J.LAP, Volume 79(6), pp 397-434. 2010

An Overview of the K Semantic Framework

Grigore Roșu and Traian Florin Șerbănuță

University of Illinois
{grosu,tserban2}@illinois.edu

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 explicit which parts of the term they read, write, 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 to challenge and expose the limitations of 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 course at the University of Illinois at Urbana-Champaign (UIUC) in Fall 2003 [36], as a means to define executable concurrent languages in rewriting logic using Maude [8]. 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 [37, 38].

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, but also to the entire research community:

> Is there any language definitional framework which, at the same time,
>
> 1. Gives a unified approach to define not only languages but also language-related abstractions, such as type checkers, type inferencers, abstract interpreters, safety policy or domain-specific checkers, etc.? The current state-of-the art is that language designers use different approaches or styles to define different aspects of a language, sometimes even to define different components of the same aspect.
> 2. Can define arbitrarily complex language features, including, obviously, all those found in existing languages, capturing also their intended computational granularity? For example, features like call-with-current-continuation and true concurrency are hard or impossible to define in many existing frameworks.

3. Is modular, i.e., adding new language features does not require modifying existing definitions of unrelated features? Modularity is crucial for scalability and reuse.
4. Supports non-determinism and concurrency, at any desired granularity?
5. Is generic, that is, not tied to any particular programming language or paradigm?
6. 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.
7. Has state-exploration capabilities, including exhaustive behavior analysis (e.g., model-checking), when one's language is non-deterministic or/and concurrent?
8. Has a corresponding initial-model or axiomatic semantics (to allow inductive or 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 requirements of an ideal language definitional framework that are more subjective and thus more difficult to quantify. 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 understanding 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, so that an external parser is not needed.

The requirements above are nevertheless ambitious. 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 [37, 38] 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 achieve a dynamic semantics of a language, but that defining type systems, proving properties about programs, model checking, etc., are "something else" and therefore are allowed to require a different "encoding" of the language. Our position is that an ideal language definitional framework should not compromise any of the 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 requirements discussed above, promoting none of them at the expense of others.

The K framework consists of two components: *K rewriting*, discussed in Section 4; and the *K technique*, discussed in Section 5. Like a term rewrite system, a K-system consists of a signature for building terms and of a set of rules for iteratively rewriting terms. Like in rewriting logic [24], K rules can be applied concurrently and unrestricted by context. Moreover, K rules contain information about what part of the matched term is left unchanged by the rule (called the *read-only* part), similar to interfaces in graph rewriting [10]. Besides offering a more compact notation for the rewrite rules, identifying the read-only part can also potentially enhance the concurrency: now two overlapping rules can be applied in parallel if they only overlap on their read-only part, similar to the concept of parallel independence in graph rewriting [10]. However, if one is not interested in the degree of concurrency allowed by a K definition, one can safely ignore the information regarding the read-only part, and view K rules as an alternative notation for rewrite rules. Our prototype implementation [42] takes this approach, to benefit from the existing infrastructure and analysis tools provided by the Maude [7] rewrite engine.

A rewriting formalism, be it rewriting logic or K rewriting, can be very general and flexible, but it does not tell *how* one can define a programming language or a calculus. In particular, even though the

underlying framework might offer support for concurrency, a bad definition may partially or totally inhibit the framework's concurrency potential. The K technique discussed in Section 5 proposes an approach and supporting notation to make the use of K rewriting, or even rewriting in general, convenient when formally defining programming languages and calculi.

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 intuition behind K rewriting, that is, how K rules can increase concurrency. Section 5 presents the K technique, explaining essentially how rewriting can be used to define programming language semantics by means of nested-cell configurations, computations and rewrite rules. Section 6 shows K at work: it introduces and shows how to give a K semantics to CHALLENGE, a programming language containing various 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. Section 7 concludes the paper, lists some recent research based on K, and gives 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 main characteristics, limitations, and 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 [23], also named relational/evaluation semantics [29], big-step semantics is "the most denotational" of the operational semantics. One can view big-step definitions as definitions of relations interpreting each language construct in an appropriate domain. Deterministic (or functional) big-step operational semantics can be easily and efficiently interpreted/implemented. 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: The transitive closure of the K rewrite relation can be seen as a big-step relation.

Small-step SOS Introduced by Plotkin [35], also called transition (or reduction) semantics, small-step semantics captures the notion of one computational step. Therefore, it stops at errors, pointing them out. 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: 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 may grow

unbounded (e.g., through loop/fixed-point unrolling), each small step may take unbounded resources in the worst case, making it difficult to interpret efficiently.
Relationship to K: K's rewriting is small-step in spirit; however, K rewrite rules can apply anywhere and concurrently, while SOS transitions only apply at the top of the term, interleaved.

Modular SOS (MSOS) Mosses [30, 31] introduced MSOS to deal with the non-modularity of existing SOS styles. The MSOS solution involves moving the non-syntactic state components into the labels on transitions, plus a discipline of only selecting needed attributes from the states.
Limitations: Being inherently an SOS, MSOS can still only define interleaving semantics to concurrent languages. While the use of labels gives MSOS the ability to modularly deal with some forms of control, such as abrupt termination, at our knowledge it still cannot support the definition of arbitrarily complex control-intensive features such call/cc. Also, MSOS is driven by the structure of the syntax and appears to have no escape mechanism to reflectively traverse, mark, modify and/or store arbitrary syntax in labels, which makes it hard or impossible to modularly define language constructs for code generation (e.g., `quote`/`unquote`/`eval` —see Section 6).
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 only to refer to configuration items of interest, which is crucial for modularity. MSOS' labels have an additional role, to yield labeled transition systems, while K's cell names are not intended to be used for that. One can label K rules as we did with the rules ρ_r and ρ_w in Section 4, and, in principle, one can incorporate in K the same complex rule labeling of rewrite logic [24]. However, we have not done that. So far, we used additional cells in the configuration to represent/store the emitted signals.

Reduction semantics with evaluation contexts Introduced by Felleisen and colleagues (see, e.g., [45]), 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 (rather than small-step SOS rules) to find the next redex; and (2) it provides the possibility to also modify the context in which a reduction occurs, making it much easier to deal with control-intensive features, in particular to define constructs like call/cc. 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" polynomial (linear best case, quadratic worst-case) in the size of the program for each computational step.However, one might obtain efficient implementations for restricted forms of context-reduction definitions by applying refocusing techniques [9].
Relationship to K: Both reduction semantics and K make use of evaluation contexts and can store or modify them. Reduction semantics "splits/plugs" syntax into contexts, while K "heats/cools" 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 [24] as a logical formalism generalizing both equational logic and term rewriting, so it is more suitable for language semantics than plain term rewriting. The rewriting logic semantics (RLS) project [27, 28] 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 [27, 28], 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 the Maude rewrite engine [43].
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, such as those in Section 2.1, need to be represented within rewriting as shown in [43]. 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 [25] on how to use equations in an ingenious way to still achieve theoretical true concurrency in the presence of sharing, but at the expense of losing executability of definitions.
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 [10] 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 in dealing with structure copying and equality testing, operations which are crucial for programming language semantics and relatively easy to provide in term rewriting systems. Finally, in spite of decades of research, to 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 read-write components.

The Chemical Abstract Machine (CHAM) Berry and Boudol [5] introduced CHAM to give language semantics within the Gamma model [3], a multiset-rewriting model of computation. The CHAM views a distributed state as a "solution" where "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 [24]. 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, to 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 in order to 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.

P-Systems Păun's membrane systems (or P-systems) [34] 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 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 allows a very limited kind of rules (symport/antiport [33]) 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.
Relationship to K: K-systems share with P-systems the ideas of cell structures and the aim to maximize concurrency; for example the read-only parts of a K rule play a similar role to promotors/inhibitors [6] from P-systems. 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). An in-depth comparison between the two formalisms, showing how one can use K to model P-systems can be found in [44].

There are several other biology-inspired computational frameworks, such as the Calculus of Looping Sequences [4] and MGS [14], which share with K the aims for concurrency and use one form or another of rewriting as their evolution mechanism.

3 K Overview by Example

The role of this section is threefold: (1) it gives the reader a better understanding of the K framework before we proceed to define it rigorously in the remainder of the paper; (2) it shows how K avoids some of the limitations of other semantic approaches; and (3) it shows that K is actually easy to use. In this section we make no distinction between K rewriting (presented in Section 4) and the K technique (detailed in Section 5), referring to these collectively as "the K framework", or more simply just "K". 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) input and 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 framework, K, to define both dynamic and static semantics of languages.

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 may have ordinary side conditions, though, as rule schemata), and they are *context-insensitive*, so K rules apply concurrently as soon as they match, without any contextual delay or restrictions.

Computations 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, capturing the intuition of computation sequentialization, with list constructor $\_ \curvearrowright \_$ (read "followed by") and unit "$\cdot$" (the empty computation). Computations give an elegant and uniform means to define and handle evaluation contexts [45] and/or continuations [13]. 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 term 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. For example, two threads can read the same location of memory 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 [5] or membrane systems [34], 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. Like other semantic approaches that can be represented in rewrite logic [43], K can also be represented in rewrite 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 rewrite 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 rewrite logic are *faithful*, in that the resulting rewrite logic theories are step-for-step equivalent with the original definitions, K cannot be captured faithfully by rewrite logic in any natural way; more precisely, the resulting rewrite logic theory may need more interleaved steps 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 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 context-free grammar (CFG) notation for syntax and for configurations, extended with specialized "algebraic" notation for lists, sets, multisets (bag) and maps. For any sort S, the sort $\mathsf{List}^{\dagger}_{\star}[S]$ (or $\mathsf{Bag}^{\dagger}_{\star}[S]$, or $\mathsf{Set}^{\dagger}_{\star}[S]$) defines the $\star$-separated lists (or bags, or sets) of elements of sort S, with identity $\dagger$. If unspecified, by default $\star$ is $\_,\_$ for lists and $\_\ \_$ for bags and sets, and $\dagger$ is "$\cdot$". For example, $\mathsf{List}[S]$ defines comma-separated lists of elements of type S, and could be expressed with the lower-level CFG productions $\mathsf{List}[S] ::= \cdot \mid S(,S)^*$. Similarly, sort $\mathsf{Map}^{\dagger}_{\star}[S_1 \mapsto S_2]$ contains a set of mappings *source* $\mapsto$ *target*, with *source* of sort S_1 and *target* of sort S_2, separated by $\star$ and with identity $\dagger$; be default, $\star$ is $\_\ \_$ and $\dagger$ is "$\cdot$".

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 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:

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

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 a cell $\langle\rangle_{\mathsf{state}}$ which holds a map from variables to integers. For example, "$\langle \langle x = 1;\ y = x + 1; \rangle_{\mathsf{k}}\ \langle \cdot \rangle_{\mathsf{state}} \rangle_{\top}$" is a configuration holding program "$x = 1;\ y = x + 1;$" and empty state, and "$\langle \langle x = 1;\ y = x + 1; \rangle_{\mathsf{k}}\ \langle x \mapsto 0, y \mapsto 1 \rangle_{\mathsf{state}} \rangle_{\top}$" is a configuration holding the same program and a state mapping x to 0 and y to 1.

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 reduction semantics with evaluation contexts, K allows one to define evaluation contexts over the language syntax. However, unlike in reduction semantics, parsing does not play any crucial role in K, because K replaces the hard-to-implement split/plug operations of evaluation contexts 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* (the sum "+" is non-deterministic, i.e., the evaluation procedure for its arguments is not fixed and the comparison "<=" is sequential):

$$\begin{aligned} a_1 + a_2 &\rightleftharpoons a_1 \curvearrowright \square + a_2 \\ a_1 + a_2 &\rightleftharpoons a_2 \curvearrowright a_1 + \square \\ a_1 \texttt{<=} a_2 &\rightleftharpoons a_1 \curvearrowright \square \texttt{<=} a_2 \\ i_1 \texttt{<=} a_2 &\rightleftharpoons a_2 \curvearrowright i_1 \texttt{<=} \square \\ \texttt{if}\, b\, \texttt{then}\, s_1\, \texttt{else}\, s_2 &\rightleftharpoons b \curvearrowright \texttt{if}\square\, \texttt{then}\, s_1\, \texttt{else}\, s_2 \end{aligned}$$

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 + 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 called *heating/cooling rules* in K, since they are reminiscent of the CHAM heating/cooling rules; for example, a_1+a_2 is "heated" into $a_1 \curvearrowright \square + a_2$, while $a_1 \curvearrowright \square + a_2$ is "cooled" into a_1+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, meaning that all specified arguments must be evaluated before evaluating the construct itself, 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 do not 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.

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 operations which are not underlined represent the read-only part of the term: they need to stay unchanged during the application of the rule. The anonymous variables "_" represent parts of the term that the current rule does not care about and, consequently, are allowed to be concurrently modified by other rules. By convention, when mentioning anonymous variables at the beginning or the end of cells, the

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

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

list/bag/set/map constructor can be omitted. For example, the lookup rule (the first rule in Figure 1) 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 `=` i`;`" reaches the top of the computation, the value of x in the store is replaced by i and the statement dissolves; "$\cdot$" is the unit (or empty) computation ("$\cdot$" tends to be used in K as 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 signify 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 $l \longrightarrow r$ as syntactic sugar for computational rules and the notation $l \rightharpoonup r$ or $l \rightharpoondown r$ 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: $s_1 \ s_2$ 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 on when rules should be computational versus structural and, in the latter case, on when one should use $l \rightharpoonup r$ or $l \rightharpoondown r$ 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 "$\rightharpoondown$" when syntax is split into computational tasks and "$\rightharpoonup$" 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 K-system lacks the potential for concurrency of the original K-system.

Original language syntax		Strictness	Semantics
$AExp ::= \ldots$	$\texttt{++}\,Id$		$\langle \frac{\texttt{++}\,x}{i +_{Int} 1} \ \_\rangle_{\mathsf{k}} \ \langle \_ \ x \mapsto \frac{i}{i +_{Int} 1} \ \_\rangle_{\mathsf{state}}$
	`read`		$\langle \frac{\texttt{read}}{i} \ \_\rangle_{\mathsf{k}} \ \langle \frac{i}{\cdot} \ \_\rangle_{\mathsf{in}}$
$Stmt ::= \ldots$	$\texttt{print}\,AExp\,\texttt{;}$	$[strict]$	$\langle \frac{\texttt{print}\,i\,\texttt{;}}{\cdot} \ \_\rangle_{\mathsf{k}} \ \langle \_ \ \frac{\cdot}{i}\rangle_{\mathsf{out}}$
	$\texttt{spawn}\,Stmt$		$\langle \frac{\texttt{spawn}\,s}{\cdot} \ \_\rangle_{\mathsf{k}} \ \frac{\cdot}{\langle s\rangle_{\mathsf{k}}}$
	`haltThread;`		$\langle \frac{\texttt{haltThread;} \curvearrowright \_}{\cdot}\rangle_{\mathsf{k}}$
			$\langle\cdot\rangle_{\mathsf{k}} \rightharpoonup \cdot$

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

3.2 K Semantics of IMP++

Figure 2 shows how the K semantics of IMP can be seamlessly extended into a semantics for IMP++. To accommodate input and output, two new cells need to be added to the configuration:

$$Configuration_{\text{IMP++}} \quad \equiv \quad \langle\langle K\rangle_{\mathsf{k}} \ \langle \mathsf{Map}[Id \mapsto Int]\rangle_{\mathsf{state}} \ \langle \mathsf{List}[Int]\rangle_{\mathsf{out}} \ \langle \mathsf{List}[Int]\rangle_{\mathsf{in}}\rangle_{\top}$$

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 increment construct "`++_`" introduces side effects for expressions: it increments the value of the identifier and evaluates to that value. The rule for the `read` construct uses the first element from the $\langle\rangle_{\mathsf{input}}$ cell (and replaces it by the unit "$\cdot$") to replace the `read` expression. The `print` construct is strict and its rule adds the value of its argument to the end of the $\langle\rangle_{\mathsf{out}}$ buffer (matches and replaces the unit "$\cdot$" at the end of the buffer). The rule for `haltThread` dissolves the current computation, and the rule for `spawn` creates a new $\langle\rangle_{\mathsf{k}}$ cell initialized with 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 instances of the look up rule can apply concurrently, in spite of the fact that they overlap on the state. Similarly, since the rules for variable assignment and increment do not update anything else in the $\langle\rangle_{\mathsf{state}}$ cell 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 rules for `read`/`print` match and change the beginning/end of the $\langle\rangle_{\mathsf{in}}/\langle\rangle_{\mathsf{out}}$ cell. That means, in particular, that multiple `read`/`print` statements by various threads need to be interleaved for the same reason as above; however, one `read` could be executed in parallel with one `print` command. 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 of IMP++ are possible to define formally thanks to the specific nature of the K rules. If 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 define

Original language syntax	Strictness	Semantics
$AExp ::= Int$		$i \rightarrow int$
$\mid Id$		$\langle \underline{x} \;\; \_ \rangle_{\mathsf{k}} \; \langle \_ \;\; x \;\; \_ \rangle_{\mathsf{vars}}$ int
$\mid AExp$ `+` $AExp$	$[strict]$	int `+` $int \rightarrow int$
$\mid AExp$ `/` $AExp$	$[strict]$	int `/` $int \rightarrow int$
$\mid$ `++` Id		$\langle \underline{\texttt{++}\, x} \;\; \_ \rangle_{\mathsf{k}} \; \langle \_ \;\; x \;\; \_ \rangle_{\mathsf{vars}}$ int
$\mid$ `read`		`read` $\rightarrow int$
$BExp ::= AExp$ `<=` $AExp$	$[strict]$	int `<=` $int \rightarrow bool$
$\mid$ `not` $BExp$	$[strict]$	`not` $bool \rightarrow bool$
$\mid BExp$ `and` $BExp$	$[strict]$	$bool$ `and` $bool \rightarrow bool$
$Stmt ::=$ `skip;`		`skip;` $\rightarrow stmt$
$\mid Id$ `=` $AExp$`;`	$[strict(2)]$	$\langle \underline{x \texttt{ = } int\texttt{;}} \;\; \_ \rangle_{\mathsf{k}} \; \langle \_ \;\; x \;\; \_ \rangle_{\mathsf{vars}}$ $stmt$
$\mid Stmt \;\; Stmt$	$[strict]$	$stmt \;\; stmt \rightarrow stmt$
$\mid$ `if` $BExp$ `then` $Stmt$ `else` $Stmt$	$[strict]$	`if` $bool$ `then` $stmt$ `else` $stmt \rightarrow stmt$
$\mid$ `while` $BExp$ `do` $Stmt$	$[strict]$	`while` $bool$ `do` $stmt \rightarrow stmt$
$\mid$ `print` $AExp$`;`	$[strict]$	`print` int`;` $\rightarrow stmt$
$\mid$ `haltThread;`		`haltThread;` $\rightarrow stmt$
$\mid$ `spawn` $Stmt$	$[strict]$	`spawn` $stmt \rightarrow stmt$
$Pgm ::=$ `var` $\mathsf{List}[Id]$`;` $Stmt$		$\langle \texttt{var}\; xl\texttt{;}\; s \rangle_{\mathsf{k}} \; \langle \cdot \rangle_{\mathsf{vars}}$ $s \curvearrowright pgm \qquad xl$ $stmt \curvearrowright pgm \rightarrow pgm$

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

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. We next 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 values. The technique is general and has been used to define more complex type systems, such as higher-order polymorphic ones [11].

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.):

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

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 are exceptions, 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 int. If we defined, e.g., the assignment strict and with rule $int = int$, then our type system would allow ill-formed programs like "$x + y = 0;$". Note how we defined the typing policy of programs "`var` xl`;` s": the declared variables xl are stored into the $\langle\rangle_{\mathsf{vars}}$ cell (expected to be 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 K rewriting

Rewriting logic deduction [24] 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 is realized through message passing, then the rules of the system need to only apply inside a process state, or to match a process together with a message addressed to it. However, this is not the case when modeling 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 simultaneously read the same parts of the global state, provided that they do not modify them. Take for example the following rules for reading/writing a variable in the state (obtained by translating the corresponding IMP++ rules into rewrite rules):

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

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

Consider a system containing only these rules, and let $\langle a \rangle_{\mathsf{k}}\ \langle a \rangle_{\mathsf{k}}\ \langle b = 3\,; \rangle_{\mathsf{k}}\ \langle a \mapsto 1\ \ b \mapsto 2 \rangle_{\mathsf{state}}$ be the (ground) term to be rewritten, i.e., two threads $\langle a \rangle_{\mathsf{k}}$ whose tasks are to read a from the state, and a thread $\langle b = 3\,; \rangle_{\mathsf{k}}$ updating the value of b. All threads could advance simultaneously: the first two by reading the value of a (since a is shared), and the third by updating the value of b (since the location of b is independent of that of a). However, this is impossible with the deduction rules of rewriting logic because "the same object cannot be shared by two simultaneous rewrites" [26].

One reason is that traditional matching modulo axioms requires that the term be rearranged to fit the pattern, and thus it limits concurrency where sharing is allowed: there is no way to re-arrange the term so that any two of the rule instances match simultaneously. To address that, we propose a special treatment for matching operators governed by axioms (Section 4.4). Another, more fundamental reason, is that the top set constructor operation and the state itself need to be shared for the rules to apply. The K rules (Section 4.2) address this issue by distinguishing the read-only part of a rule pattern, which can be shared, and by making the change local to the exact position in which the update must be applied.

4.1 Preliminaries

A signature Σ is a pair (S, F) where S is a set of *sorts* and F is a set of operations $f : w \to s$, where f is an operation symbol, $w \in S^*$ is its arity, and $s \in S$ is its result sort. If w is the empty word ϵ then f is a constant. 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 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. One can associate a signature to any CFG, so that well-formed words in the CFG language are associated corresponding terms in the signature (their AST).

Substitutions A substitution is a mapping yielding terms (possibly with variables) for variables. Any substitution $\psi : \mathcal{X} \to T_\Sigma(\mathcal{Y})$ naturally extends to terms, yielding a homonymous mapping $\psi : T_\Sigma(\mathcal{X}) \to T_\Sigma(\mathcal{Y})$. When $\mathcal{X}$ is finite and small, the application of substitution $\psi : \{x_1, \ldots, x_n\} \to T_\Sigma(\mathcal{Y})$ to term t can be

written as $t[\psi(x_1)/x_1, \ldots, \psi(x_n)/x_n] ::= \psi(t)$. This notation allows one to use substitutions by need, without formally defining them.

Contexts Given an ordered set of variables, $\mathscr{W} = \{\Box_1, \ldots, \Box_n\}$, 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 exactly once). The instantiation of a $\mathscr{W}$-context C with an n-tuple $\overline{t} = (t_1, \ldots, t_n)$, written $C[\overline{t}]$ or $C[t_1, \ldots, t_n]$, is the term $C[t_1/\Box_1, \ldots, t_n/\Box_n]$. One can alternatively regard $\overline{t}$ as a substitution $\overline{t} : \mathscr{W} \to T_\Sigma(X)$, defined by $\overline{t}(\Box_i) = t_i$, in which case $C[\overline{t}] = \overline{t}(C)$. A Σ-context is a $\mathscr{W}$ context over Σ where $\mathscr{W}$ is a singleton.

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 only discuss *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 $(\forall \mathcal{X})\ l \to r$ *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 *rewrites* to $C[\theta[r]]$. A (Σ-)*rewrite-system* $\mathcal{R} = (\Sigma, R)$ is a set R of Σ-rewrite rules.

4.2 K rules and K-systems

K rules are schemata 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; however, each rule identifies a read-only pattern, the local context of the rule. This pattern is used to glue together read-write patterns, that is, subparts to be rewritten. Moreover, through its variables, it also provides information which can be used and shared by the read-write patterns. To some extent, the read-only pattern plays here the same role played by interfaces in graph rewriting [10].

Similar in spirit with the distinction between equations and rules in rewriting logic, K rules are of two types: *structural* (which rearrange the term without altering the flow of computation) and *computational* (which advance the state of the system). 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 a signature $\Sigma = (S, F)$ is a tuple $(\mathcal{X}, p, L, R)$, where:*

- *$\mathcal{X}$ is an S-sorted set, called the **variables** of the rule ρ;*
- *p 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 "read-write" part of ρ.*

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

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

The variables in $\mathcal{W}$ are only used to formally identify the positions in p where rewriting takes place; 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 $\mathcal{X}$ is clear, it can be omitted. Structural variables, that is, variables which are only introduced for putting the relevant terms into context, and thus are neither used in $R(w)$ for any $w \in \mathcal{W}$, nor constrained in any other way (like occurring multiple times among the variables of the terms in $\{p\} \cup \{L(w) \mid w \in \mathcal{W}\}$), are termed *anonymous* or *nameless* variables. We take the liberty to denote all anonymous variables by the "_" symbol (like in Prolog), but recall that they are all distinct.

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 state, respectively:

$$\rho_r\colon \langle \underline{x} \curvearrowright \_ \rangle_{\mathsf{k}}\ \langle \_\ x \mapsto i \rangle_{\mathsf{state}} \qquad \Big|\qquad \rho_w\colon \langle \underline{x = i'\,;} \curvearrowright \_ \rangle_{\mathsf{k}}\ \langle \_\ x \mapsto \underline{\_} \rangle_{\mathsf{state}}$$

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

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

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

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

4.3 Relation to rewrite rules

Here we analyze the relationship between rewrite rules and K rules, showing that the latter are a conservative extension of the former. Consider the two mappings defined below:

$$R2K((\forall \mathcal{X})\ l \to r) = (\forall \mathcal{X})\ \underline{l} \;\Big/\; \overline{r} \qquad \Big|\qquad K2R((\forall \mathcal{X})\ p[\ \underline{L}\ \Big/\ \overline{R}\ ]) = (\forall \mathcal{X})\ L(p) \to R(p)$$

R2K associates to each rewrite rule a K rule quantified by the same variables, having as the read-only pattern just a hole $\Box$ which is mapped by L and R to the left-hand-side and right-hand-side of the rewrite rule, respectively. Conversely, *K2R* associates to each K-rule a rewrite rule by forgetting the additional information contained by the K rule and flattening it by applying L and R to the read-only pattern to obtain the lhs and the rhs of the rewrite rule, respectively.

K rules faithfully capture conventional rewrite rules, since $K2R(R2K(\varrho)) = \varrho$ 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, because it is not always straightforward to determine what should be shared and what not.

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 $R2K_{max}$. Given a rewrite rule $\varrho\colon (\forall \mathcal{X})\ l \to r$, the *maximally sharing K rule* associated to ϱ is $R2K_{max}(\varrho) = (\forall \mathcal{X})\ p[\ \underline{L}\ \Big/\ \overline{R}\ ]$ satisfying that $K2R(R2K_{max}(\varrho)) = \varrho$, and that, for any other K rule $\rho'\colon (\forall \mathcal{X})\ p'[\ \underline{L'}\ \Big/\ \overline{R'}\ ]$ such that $K2R(\rho') = \varrho$, p' is a specialization of p, that is, there exists a substitution $\theta : vars(p) \to T_\Sigma(\mathcal{X} \cup vars(p'))$ such that $p' = \theta(p)$.

It might seem that maximal sharing is always desirable; however maximal sharing does not always model the intended behavior. Consider the rewrite rule $a\ * \to b\ *$, where a, b, and $*$ are constants of sort s and "$\_\ \_ : ss \to s$" is an associative and commutative (AC) 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 is $\underline{a}\ *\ \Big/\ \overline{b}$; as seen shortly, such a rule

allows a term $a\ a\ *$ to rewrite in one concurrent step to $b\ b\ *$. However, if $*$ is to be regarded as a token ensuring mutual exclusion, then the corresponding K rule is $\frac{a\ *}{b\ *}$; as seen shortly, such rules cannot be applied concurrently on $a\ a\ *$, requiring, due to the overlapping of the token "$*$", two interleaved steps.

The remainder of this section gives a theoretical account of the K rewriting method, arguing that the read-only information contained in K rules can be used to enhance the potential of parallelism of rewriting. Nevertheless, K-Maude [42], our current prototype implementation of the K framework, simply reduces K rules to rewrite rules through the *K2R* transformation, to make them directly executable in the Maude rewrite engine.

4.4 Matching K rules

One could give a straightforward 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* $\rho : (\forall \mathcal{X})\ p[\ \frac{L}{R}\ ]$ ***matches*** *a* Σ*-term* t *using context* C *and substitution* θ *if and only if its corresponding rewrite rule* $K2R(\rho)$ *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 rules in parallel we can observe the benefits of the K rules.

As seen in Section 3, the K framework employs lists, sets, bags, and maps to give semantics. Moreover, it is precisely the bag structure of the configuration that allows K to give truly concurrent semantics to languages (partly seen in the semantics of IMP++ in Section 3, and heavily used in Section 6). However, concurrent matching modulo (ACU) axioms in the presence of sharing appears to be a rather difficult problem. Recall the motivating example from the beginning of Section 4, in which we attempted to concurrently read and write in the state. The structure of K-rules introduced above, explicitly representing read-only parts of the rules, allows multiple rules to overlap on the $\langle\rangle_{\mathsf{store}}$ cell at the same time. However, the problem of matching multiple rules simultaneously still remains, because the traditional treatment of matching modulo axioms requires the term to be rearranged (using the axioms) to conform to the rule, and thus prevents multiple rules to actually match at the same time. The remainder part of this subsection shows how concurrency can be enhanced by handling matching modulo (ACU) axioms differently.

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

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

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

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

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

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

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

4. Resolving Associativity For each rule containing subterms of the form $t_1 \star t_2 \star \cdots \star t_n$ where $\star$ is an associative operator, generate rules containing all possible ways to put parentheses so that each occurrence of $\_ \star \_$ has only two arguments. Note that matching terms containing subterms built from new variables using only $\star$ these rules need not be considered, as they will be equivalent with rules containing just one new variable instead of that subterm. Keeping this in mind, the following patterns are the final concrete patterns associated to the ones presented above: $\langle x \mapsto v \rangle_{\mathsf{state}}$, $\langle \sigma \ x \mapsto v \rangle_{\mathsf{state}}$, $\langle x \mapsto v \ \sigma \rangle_{\mathsf{state}}$, $\langle \sigma_1 \ (\sigma_2 \ x \mapsto v) \rangle_{\mathsf{state}}$, $\langle (\sigma_1 \ x \mapsto v) \ \sigma_2 \rangle_{\mathsf{state}}$, $\langle \sigma_1 \ (x \mapsto v \ \sigma_2) \rangle_{\mathsf{state}}$, and $\langle (x \mapsto v \ \sigma_1) \ \sigma_2 \rangle_{\mathsf{state}}$, and so on.

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

Example: Matching the IMP++ variable read/write rules Let us now show that interpreting K rules as rule schemata as described above allows multiple concurrent matching instances for our motivating example introducing Section 4. Recall the two rules (in their K form):

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

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

4.5 K Concurrent Rewriting

In Section 4.2 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 rewriting logic. In this section we intuitively describe how a term is matched by multiple rules, and indicate how all of the matched rules can be applied concurrently. A formalization of this intuitive description can be found in [38].

As described above, the intuition in a K rule $(\forall \mathcal{X})\ p[\ \underset{R}{L}\ ]$ 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 read-write, 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 instances 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 read-write 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 rule. 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 [10] 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 conclude this section by showing how we can finally achieve the goal of concurrently advancing all threads from the motivating example at the beginning of this section. To do that, we use the coloring policies described above to combine the concrete matching rules associated to rule schemata ρ_r and ρ_w from the end of previous subsection. We will use underlining to represent red, and boxing to represent green. Let us start with the original term $((\langle a\rangle_{\mathsf{k}}\ \langle a\rangle_{\mathsf{k}})\ \langle b = 3;\rangle_{\mathsf{k}})\ \langle a \mapsto 1\ \ b \mapsto 2\rangle_{\mathsf{state}}$, and let us match rule $\rho_{r,2}$ first, yielding the coloring:

$$((\langle a\rangle_{\mathsf{k}}\ \boxed{\ }\ \boxed{\langle}\ \underline{a}\ \boxed{\rangle_{\mathsf{k}}})\ \boxed{\ }\ \langle b = 3;\rangle_{\mathsf{k}})\ \boxed{\ }\ \boxed{\langle}\ \boxed{a}\ \boxed{\mapsto}\ \boxed{1}\ \boxed{\ }\ b \mapsto 2\boxed{\rangle_{state}}$$

Rule $\rho_{r,1}$ also matches since the colors are consistent, and colors the first continuation cell in green and the a inside it in red: $((\boxed{\langle}\ \underline{a}\ \boxed{\rangle_k}\ \square\ \boxed{\langle}\ \underline{a}\ \boxed{\rangle_k})\ \square\ \langle b = 3\,;\rangle_k)\ \square\ \boxed{\langle}\ \boxed{a}\ \boxed{\mapsto}\ \boxed{1}\ \square\ b \mapsto 2\ \boxed{\rangle_{state}}$.

Finally, rule $\rho_{w,1}$ can also match without conflicts, coloring all remainder positions:

$$((\boxed{\langle}\ \underline{a}\ \boxed{\rangle_k}\ \square\ \boxed{\langle}\ \underline{a}\ \boxed{\rangle_k})\ \square\ \boxed{\langle}\ \underline{b = 3\,;}\ \boxed{\rangle_k})\ \square\ \boxed{\langle}\ \boxed{a}\ \boxed{\mapsto}\ \boxed{1}\ \square\ \boxed{b}\ \boxed{\mapsto}\ \underline{2}\ \boxed{\rangle_{state}}.$$

Since all matches succeeded, we can apply all three rules simultaneously, obtaining the term

$$((\langle 1\rangle_k\ \langle 1\rangle_k)\ \langle\cdot\rangle_k)\ \langle a \mapsto 1\ \ b \mapsto 3\rangle_{\mathsf{state}}.$$

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 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, K rewriting (discussed in Section 4) can be used in various ways in various applications; in other words, it 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 K rewriting, 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, there are no rigid rules for how to write an SOS configuration [35]: one may use the angle-bracket notation $\langle code, state, \ldots\rangle$, the square bracket notation $[code, state, \ldots]$, or even the simple tuple notation $(code, state, \ldots)$; 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 [29]) 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 implementation of K [42] relies on our current notation as described in this section; therefore, to use our tool 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 K rewriting as a back-end 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 [43], the K technique can also be cast as a definitional methodology or style within other computational frameworks. In [38, 42] 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 10, which is a cell term over the simple cell grammar described below. By defining the configuration you have three benefits:

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

In K definitions, the programming language, calculus or system configuration is represented as a potentially *nested cell* structure. This is similar in spirit to how configurations are represented in chemical abstract machines (CHAMs, see [5]) or in membrane systems (P-systems, see [34]), 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:

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

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 stands for "*top*") and k (which stays for "*computation*"). They are built-in in our implementation of K in Maude [42], so one needs not declare them in each language definition. The white-space or "invisible" label "$\_$" 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 transformation (Section 5.5).

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

We have seen so far three K configurations, for IMP, for IMP++, and for their type system:

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

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, Figure 10 presents the K configuration of CHALLENGE (see Section 6), a language conceived to challenge and expose the limitations of the language definitional frameworks. Figure 10 shows both a textual representation of CHALLENGE configurations (as the ones for IMP and IMP++ above), as well as a graphical one, which can be automatically generated by the K2Latex component of our current implementation of K in Maude [42]. The CHALLENGE configurations have five 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 $\langle\rangle_{\mathsf{agent}}$ 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 $\langle\rangle_{\mathsf{thread}}$ cells grouped in a $\langle\rangle_{\mathsf{threads}}$ cell; 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 [31], 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 configuration items do not need to be touched. This is an important aspect of K, which significantly contributes to its modularity.

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 transformation* (see Section 5.5), 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 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 [45] in K, then the only computations 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: Using 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 standing for *computational structures* or *computations*, and terms t of sort K have the abstract syntax tree (AST) representation $l(t_1, \ldots, t_n)$, where l is some K *label* and $t_1, \ldots, t_n$ 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:

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

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:

$$\begin{array}{rcl} \mathit{KLabel}_{\mathrm{IMP}} & ::= & \mathit{Int} \mid \mathit{Id} \mid \texttt{\_+\_} \mid \texttt{\_/\_} \mid \texttt{\_<=\_} \mid \texttt{not\_} \mid \texttt{\_and\_} \mid \texttt{skip;} \mid \texttt{\_=\_;} \mid \texttt{\_ \_} \\ & \mid & \texttt{if\_then\_else\_} \mid \texttt{while\_do\_} \mid \texttt{var\_; \_} \end{array}$$

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 [15] and followed by many other similar 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; e.g, labels corresponding to semantic domain values which may have not been automatically included in the syntax

[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, for simplicity we prefer to keep only one computation sort for now.

of the language, such as the *Bool* domain in the case of IMP. We take the liberty to call *K constants* those computations which are labels applied to, and always intended to be applied to, an empty list of arguments (e.g., `skip;`(), `true`(), 1(), 2(), etc.)

Most K labels always take a fixed number of arguments; e.g., the label `if_then_else_` above takes 3 arguments. Even though the simplistic syntax of K cannot enforce the fixed number of arguments in the semantics, one can show that the semantic rules never change the number of arguments of such labels, so they will always have the original number of arguments as given by the original parsing of the program. There are syntactic language constructs, however, which are allowed to take an arbitrary number of arguments. A typical example is, for example, lists of expressions. Consider, for example, the CHALLENGE language discussed in Section 6. Like many other languages, CHALLENGE includes functions (as λ-abstractions). The function application construct, `_(_)`, is defined to take an expression as first argument (expected to evaluate to a function value) and a comma-separated list of expressions as second argument: *Exp* (List[*Exp*]). A list of expressions "e_1`,`e_2`,`...`,`e_n" is captured as a K computation of the form "`_,_`$(t_1, t_2, ..., t_n)$" (with `_,_`() for the empty list), where t_i is the K representation of e_i. Therefore, lists of expressions are regarded as labels applied to an arbitrary number of arguments; the name of the label is inspired from list constructs being thought of as binary associative operations.

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

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

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 λ`_._; indeed, in some $\lambda$-calculi, $\lambda$-abstractions $\lambda x.e$ (or $\lambda$`_._(x, e) in AST form), are values (or finished computations).

There is yet another category of labels that turns out to be useful in semantic definitions, namely *hybrid* labels, which are intended to "hold data", i.e., take lists of K results into a K result:

$$\begin{array}{rcl} \mathit{KResult} & ::= & \mathit{KHybridLabel}(\mathsf{List}\sim[K]) \\ \mathit{KHybridLabel} & ::= & \text{(one per construct that does not reduce once its arguments are reduced)} \end{array}$$

For example, the "list" label `_,_` above should be declared hybrid, since we want `_,_`$(t_1, t_2, ..., t_n)$ to be considered evaluated in the semantics whenever each t_i is evaluated. On the other hand, labels like `_+_` are obviously not hybrid. In fact, hybrid labels are rather rare. It may also be worth noting that, unlike the result labels, the hybrid labels are more of a convenience than a necessity. Indeed, one can always introduce a new result label for any label intended to be hybrid, e.g. `_,`$_{result}$`_`, together with a rule replacing the label with its result counterpart whenever its arguments become results, e.g., "`_,_`$(t_1, t_2, ..., t_n) \rightharpoonup$ `_,`$_{result}$`_`$(t_1, t_2, ..., t_n)$ when $t_1, t_2, ..., t_n \in$ *KResult*" (a structural rule, see Section 5.3). However, this would be inconvenient in many cases.

We take the liberty to write language or calculus syntax either in AST form, like in λ`_._$(x, e)$, `if_then_else_`$(b, s_1, s_2)$, and `skip`(), or in mixfix form, like in $\lambda x.e$, `if` $b$ `then` $s_1$ `else` $s_2$, and `skip`. For example, in Figure 1 we preferred to write the rule for addition as $i_1$ `+` $i_2 \rightarrow i_1 +_{Int} i_2$ instead of `_+_`$(i_1(), i_2()) \rightarrow (i_1 +_{Int} i_2)()$. In our Maude implementation of K [42], thanks to Maude's builtin support for mixfix notation and 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. 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 fragments 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 type constraints in k_1 then do the same for k_2", etc. Figure 4 shows examples of computations making use of the $\mathsf{List}_{\curvearrowright}[K]$ structure of K (we use parentheses for disambiguation). The "$\cdot$" in the first and last computations in Figure 4 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 $\square$ in Figure 4 are "`_+`$\square$", "$\square$`+_`", and "`if`$\square$`then_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 [42], all these are generated automatically as constants of sort *KLabel* after a simple analysis of the language syntax.

$(\texttt{if true then} \cdot \texttt{else} \cdot) \curvearrowright \texttt{while false do} \cdot$
$a_1 \curvearrowright \square + a_2$
$a_2 \curvearrowright a_1 + \square$
$a_3 \curvearrowright (a_1 + a_2) + \square$
$a_3 \curvearrowright (a_1 \curvearrowright \square + a_2) + \square$
$b \curvearrowright \texttt{if} \square \texttt{then} \, s_1 \, \texttt{else} \, s_2$
$b \curvearrowright \texttt{if} \square \texttt{then} \, (s \curvearrowright \texttt{while} \, b \, \texttt{do} \, s) \, \texttt{else} \cdot$

Figure 4: Examples of K computations

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 $c[k, x_1, \ldots, x_n]$ for a multicontext c and a freezer is needed to freeze everything except k, then its name is "$c[\square, \_, \ldots, \_]$".

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 $c[\square, k_1, \ldots, k_n]$ instead of $c[\square, \_, \ldots, \_](k_1, \ldots, k_n)$. 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 "$\mathcal{P} ::= (\mathit{Exp} @ \mathcal{P}) @ (\mathit{Exp} @ \mathit{Exp})$".

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 K rewriting; how the K rewriting 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 [42] does so.

The K framework builds upon K rewriting. As mentioned in Section 4, K rules can be split into computational and structural. From here on, we distinguish them as shown in Figure 5. 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 "read-write" 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 so that computational rules can apply. 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 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 some language designers may prefer it to be computational.

Computational rules

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

Structural rules

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

Figure 5: K rules

Recall from Section 3.1 that we prefer to use the conventional rewrite rule notations "$l \to r$" and "$l \rightharpoonup r$" for computational and structural K rules, respectively, when $p = \square$ (that is, when there is only one read-write part, the entire pattern, and no read-only part). There is not much to say about K rules besides what has already been said in Sections 3 and 4. We would only like to elaborate a bit further on 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. 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, freezers 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 the 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 two heating/cooling pairs of K rules in Figure 6 (a_1 and a_2 range over computations in K). The symbol "$\rightleftharpoons$" is borrowed from the chemical abstract machine (CHAM) [5], as a shorthand for combinations of a heating rule ("$\rightharpoonup$") and a cooling rule ("$\leftharpoondown$"). Indeed, one

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

Figure 6: Rules for _+_ *strict*

can think of the first rule 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. These heating/cooling rules can be applied at any moment and in any direction, since they are regarded not as computational steps but as structural rearrangements. For example, one can use the heating/cooling rules for "`_+_`" to pick and pass in front either a_1 or a_2, then rewrite it one step only using semantic rules (defined shortly), 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 $c[\Box, N_1, \ldots, N_n]$, 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 $c[\Box, \_, \ldots, \_]$ like discussed in Section 5.2, together with a heating/cooling rule pair "$c[k, k_1, \ldots, k_n] \rightleftharpoons k \curvearrowright c[\Box, k_1, \ldots, k_n]$". 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:

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

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 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, Figure 7 shows the computational class of the expression $x * (y + 2)$ in the context of a language definition with non-deterministically strict binary $+$ and $*$. 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.

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

Figure 7: Computational class

5.4 Strict and Hybrid 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 `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 rule pair "`if` b `then` s_1 `else` $s_2 \rightleftharpoons b \curvearrowright$ `if` $\Box$ `then` s_1 `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 with 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 `_<=_` desugars into the rules in Figure 8 ($a_1, a_2 \in K$ and $r_1 \in KResult$). 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.

$$a_1 \texttt{<=} a_2 \rightleftharpoons a_1 \curvearrowright \square \texttt{<=} a_2$$
$$r_1 \texttt{<=} a_2 \rightleftharpoons a_2 \curvearrowright r_1 \texttt{<=} \square$$

Figure 8: Rules for `_<=_` *seqstrict*

The strictness attributes also apply to labels taking an arbitrary number of arguments. For example, the expression-list construct `_,_` of CHALLENGE in Section 6 is declared *strict*, meaning that all the expressions appearing in the list need to evaluate whenever the expression-list is asked to evaluate. The general desugaring rules of strictness can be best given in terms of K label representations. For example, a *strict* attribute associated to $label \in KLabel$ is syntactic sugar for rules like the ones below ($kl_1, kl_2 \in \mathsf{List}[K]$, $k \in K$, $r \in KResult$):

$$label(kl_1, k, kl_2) \rightleftharpoons k \curvearrowright label(\_, \square, \_)(klist(kl_1), klist(kl_2))$$
$$r \curvearrowright label(\_, \square, \_)(klist(kl_1), klist(kl_2)) \rightleftharpoons label(kl_1, r, kl_2)$$

where $label(\_, \square, \_)$ and *klist* are labels. This way, the lists kl_1 and kl_2 are unambiguously frozen until k is reduced to a result. For *seqstrict*, replace $kl_1 \in \mathsf{List}[K]$ by $rl_1 \in \mathsf{List}[KResult]$ above.

Our most general strictness declaration in K, also supported by our current implementation [42], is to declare a certain syntactic context (a derived term) strict in a certain position designated by a special marker $\square$. For example, in the K definition of CHALLENGE in Section 6, we declare the context "`*` $\square$ `=` `_`", with the meaning that the assignment statement applied to a pointer needs to first evaluate the pointer expression before other semantic rules can apply.

Language constructs can also be annotated with the *hybrid* attribute, to indicate that their corresponding label is hybrid (see Section 5.2).

5.5 Context Transformation

We next introduce one of the most advanced feature of K, the *context transformation*, which gives K an additional degree of modularity. The process of context transformation 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 transformation 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 a necessary step in order to make use of K's context transformation. Assuming that the various cell-labels forming the configuration are distinct, then one can use the structure of the configuration to *automatically* transform abstract rules, 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. The context transformation process can be thought of as being applied statically on all rules, before the K-system is executed.

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

$$\langle \underset{\cdot}{\texttt{print}\, i\texttt{;}} \;\; \_ \rangle_{\mathsf{k}} \; \langle \_ \;\; \underset{i}{\cdot} \rangle_{\mathsf{out}}$$

This rule captures the semantics of the output statement in the most abstract and compact possible way: if "`print` i`;`" is the next computational task, then append i to the end of the $\langle\rangle_{\mathsf{out}}$ buffer and dissolve the `print` statement. This rule 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 is shown in Figure 10. 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 transformation 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 $\langle\rangle_{\mathsf{out}}$ mentioned in the rule above together: to wrap them with the cells above them declared in the CHALLENGE configuration until a common cell contents is reached, namely to transform the rule into the following one:

$$\langle \_ \;\; \langle \_ \;\; \langle \_ \;\; \langle \_ \;\; \langle \underset{\cdot}{\texttt{print}\, i\texttt{;}} \;\; \_ \rangle_{\mathsf{k}} \;\; \_ \rangle_{\mathsf{thread}} \;\; \_ \rangle_{\mathsf{threads}} \;\; \_ \rangle_{\mathsf{agent}} \;\; \_ \rangle_{\mathsf{agents}} \; \langle \_ \;\; \langle \_ \;\; \underset{i}{\cdot} \rangle_{\mathsf{out}} \;\; \_ \rangle_{\mathsf{I/O}}$$

Context transformation can be defined as the process of customizing the paths to the various cells used in a rule according to the configuration of the target language. As part of this customization process, 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.

5.5.1 The Locality Principle

The rule above was rather simple, in that there was no confusion on how to complete the paths to the referred cells. Consider instead a general-purpose K rule for pointer dereferencing:

$$\langle \underset{v}{\texttt{*}\, n} \;\; \_ \rangle_{\mathsf{k}} \; \langle \_ \;\; n \mapsto v \;\; \_ \rangle_{\mathsf{store}}$$

This says that if dereferencing of location n is the next computational task and if v is stored at location n, then $\texttt{*}\, n$ 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 transformation operation is necessary to adapt this rule to CHALLENGE's concrete configuration. However, without care, there are two ways to do it:

$$\langle \_ \;\; \langle \_ \;\; \langle \underset{v}{\texttt{*}\, n} \;\; \_ \rangle_{\mathsf{k}} \;\; \_ \rangle_{\mathsf{thread}} \;\; \_ \rangle_{\mathsf{threads}} \; \langle \_ \;\; n \mapsto v \;\; \_ \rangle_{\mathsf{store}}$$

$$\langle \_ \;\; \langle \_ \;\; \langle \_ \;\; \langle \underset{v}{\texttt{*}\, n} \;\; \_ \rangle_{\mathsf{k}} \;\; \_ \rangle_{\mathsf{thread}} \;\; \_ \rangle_{\mathsf{threads}} \;\; \_ \rangle_{\mathsf{agent}} \; \langle \_ \;\; \langle \_ \;\; n \mapsto v \;\; \_ \rangle_{\mathsf{store}} \;\; \_ \rangle_{\mathsf{agent}}$$

The first 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 make sense according to the configuration of CHALLENGE.

To avoid such conflicts, context transformation relies on the *locality principle*: rules are transformed in a way that makes them as local as possible, or, in other words, in a way that the resulting rule matches as deeply as possible in the concrete configuration. Thus, the locality principle rules out the second rule transformation above, since 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 context transformation of the dereferencing rule above, then one should have written the abstract rule, for example, as follows:

$$\langle \_ \ \langle \underline{*n \atop v} \ \_\rangle_{\mathsf{k}} \ \_\rangle_{\mathsf{agent}} \langle \_ \ n \mapsto v \ \_\rangle_{\mathsf{store}}$$

Now there is only one way to transform it to fit the configuration of CHALLENGE, namely like in the second CHALLENGE-concrete rule above. Indeed, the $\langle\rangle_{\mathsf{store}}$ cell can only be 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, which must reside in a $\langle\rangle_{\mathsf{threads}}$ cell. Context transformation applies at all levels in the rule context.

Let us 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 bound to their addresses in the environment and their addresses to their values in the store):

$$\langle \underline{x \atop k} \ \_\rangle_{\mathsf{k}} \ \langle \_ \ x \mapsto n \ \_\rangle_{\mathsf{env}} \ \langle \_ \ n \mapsto k \ \_\rangle_{\mathsf{store}}$$

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:

$$\langle \_ \ \langle \_ \ \langle \underline{x \atop k} \ \_\rangle_{\mathsf{k}} \ \langle \_ \ x \mapsto n \ \_\rangle_{\mathsf{env}} \ \_\rangle_{\mathsf{thread}} \ \_\rangle_{\mathsf{threads}} \ \langle \_ \ n \mapsto k \ \_\rangle_{\mathsf{store}}$$

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.

5.5.2 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 $\langle\rangle_{\mathsf{agent}}$ 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 "`rendezvous` v`;`" blocks until another thread also reaches an identical "`rendezvous` v`;`" statement, and, in that case, both threads unblock and continue their execution. The following K rule captures this desired behavior of rendezvous synchronization:

$$\langle \underline{\texttt{rendezvous}\ v\texttt{;} \atop \cdot} \ \_\rangle_{\mathsf{k}} \ \langle \underline{\texttt{rendezvous}\ v\texttt{;} \atop \cdot} \ \_\rangle_{\mathsf{k}}$$

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 Section 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 transformation 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 transformation 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 10 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:

$$\langle \_ \ \langle \underline{\texttt{rendezvous}\ v\texttt{;} \atop \cdot} \ \_\rangle_{\mathsf{k}} \ \_\rangle_{\mathsf{thread}} \ \langle \_ \ \langle \underline{\texttt{rendezvous}\ v\texttt{;} \atop \cdot} \ \_\rangle_{\mathsf{k}} \ \_\rangle_{\mathsf{thread}}$$

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 transformation 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.

5.5.3 Context transformation with default values

When creating a new agent, or spawning a new thread, a new cell for that agent/thread must be created, together with all the structure of nested cells below it. Moreover, all leaf cells need to be initialized with appropriate values. However, while the structure of the newly created cell can be quite complex, only a few cells are initialized with non-default values. Also, having to specify the entire cell makes the rule depend on a particular configuration, and thus non-modular: if the designer decides to add a new cell or to re-organize existing cells, the rule would have to be updated. To address all these issues, we allow partially specified configurations to appear in the replacement part of a K rule, and context-transform them by adding all missing cells and initializing the leaf ones with their default values. The default values can either be given for each sort or for each cell once and for all; in our K-Maude tool [42], for example, we place them inside each cell when we define the configuration. The rule of agent creation can then be written as ("_" variables are used in the replacement part to indicate that it must be context-transformed):

$$\frac{\langle \underline{\texttt{new-agent}\ s} \quad \_\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}}\ \langle \frac{n}{n +_{Int} 1}\rangle_{\mathsf{nextAgent}}}{n} \quad \frac{\cdot}{\langle \_ \quad \langle s\rangle_{\mathsf{k}}\langle n\rangle_{\mathsf{me}}\ \langle m\rangle_{\mathsf{parent}} \quad \_\rangle_{\mathsf{agent}}}$$

Upon applying the context transformation, the $\langle\rangle_{\mathsf{agent}}$ cell is completed to:

$$\langle\langle\langle\langle s\rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{env}}\ \langle\cdot\rangle_{\mathsf{holds}}\ \langle\cdot\rangle_{\mathsf{fstack}}\rangle_{\mathsf{thread}}\rangle_{\mathsf{threads}}\ \langle\cdot\rangle_{\mathsf{store}}\ \langle\cdot\rangle_{\mathsf{ptr}}\ \langle 0\rangle_{\mathsf{nextLoc}}\ \langle\cdot\rangle_{\mathsf{aspect}}\ \langle\cdot\rangle_{\mathsf{busy}}\ \langle n\rangle_{\mathsf{me}}\ \langle m\rangle_{\mathsf{parent}}\rangle_{\mathsf{agent}}$$

6 The CHALLENGE Language

This section gives the K semantic definition of CHALLENGE, a non-trivial experimental programming language aimed at challenging existing or future language definitional frameworks. All language constructs of CHALLENGE can be found in existing languages, possibly in a more general form and possibly using a different syntax. None of CHALLENGE's features are artificially crafted and the language, as a whole, is plausible and powerful.

The rationale for choosing features that already exist in popular languages is to avoid criticism, from proponents of frameworks that cannot handle the proposed features, that those features are useless, and thus do not deserve attention. Our standpoint is that the very fact that a feature exists in a mainstream language is sufficient evidence that the feature is useful, so an ideal semantic framework should unarguably support it. While a language designer may chose not to include such a feature in her language, a language-design framework designer aiming at an ideal framework has no choice and should support it. The CHALLENGE's features, and particularly their combination, have been chosen to satisfy two criteria:

1. To capture *the essence* of a certain category of conventional language concepts; if a semantic framework cannot handle a certain CHALLENGE feature then that framework cannot handle a potentially wide range of desirable features;

2. To illustrate how one *modularly* defines a complex language in K: each new feature is added without changing any of the already defined features. We here assume that one knows upfront the entire language and that at each step one takes the globally best design decisions.

6.1 CHALLENGE Syntax and Informal Meaning

Figure 9 shows the syntax of CHALLENGE, which is split into two syntactic categories: expressions (*Exp*) and statements (*Stmt*). To save space and give the reader an early intuition of the evaluation strategies of the various constructs, the syntax of CHALLENGE in Figure 9 is annotated with K strictness attributes (Section 5.4). The language starts with arithmetic expressions, then gradually grows in complexity by adding: expressions with side effects; memory allocation and pointers; lists; aspects; functions; recursion;

```
Exp ::= Bool | Int | Real | Id
      | Exp binaryArithOps Exp [strict]
      | unaryArithOps Exp [strict]
      | ++ Id
      | malloc Exp [strict]
      | & Id
      | * Exp [strict]
      | [List[Exp]] [strict hybrid]
      | Exp : Exp [strict]
      | head Exp [strict]
      | tail Exp [strict]
      | λ (List[Id]) . Stmt
      | Exp (List[Exp]) [strict]
      | μ Id . Exp
      | callcc Exp [strict]
      | read
      | randomBool
      | newAgent Stmt
      | me | parent
      | receive
      | receiveFrom Exp [strict]
      | quote Exp
      | unquote Exp
      | lift Exp
      | eval Exp [strict]
Stmt ::= {} | {Stmt} | Stmt Stmt
      | var List[Id];
      | Exp; [strict]
      | Exp = Exp; [strict(2)]
      | if Exp then Stmt else Stmt [strict(1)]
      | while Exp do Stmt
      | return Exp; [strict]
      | aspect Stmt
      | print Exp; [strict]
      | free Exp; [strict]
      | spawn Stmt
      | acquire Exp; [strict]
      | release Exp; [strict]
      | rendezvous Exp; [strict]
      | send Exp to Exp; [strict]
      | sendSynch Exp to Exp; [strict]
      | barrier;
      | broadcast Exp; [strict]
      | haltThread;
      | haltAgent;
      | haltSystem;
```

Figure 9: K annotated CHALLENGE syntax

call with current continuation; non-determinism; multithreading with shared memory and synchronization; agents with synchronous and asynchronous communication, and with broadcasting and barriers; and constructs for code generation and evaluation. We next informally explain the various CHALLENGE constructs and briefly give our reasons for including them in CHALLENGE.

Expressions can be both arithmetic and boolean, and include both integer and real numbers. The arithmetic/boolean expression constructs and their evaluation strategies are similar to those of IMP, which were discussed in Section 3.1. However, we keep the integer and real numbers separate and all the arithmetic and relational operations are overloaded to work with both. Expressions can have side effects, both minimal (as given, e.g., by the variable increment construct "`++`x") and arbitrarily complex (as given, e.g., by λ and μ in combination with arbitrary pointer passing, assignment and arithmetic). Side effects are common in programming languages and their inclusion may require non-modular changes when one uses certain semantic frameworks.

CHALLENGE allows dynamic memory allocation via the `malloc` expression construct and allows memory deallocation with the `free` statement construct. Like in C, "`malloc e`" evaluates `e` (say to n), allocates a block of n contiguous locations, and returns the location where the block starts; `free` can only be called on a location which has been previously returned by a `malloc`, so that the number of locations to free is known. CHALLENGE also allows unrestricted pointer arithmetic, accessing the address at which a variable is allocated (with the expression construct `&`x), dereferencing locations (with the expression construct `*`p), and writing values to locations (using statements of the form `*`p `=` e). Not all languages allow as explicit and direct memory access as CHALLENGE. Nevertheless, a semantic framework able to naturally deal with the memory operations of CHALLENGE can likely support arbitrarily complex memory operations in programming language semantic definitions, including, for example, arrays.

While pointers and explicit memory allocation are powerful enough to mimic any data-structures and operations on them, many languages provide more explicit (built-in or user-defined) data-structures, such as lists, trees, etc. For simplicity, we only include (arbitrary) lists with operations on them (whose syntax is borrowed from Haskell), where x`:`l prepends element x to list l and `head` l and `tail` l give the first element of list l and the remaining elements, respectively.

Any semantic framework must support definitions of languages with functions. Higher-order functions tend to be more complex and there are many higher-order functional languages. We therefore only include higher-order functions in CHALLENGE, and we do it by means of λ-expressions. To stress the framework's capability to syntactically and semantically deal with lists of terms, we allow λ-abstractions taking arbitrary lists of parameters (λ (List[*Id*]) . *Stmt*) and λ-applications taking arbitrary lists of arguments (*Exp* (List[*Exp*])) in CHALLENGE. For example, $(\lambda() \,.\, s)()$, $(\lambda(x) \,.\, s)e$, and $(\lambda(x_1, x_2) \,.\, s)(e_1, e_2)$ are all well-formed CHALLENGE expressions when $x, x_1, x_2 \in$ *Id*, s is a well formed statement, and e, e_1, e_2 are well-formed expressions. Note that we defined the λ-application to be strict both in its first argument (expected to evaluate to a function) and in its second argument; its second argument is a list of expressions, so the application is implicitly strict in all these expressions in the list (i.e., each of the expressions in the list will be evaluated before the function is applied; how this works is explained in Section 5); therefore, CHALLENGE is call-by-value. To stress the framework's ability to define constructs that change the execution flow abruptly, the body of a λ-expression is a statement; to exit it, one needs to use an explicit "`return` *Exp*" statement, which is strict in its argument.

Since recursion is a fundamental design and semantic concept in programming languages, and since recursion can have many apparently different facets, CHALLENGE allows several different means to achieve it: through explicit use of a provided fixed-point μ construct (μ *Id* . *Exp*); through iteration using the provided `while` loop statement construct (`while` *Exp* `do` *Stmt*); through using conventional λ-calculus encodings based on passing functions to themselves (which only work in untyped settings); through assigning to store locations λ-abstractions whose closed environments refer to those locations; and through using continuations via callcc. Even though in theory only one approach to recursion suffices, we strongly believe that a language definitional framework should naturally support all these approaches in practice to give the language designer flexibility.

Many languages have control-intensive constructs, such as abrupt termination, exceptions, break/continue of loops, and even callcc (from "call with current continuation"). Therefore, any language definitional framework must support language constructs that abruptly change the execution flow. Since callcc is as

complex a control-intensive construct as it can be, we included it as part of CHALLENGE (`callcc` *Exp*). We also included `return`, as explained above, and several abrupt termination statements (for threads, agents and program), which we discuss shortly.

Some semantic frameworks, particularly those of a denotational nature, cannot handle non-determinism satisfactorily: while they still allow, in some cases, to define a collecting semantics (i.e., a semantics that collects all behaviors), such a semantics is inappropriate for obtaining an interpreter for the defined language when executing it. To illustrate a framework's capability to deal with non-determinism, CHALLENGE includes the simplest imaginable explicit non-deterministic construct, namely a `randomBool` boolean constant which non-deterministically evaluates to *true* or *false* each time it is evaluated. Concurrency, discussed below, induces implicit non-determinism.

Concurrent software in no longer the exception to the rule, it has become the norm. Any practical language definitional framework should provide good support for concurrency. Therefore, CHALLENGE contains language constructs that stress the underlying framework in several ways with respect to concurrency. It allows dynamic creation and termination of both agents (`newAgent` *Stmt*) and threads (`spawn` *Stmt*). The threads live inside agents and the threads within a given agent communicate and synchronize with each other by means of shared memory, re-entrant acquisition (`acquire` *Exp*) and releasing (`release` *Exp*) of locks (any value can serve as a lock), and by rendezvous barriers (`rendezvous` *Exp*). Agents communicate by means of asynchronous (`send` *Exp* `to` *Exp*) and synchronous (`sendSynch` *Exp* `to` *Exp*) message-send commands (the first argument is the value being passed and the second argument is the receiving agent's name), by targeted (`receiveFrom` *Exp*, where the argument is the sending agent's name) and non-targeted (`receive`) expression receive constructs, and by broadcasting (`broadcast` *Exp*) and global barriers (`barrier`). A framework supporting the above common concurrency features likely can support any degree of concurrency that a language designer may want to include in her language.

To keep a tight connection between the concurrency features and the rest of the language, CHALLENGE includes several other features that do not dictate the semantics of the concurrent ones but interact with them. For example, since there are three types of executing entities, namely threads, agents, and the entire system, it makes sense to have three abrupt termination statements, one for the current thread (`haltThread`), one for the current agent (`haltAgent`) and one for the entire system (`haltSystem`); as expected, the former releases the locks held by the thread. CHALLENGE also has an output that is shared by all threads and agents (via the `print` statement). The constructs `malloc`/`free` act on the memory of the current agent only (so the agent's threads compete on memory), and `callcc` acts on the computation of the current thread only.

Aspect-oriented programming is an important and increasingly growing trend in programming languages. There are many different approaches to aspects (e.g., dynamic versus static aspects) and we cannot possibly include all of them in CHALLENGE. We only include one aspect-oriented feature, namely executing an aspect statement whenever a function is invoked. To make it more interesting, we combine static with dynamic intuitions of aspects in that we weave the current aspect within the body of a function at function declaration time, but we allow one to dynamically change the aspect code (the latter is done via a statement "`aspect` s" setting the argument statement s as the aspect of the current thread. Many variations are possible, particularly in combination with concurrency, but we prefer to keep CHALLENGE minimal. We added the particular aspect feature above for two reasons: first, executing additional code when a function is called is one of the most common aspects; second, it may be inconvenient to give it semantics in purely syntactic approaches (one needs to distinguish between already-weaved and not-yet-weaved functions) or in approaches which reduce desired language features to their "builtin" similar features (e.g., desired functions to their builtin functions, desired recursion to their fixed-points, etc.).

Finally, there are several programming languages providing support for dynamic code generation and execution (e.g., Lisp, Scheme, APL, COBOL, Javascript, Python, MetaOCaml, Jumbo, etc., and almost all assembler languages). We therefore add support for code generation and execution to CHALLENGE, to challenge the framework's meta/reflective capabilities. The expression "`quote` e" freezes expression e into a special code value, and "`eval` e" first evaluates e to a code value, say c, and then unfreezes c and evaluates it in the current environment. To allow for arbitrarily complex code generation and reuse, one can use expressions like "`unquote` e" inside a quoted expression, with the following semantics: evaluate e in the

current environment to a code value, then plug that code value into the quoted expression. Quoting of code can be nested and an expression must be unquoted as many times as quoted in order to be evaluated. To lift values to code values, we also include a "`lift` e" construct which evaluates e and creates a code value from its result. Therefore, like in Jumbo and related languages (but unlike in Lisp and Scheme), we keep a clear separation between code and data in CHALLENGE. Code generation may be hard or impossible to define modularly in many language semantic frameworks, because they do not allow for a meta/reflective view of syntax. Indeed, most frameworks drive a language semantics according to the rigid structure of its syntax, where the individuality of each language construct is important; thus, one would most likely need to define quoting/unquoting for each language construct, which is non-modular. In K, we can define the semantics of `quote`/`unquote`/`eval`/`lift` with a few language-independent rules, using its reflective capabilities for syntax.

As argued above, the purpose of CHALLENGE is to present a minimal set of non-artificial features that we believe any ideal language definitional framework should be able to define modularly. In what follows we show how one can use K to give modular semantics to CHALLENGE. The fact that one can indeed define CHALLENGE in K obviously does not automatically make K an ideal language definitional framework; it only shows that K passes this minimal test.

Once a language syntax is defined in a tool-supported executable semantics framework, one can write and parse several programs that one would like to eventually use to test the semantics. In our experience, it is worth writing tens of programs using each language construct in isolation or combinations of them. These will help the designer not only make aesthetic syntactic changes early in the process if needed, but also have a better understanding of the language overall. Finally, it is a good idea to also write a few more complex programs making intensive use of large subsets or language constructs, even all if possible, such as, for example, the program in Figure 12.

6.2 CHALLENGE Configuration

Figure 10 shows the complete K-configuration of CHALLENGE, both in term representation and in graphical representation. The cell contents is self-explanatory, but more detail will be given as we define the various language constructs in Section 6.3. For notational simplicity and to keep the figure within the page width, we allowed more generous contents for some of the cells in Figure 10; in the semantics, for example, the cell $\langle\rangle_{\mathsf{busy}}$ will take a set of K results only.

6.3 CHALLENGE Semantics

Here we discuss the K semantics of CHALLENGE, skipping all those constructs whose semantics are the same as in IMP++ (Section 3). Recall from Section 5.2 that language syntax is sunk into K computations by associating a K label to each language construct (including constants) and that, to ease writing, we continue to use the mixfix notation instead of the AST notation in computations, that is, we write $e_1 + e_2$ instead of $\_+\_(e_1, e_2)$, 3 instead of 3(), etc. In general, to reduce clutter in computations we write *label* instead of *label*() for any $label \in KLabel$. We start by defining basic values as result computations (or more precisely as result labels; see Section 5.2):

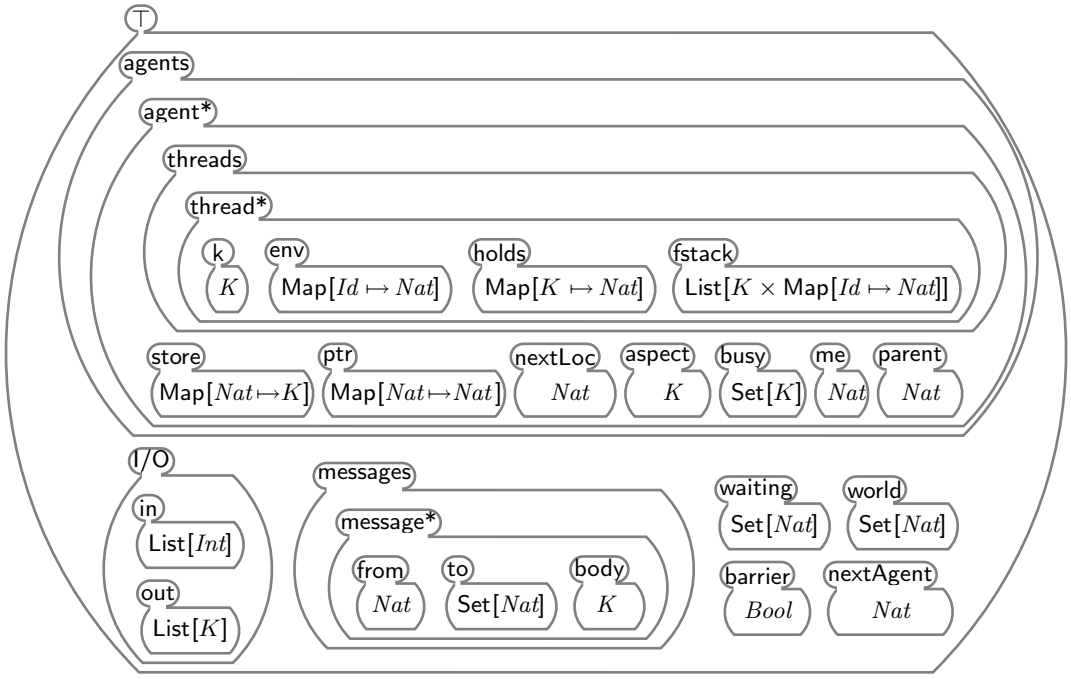
$$KResultLabel \quad ::= \quad Bool \quad | \quad Int \quad | \quad Real$$

Other result labels are added by need as we define the semantics below. The variables used in the subsequent K rules range over syntactic/semantic categories as follows: $r_1, r_2 \in Real$; $i, i_1, i_2 \in Int$; $m, n, n', l, l_1, l_2, ... \in Nat$; $v \in KResult$; $vl \in \mathsf{List}[KResult]$; $x, y \in Id$; $xl \in \mathsf{List}[Id]$; $\rho, \varrho \in \mathsf{Map}[Id \mapsto Nat]$; $\sigma \in \mathsf{Map}[Nat \mapsto K]$; $fs \in \mathsf{List}[K \times \mathsf{Map}[Id \mapsto Nat]]$; $h \in \mathsf{Map}[KResult \mapsto Nat]$; $busy \in \mathsf{Set}[KResult]$; $e, s_1, s_2, k, k_1, k_2 \in K$; $label \in KLabel$; $kl, kl_1, kl_2 \in \mathsf{List}[K]$.

Note that we allow our stores to hold arbitrary computations, not only results as one would expect. We allow this generality because it simplifies the semantics of recursion.

Arithmetic operations All arithmetic operations have the same semantics as in IMP++ on the types of values that IMP++ had. We additionally extend their semantics to *Real* values:

$$
\begin{aligned}
\mathit{Configuration}_{\textsc{Challenge}} &\equiv \langle \mathit{Agents}_{\textsc{Challenge}}\ \mathit{I/O}_{\textsc{Challenge}}\ \mathit{Messages}_{\textsc{Challenge}}\ \mathit{Rest}_{\textsc{Challenge}} \rangle_{\top} \\
\mathit{Agents}_{\textsc{Challenge}} &\equiv \left\langle \left\langle \begin{array}{l} \mathit{Threads}_{\textsc{Challenge}} \langle \mathsf{Map}[\mathit{Nat} \mapsto K] \rangle_{\mathsf{store}} \langle \mathit{Nat} \rangle_{\mathsf{nextLoc}} \langle K \rangle_{\mathsf{aspect}} \\ \langle \mathsf{Set}[K] \rangle_{\mathsf{busy}}\ \langle \mathit{Nat} \rangle_{\mathsf{me}}\ \langle \mathit{Nat} \rangle_{\mathsf{parent}}\ \langle \mathsf{Map}[\mathit{Nat} \mapsto \mathit{Nat}] \rangle_{\mathsf{ptr}} \end{array} \right\rangle_{\mathsf{agents}*} \right\rangle_{\mathsf{agents}} \\
\mathit{Threads}_{\textsc{Challenge}} &\equiv \left\langle \left\langle \begin{array}{l} \langle K \rangle_{\mathsf{k}}\ \langle \mathsf{Map}[\mathit{Id} \mapsto \mathit{Nat}] \rangle_{\mathsf{env}}\ \langle \mathsf{Map}[K \mapsto \mathit{Nat}] \rangle_{\mathsf{holds}} \\ \langle \mathsf{List}[K \times \mathsf{Map}[\mathit{Id} \mapsto \mathit{Nat}]] \rangle_{\mathsf{fstack}} \end{array} \right\rangle_{\mathsf{thread}*} \right\rangle_{\mathsf{threads}} \\
\mathit{I/O}_{\textsc{Challenge}} &\equiv \langle \langle \mathsf{List}[\mathit{Int}] \rangle_{\mathsf{in}}\ \langle \mathsf{List}[K] \rangle_{\mathsf{out}} \rangle_{\mathsf{I/O}} \\
\mathit{Messages}_{\textsc{Challenge}} &\equiv \langle \langle \langle \mathit{Nat} \rangle_{\mathsf{from}}\ \langle \mathit{Nat} \rangle_{\mathsf{to}}\ \langle K \rangle_{\mathsf{body}} \rangle_{\mathsf{message}*} \rangle_{\mathsf{messages}} \\
\mathit{Rest}_{\textsc{Challenge}} &\equiv \langle \mathsf{Set}[\mathit{Nat}] \rangle_{\mathsf{waiting}} \langle \mathsf{Set}[\mathit{Nat}] \rangle_{\mathsf{world}} \langle \mathit{Bool} \rangle_{\mathsf{barrier}} \langle \mathit{Nat} \rangle_{\mathsf{nextAgent}}
\end{aligned}
$$



$\langle\rangle_{\top}$: top cell, holding everything	$\langle\rangle_{\mathsf{me}}$: agent's id
$\langle\rangle_{\mathsf{agents}}$: all the agents	$\langle\rangle_{\mathsf{parent}}$: agent's parent id
$\langle\rangle_{\mathsf{agent}*}$: one agent, can multiply	$\langle\rangle_{\mathsf{I/O}}$: the system's input/output
$\langle\rangle_{\mathsf{threads}}$: all agent's threads	$\langle\rangle_{\mathsf{in}}$: the system input
$\langle\rangle_{\mathsf{thread}*}$: one thread, can multiply	$\langle\rangle_{\mathsf{out}}$: the system output
$\langle\rangle_{\mathsf{k}}$: thread's computation	$\langle\rangle_{\mathsf{messages}}$: all the pending messages
$\langle\rangle_{\mathsf{env}}$: thread's environment	$\langle\rangle_{\mathsf{message}*}$: one message, can multiply
$\langle\rangle_{\mathsf{holds}}$: thread's locks	$\langle\rangle_{\mathsf{from}}$: the message sender's id
$\langle\rangle_{\mathsf{fstack}}$: thread's function stack	$\langle\rangle_{\mathsf{to}}$: the message receivers' ids
$\langle\rangle_{\mathsf{store}}$: agent's store	$\langle\rangle_{\mathsf{body}}$: the message's value/body
$\langle\rangle_{\mathsf{ptr}}$: agent's store allocation map	$\langle\rangle_{\mathsf{waiting}}$: the ids of agents waiting at barrier
$\langle\rangle_{\mathsf{nextLoc}}$: the next available location	$\langle\rangle_{\mathsf{world}}$: the ids of all the agents
$\langle\rangle_{\mathsf{aspect}}$: agent's aspect	$\langle\rangle_{\mathsf{barrier}}$: a flag saying if barrier is active
$\langle\rangle_{\mathsf{busy}}$: agent's busy locks	$\langle\rangle_{\mathsf{nextAgent}}$: the next available agent id

Figure 10: The configuration of the CHALLENGE language in both term (top) and graphical (middle) representation, with short explanation of cell contents (bottom)

$$r_1 + r_2 \to r_1 +_{Real} r_2 \qquad r_1 + i_2 \to r_1 +_{Real} i2r(i_2) \qquad i_1 + r_2 \to i2r(i_1) +_{Real} r_2$$

We chose to convert integers to reals (using an assumed *i2r* function) in order to sum them with reals. Our choice here is, of course, adhoc; another possibility is, e.g., to disallow such situations (by dropping the last two rules), etc. The other arithmetic operations are defined similarly.

Variable declaration Newly declared variables are assigned new locations in the environment and the store is initialized with 0 at those locations. The $\langle\rangle_{\mathsf{nextLoc}}$ cell holds the next available location. $\rho[l_1 \,..\, l_n/xl]$ (re)assigns in ρ the identifier list xl to the locations $l_1 \,..\, l_n$, and $l_1 \,..\, l_n \mapsto v$ creates mappings from all locations $l_1 \,..\, l_n$ to v (i.e., $l_1 \mapsto v$, $l_2 \mapsto v$, etc.):

$$\langle \underline{\texttt{var}\ xl\,\texttt{;}\quad \_}_{\cdot} \rangle_{\mathsf{k}}\ \langle \underline{\qquad \rho \qquad \_}_{\rho[n \,..\, n'/xl]} \rangle_{\mathsf{env}}\ \langle \_\ \underline{\qquad \cdot \qquad \_}_{n \,..\, n' \mapsto 0} \rangle_{\mathsf{store}}\ \langle \underline{\qquad n \qquad \_}_{n' +_{Int} 1} \rangle_{\mathsf{nextLoc}}$$

where n' is $n +_{Int} |xl| -_{Int} 1$

Variable lookup When a program variable reaches the top of the computation, it is replaced by whatever is in the store at the location corresponding to that variable in the environment:

$$\langle \underline{x \quad \_}_{k} \rangle_{\mathsf{k}}\ \langle \_\ \ x \mapsto n\ \ \_ \rangle_{\mathsf{env}}\ \langle \_\ \ n \mapsto k\ \ \_ \rangle_{\mathsf{store}}$$

Variable assignment The "strict(2)" attribute of the assignment construct says that the right-hand-side of an assignment will be eventually evaluated. The resulting value is then written into the store at the location the variable points to in the environment, and the assignment is dissolved:

$$\langle \underline{x = v\,\texttt{;}\quad \_}_{\cdot} \rangle_{\mathsf{k}}\ \langle \_\ \ x \mapsto n\ \ \_ \rangle_{\mathsf{env}}\ \langle \_\ \ n \mapsto \underline{\quad \_ \quad}_{v} \ \_ \rangle_{\mathsf{store}}$$

Variable increment The semantics of variable increment is a mixture of the two rules above:

$$\langle \underline{\texttt{++}\,x \quad \_}_{i +_{Int} 1} \rangle_{\mathsf{k}}\ \langle \_\ \ x \mapsto n\ \ \_ \rangle_{\mathsf{env}}\ \langle \_\ \ n \mapsto \underline{\quad i \quad \_}_{i +_{Int} 1} \rangle_{\mathsf{store}}$$

Basic statement constructs The following three rules give semantics to the basic statement constructs that were not already discussed as part of IMP++ (`{}` is the same as IMP's `skip`). The first gives semantics to blocks and says that once the statement in the block is executed, the old environment must be resumed. The old environment is saved as item $\mathsf{env}(\rho)$ in the computation right after the statement, where env wraps an environment in a K label, i.e., $KLabel ::= \mathsf{env}(\mathsf{Map}[Id \mapsto Nat])$; the second rule below, which is structural, restores the environment once the statement is processed. The third rule below discards the residual value obtained after evaluating expression e (e.g., $\texttt{++}x$) in statement "$e\,\texttt{;}$" (recall from Figure 9 that "$Exp\,\texttt{;}$" is strict):

$$\langle \underline{\texttt{\{}\,s\,\texttt{\}} \quad \_}_{s \curvearrowright \mathsf{env}(\rho)} \rangle_{\mathsf{k}}\ \langle \rho \rangle_{\mathsf{env}} \qquad \Big| \qquad \langle \underline{\mathsf{env}(\rho) \quad \_}_{\cdot} \rangle_{\mathsf{k}}\ \langle \underline{\quad \_ \quad}_{\rho} \rangle_{\mathsf{env}} \qquad \Big| \qquad v\ \texttt{;} \to \cdot$$

The restoration of the environment after a block is crucial, because we allow variable declarations anywhere and their scope is until the end of the current block. Thus, a block-local variable may shadow an outside one which happens to have the same name. The main reason for which we chose an environment-based instead of a substitution-based definitional style for CHALLENGE is that the later is known to be inconvenient when defining languages with such unrestricted variable declarations in combination with referencing, dereferencing, and assignment constructs.

Referencing and dereferencing The next rules give the semantics of `&`x (the reference location of program variable x) and `*`n (the value at location n –note that `*`e was declared strict in Figure 9):

$$\langle \underline{\texttt{\&}\, x}_{n} \;\_\rangle_{\mathsf{k}} \; \langle \_ \;\; x \mapsto n \;\; \_\rangle_{\mathsf{env}} \qquad\qquad \langle \underline{\texttt{*}\, n}_{v} \;\_\rangle_{\mathsf{k}} \; \langle \_ \;\; n \mapsto v \;\; \_\rangle_{\mathsf{store}}$$

Memory allocation and deallocation The following two rules are almost dual to each other:

$$\langle \frac{\texttt{malloc}\, n}{l} \;\_\rangle_{\mathsf{k}} \; \langle \_ \; \frac{\cdot}{l \mapsto n} \; \_\rangle_{\mathsf{ptr}} \; \langle \_ \; \frac{\cdot}{l \,..\, (l +_{Int} n -_{Int} 1) \mapsto 0} \; \_\rangle_{\mathsf{store}} \; \langle \frac{l}{l +_{Int} n} \rangle_{\mathsf{nextLoc}}$$

$$\langle \frac{\texttt{free}\, l\texttt{;}}{\cdot} \;\_\rangle_{\mathsf{k}} \; \langle \_ \; \frac{l \mapsto n}{\cdot} \; \_\rangle_{\mathsf{ptr}} \; \langle \frac{\sigma}{\sigma[\bot / l \,..\, l +_{Int} n -_{Int} 1]} \rangle_{\mathsf{store}}$$

Above, $\sigma[\bot/l \,..\, l +_{Int} n -_{Int} 1]$ "undefines" the map σ in each of the locations in the list $l +_{Int} n -_{Int} 1$.

Assigning to a dereferenced location Since the assignment operation is strict in its second argument (see Figure 9), the expression e_2 in a reference assignment statement of the form "`*`e_1`=`e_2`;`" is eventually evaluated. To give this statement a semantics, we also need to evaluate e_1 (interleaved with the evaluation of e_2). The simplest way to do this is to declare a strict context like below. Once e_1 is also evaluated, the semantics of reference assignment is straightforward:

context: `*` $\square$ `=` `_` `;`

$$\langle \frac{\texttt{*}\, n \,\texttt{=}\, v\texttt{;}}{\cdot} \;\_\rangle_{\mathsf{k}} \; \langle \_ \;\; n \mapsto \frac{\_}{v} \;\; \_\rangle_{\mathsf{store}}$$

Note that even though the CHALLENGE syntax in Figure 9 allows assignment statements of the form "e_1`=`e_2`;`" for arbitrary e_1, we only gave it a semantics for the cases when e_1 is either a variable or a dereferencing expression of the form `*`e'_1; any other e_1 would stuck the computation/evaluation.

Lists CHALLENGE lists are comma-separated sequences of expressions surrounded by square brackets. Lists of expressions evaluate to corresponding lists of results, obtained by evaluating each of the expressions in the list, and the obtained lists of results are themselves results. This is achieved by the *strict* and *hybrid* attributes of the bracket list construct in Figure 9, with one important addition: the label "`_,_`" that corresponds to List[*Exp*] needs to also be declared *strict* and *hybrid*, so that the expression-list argument of the bracket construct is evaluated to a result-list, as discussed in Section 5.2. Below is the complete semantics of lists:

`_,_` [*strict hybrid*]

v `:` `[_,_`(vl)`]` $\rightarrow$ `[_,_`(v, vl)`]`

`head [_,_`(v, vl)`]` $\rightarrow v$

`tail [_,_`(v, vl)`]` $\rightarrow$ `[_,_`(vl)`]`

We used the K labeled representation of lists above to make it clear that there is a `_,_` list construct inside each square-bracket CHALLENGE list. In our K-Maude tool [42], however, one does not need to use the labeled representation; for example, the last rule above is "`rule tail [V,Vl] => [Vl]`".

Aspects CHALLENGE allows very simple aspects, whose semantics is as follows. Using a statement "`aspect` s", one can dynamically set the current aspect (held in a specific cell) to statement s:

$$\langle \frac{\texttt{aspect}\, s}{\cdot} \;\_\rangle_{\mathsf{k}} \; \langle \frac{\_}{s} \rangle_{\mathsf{aspect}}$$

The rest of the aspect semantics is formally defined in the next rule for function evaluation. Informally, whenever a λ-abstraction evaluates to a closure, the current aspect is weaved at the beginning of its body. This way, the aspect is executed whenever the function is invoked.

Functions as closures It is convenient to evaluate CHALLENGE λ-abstractions to closures. A *closure* is a special value wrapping a function together with its defining environment. Closures ensure that the identifiers which are free in the wrapped λ-abstraction are not escaped and always refer to the same store locations each time the λ-abstraction is applied. Since closures evaluate no further, we can define them as K result labels, $\mathit{KResultLabel} ::= \mathit{closure}_\lambda(\mathsf{List}[\mathit{Id}], K, \mathsf{Map}[\mathit{Id} \mapsto \mathit{Nat}])$; an alternative, unexplored here, is to only define $\mathit{closure}_\lambda$ as a K result label. The semantics of λ-abstractions is given by the following rule, which also weaves the current aspect:

$$\langle \frac{\lambda xl \,.\, s_1}{\mathit{closure}_\lambda(xl, s_2\ s_1, \rho)} \ \_\rangle_{\mathsf{k}}\ \langle \rho \rangle_{\mathsf{env}}\ \langle s_2 \rangle_{\mathsf{aspect}}$$

Function application Upon function call, the evaluated arguments (which, since $\_\,,\,\_$ is strict and hybrid, are grouped as a list of result computations under a $\_\,,\,\_$ label) 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 and remainder of computation are saved as a pair in the function stack $\langle\rangle_{\mathsf{fstack}}$ to allow restoration upon return (a return must always be used to exit a function):

$$\left(\begin{array}{c} \langle \frac{\mathit{closure}_\lambda(xl, s, \rho)}{s} \ \_\,,\,\_(vl) \curvearrowright k \rangle_{\mathsf{k}}\ \langle \frac{\varrho}{\rho[n \,..\, n +_{\mathit{Int}} |xl| -_{\mathit{Int}} 1 \,/\, xl]} \rangle_{\mathsf{env}}\ \langle \frac{\cdot}{(k, \varrho)} \ \_\rangle_{\mathsf{fstack}} \\ \\ \langle \_ \ \frac{\cdot}{n \,..\, n +_{\mathit{Int}} |xl| -_{\mathit{Int}} 1 \mapsto vl} \ \_\rangle_{\mathsf{store}}\ \langle \frac{n}{n +_{\mathit{Int}} |xl|} \rangle_{\mathsf{nextLoc}} \end{array} \right)$$

When a return statement is reached, the original environment and computation are restored:

$$\langle \frac{\mathtt{return}\ v; \curvearrowright \_}{v \curvearrowright k} \rangle_{\mathsf{k}}\ \langle \frac{\_}{\rho} \rangle_{\mathsf{env}}\ \langle \frac{(k, \rho)}{\cdot} \ \_\rangle_{\mathsf{fstack}}$$

Recursion The standard fix-point semantics of $\mu x \,.\, e$ is given by unrolling it to $e[\mu x \,.\, e/x]$. To achieve this in an environment based definition one needs to evaluate e in an environment where x is bound to e itself. However, similarly to the semantics of λ, this enriched environment must be enclosed together with e to account for the other free variables of e. We can achieve this using a similar closure construct as for λ, namely $\mathit{KLabel} ::= \mathit{closure}_\mu(\mathit{Exp}, \mathsf{Map}[\mathit{Id} \mapsto \mathit{Nat}])$. Note, however, that $\mathit{closure}_\mu(e, \rho)$ is not a result label anymore, because it must schedule e for evaluation, as shown below. Once e is evaluated, the original environment must be restored:

$$\langle \frac{\mu x \,.\, e}{e \curvearrowright \mathsf{env}(\rho)} \ \_\rangle_{\mathsf{k}}\ \langle \frac{\rho}{\rho[n/x]} \rangle_{\mathsf{env}}\ \langle \_ \ \frac{\cdot}{n \mapsto \mathit{closure}_\mu(e, \rho[n/x])} \ \_\rangle_{\mathsf{store}}\ \langle \frac{n}{n +_{\mathit{Int}} 1} \rangle_{\mathsf{nexLoc}}$$

During the evaluation of e, whenever x is encountered the μ closure needs to be unrolled:

$$\langle \frac{\mathit{closure}_\mu(e, \varrho)}{e \curvearrowright \mathsf{env}(\rho)} \ \_\rangle_{\mathsf{k}}\ \langle \frac{\varrho}{\varrho} \rangle_{\mathsf{env}} \qquad \Big| \qquad v \curvearrowright \mathsf{env}(\rho) \rightharpoonup \mathsf{env}(\rho) \curvearrowright v$$

The other rule above (on the right) passes an already evaluated expression after the environment recovery item in the computation, to allow our original environment recovery rule (see the rules for the statement block above) to apply. The environment-based semantics of μ is more involved than the substitution-based one, but environments make the semantics of other constructs (e.g., declarations, referencing, dereferencing, assignment) easier. There is no absolute better style.

Note that e can be arbitrarily complex in its full generality, in particular it can generate several side effects before reaching x and x need not necessarily appear inside a λ-abstraction. A common type of recursion, however, is to restrict e to only λ-abstractions, in which case $\mu x \,.\, e$ defines a recursive function (indeed, $\mu x \,.\, \lambda y \,.\, e'$ then becomes $\mathtt{let\ rec}\ x\ y = e'\ \mathtt{in}\ x$, the usual mechanism to define recursive functions in functional languages). If such a restriction is acceptable, then we can have much more elegant and compact semantics for μ, such as, for example

$$\langle \underset{\texttt{var}\, x\,;\, \curvearrowright x = e\,;\, \curvearrowright x \curvearrowright \mathsf{env}(\rho)}{\overset{\mu x\,.\,e}{\cdots\cdots\cdots\cdots\cdots\cdots}} \_ \rangle_{\mathsf{k}}\ \langle \rho \rangle_{\mathsf{env}}$$

Alternatively, one could completely desugar $\mu x\,.\,e$ into $(\lambda()\,.\ \{\,\texttt{var}\, x\,;\ x = e\,;\ \texttt{return}\, x\,;\,\})()$. These semantics mimic the way `let rec` can be expressed using assignment in languages like Ocaml and SML. For example, Figure 11 shows one possible way to implement factorial in Ocaml. All the steps in our previous semantic rules can be observed in this definition: declaring x, assigning it to an expression referencing x, and then escaping its value outside the defining environment. One could also give μ a more direct semantics, under the same restriction, by declaring $\mu x\,.\,e$ to be strict in e, and, once e evaluates to a closure, to assign that closure to a fresh location n in the store, while modifying its environment to contain a mapping of x to n:

```
let fact =
 let x = ref (fun y -> y)
 in x := (fun y ->
      if y > 0
      then y*(!x (y-1))
      else 1) ;
    !x;;
```

Figure 11: Recursion via assignment

$$\langle \underset{closure_\lambda(y,s,\rho[n/x])}{\mu x\,.\,closure_\lambda(y,s,\rho)} \_ \rangle_{\mathsf{k}}\ \langle \_\ \underset{n \mapsto closure_\lambda(y,s,\rho[n/x])}{\cdot} \_ \rangle_{\mathsf{store}}\ \langle \underset{n +_{Int} 1}{n} \rangle_{\mathsf{nextLoc}}$$

Call with current continuation Callcc packages the reminder of the computation as a value and passes it to its argument (expected to evaluate to a function). When the packaged computation is applied on a value, the current computation is discarded and replaced by the packaged one which is also passed the value. Note that the environment and function stack also need to be packed:

$$\langle \underset{v\ cc(k,\rho,fs)}{\texttt{callcc}\ v} \curvearrowright k \rangle_{\mathsf{k}}\ \langle \rho \rangle_{\mathsf{env}}\ \langle fs \rangle_{\mathsf{fstack}} \quad\Big|\quad \langle \underset{v \curvearrowright k}{cc(k,\rho,fs)\ v\ \curvearrowright \_} \rangle_{\mathsf{k}}\ \langle \underset{\rho}{\_} \rangle_{\mathsf{env}}\ \langle \underset{fs}{\_} \rangle_{\mathsf{fstack}}$$

In the above, we used cc to pack the various items discussed above into a K result label. Formally, $KResultLabel ::= cc(K, \mathsf{Map}[Id \mapsto Nat], \mathsf{List}[K \times \mathsf{Map}[Id \mapsto Nat]])$.

Sequential non-determinism `randomBool` can arbitrarily evaluate to either *true* or *false*.

$$\langle \underset{\texttt{true}}{\texttt{randomBool}} \_ \rangle_{\mathsf{k}} \quad\Big|\quad \langle \underset{\texttt{false}}{\texttt{randomBool}} \_ \rangle_{\mathsf{k}}$$

Thread creation and termination Threads execute in parallel within an agent, sharing its store. A $\langle\rangle_{\mathsf{thread}}$ cell groups together several cells related to a thread, namely its computation, its environment, its function stack, and its acquired resources. A "`spawn` s" command creates a new thread, whose computation is s and whose environment is the same as that of the creating thread (this is how threads share memory). The initialization of the unspecified cells ($\langle\rangle_{\mathsf{holds}}$ and $\langle\rangle_{\mathsf{fstack}}$) with their default values is taken care of by the context transformation step—see Section 5.5.3:

$$\langle \_\ \underset{\cdot}{\langle \texttt{spawn}\ s \rangle_{\mathsf{k}}} \_ \rangle_{\mathsf{k}}\ \langle \rho \rangle_{\mathsf{env}}\ \_ \rangle_{\mathsf{thread}}\ \underset{\langle \_\ \langle s \rangle_{\mathsf{k}}\ \langle \rho \rangle_{\mathsf{env}}\ \_ \rangle_{\mathsf{thread}}}{\cdot}$$

When the computation of a thread is empty, the thread is dissolved and its resources released:

$$\langle \underset{\cdot}{\_\ \langle \cdot \rangle_{\mathsf{k}}\ \langle h \rangle_{\mathsf{holds}}\ \_ \rangle_{\mathsf{thread}}}\ \langle \underset{busy\ -_{set}\ dom(h)}{busy} \rangle_{\mathsf{busy}},$$

Above, $\langle\rangle_{\mathsf{busy}}$ holds the names of all the resources held by any of the threads; dom and $\_ -_{set} \_$ define the map domain and the set difference, respectively, both easy to define and assumed given.

Thread synchronization To attain mutual exclusion, threads can be synchronized by means of locks. In CHALLENGE, any K result can be used as a lock. A lock can only be acquired if it is not busy, indicated by its absence from the set cell $\langle\rangle_{\mathsf{busy}}$ which can be tested using the built-in membership predicate for sets $\in_{set}$. The same thread can acquire same lock multiple times (due, e.g., to recursion or synchronized functions), so $\langle\rangle_{\mathsf{holds}}$ must maintain a counter for each lock:

$$\langle \underline{\texttt{acquire}\ v\texttt{;}}_{\cdot}\ \_\rangle_{\mathsf{k}}\ \langle \_\ \underline{\cdot}_{v \mapsto 0}\ \_\rangle_{\mathsf{holds}}\ \langle busy\ \underline{\cdot}_{v}\rangle_{\mathsf{busy}} \qquad \text{where } \neg_{Bool}(v \in_{set} busy)$$

$$\langle \underline{\texttt{acquire}\ v\texttt{;}}_{\cdot}\ \_\rangle_{\mathsf{k}}\ \langle \_\ v \mapsto \underline{n}_{s_{Nat}(n)}\ \_\rangle_{\mathsf{holds}}$$

$$\langle \underline{\texttt{release}\ v\texttt{;}}_{\cdot}\ \_\rangle_{\mathsf{k}}\ \langle \_\ v \mapsto \underline{s_{Nat}(n)}_{n}\ \_\rangle_{\mathsf{holds}}$$

$$\langle \underline{\texttt{release}\ v\texttt{;}}_{\cdot}\ \_\rangle_{\mathsf{k}}\ \langle \_\ \underline{v \mapsto 0}_{\cdot}\ \_\rangle_{\mathsf{holds}}\ \langle \_\ \underline{v}_{\cdot}\ \_\rangle_{\mathsf{busy}}$$

Rendezvous synchronization Two threads can only pass together a barrier specified by a lock v:

$$\langle \underline{\texttt{rendezvous}\ v\texttt{;}}_{\cdot}\ \_\rangle_{\mathsf{k}}\ \langle \underline{\texttt{rendezvous}\ v\texttt{;}}_{\cdot}\ \_\rangle_{\mathsf{k}}$$

The semantics of `rendezvous` is more difficult to give in other definitional frameworks, because it requires the ability to access and reduce redexes of two computational units simultaneously.

Agents A CHALLENGE agent is a collection of threads grouped in an $\langle\rangle_{\mathsf{agent}}$ cell. Besides its threads, an $\langle\rangle_{\mathsf{agent}}$ cell also contains a $\langle\rangle_{\mathsf{me}}$ subcell holding agent's unique id and a $\langle\rangle_{\mathsf{parent}}$ subcell holding the id of its creating agent. The cell $\langle\rangle_{\mathsf{nextAgent}}$ provides fresh agent ids. The cell $\langle\rangle_{\mathsf{world}}$ holds the set of all active agents. Due to context transformation (Section 5.5), the rule for new agent creation needs specify only those cells which contain non-default values: $\langle\rangle_{\mathsf{k}}$—initialized with the given statement, $\langle\rangle_{\mathsf{parent}}$—initialized with the contents of the $\langle\rangle_{\mathsf{me}}$ cell of the creating agent, and $\langle\rangle_{\mathsf{me}}$—initialized with a new agent id, which is also the return value of the `newAgent` construct:

$$\langle \underline{\texttt{newAgent}\ s}_{n}\ \_\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}}\ \langle \underline{n}_{n +_{Int} 1}\rangle_{\mathsf{nextAgent}}\ \langle \_\ \underline{\cdot}_{n}\ \_\rangle_{\mathsf{world}}\ \underline{\cdot}_{\langle \_\ \langle s\rangle_{\mathsf{k}}\ \langle n\rangle_{\mathsf{me}}\ \langle m\rangle_{\mathsf{parent}}\ \_\rangle_{\mathsf{agent}}}$$

When all threads of an agent terminate, the agent can be dissolved and unregistered from $\langle\rangle_{\mathsf{world}}$:

$$\underline{\langle \_\ \langle m\rangle_{\mathsf{me}}\ \langle\cdot\rangle_{\mathsf{threads}}\ \_\rangle_{\mathsf{agent}}}_{\cdot}\ \langle \_\ \underline{m}_{\cdot}\ \_\rangle_{\mathsf{world}}$$

The semantics of the `me` and `parent` expression constructs is straightforward:

$$\langle \underline{\texttt{me}}_{m}\ \_\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}} \qquad\qquad \langle \underline{\texttt{parent}}_{n}\ \_\rangle_{\mathsf{k}}\ \langle n\rangle_{\mathsf{parent}}$$

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

$$\langle \underline{\texttt{send}\ v\ \texttt{to}\ n\texttt{;}}_{\cdot}\ \_\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}}\ \underline{\cdot}_{\langle\langle m\rangle_{\mathsf{from}}\ \langle n\rangle_{\mathsf{to}}\ \langle v\rangle_{\mathsf{body}}\rangle_{\mathsf{message}}}$$

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

$$\langle m\rangle_{\mathsf{me}}\ \langle \underset{\cdot}{\texttt{broadcast}\ v\texttt{;}}\ \ \_\rangle_{\mathsf{k}}\ \langle w\rangle_{\mathsf{world}}\ \frac{\cdot}{\langle\langle m\rangle_{\mathsf{from}}\ \langle w\rangle_{\mathsf{to}}\ \langle v\rangle_{\mathsf{body}}\rangle_{\mathsf{message}}}$$

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

$$\langle \underset{v}{\texttt{receiveFrom}\ n}\ \ \_\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}}\ \langle\langle n\rangle_{\mathsf{from}}\ \langle \_\ \ \underset{\cdot}{m}\ \ \_\rangle_{\mathsf{to}}\ \langle v\rangle_{\mathsf{body}}\rangle_{\mathsf{message}}$$

$$\langle \underset{v}{\texttt{receive}}\ \ \_\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}}\ \langle \_\ \ \langle \_\ \ \underset{\cdot}{m}\ \ \_\rangle_{\mathsf{to}}\ \langle v\rangle_{\mathsf{body}}\ \ \_\rangle_{\mathsf{message}}$$

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

$$\langle \_\ \ \langle\cdot\rangle_{\mathsf{to}}\ \ \_\rangle_{\mathsf{message}} \rightharpoonup \cdot$$

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

$$\langle \_\ \ \langle \underset{\cdot}{\texttt{sendSynch}\ v\ \texttt{to}\ n\texttt{;}}\ \ \_\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}}\ \ \_\rangle_{\mathsf{agent}}\ \langle \_\ \ \langle \underset{v}{\texttt{receiveFrom}\ m}\ \ \_\rangle_{\mathsf{k}}\ \langle n\rangle_{\mathsf{me}}\ \ \_\rangle_{\mathsf{agent}}$$

$$\langle \underset{\cdot}{\texttt{sendSynch}\ v\ \texttt{to}\ n\texttt{;}}\ \ \_\rangle_{\mathsf{k}}\ \langle \_\ \ \langle \underset{v}{\texttt{receive}}\ \ \_\rangle_{\mathsf{k}}\ \langle n\rangle_{\mathsf{me}}\ \ \_\rangle_{\mathsf{agent}}$$

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

$$\langle n\rangle_{\mathsf{me}}\ \langle\langle\langle \texttt{barrier;}\ \ \_\rangle_{\mathsf{k}}\ \ \_\rangle_{\mathsf{thread}}\rangle_{\mathsf{threads}}\ \langle \texttt{true}\rangle_{\mathsf{barrier}}\ \langle w\ \underset{n}{\cdot}\rangle_{\mathsf{waiting}}\quad \text{when } \neg_{Bool}(n \in_{Set} w)$$

It is important to note that the above rule requires the agent to have only one running thread when attempting to globally synchronize. Otherwise, it may happen that while one thread is waiting at the barrier, other threads send messages to other agents. We do not want that in CHALLENGE.

When all agents in the $\langle\rangle_{\mathsf{world}}$ wait at the barrier, the barrier is lifted (so they can all proceed):

$$\langle \underset{\texttt{false}}{\texttt{true}}\ \rangle_{\mathsf{barrier}}\ \langle w\rangle_{\mathsf{waiting}}\ \langle w\rangle_{\mathsf{world}}\qquad \text{when } \neg_{Bool}(w =_{Bool} \cdot)$$

Once the barrier is off, waiting agents unregister from the $\langle\rangle_{\mathsf{waiting}}$ cell and proceed:

$$\langle n\rangle_{\mathsf{me}}\ \langle \underset{\cdot}{\texttt{barrier;}}\ \ \_\rangle_{\mathsf{k}}\ \langle \texttt{false}\rangle_{\mathsf{barrier}}\ \langle \_\ \ \underset{\cdot}{n}\ \ \_\rangle_{\mathsf{waiting}}$$

Once all the agents proceeded, the barrier is lowered so that it can be used again:

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

Abrupt termination We have three abrupt termination statements. `haltThread` empties the current thread's computation, inducing thread dissolution and resource freeing (see the thread termination rule above). `haltAgent` empties the current agent's $\langle\rangle_{\mathsf{threads}}$ cell, inducing its dissolution (see the agent termination rule above). finally, `haltSystem` empties the $\langle\rangle_{\mathsf{agents}}$ cell:

$$\langle \texttt{haltThread;}\ \ \_\rangle_{\mathsf{k}} \rightarrow \langle\cdot\rangle_{\mathsf{k}}$$

$$\langle \_\ \ \langle \texttt{haltAgent;}\ \ \_\rangle_{\mathsf{k}}\ \ \_\rangle_{\mathsf{threads}} \rightarrow \langle\cdot\rangle_{\mathsf{threads}}$$

$$\langle \_\ \ \langle \texttt{haltSystem;}\ \ \_\rangle_{\mathsf{k}}\ \ \_\rangle_{\mathsf{agents}} \rightarrow \langle\cdot\rangle_{\mathsf{agents}}$$

Code generation We next define runtime code generation language-independently, in the sense that the particular concrete syntax or meaning of the existing language constructs plays no role. To obtain this degree of language-independence, the reflective capabilities of K w.r.t. syntax are crucial. Indeed, since any fragment of program can be uniformly regarded as a K computation, we only need to consider the general-purpose K syntactic constructs, namely "$K ::= KLabel(\mathsf{List}\text{-}[K]) \mid \mathsf{List}_{\curvearrowright}[K]$". To distinguish between existing code and generated code, we start by reflecting K's syntax within itself, that is, by creating labels corresponding to the K constructs the same way we created labels to sink concrete language syntax into K:

$$KLabel \quad ::= \quad \boxed{KLabel}\ [strict] \quad \mid \quad \_\ \boxed{\curvearrowright}\ \_\ [strict] \quad \mid \quad \_\ \boxed{\curvearrowright\!\!\!\!\!\!\sim}\ \_\ [strict]$$

We boxed these new labels to easily distinguish them. For now, we can think of computations using boxed labels as computations being generated using the `quote`/`unquote` mechanism. They were declared *strict* because they will be used in a way that their arguments will still need to be processed and hereby code generated for them. We did not need reflective versions of the units of the two K builtin lists, namely of "$\lrcorner$" and "$\cdot$", because we can reuse the core ones instead.

There are two important aspects of code generation. One important aspect is that code quoting and unquoting can be nested, and an unquoted code can be evaluated only if it is guarded by as many `unquote` constructs as `quote` ones; to achieve this we need to maintain a counter for the depth of quotation –we prefer to do it as part of a special K label. The other important aspect is that the generated code may need to be eventually evaluated; to avoid "unboxing" the labels at evaluation time, we prefer instead to introduce a special *code* K result label that wraps the generated code as code over the core labels instead of over boxed labels:

$$\begin{array}{rcl} KLabel & ::= & quote[Nat] \\ KResultLabel & ::= & code \end{array}$$

Initially, a quoted expression starts at the quoting level 0, and acts as a morphism on all K constructors, transforming them into their boxed variants, except for the `quote`/`unquote` labels, which need to increment/decrement the counter. If the quoting level is 0 and an $\texttt{unquote}(k)$ expression is encountered, then k is released for evaluation; its evaluation will indeed take place, because the boxed labels were all defined strict. Each computation rooted in a boxed label eventually evaluates to a code fragment; as this happens, the generated fragments of code are glued together into bigger code fragments, this way eliminating all the boxed labels:

$$\begin{array}{l} \langle\ \underset{\cdots}{\texttt{quote}}\ (k)\ \ \_ \rangle_{\mathtt{k}} \\ \ quote[0] \end{array}$$

$$\begin{array}{l} quote[n](\texttt{quote}(k)) \rightharpoonup \boxed{\texttt{quote}}(quote[s_{Nat}(n)](k)) \\ quote[0](\texttt{unquote}(k)) \rightharpoonup k \\ quote[s_{Nat}(n)](\texttt{unquote}(k)) \rightharpoonup \boxed{\texttt{unquote}}(quote[n](k)) \end{array}$$

$$\begin{array}{l} quote[n](label(kl)) \rightharpoonup \boxed{label}(quote[n](kl)) \quad \text{when } label \neq \texttt{quote}\ ,\ label \neq \texttt{unquote} \\ \boxed{label}(code(kl)) \rightharpoondown code(label(kl)) \end{array}$$

$$\begin{array}{l} quote[n](\lrcorner) \rightharpoondown code(\lrcorner) \\ quote[n](kl_1, kl_2) \rightharpoonup quote[n](kl_1)\ \boxed{\ }\ quote[n](kl_2) \quad \text{when } kl_1 \neq \lrcorner\ ,\ kl_2 \neq \lrcorner \\ code(kl_1)\ \boxed{\ }\ code(kl_2) \rightharpoondown code(kl_1, kl_2) \end{array}$$

$$\begin{array}{l} quote[n](\cdot) \rightharpoondown code(\cdot) \\ quote[n](k_1 \curvearrowright k_2) \rightharpoonup quote[n](k_1)\ \boxed{\curvearrowright\!\!\!\!\!\!\sim}\ quote[n](k_2) \quad \text{when } k_1 \neq \cdot\ ,\ k_2 \neq \cdot \\ code(k_1)\ \boxed{\curvearrowright\!\!\!\!\!\!\sim}\ code(k_2) \rightharpoondown code(k_1 \curvearrowright k_2) \end{array}$$

$$\begin{array}{l} \texttt{lift}(v) \rightharpoondown code(v) \\ \texttt{eval}(code(k)) \rightharpoondown k \end{array}$$

Section 6.4 shows a program using code generation (together with many other CHALLENGE features).

6.4 A Challenge Program

```
 1 var in, exit, g, length, array, a, n, i, j, t, mergeSortAgent; in = [];
 2 callcc lambda k.{callcc lambda k.{g=k;};i=read;if i==0 then k(0);else {in=i:in;g(0);}};
 3 callcc lambda k.{ spawn k(0) ;
 4   mergeSortAgent = newAgent {var split, merge, mergeSort;
 5     split = lambda l . { var t; if l == [] then return [[],[]];
 6       else if tail l == [] then return [[head l],[]];
 7       else {t=split(tail tail l); return [head l:head t, head tail l:head tail t];}};
 8     merge = lambda (left,right) . {
 9       if left == [] then return right; else if right == [] then return left;
10       else if head left <= head right then return head left : merge(tail left, right);
11       else return head right : merge(left, tail right);};
12     var m,c,code; m = receive; send & c to m; code = eval receiveFrom m; aspect code();
13     mergeSort = lambda l.{ if l==[] then return []; else if tail l == [] then return l;
14       else {var t, left, right; t = split(l);
15         spawn {left=mergeSort(head t); rendezvous & left;}
16         right=mergeSort(head tail t); rendezvous & left; return merge(left,right);}};
17     while true do {m = receive; spawn send [mergeSort(head tail m),c] to head m;}
18   };       //  end of the creation of MergeSortAgent
19   send me to mergeSortAgent ;
20   var z ; z = lift receiveFrom mergeSortAgent ;
21   send quote(lambda().{acquire unquote z; *unquote z=*unquote z+1; release unquote z;})
22      to mergeSortAgent ;
23   send [me,in] to mergeSortAgent ;
24   spawn {print receive ; haltSystem ;}
25   haltThread ;
26 };         //  end of the main thread in the main agent
27 n = (mu length.lambda l.{if l==[] then return 0; else return 1 + length(tail l);})(in);
28 array = malloc n; a = array; while not in == [] do {* a = head in; in = tail in; ++a;}
29 i=-1; while ++i+2<=n do {j=i; while ++j+1<=n do { if not *(array+i)<=*(array+j)
30         then {t=*(array+i);*(array+i)=*(array+j);*(array+j)=t;} else {} }}
31 send(mu f.lambda(a,n).{if n==0 then return[];else return *a:f(a+1,n-1);})(array,n)to me;
32 free array ;
```

Figure 12: A complex complete Challenge program involving almost all the language constructs

Here we discuss a non-trivial Challenge program that can be used as a stress-test for the semantics. It makes use of almost all the language constructs of Challenge. The overall purpose of the program is to sort the elements of a list. However, it does it in a specific way that makes intensive use of Challenge's features. Even though this approach to sorting may appear unnecessarily complex in a uni-processor implementation of Challenge, it may in fact provide an efficient sorting procedure on multi-core implementations. The idea is that the main agent first reads the data to sort in a list, and then starts a local thread and an agent. The local thread is given the task to allocate the data in memory and then sort it using a selection sort. The agent is given the task to sort the list using a multi-threaded variant of merge-sort, where a thread is created at each split to take care of sorting one half of the list. When either the selection-sort thread or the merge-sort agent terminates, it sends a message to the original agent, which spawned a new thread waiting for such a message. Then the sorted list is printed and the system is halted.

Line 2 reads the numbers to sort and puts them in the list `in`. To test `callcc`, we used it both for exceptions and for looping (the use of `callcc` for these operations is folklore): the first `callcc` captures the exit continuation (which is invoked whenever a `0` is read), and the second `callcc` captures the looping continuation and "leaks" it into the variable `g`.

The third `callcc`, at lines 3-26 is more subtle and cannot be simulated with other constructs. Before we explain it, let us first discuss the lines 27-30 and then a programming scenario that motivates the architecture of our program. Line 27 calculates the length of the input list by defining and applying a recursive function. Line 28 allocates a block of memory and writes the list elements in it. Lines 29-30 implement selection sort using direct memory access. Suppose that, originally, one's program consisted only of this selection

sort code; in other words, suppose that the `callcc` at lines 3-26 was not originally planned. A possible continuation of the original program could have been to print the sorted list (calculated with the code in the first argument of the `send` command in line 31), and then free the allocated memory (line 32). Now suppose that the original program was sometimes slow and one wanted to improve it with minimal work, taking advantage of concurrency. One possibility to do it is to start two threads, one executing selection sort as before and another running a different sorting algorithm, and then collect the answer from whichever thread finishes first. To achieve that with minimal programming, one can do the following: (1) shift the main computation from selection sort to a manager performing the above (the `callcc` at line 3), as explained next; (2) create a new thread that executes the old selection sort code (the `spawn` at line 3); (3) initiate a concurrent sorting procedure (lines 4-23; this is discussed below); (4) spawn a thread that receives the sorted list from whomever finishes first, then prints it and halt the system (line 24); (5) halt the "manager" thread, to void executing the subsequent selection sorting again. No change was needed in the original selection sorting code, except for replacing the print statement with sending a message.

Let us next discuss step (3) above, the initiation of merge sorting. Since merge sort admits elegant parallel implementations, to avoid loading the original agent and for separation of concerns reasons we prefer to create a new agent that offers merge-sort services (lines 4-18). The procedures for splitting (lines 5-7) and merging (lines 8-11) are standard; note, however, that we achieve recursion by means of side effects and assignment (e.g., `split` is allocated a location at line 4, then at line 5 a closure is written at that location that sees `split` in its environment so that it is properly retrieved when `split` is invoked at line 7). The `mergeSort` procedure spawns a thread to sort one half of the list and write the sorted result in a local variable `left`, and it proceeds to sort the other half. To make sure that the spawned thread is complete before the actual merging of the two sorted halves, the address of `left` is used as a rendezvous barrier for synchronization.

For the sake of testing our semantics for intra-agent message communication, code generation and aspects, we assume that a certain protocol must be obeyed when initializing and communicating with a merge-sort agent: first, one should send one's id to the merge-sort agent one wants to initialize (the `send` statement at line 19 and the corresponding `m=receive` at line 12); second, the merge-sort agent sends back to the initiating agent a local address, where an aspect state can be stored (the `send` at line 12); third, the initiating agent takes the sent location (line 20) and generates some code based on it –incrementing the value at the sent location–, which is sent to the merge-sort agent where it makes sense (lines 21-22); fourth, the generated code is set as an aspect by merge-sort agent, so it will be executed each time a function is invoked (end of line 12). Now the initialization of the merge-sort agent is complete, so it is ready to accept tasks (line 17). A task consists of a received message containing the sender agent's id and a list to sort (e.g., see line 23). The merge-sort agent spawns a new thread for each task and, once completed, sends back to the sender the sorted list as well as the state of the aspect.

The CHALLENGE program above, as well as many others together with other language definitions and corresponding programs, can be reached from `http://k.cs.uiuc.edu`, the K website. We are currently using the K-Maude tool [42] to execute and formally analyze K definitions. To execute the program above in K-Maude, one needs to rewrite a configuration containing the program in the $\langle\rangle_{\mathsf{k}}$ cell of some agent and a list of integers, including a 0, in the $\langle\rangle_{\mathsf{in}}$ cell.

7 Conclusion, Current Work and Future Work

We presented the K semantic framework, consisting of a general-purpose concurrent rewriting approach together with a definitional technique specialized for concurrent programming languages or systems. We hope that we presented compelling arguments that K brings together the advantages of the existing language definitional frameworks and avoids some of 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 [12]. More recently, the same definition was adapted and used to verify security-related properties for Java programs [1, 2].

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 [17], which can be used to analyze real C programs. Additionally, [41] uses K to define a symbolic semantics for pointer allocation in a C-like language, aiming at runtime-verifying memory safety. A K semantic definition of Scheme R5RS is given in [32], and one for the Beta language in [16].

There are also several tools and techniques based on K. For example, [42] describes the K-Maude prototype, a Maude implementation of the K framework which fully supports the K definitional style as portrayed here. A module system for K aiming at maximizing modularity and code-reuse is discussed in [21]. The potential for efficient executability of K definitions has been first empirically noted in [22]. An experimental object-oriented programming language is defined using K in [18–20], together with several formal analyses and optimizations based on it. The K framework is compared in [44] with P-systems [34] and shown that it can be systematically used to define, execute and analyze a large body of 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 [39, 40]. As shown in [11], 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 interpreters for programming languages from their K semantics.

Acknowledgments This research was supported by NSF grants CCF-0916893, CNS-0720512, and CCF-0448501, by NASA contract NNL08AA23C, and by several Microsoft gifts. The authors would like to thank Mark Hills, Narciso Martí Oliet, Patrick Meredith, Peter Mosses, as well as the anonymous reviewers for insightful suggestions on earlier drafts of this paper. Warm thanks also go to the members of the FSL group at Urbana and Dorel Lucanu's group in Iasi, Romania, who provided invaluable feedback during the last 7 years of research on K and applications. Last but not least, we are grateful to Gheorghe Păun and Gheorghe Stefănescu for noticing the interesting relationships between P-systems and K, and for encouraging us to write and submit this paper.

References

[1] M. Alba-Castro, M. Alpuente, and S. Escobar. Automatic certification of Java source code in rewriting logic. In S. Leue and P. Merino, editors, *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, pages 200–217. Springer, 2007.

[2] M. Alba-Castro, M. Alpuente, S. Escobar, P. Ojeda, and 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)*, volume 248 of *Electr. Notes Theor. Comput. Sci.*, pages 19–29, 2009.

[3] J.-P. Banâtre and D. L. Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15(1):55–77, 1990.

[4] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, and A. Troina. A calculus of looping sequences for modelling microbiological systems. *Fundam. Inform.*, 72(1–3):21–35, 2006.

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

[6] P. Bottoni, C. Martín-Vide, G. Păun, and G. Rozenberg. Membrane systems with promoters/inhibitors. *Acta Informatica*, 38(10):695–720.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

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

[9] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. RS RS-04-26, BRICS, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. 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.

[10] H. Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer, 1978.

[11] C. Ellison, T. F. Șerbănuță, and G. Roșu. 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*, volume 5486 of *Lecture Notes in Computer Science*, pages 135–151. Springer, 2009.

[12] A. Farzan, F. Chen, J. Meseguer, and G. Roșu. Formal analysis of Java programs in JavaFAN. In *Proceedings of Computer-aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 501 – 505, 2004.

[13] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, pages 193–219, Ebberup, Denmark, Aug. 1986.

[14] J.-L. Giavitto and O. Michel. MGS: a rule-based programming language for complex objects and collections. In M. van den Brand and R. Verma, editors, *RULE'01*, volume 59 of *Electronic Notes in Theoretical Computer Science*, pages 286–304. Elsevier Science Publishers, 2001.

[15] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[16] M. Hills, T. B. Aktemur, and G. Roșu. An Executable Semantic Definition of the Beta Language using Rewriting Logic. Technical Report UIUCDCS-R-2005-2650, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[17] M. Hills, F. Chen, and G. Roșu. 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)*, volume To Appear of *ENTCS*. Elsevier, 2008.

[18] M. Hills and G. Roșu. 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*, pages 827–828, New York, NY, USA, 2007. ACM.

[19] M. Hills and G. Roșu. Kool: An application of rewriting logic to language prototyping and analysis. In F. Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 246–256. Springer, 2007.

[20] M. Hills and G. Roșu. 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)*, volume 4468 of *LNCS*, pages 107–121. Springer, 2007. also appeared as Technical Report UIUCDCS-R-2007-2809, January 2007.

[21] M. Hills and G. Roșu. Towards a module system for K. In A. Corradini and U. Montanari, editors, *WADT*, volume 5486 of *Lecture Notes in Computer Science*, pages 187–205. Springer, 2008.

[22] M. Hills, T. F. Șerbănuță, and G. Roșu. 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)*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 215–231. Elsevier Science, July 2007. also appeared as Technical Report UIUCDCS-R-2005-2667, December 2005.

[23] G. Kahn. Natural semantics. In F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

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

[25] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In U. Montanari and V. Sassone, editors, *CONCUR*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer, 1996.

[26] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In U. Montanari and V. Sassone, editors, *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer, 1996.

[27] J. Meseguer and G. Roșu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. A. Basin and M. Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Computer Science*, pages 1–44. Springer, 2004.

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

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

[30] P. D. Mosses. Pragmatics of modular SOS. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002.

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

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

[33] A. Păun and G. Păun. The power of communication: P systems with symport/antiport. *New Generation Computing*, 20(3):295–305, 2002.

[34] G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 2000.

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

[36] G. Roșu. CS322, Fall 2003 - Programming Language Design: Lecture Notes. Technical Report UIUCDCS-R-2003-2897, Department of Computer Science, University of Illinois at Urbana-Champaign, December 2003. Lecture notes of a course taught at UIUC.

[37] G. Roșu. K: A rewriting-based framework for computations – preliminary version. Technical Report Department of Computer Science UIUCDCS-R-2007-2926 and College of Engineering UILU-ENG-2007-1827, University of Illinois at Urbana-Champaign, 2007.

[38] G. Roșu. *Programming Languages—A Rewriting Approach—*. draft. accesible online at: http://fsl.cs.uiuc.edu/pub/pl.pdf.

[39] G. Roșu, C. Ellison, and W. Schulte. From rewriting logic executable semantics to matching logic program verification. Technical Report http://hdl.handle.net/2142/13159, University of Illinois, July 2009.

[40] G. Roșu and W. Schulte. Matching logic — extended report. Technical Report Department of Computer Science UIUCDCS-R-2009-3026, University of Illinois at Urbana-Champaign, January 2009.

[41] G. Roșu, W. Schulte, and T. F. Șerbănuță. Runtime verification of C memory safety. In *Runtime Verification (RV'09)*, volume 5779 of *Lecture Notes in Computer Science*, pages 132–152, 2009.

[42] T. F. Șerbănuță and G. Roșu. 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.

[43] T. F. Șerbănuță, G. Roșu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207:305–340, 2009.

[44] T. F. Șerbănuță, G. Stefanescu, and G. Roșu. Defining and executing P systems with structured data in K. In D. W. Corne, P. Frisco, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Workshop on Membrane Computing (WMC'08)*, volume 5391 of *Lecture Notes in Computer Science*, pages 374–393. Springer, 2009.

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