73 of Lecture Notes in Computer Science, Springer, 1978, pp. 1–69.

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

[19] G. Paun, Computing with membranes, Journal of Computer and System Sciences 61 (2000) 108–143.

[20] T. F. Serbănuţă, G. Stefănescu, G. Rosu, Defining and executing P systems with structured data in K, in: D. W. Corne, P. Frisco, G. Paun, G. Rozenberg, A. Salomaa (Eds.), Workshop on Membrane Computing (WMC'08), Vol. 5391 of Lecture Notes in Computer Science, Springer, 2009, pp. 374–393.

[21] M. Felleisen, D. P. Friedman, Control operators, the SECD-machine, and the lambda-calculus, in: 3rd Working Conference on the Formal Description of Programming Concepts, Ebberup, Denmark, 1986, pp. 193–219.

[22] C. Ellison, T. F. Serbănuţă, G. Rosu, A rewriting logic approach to type inference, in: Recent Trends in Algebraic Development Techniques — 19th International Workshop, WADT 2008, Pisa, Italy, June 13-16, 2008, Revised Selected Papers, Vol. 5486 of Lecture Notes in Computer Science, Springer, 2009, pp. 135–151.

[23] J. Meseguer, Rewriting as a unified model of concurrency, in: J. C. M. Baeten, J. W. Klop (Eds.), CONCUR, Vol. 458 of Lecture Notes in Computer Science, Springer, 1990, pp. 384–400.

[24] T. F. Serbanută, G. Rosu, K-Maude: A rewriting based tool for semantics of programming languages, in: Proceedings of the 8th International Workshop on Rewriting Logic and its Applications (WRLA'10), Lecture Notes in Computer Science, 2010, to appear.

[25] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, J.-P. Jouannaud, Introducing OBJ, in: J. Goguen (Ed.), Applications of Algebraic Specification using OBJ, Cambridge, 1993.

[26] A. Farzan, F. Chen, J. Meseguer, G. Rosu, Formal analysis of Java programs in JavaFAN, in: Proceedings of Computer-aided Verification (CAV'04), Vol. 3114 of LNCS, 2004, pp. 501 – 505.

[27] M. Alba-Castro, M. Alpuente, S. Escobar, Automatic certification of Java source code in rewriting logic, in: S. Leue, P. Merino (Eds.), FMICS, Vol. 4916 of Lecture Notes in Computer Science, Springer, 2007, pp. 200–217.

[28] M. Alba-Castro, M. Alpuente, S. Escobar, P. Ojeda, D. Romero, A tool for automated certification of Java source code in Maude, in: Proceedings of the Eighth Spanish Conference on Programming and Computer Languages (PROLE 2008), Vol. 248 of Electr. Notes Theor. Comput. Sci., 2009, pp. 19–29.

[29] M. Hills, F. Chen, G. Rosu, A Rewriting Logic Approach to Static Checking of Units of Measurement in C, in: Proceedings of the 9th International Workshop on Rule-Based Programming (RULE'08), Vol. To Appear of ENTCS, Elsevier, 2008.

[30] G. Rosu, W. Schulte, T. F. Serbanuta, Runtime verification of C memory safety, in: Runtime Verification (RV'09), Vol. 5779 of Lecture Notes in Computer Science, 2009, pp. 132–152.

[31] G. R. Patrick Meredith, Mark Hills, An Executable Rewriting Logic Semantics of K-Scheme, in: D. Dube (Ed.), Proceedings of the 2007 Workshop on Scheme and Functional Programming (SCHEME'07), Technical Report DIUL-RT-0701, Laval University, 2007, pp. 91–103.

[32] M. Hills, T. B. Aktemur, G. Rosu, An Executable Semantic Definition of the Beta Language using Rewriting Logic, Tech. Rep. UIUCDCS-R-2005-2650, Department of Computer Science, University of Illinois at Urbana-Champaign (2005).

[33] M. Hills, G. Rosu, Towards a module system for K, in: A. Corradini, U. Montanari (Eds.), WADT, Vol. 5486 of Lecture Notes in Computer Science, Springer, 2008, pp. 187–205.

[34] M. Hills, T. F. Serbănută, G. Rosu, A rewrite framework for language definitions and for generation of efficient interpreters, in: Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA'06), Vol. 176 of Electronic Notes in Theoretical Computer Science, Elsevier Science, 2007, pp. 215–231, also appeared as Technical Report UIUCDCS-R-2005-2667, December 2005.

[35] M. Hills, G. Rosu, A Rewriting Approach to the Design and Evolution of Object-Oriented Languages, in: OOPSLA '07: Companion to the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, ACM, New York, NY, USA, 2007, pp. 827–828.

[36] M. Hills, G. Rosu, Kool: An application of rewriting logic to language prototyping and analysis, in: F. Baader (Ed.), RTA, Vol. 4533 of Lecture Notes in Computer Science, Springer, 2007, pp. 246–256.

[37] M. Hills, G. Rosu, On Formal Analysis of OO Languages using Rewriting Logic: Designing for Performance, in: Proceedings of the 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07), Vol. 4468 of LNCS, Springer, 2007, pp. 107–121, also appeared as Technical Report UIUCDCS-R-2007-2809, January 2007.

[38] G. Rosu, W. Schulte, Matching logic — extended report, Tech. Rep. Department of Computer Science UIUCDCS-R-2009-3026, University of Illinois at Urbana-Champaign (January 2009).

[39] G. Rosu, C. Ellison, W. Schulte, From rewriting logic executable semantics to matching logic program verification, Tech. Rep. http://hdl.handle.net/2142/13159, University of Illinois (July 2009).
URL http://hdl.handle.net/2142/13159