Chapter 5

The K Semantic Framework

This chapter introduces K, a rewriting-based executable semantic framework which will be used in the remainder of this book. K was first introduced by the author in the lecture notes of a programming language design course at the University of Illinois at Urbana-Champaign (UIUC) in Fall 2003 [43], as a means to define concurrent languages in rewriting logic using Maude. Since 2003, K has been used continuously in teaching programming languages, in seminars, and in research.

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

The K framework consists of two components:

1. The *K concurrent rewrite abstract machine*, abbreviated KRAM and discussed in Section 5.4;

2. The *K technique*, discussed in Section 5.5.

Like conventional rewrite systems, a K-system consists of a signature for building terms and of a set of rules for iteratively rewriting terms. Like in rewriting logic (Section 2.7), K rules can be applied concurrently and unrestricted by context. The novelty of the KRAM (Section 5.4) is that its rules contain, besides expected information saying how the original term is modified (the *write data*), also information about what parts of the term are shared with other rules (the *read-only data*). This additional information provided as part of the rules allows K to be a suitable rewrite-based framework for semantically defining truly concurrent programming languages and calculi whose threads or processes may be desired to share data. The K concurrent rewrites associated to a K system may require several interleaved rewrites in the rewrite logic theory straightforwardly associated to the K-system (i.e., by forgetting the sharing information).

Even though the KRAM aims at maximizing the amount of concurrency that can be achieved in a rewriting setting, it does not tell *how* one can define a programming language or a calculus as a K-system. In particular, a bad K definition may partially or totally inhibit KRAM's potential for concurrency. The K technique discussed in Section 5.5 proposes a definitional methodology that makes the use of the KRAM convenient when formally defining programming languages and calculi. Moreover, the K technique can also be and actually has already been intensively used as a technique to define languages and calculi as conventional term rewrite systems or as rewrite logic theories. There is one important thing lost in the translation of K into rewriting logic though, namely the degree of true concurrency of the original K-system. Ignoring this true concurrency aspect, the relationship between K and rewriting logic in general and Maude in particular is the same as that between any of the conventional semantic styles discussed in Chapter 3 and rewriting logic and Maude: the latter can be used to execute and analyze K-systems.

This chapter is structured as follows:

- Section 5.1 discusses informally the requirements that have led to the design and development of the K framework, and hereby, it highlights the objectives that K attempts to achieve. These requirements are derived from what we believe the characteristics of an ideal semantic framework should be, and the motivation for the K framework comes from the observation that the existing semantic frameworks fail to satisfy these requirements.

- Section 5.2 gives a quick overview of the K framework by using it to define the IMP language (Section 3.1) and its extension IMP++ (Section 3.8). This section should give the reader quite a clear feel for what K is about and how it operates.

- Section 5.4 describes the K concurrent rewrite abstract machine (KRAM) and is the most technical part of this chapter. However, it formalizes a relatively intuitive process of concurrent rewriting with sharing, so the reader interested more in using K then in the technical details of its core concurrent rewriting machinery may safely skip this section.

- Section 5.5 presents the K technique, explaining essentially how the KRAM or other rewrite infrastructures can be used to define programming language semantics by means of nested-cell configurations, computations anf rewrite rules.

- Section 5.7 shows K at work: it introduces and at the same time shows how to give a K semantics to CHALLENGE, a programming language containing varied features known to be problematic to define in other frameworks. CHALLENGE was conceived as a means to challenge the various language definitional frameworks and to expose their limitations.

5.1 Quest for an Ideal Language Definitional Framework

Chapter 3 showed that any conventional language definitional style can be faithfully, step-for-step, captured by a rewriting logic theory. It may then seem "obvious" to the hasty reader that rewriting logic is perhaps the ideal language definitional framework and thus naturally ask the following:

What is the need for yet another language definitional framework that can be embedded in rewriting logic, K in this case, if rewriting logic is already so powerful?

Unfortunately, in spite of its definitional strength as a computational logic framework, rewriting logic does not give, and does not intend to give, the programming language designer any recipe on *how* to define a language. It essentially only suggests the following: however one wants to formally define a programming language or calculus, one can probably also do it in rewriting logic following the same intuitions and style. Therefore, rewriting logic can be regarded as a *meta-framework* that supports definitions of programming languages and calculi among many other things, providing the language designer with a means to execute and formally analyze languages in a generic way, but only *after* the language is already defined. Additionally, as discussed in Section 2.7 and in more depth in Section 5.2, the natural rewriting logic definition of a concurrent programming language may enforce interleaving in situations where true concurrency is meant.

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

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

1. *Gives a strong intuition, even precise recipes, on how to define a language?*
2. *Same for language-related definitions, such as type checkers, type inferencers, abstract interpreters, safety policy or domain-specific checkers, etc.?*
3. *Can define arbitrarily complex language features, including, obviously, all those found in existing languages, capturing also their intended computational granularity?*
4. *Is modular, that is, adding new language features does not require to modify existing definitions of unrelated features? Modularity is crucial for scalability and reuse.*
5. *Supports non-determinism and concurrency, at any desired granularity?*
6. *Is generic, that is, not tied to any particular programming language or paradigm?*
7. *Is executable, so one can "test" language or formal analyzer definitions, as if one already had an interpreter or a compiler for one's language? Efficient executability of language definitions may even eliminate the need for interpreters or compilers.*
8. *Has state-exploration capabilities, including exhaustive behavior analysis (e.g., finite-state model-checking), when one's language is non-deterministic or/and concurrent?*
9. *Has a corresponding initial-model (to allow inductive proofs) or axiomatic semantics (to allow Hoare-style proofs), so that one can formally reason about programs?*

The list above contains a *minimal* set of desirable features that an ideal language definitional framework should have. Unfortunately, the current practice is to take the above features one at a time, temporarily or permanently declaring the others as "something else". We next describe how current practices and language definitional styles fail to satisfy the above-mentioned requirements.

1. *Gives a strong intuition, even precise recipes, on how to define a language?*

   To formalize one's intuition about a language feature, it is common practice to use a big-step or a small-step SOS definition, with or without evaluation contexts, typically on paper, without any machine support. Sometimes this so-called "formal" process is pushed to extreme in what regards its informality, in the sense that one can see definitions of some language features using one definitional style and of other features using another definitional style, without ever proving that the two definitional styles can co-exist in the claimed form for the particular language under consideration. For example, one may use a big-step SOS to give semantics to a code-self-generation extension of Java, while using a small-step SOS to define the concurrency semantics of Java. However, common sense tells that once one has concurrency and shared memory, one cannot have a big-step SOS definition. An ideal language definitional framework should provide a uniform, compact and rigorous way to modularly define various language features, avoiding the need to define different language features following different styles.

2. *Same for language-related definitions, such as type checkers, type inferences, abstract interpreters, safety policy or domain-specific checkers, etc.?*

   To define a type system or a (domain-specific or not) safety policy for a language, one may follow a big-step-like definitional style, or even simply provide an algorithm to serve as a formal definition. While this appears to be, and in many cases indeed is acceptable, there can be a significant "formal gap" between the actual language semantic definition and its type system or safety policy regarded as mathematical objects, because in order to carry out proofs relating the two one needs one common formal ground. In practice, one typically ends up "encoding" the two in yet another framework, claimed to be "richer", and then carry out the proofs within that framework. But how can one make sure that the encodings are correct? Do they serve as alternative definitions for that sole purpose?

   An ideal language definitional framework should have all the benefits of the "richer" framework, at no additional notational or logical complexity, yet naturally capturing the complete meaning of the defined constructs. In other words, in an ideal framework one should define a language as a mathematical object, say $\mathcal{L}$, and a type system or other abstract interpretation of it as another mathematical object over the same formalism, say $\mathcal{L}'$, and then carry out proofs relating $\mathcal{L}$ and $\mathcal{L}'$ using the provided proof system of the definitional framework. $\mathcal{L}$, $\mathcal{L}'$, as well as other related definitions, should be human readable and easy to understand enough so that one does not feel the drive to give alternative, more intuitive definitions using a more informal notation or framework.

3. *Can define arbitrarily complex language features, including, obviously, all those found in existing languages, capturing also their intended computational granularity?*

   Some popular language definitional frameworks are incapable of defining even existing language features. The fact that a particular language feature is supported in some existing language serves as the strongest argument that that feature may be desirable, so an ideal language definitional framework must simply support it; in other words, one cannot argue against the usefulness of that feature just because one's favorite definitional framework does not support it. For example, since in standard SOS definitions (not including reduction semantics with evaluation contexts) the "control flow" information of a program is captured within the structure of the "proof", and since proof derivations are not first class objects in these formalisms, it makes it very hard, virtually impossible in these formalisms to define complex control intensive language constructs like, e.g., call-with-current-continuation (callcc).

   Another important example showing that conventional definitional frameworks (e.g., SOS) fail to properly support existing common language features, is concurrency. Most frameworks enforce an interleaving semantics, which may not necessarily always be the desired approach to concurrency. In particular, an implementation of a multi-threaded system in which two threads can concurrently read a shared variable would be dissallowed, because it disobeys the "formal" interleaving-based language definition. Concurrency is further discussed in item 5.

Some frameworks provide a "well-chosen" set of constructs, shown to be theoretically sufficient to define any computable function or algorithm, and then propose *encodings* of other language features into the set of basic ones; examples in this category are Turing machines or the plethora of (typed or untyped) λ-calculi, or π-calculi, etc. While these basic constructs yield interesting and meaningful idealized programming languages, using them to encode other language features is, in our view, inappropriate. Indeed, encodings hide the intended *computational granularity* of the defined language constructs; for example, a variable lookup intended to be a one-step operation in one's language should take precisely one step in an ideal framework (not hundreds/thousands of steps as in a Turing machine or lambda calculus encoding, not even two steps: first get location, then get value).

4. *Is modular, that is, adding new language features does not require to modify existing definitions of unrelated features? Modularity is crucial for scalability and reuse.*

   As Mosses pointed out in [38] and as shown in Chapter 3 and discussed in Section 3.9, big-step and small-step SOS are non-modular; Plotkin himself had to modify the definition of simple arithmetic expressions (in the original notes on SOS [42]) three times as his initial language evolved. As seen in Section 3.8, to add an innocent abrupt termination statement to a language defined using SOS, say a `halt`, one needs to more than double the total number of rules: each language construct needs to be allowed to "propagate" the halting signal potentially generated by its arguments. Also, as one needs to add more items into configurations to support new language features, in SOS one needs to change again every rule to include the new items; note that there are no less than $7 + n * 10$ configuration items, where n is the number of threads, in the configuration of SKOOL in Section B.4 (which is a comparatively simple language) as shown in Figure **??**. It can easily become very annoying and error prone to modify a large portion of unrelated existing definitions when adding a new feature.

   A language designer may be unwilling to add a new feature or improve the definition of an existing one, just because of the large number of required changes. Informal writing conventions are sometimes adopted to circumvent the non-modularity of SOS. For example, in the definition of Standard ML [35], Milner and his collaborators propose a "store convention" to avoid having to mention the store in every rule, and an "exception convention" to avoid having to double the number of rules for the sole purpose of supporting exceptions. As rightfully noticed by Mosses [38], such conventions are not only adhoc and language specific, but may also lead to erroneous definitions. Mosses' Modular SOS [38] (MSOS) brings modularity to SOS in a formal and elegant way, by grouping the non-syntactic configuration items into transition labels, and allowing rules to mention only those items of interest from each label. As discussed in Section 3.4, MSOS still inherits all the remaining limitations of SOS.

5. *Supports non-determinism and concurrency, at any desired granularity?*

   By inherently enforcing an interleaving semantics for concurrency, existing reduction semantics definitions (including ones based on evaluation contexts) can only capture a projection of concurrency (when one's goal is to define a truly concurrent language), namely its resulting non-determinism. Proponents of existing reduction semantics approaches may argue that the resulting non-deterministic behavior of a concurrent system is all what matters, while proponents of true concurrency may argue that a framework which does not support naturally concurrent actions, i.e., actions that take place *at the same time*, is not a framework for concurrency. We do not intend to discuss the (admittedly important but debatable) distinctions between non-determinism and interleaving vs. true concurrency here. The fact that there are language designers who desire an interleaving semantics while others who desire a true concurrency semantics for their language is strong evidence that an ideal language definitional framework should simply support both, preferably with no additional settings of the framework, but rather via particular definitional methodologies within the framework.

6. *Is generic, that is, not tied to any particular programming language or paradigm?*

   A non-generic framework, i.e., one building upon a particular programming language or paradigm, may be hard or impossible to use at its full strength when defining a language that crosses the boundaries of the underlying language or paradigm. For example, a framework enforcing object or thread communication via explicit send and receive messages may require artificial encodings of languages that opt for a different communication approach (e.g., shared memory), while a framework enforcing static typing of programs in the defined language may be inconvenient for defining dynamically typed or untyped languages. In general, a framework providing and enforcing particular ways to define certain types of language features would lack genericity. Within an ideal framework, one can and should develop and adopt methodologies for defining certain types of languages or language features, but these should not be enforced. This genericity requirement is derived from the observation that today's programming languages are so diverse and based on orthogonal, sometimes even conflicting paradigms, that, regardless of how much we believe in the superiority of a particular language paradigm, be it object-oriented, functional or logical, a commitment to any existing paradigm would significantly diminish the strengths of a language definitional framework.

7. *Is executable, so one can "test" language or formal analyzer definitions, as if one already had an interpreter or a compiler for one's language? Efficient executability of language definitions may even eliminate the need for interpreters or compilers.*

   Most existing language definitional frameworks are, or until relatively recently were, lacking tool support for executability. Without the capability to execute language definitions, it is virtually impossible to debug or develop large and complex language definitions in a reasonable period of time. The common practice today is still to have a paper definition of a language using one's desired formalism, and *then* to implement an interpreter for the defined language following in principle the paper definition. This approach, besides the inconvenience of having to define the language twice, guarantees little to nothing about the appropriateness of the formal, paper definition. Compare this approach to an approach where there is *no gap* between the formal definition and its implementation as an interpreter. While any definition is by definition correct, one gets significantly more confidence in the appropriateness of a language definition, and is less reluctant to change it, when one is able to run it *as is* on tens or hundreds of programs. Recently, executability engines have been proposed both for MSOS (the MSOS tool, implemented by Braga and collaborators in Maude [8]) and for reduction semantics with evaluation contexts (the PLT Redex tool, implemented by Findler and his collaborators in Scheme [23]). A framework providing *efficient* support for executability of formal language definitions may eliminate entirely the need to implement interpreters, or type checkers or type inferencers, for a language, because one can use directly the formal definition for that purpose.

8. *Has state-exploration capabilities, including exhaustive behavior analysis (e.g., finite-state model-checking), when one's language is non-deterministic or/and concurrent?*

   While executability of language definitions is indispensable when designing non-trivial languages, one needs richer tool support when the language is concurrent or non-deterministic. Indeed, it may be that one's definition is appropriate for particular thread or process interleavings (e.g., when blocks are executed atomically), but that it has unexpected behaviors for other interleavings. Moreover, somewhat related to the desired computational granularity of language constructs mentioned in item 3 above, one may wish to exhaustively investigate all possible interleavings or executions of a particular concurrent program, to make sure that no undesired behaviors are present and no desired behaviors are excluded. When the state space of the analyzed program is large, manual analysis of behaviors may not be feasible; therefore, model-checking and/or safety property analysis (through systematic state-space exploration) are also desirable as intrinsic components of an ideal language definitional framework.

9. *Has a corresponding initial-model (to allow inductive proofs) or axiomatic semantics (to allow Hoare-style proofs), so that one can formally reason about programs?*

   To prove properties about programs in a defined programming language, or properties about the programming language itself, as also mentioned in item 2 above, the current practice is to encode/redefine the language semantics in a "richer" framework, such as a theorem prover, and then carry out the desired proofs there. Redefining the semantics of a fully fledged programming language in a different formalism is a highly nontrivial, error prone and tedious task, possibly taking months; automated translations may be possible when the original definition of the language is itself formal, though one would need to validate the translator. In addition to the "formal gap" mentioned in item 2 due to the translation itself, this process of redefining the language is simply inconvenient. An ideal language definitional framework should allow one to have, for each language, "one definition serving all purposes", including all those mentioned above.

   Current program verification approaches are based on axiomatic semantics (in the style of Hoare logic) of the language under consideration and on implementations of it in program verifiers. In fact, implementing program verifiers is still an art, one that few master. Existing program verifiers are based "in principle" on some implicit axiomatic semantics which is hand-crafted in the prover; a formal semantics is in fact not required and typically not given at all, thus creating an obvious gap between the implementation of a program verifier and the language semantics. Moreover, since axiomatic semantics are not executable and thus not testable, the underlying axiomatic semantics can be itself untrustable; at minimum, an alternative executable semantics of the language is needed and a proof that the axiomatic semantics is sound for it. Thus there is a double gap between a language definition and a program verifier for it. The very fact that one needs various semantics of a language for various purposes shows that none of these semantics is "ideal": as already stated above, an ideal language semantic definition should serve all the purposes.

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

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

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

5.2 K Overview by Example

Here we briefly describe the K framework, what it offers and how it can be used. We use as concrete examples the IMP language (Section 3.1) and its extension IMP++ (Section 3.8), discussed in Chapter 3 in the context of the various existing language definitional frameworks. We define both an executable semantics and a type system for these languages. The type system is included mainly for demonstration purposes, to show that one can use the same definitional framework, K, to define both formal language semantics and language abstractions. The role of this section is threefold: first, it gives the reader a better understanding of the K framework before we proceed to define it rigorously in the remainder of this chapter; second, it shows how K avoids the limitations of the various more conventional semantic approaches discussed in Chapter 3; and third, it shows that K is actually easy to use, in spite of the intricate K concurrent abstract machine technicalities discussed in Section 5.4 — indeed, users of the K framework need not be familiar with all those intricate details, the same way users of a concurrent programming language need not be aware of the underlying details of the concurrent computing architecture on which their programs are executed. In fact, in this section we make no distinction between the K rewrite abstract machine and the K technique, refering to these collectively as "the K framework", or more simply just "K".

Programming languages, calculi, as well as type systems or formal analyzers can be defined in K by making use of special, potentially nested *(K) cell* structures, and *(K) (rewrite) rules*. There are two types of K rules: *computational rules*, which count as computational steps, and *structural rules*, which do not count as computational steps. The role of the structural rules is to rearrange the term so that the computational rules can apply. K rules are *unconditional* (they can be thought of as rule schemata and may have ordinary side conditions, though), and they are *context-insensitive*, so

K rules apply concurrently as soon as they match, without any contextual delay or restrictions.

One sort has a special meaning in K, namely the sort K of *computations*. The intuition for terms of sort K is that they have computational contents, such as programs or fragments of programs have; indeed, computations extend the syntax of the original language. Computations have a list structure with "$\curvearrowright$" (read "followed by") concatenating two computations and "$\cdot$" the empty computation; the list structure captures the intuition of computation sequentialization. Computations give an elegant and uniform means to define and handle evaluation contexts (Section 3.8.1) and/or continuations [52]. Indeed, a computation "$v \curvearrowright c$" can be thought of as "$c[v]$, that is, evaluation context c applied to v" or as "passing v to continuation c". Computations can be handled like any other terms in a rewriting environment, that is, they can be matched, moved from one place to another in the original term, 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 (Section 2.7) in particular, is that K allows rewrite rules to apply *concurrently* even in cases when they overlap, provided that they do not change the overlapped portion of the term. This allows for *truly concurrent semantics* to programming languages and calculi. For example, two threads that read the same location of memory can do that concurrently, even though the corresponding rules overlap on the store location being read. The distinctive feature of K compared to other frameworks for true concurrency, like chemical abstract machines (Section 3.6) or membrane systems (Section 9.7), 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 (CHAMs; Section 3.6) and reduction semantics with evaluation contexts (RSEC; Section 3.8.1), at the same time avoiding what might be called the "rigidity to chemistry" of the former and the "rigidity to syntax" of the latter. Any CHAM and any RSEC definition can be captured in K with minimal (in our view *zero*) representational distance. K can support concurrent language definitions with either an interleaving or a true concurrency semantics.

Like the other semantic approaches that can be represented in rewriting logic (Chapter 3), K can also be represented in rewriting logic and thus K definitions can be executed on existing rewrite engines, thus providing "interpreters for free" directly from formal language definitions; additionally, general-purpose formal analysis techniques and tools developed for rewriting logic, such as state space exploration for safety violations or model-checking, give us corresponding techniques and tools for the defined languages, at no additional development cost. Unlike the other semantic approaches whose representations in rewriting logic are *faithful*, in that the resulting rewriting logic theories are step-for-step equivalent with the original definitions, K cannot be captured faithfully by rewriting logic in any natural way. On the one hand, there is no clear way to represent the K structural rules in rewriting logic in a general way that properly captures the intuition that the K computational rules take place "modulo" the structural ones; this results in the rewrite logic theory to potentially miss some of the non-deterministic behaviors of the original K definition. On the other hand, the corresponding rewrite logic theory may need more interleaved steps in order to capture one concurrent step in the original K definition; this results in the rewrite logic theory to potentially miss some of the concurrent behaviors of the original K definition.

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

Figure 5.1: K definition of IMP: syntax (left), annotations (middle) and semantics (right); $x \in \mathit{VarId}$, $\mathit{xl} \in \mathbf{List}\{\mathit{VarId}\}$, $i, i_1, i_2 \in \mathit{Int}$, $t \in \mathit{Bool}$, $b \in \mathit{BExp}$, $s, s_1, s_2 \in \mathit{Stmt}$ (b, s, s_1, s_2 can also be in K)

5.2.1 K Semantics of IMP

Figure 5.1 shows the complete K definition of IMP, except for the configuration; the IMP configuration is explained separately below. The left column in Figure 5.1 gives the IMP syntax (identical to the one in Section 3.1.1), which uses the algebraic CFG notation introduced in Section 2.5. The middle column contains special syntax K annotations, called *strictness attributes*, stating the evaluation strategy of some language constructs. Finally, the right column gives the semantic rules.

K makes intensive use of the algebraic CFG notation (Section 2.5) to define configurations, in particular of list, set, multiset and map structures. Like in the CHAM or P-systems, program or system configurations in K are organized as potentially nested structures of *cells* (we call them cells instead of molecules like in CHAM or membranes like in P-systems to avoid confusion with terminology in CHAM/P-systems as well as confusion with terminology in chemistry or biology — note that "cell" has a more wide-spread use, e.g., "memory cell", etc.). However, unlike in CHAM/P-systems which only provide multisets (or bags), K also provides list, set and map cells in addition to multiset (called bag) cells; K's cells may be labelled to distinguish them from each other. We use angle brackets as cell wrappers. The K configuration of IMP can be defined as follows:

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

In words, IMP configurations consist of a top cell $\langle \ldots\rangle_{\top}$ containing two other cells inside: a cell $\langle \ldots\rangle_{\mathsf{k}}$ which holds a term of sort K (terms of sort K are called computations and extend the original language syntax as explained in the next paragraph) and and a cell $\langle ... \rangle_{\mathsf{state}}$ which holds a map from variables to integers. In Chapter 3, we used the sort *State* as an alias (introduced in Section 3.1.2) for the map sort from variables to integers; since in this chapter we are going to have many maps, lists and bags, we prefer to avoid using aliases. As examples of IMP K configurations, $\langle\langle \texttt{x := 1; y := x+1} \rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{state}}\rangle_{\top}$ is a configuration holding program "`x := 1; y := x+1`" and empty state, and $\langle\langle \texttt{x := 1; y := x+1} \rangle_{\mathsf{k}}\ \langle \texttt{x} \mapsto 0, \texttt{y} \mapsto 1\rangle_{\mathsf{state}}\rangle_{\top}$ is a configuration holding the same program and a state mapping `x` to 0 and `y` to 1. When we add threads (in IMP++), the configurations can hold multiple $\langle ... \rangle_{\mathsf{k}}$ cells (its bag structure allows that).

K provides special notational support for *computational structures*, or simply *computations*. Computations have the sort K, which is therefore builtin in the K framework; the intuition for terms of sort K is that they have computational contents, such as, for example, a program or a fragment of program has. Computations extend the original language/calculus/system syntax with special "$\curvearrowright$"-separated lists "$T_1 \curvearrowright T_2 \curvearrowright \cdots \curvearrowright T_n$" comprising *(computational) tasks*, thought of as having to be "processed" sequentially ("$\curvearrowright$" reads "followed by"). The identity of the "$\curvearrowright$" associative operator is "$\cdot$". Like in reduction semantics with evaluation contexts (RSEC, see Section 3.5), K allows one to define evaluation contexts over the language syntax. However, unlike in RSEC, parsing does not play any crucial role in K, because K replaces the hard-to-implement split/plug operations of RSEC by plain, context-insensitive rewriting. Therefore, instead of defining evaluation contexts using grammars and relying on splitting syntactic terms (via parsing) into evaluation contexts and redexes, in K we define evaluation contexts using special rewrite rules. For example, the evaluation contexts of sum, comparison and conditional in IMP can be defined as follows, by means of *structural rules* (recall that the sum "`+`" was non-deterministic and the comparison "`<=`" was sequential):

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

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

The right-hand sides of the structural rules above contain, besides the task sequentialization operator "$\curvearrowright$", *freezer* operators containing "$\square$" in their names, such as "$\square$ + _", "_ + $\square$", etc. The first rule above says that in any expression of the form $a_1 + a_2$, a_1 can be scheduled for processing while a_2 is being held for future processing. Since the rules above are bi-directional, they can be used at will to structurally re-arrange the computations for processing. Thus, when iteratively applied left-to-right they fulfill the role of *splitting* syntax into an evaluation context (the tail of the resulting sequence of computational tasks) and a redex (the head of the resulting sequence), and when applied right-to-left they fulfill the role of *plugging* syntax into context. Such structural rules are often called *heating/cooling rules* in K, because they are reminiscent of the CHAM heating/cooling rules; for example, $a_1 + a_2$ is "heated" into $a_1 \curvearrowright \square + a_2$, while $a_1 \curvearrowright \square + a_2$ is "cooled" into $a_1 + a_2$. Heating/cooling rules can be used to define any evaluation context, not only strictness of operations. 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 5.1 and 5.2: "*strict*" means non-deterministically strict in all enlisted arguments (given by their positions) or by default in all arguments if none enlisted, and "*seqstrict*" is like *strict* but each argument is fully processed before moving to the next one (see the second structural rule of "`<=`" above).

The structural rules corresponding to strictness attributes (or the heating/cooling rules) decompose and eventually push the tasks that are ready for processing to the top (or the left) of the computation. Semantic rules then tell how to process the atomic tasks. The right column in Figure 5.1 shows the semantic K rules of IMP. To understand them, let us first discuss the important notion of a *K rule*, which is a strict generalization of the usual notion of a rewrite rule. To take full advantage of K's support for concurrency, K rules explicitly mention the parts of the term that they read, write, or don't care about. The underlined parts are those which are written by the rule; the term underneath the line is the new subterm replacing the one above the line.

All writes in a K rule are applied in *one parallel step*, and, with some reasonable restrictions discussed in Section 5.4 (that avoid read/write and write/write conflicts), writes in multiple K rule instances can also apply in parallel. The elipses "..." represent the volatile part of the term, that is, that part that the current rule does not care about and, consequently, can be concurrently modified by other rules. The operations which are not underlined represent the read-only part of the term: they need to stay unchanged during the application of the rule. For example, the lookup rule in Figure 5.1 (first one) says that once program variable x reaches the top of the computation, it is replaced by the value to which it is mapped in the state, regardless of the remaining computation or the other mappings in the state. Similarly, the assignment rule says that once the assignment statement "$x := i$" reaches the top of the computation, the value of x in the store is replaced by i and the statement dissolves; in K, "_" is a nameless variable of any sort and "$\cdot$" is the unit (or empty) computation (in practice, "$\cdot$" tends to be a polymorphic unit of most if not all list, set and multiset structures). The rule for variable declarations in Figure 5.1 (last one) expects an empty state and allocates and initializes with 0 all the declared variables; the dotted or dashed lines signifies that the rule is structural, which is discussed next.

K rules are split in two categories: *computational rules* and *structural rules*. Computational rules capture the intuition of computational steps in the execution of the defined system or language, while structural rules capture the intuition of structural rearrangement, rather than computational evolution, of the system. We use dashed or dotted lines in the structural rules to convey the idea that they are lighter-weight than the computational rules. Ordinary rewrite rules are a special case of K rules, when the entire term is replaced; in this case, we prefer to use the standard notation "$l \rightarrow r$" as syntactic sugar for computational rules and the notation "$l \rightharpoondown r$" or "$l \rightharpoonup r$" as syntactic sugar for structural rules. We have seen several structural rules at the beginning of this section, namely the heating/cooling rules corresponding to the strictness attributes. Figure 5.1 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 when rules should be computational versus structural and, in the latter case, when one should use "$l \rightharpoondown r$" or "$l \rightharpoonup r$" as syntactic sugar. We (subjectively) prefer to use structural rules for desugaring (like for sequential composition), loop unrolling and declarations, and we prefer to use "$\rightharpoonup$" when syntax is split into computational tasks and "$\rightharpoondown$" when computational tasks are put back into the original syntax.

Each K rule can be "desugared" into a standard term rewrite rule by combining all its changes into one top-level change. The relationship between K rules and conventional term rewriting and rewriting logic is discussed in Section 5.4. The main point is that the resulting rewrite system/theory associated to a K system lacks the potential for concurrency of the original K system.

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

Figure 5.2: K definition of IMP++ (extends that of IMP in Figure 5.1, *without changing anything*)

5.2.2 K Semantics of IMP++

In spite of its simplicity, IMP++ revealed limitations in each of the conventional semantic approaches (see Section 3.9); for example, big-step and small-step SOS as well as denotational semantics were heavily non-modular, modular SOS required artificial syntactic extensions of the language in order to attain modularity, reduction semantics with evaluation contexts lacked modularity in some cases, the CHAM relied on a heavy airlock operation to match information in solution molecules, and all these approaches have limited or no support for true concurrency (the CHAM provides more support for true concurrency than the others, but it still unnecessarily enforces interleaving in some cases). In this section we show that K avoids these limitations, at least in the case of IMP++.

Figure 5.2 shows how the K semantics of IMP can be seamlesly extended into a semantics for IMP++. To accomodate the output, a new cell needs to be added to the configuration:

$$Configuration_{\mathrm{IMP++}} \;\equiv\; \langle\langle K\rangle_{\mathsf{k}}\; \langle \mathbf{Map}\{\,VarId \mapsto Int\}\rangle_{\mathsf{state}}\; \langle \mathbf{List}\{\,Int\}\rangle_{\mathsf{output}}\;\rangle_{\top}$$

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

We conclude this section with a discusion on the concurrency of the K definition of IMP++. Since in K rule instances can share read-only data, various (actually all matching) instances of the look up rule can apply concurrently, in spite of the fact that they overlap on the state subterm. Similarly, since the rules for variable assignment and increment declare volatile everything else in the state except the mapping corresponding to the variable, mutiple assignments, increments and reads of distinct variables can happen concurrently. However, if two threads want to write the same variable, or if one wants to write it while another wants to read it, then the two corresponding rules need to interleave, because the two rule instances are in a concurrency conflict. Note also that the rule for `output` matches and changes the end of the output cell; that means, in particular, that multiple outputs by various threads need to be interleaved for the same reason as above. On the

Original language syntax	K Strictness	K Semantics
$AExp ::= Int$		$i \rightarrow int$
$\mid$ $VarId$		$\langle \underset{int}{\underline{x}} \cdots\rangle_{\mathsf{k}} \; \langle\cdots x \cdots\rangle_{\mathsf{vars}}$
$\mid$ $AExp$ `+` $AExp$	$[strict]$	int `+` $int \rightarrow int$
$\mid$ $AExp$ `/` $AExp$	$[strict]$	int `/` $int \rightarrow int$
$\mid$ `++` $VarId$		$\langle \underset{int}{\underline{\texttt{++}\,x}} \cdots\rangle_{\mathsf{k}} \; \langle\cdots x \cdots\rangle_{\mathsf{vars}}$
$BExp ::= AExp$ `<=` $AExp$	$[strict]$	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` $\rightarrow stmt$
$\mid$ $VarId$ `:=` $AExp$	$[strict(2)]$	$\langle \underset{stmt}{\underline{x \texttt{ := } int}} \cdots\rangle_{\mathsf{k}} \; \langle\cdots x \cdots\rangle_{\mathsf{vars}}$
$\mid$ $Stmt$ `;` $Stmt$	$[strict]$	$stmt$ `;` $stmt \rightarrow stmt$
$\mid$ `if` $BExp$ `then` $Stmt$ `else` $Stmt$	$[strict]$	`if` $bool$ `then` $stmt$ `else` $stmt \rightarrow stmt$
$\mid$ `while` $BExp$ `do` $Stmt$	$[strict]$	`while` $bool$ `do` $stmt \rightarrow stmt$
$\mid$ `output` $(AExp)$	$[strict]$	`output` $int \rightarrow stmt$
$\mid$ `halt`		`halt` $\rightarrow stmt$
$\mid$ `spawn` $(Stmt)$	$[strict]$	`spawn` $(stmt) \rightarrow stmt$
$Pgm ::=$ `vars` $\mathbf{List}\{VarId\}$ `;` $Stmt$		$\langle \underset{s \curvearrowright pgm}{\underline{\texttt{vars}\; xl \,;\, s}}\rangle_{\mathsf{k}} \; \langle \underset{xl}{\underline{\cdot}} \rangle_{\mathsf{vars}}$ $stmt \curvearrowright pgm \rightarrow pgm$

Figure 5.3: K type system for IMP++ (and IMP)

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 concurrency aspects of IMP++ are possible to define formally thanks to the specific nature of the K rules. If instead we used standard rewrite rules instead of K rules, than many of the concurrent steps above would need to be interleaved because rewrite rule instances which overlap cannot be applied concurrently.

5.2.3 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 a type checker for a desired typing policy, and then use it to discard as inappropriate programs that do not obey the desired typing policy. In this section we show how to define a type system for IMP/IMP++ using the very same K framework. The type system is defined like an (executable) semantics of the language, but one in the more abstract domain of types rather than in the concrete domain of integer and boolean values. The technique is general and has been used to define more complex type systems, such as higher-order polymorphic ones (see Section 7.15).

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.):

$$\begin{array}{rcl} K & ::= & \ldots \mid \mathit{int} \mid \mathit{bool} \mid \mathit{stmt} \mid \mathit{pgm} \\ \mathit{Configuration}^{\mathit{Type}}_{\mathrm{IMP++}} & \equiv & \langle\langle K\rangle_{\mathsf{k}}\ \langle \mathbf{List}\{\mathit{VarId}\}\rangle_{\mathsf{vars}}\rangle_{\mathsf{T}} \end{array}$$

Figure 5.3 shows the IMP/IMP++ type system as a K system over such configurations. Constants reduce to their types, and types are straightforwardly propagated through each language construct. Note that almost each language construct is strict now, because we want to type all its arguments in almost all cases in order to apply the typing policy of the construct. Two constructs make exception, namely the increment and the assignment. The typing policy of these constructs is that they take precisely a variable and not something that types to an integer. If we defined, e.g., the assignment strict and with rule "$\mathit{int} \texttt{:=} \mathit{int} \to \mathit{stmt}$", then our type system would allow ill-formed programs like "`x+y := 0`". Note how we defined the typing policy of programs "`vars` xl `;` s": the declared variables xl are stored into the $\langle\ldots\rangle_{\mathsf{vars}}$ cell (which is expected to initially be empty) and the statement is scheduled for typing (using a structural rule), placing a "reminder" in the computation that the *pgm* type is expected; once/if the statement is correctly typed, the type *pgm* is generated.

5.3 K in Rewrite Logic

In this section we discuss easy and intuitive ways to encode K systems as rewrite systems. Even though we have not defined the K rewrite abstract machine (Section 5.4) or the K technique (Section 5.5) in depth yet, the informal presentation of the K framework in Section 5.2 suffices to give our reader a reasonably good idea of what K is and how it works. The role of this section is to extend that informal understanding of K and give the reader a reasonably good idea of how she can write K definitions in rewrite logic and then use rewrite engines to execute and formally analyze such definitions. To a large extent, this section is equivalent in style and purpose to similar sections in Chapter 3 explaining how each of the conventional semantic approaches can be represented in rewrite logic and then executed using rewrite engines (e.g., Section 3.2.3, 3.3.3, 3.4.2, etc.).

Since K is a rewrite-based framework, it is conceptually closer to rewrite logic than any of the other semantic frameworks in Chapter 3. In spite of this closeness, however, there appears to be no immediate faithful embedding of K into rewrite logic. This is mainly because of two reasons:

1. Rewrite logic does not provide direct semantic concurrency support for subterm sharing, which has the effect that rewrite logic rules associated to K rules need to interleave even in situations where their original K rules could proceed truly concurrently in the K framework. As an example, suppose that a, a' and b are terms of sort S, that "a, a, b" is a term of sort $\mathbf{Bag\{}S\mathbf{\}}$, and that we have the following K rule

   $$\underset{a'}{\underline{a}}, b$$

   which rewrites a to a' in the presence of a (possibly shared) b. Then in K we can rewrite the term "a, a, b" in one concurrent step to "a', a', b" (see Section 5.4). This example pushes to its essence the practical situation where two threads (here simulated by each of the two a's) proceed concurrently when they only read (but do not change) the shared memory (here simulated by b). We are going to translate such a K rule into a rewrite rule

   $$a, b \to a', b$$

   which applies modulo the associativity and commutativity of the binary comma operation that implicitly constructs $\mathbf{Bag\{}S\mathbf{\}}$. This rewrite rule lacks the concurrency of the original K rule: no rule instances can overlap in rewrite logic, so two rule instances of the rule above matching the same b cannot proceed concurrently, they need to interleave. All our embeddings of K into rewrite logic share this limitation, the effect of which is that truly concurrent operations in the original K semantics become interleaved operations in the resulting rewrite logic theories. Therefore, we cannot obtain a practical concurrent-step-for-concurrent-step faithful embedding of K into rewrite logic (theoretically, one could eliminate subterm sharing by subterm copying via equations and multiset rewriting, as discussed in Section 5.4 and in [26], but that is impractical: hard or impossible to execute and to analyze formally, hard to read).

2. Rewrite logic does not provide support for structural rules (i.e., rules whose application does not count as computational). In K, computational rules apply "modulo" the structural ones. In other words, given a term t to rewrite using a K system, the structural rules can be used to derive a set of terms from t, each of those terms being thought of as a "computational representation" of t; then a computational rule can non-deterministically "pick" any term from the set and irreversibly rewrite it to another term. Then the process continues, i.e., the

structural rules generate a new set of terms, and so on. Even though this process is reminiscent of how rewrite rules apply in rewrite logic modulo equations, in K the equations are replaced by structural rules. Metaphorically, one can regard K's structural rules as "half-equations": they are used like the equations to compute classes of terms on which computational rules apply, but they are not necessarily reversible as the equations are. Indeed, one may have good reasons to not want the structural rules for sequential composition, `while` loops, `vars` declarations in the K semantics of IMP in Figure 5.1 to be reversible. In some sense, K's structural rules are more general than rewrite logic's equations, because one can achieve the same effect of an equation $t = t'$ by replacing it with a pair of structural rules $t \rightleftharpoons t'$. It appears impossible to achieve the desired "modulo" meaning of the structural K rules in rewrite logic.

Because of the reasons above, in this section we present our embeddings of K into rewrite logic at a rather conceptual level, discussing for each of them its advantages and disadvantages. The true concurrency and computational granularity aspects tend to be ignored by or unavailable in most conventional semantic frameworks. For example, all the approaches in Chapter 3 except for the chemical abstract machine (CHAM) assume an interleaving semantics, which means that the limitation of the true concurrency of K rewriting to that of rewrite logic is basically meaningless for those approaches, because they lack true concurrency by their very nature. The true concurrency limitation above is also meaningless for the CHAM, because, even though it advocates true concurrency, the CHAM rewriting does not allow for sharing (or, in other words, sharing yields interleaving for the involved rule instances). Therefore, most of the language designers using our embeddings of K into rewrite logic discussed in this section may not be affected much by the limitations above of the resulting rewrite logic theories. To avoid repetitively mentioning that the resulting rewrite logic theories lack the true concurrency of their original K definitions, in this section we temporarily restrict K's concurrency to the one of rewrite logic, that is, the explicit sharing specified in K rules plays no semantic role in this section.

5.3.1 (Almost Faithful) Embeddings of K into Rewrite Logic

As briefly discussed in Section 5.2 and in detail discussed in Section 5.4, there are two types of K rules: structural, whose role is to only rearrange the term without modifying its computational contents, and computational, whose role is to capture the irreversible computational steps. Notationally, the structural rules use dotted lines when written in two-dimensional form and $\rightharpoonup$ or $\rightharpoondown$ when written in unidimensional form, and the computational rules use full lines when written two-dimensionally and the usual rewrite relation notation $\rightarrow$ when written in unidimensional form. The unidimensional form is syntactic sugar for the particular case when the entire term is underlined (or rewritten), so we only discuss the more general, two-dimensional representations of the K rules.

All four embeddings discussed in this section, as well as all the other embeddings that we have experimented with but do not discuss here, translate K computational rules of the form

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

(p is called the local context, or pattern of the K rule) into corresponding rewrite rules of the form

$$p[l_1, l_2, ..., l_n] \rightarrow p[r_1, r_2, ..., r_n]$$

In order for the above to work, we need to make explicit the anonymous variables ("_") and to desugar the cell comprehension notation ("..."). For example, the assignment K rule in Figure 5.1

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

is translated into a rewrite rule like the one below

$$\langle X := I \curvearrowright \mathit{Rest}\rangle_{\mathsf{k}}\ \langle X \mapsto J\ \&\ \sigma\rangle_{\mathsf{state}} \to \langle \mathit{Rest}\rangle_{\mathsf{k}}\ \langle X \mapsto I\ \&\ \sigma\rangle_{\mathsf{state}}$$

where the variables have the following sorts: X has sort *VarId*; I, J have sort *Int*; *Rest* has sort K; and σ has sort *State*. To respect the well-established convention of rewrite logic, we used capital letters for variables; in K we prefer to use lower case letters for variables.

All four embeddings translate structural rules which are not reversible (i.e., structural rules which are not heating/cooling rules) precisely the same way, making therefore no distinction between such structural rules and computational rules in the translation. The only thing which is lost in this translation is the computational granularity of the original K definition (in addition to the true concurrency of the original K definition, which we decided to temporarily drop in this section).

What distinguishes the various embeddings of K into rewrite logic is how they represent the reversible structural rules (the heating/cooling rules). None of the embeddings below is perfect. The first two have a more theoretical relevance than practical, in that the resulting rewrite logic theories are not executable but they capture all the behaviors of the original K definition (albeit with a different computational granularity than that of the original K system). The third and fourth embeddings yield executable rewrite logic theories, but their executability comes at a loss of non-deterministic behaviors when executed on current rewrite logic engines. The reason for which we call our embeddings of K into rewrite logic "almost faithful" is twofold: first, they all miss some of the behaviors of the original K definition because of the reasons discussed in the preamble of Section 5.3; second, even though our third and fourth embeddings yield executable rewrite logic theories, they actually lose even more of the behaviors of the original K definition when executed.

First Impractical Embedding of K into Rewrite Logic

As mentioned, all our embeddings of K into rewrite logic translate both computational and non-reversible (i.e., non-heating/cooling) structural rules into rewrite rules. Our first embedding takes the simplest and most uniform approach to deal with the remaining reversible structural rules, namely to consider no difference between them and the other K rules, translating them all into rewrite rules as above. This way, each rewrite sequence in the original K system can be mirrored into a corresponding sequence in the corresponding rewrite theory and viceversa, except for the structural versus computational aspect. Therefore, one can use reachability analysis on the resulting rewrite system, e.g., the search capabilities of Maude, to find all the rewrite sequences that the original K system can yield. This reachability analysis can be very expensive and impractical, but it is theoretically important to understand that it is possible and that, indeed, the resulting rewrite system does not miss any of the behaviors of the original K system (assuming the temporarily accepted restrictions and limitations discussed in the preamble of Section 5.3).

There is a big practical problem, though, with this embedding, namely its execution capability. Consider for example a pair of structural (heating/cooling) rules

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

which, by this embedding, result into the following rewrite rules:

$$\begin{array}{rcl} a_1 + a_2 & \to & a_1 \curvearrowright \square + a_2 \\ a_1 \curvearrowright \square + a_2 & \to & a_1 + a_2 \end{array}$$

These two rules are inverse to each other so they most likely yield infinite rewriting when executed on rewrite engines. Therefore, the rewrite theories resulting from this embedding are not executable.

One could argue that the non-termination problem above is inherent in the original K definition as well, because there is nothing to stop one from applying the two structural (heating/cooling) rules above indefinitely. While this is true in principle, recall that in K computational rules apply modulo the structural rules; in particular, one would expect that practical implementations of K recognize such structural rules and handle them in a special manner. As an analogy, many practical implementations of conventional rewriting provide special support for rewriting *modulo* certain equational properties such as associativity or commutativity; indeed, like our heating/cooling rules above, commutativity would yield infinite rewriting if applied blindly as two rules inverse to each other. Also, like associativity and commutativity, the heating/cooling rules can only yield a finite number of computational representations of a given term, so an implementation can, again in principle, enumerate all of them in order to pick one on which a computational rule can apply.

Second Impractical Embedding of K into Rewrite Logic

As seen above, simply replacing each structural K rule by a rewrite logic rule yields non-termination whenever the original K definition includes any heating/cooling pair of structural rules. Since as discussed above each structural rule can be regarded as a "half-equation" in rewrite logic, it means that each pair of heating/cooling rules $l \rightleftharpoons r$ can be regarded as an equation $l = r$ in rewrite logic. Let us here assume an embedding transformation of K into rewrite logic which translates each pair of heating/cooling rules $l \rightleftharpoons r$ into an equation $l = r$ and each remaining structural K rule into a rewrite logic rewrite rule as if it was a computational K rule. Since in rewrite logic the rewrite rules apply modulo the equations, in theory none of the behaviors of the original K definition is lost.

Like the first embedding above, this embedding is also impractical: its resulting rewrite logic theories are not executable. Indeed, consider the equations

$$\begin{array}{l} a_1 + a_2 = a_1 \curvearrowright \square + a_2 \\ a_1 + a_2 = a_2 \curvearrowright a_1 + \square \end{array}$$

corresponding to the heating/cooling rules for IMP addition in Section 5.2.1, and consider an expression $7 + x$. In order to evaluate it one needs to first lookup x, and in order to do so one needs to heat the expression to $x \curvearrowright 7 + \square$ applying the second equation above. However, if the first equation is picked by a rewrite engine instead of the second, then the expression is heated to $7 \curvearrowright \square + x$ and now the rewrite process is stuck. Reachability analysis, e.g., using Maude's search command, does not work either, because only the rules are expected to generate new state-space, not the equations, so if the first equation is picked, then the second one will never be tried. The above is enough evidence that this embedding yields non-executable rewrite theories, so it is also impractical. It is interesting to note, as a side point, that the equations corresponding to the heating/cooling rules for non-deterministic constructs like the `+` above make the resulting rewrite theory non-coherent (see Section 2.7), thus also violating a basic theoretical executability requirement in rewrite logic.

First Practical Embedding of K into Rewrite Logic

One way to avoid the non-termination problem of the first embedding above is to restrict the applications of the rewrite rules corresponding to the heating/cooling K rules so that they can only apply in complementary situations. For example, it makes sense to apply the rewrite rule "$a_1 + a_2 \;\rightarrow\; a_1 \curvearrowright \square + a_2$" (corresponding to a heating rule) whenever a_1 is not a result, so it needs to be pulled out from its addition context to be further processed, and to apply the rewrite rule "$a_1 \curvearrowright \square + a_2 \;\rightarrow\; a_1 + a_2$" (corresponding to a cooling structural rule) whenever a_1 is a result, so it needs no further processing, meaning that it can be plugged back into its original addition context.

No matter how we decide to break the reversibility of the heating/cooling rules by splitting the cases in which only the first rule above applies from the cases in which only the second rule above applies, losing behaviors may be unavoidable. For example, suppose that we split the two cases as above, depending upon whether a_1 is a result or not, and suppose that a_1 is a not a result. Then the first rule applies and a_2 gets frozen until a_1 eventually reduces to an integer, when the second rule plugs it back into the addition context. Now, assuming that the original K definition also included a heating/cooling structural rule "$a_1 + a_2 \;\rightleftharpoons\; a_2 \curvearrowright a_1 + \square$" like the IMP language does (see Section 5.2.1), which is also translated into a pair of rewrite rules like the one for a_1, then the resulting rewrite theory loses those interleaved behaviors in which a_1 is reduced one step, then a_2 is reduced one step, then a_1 is again reduced one step, and so on; the remaining behaviors for `+` are then only those corresponding to non-deterministic choice: one of its arguments is non-deterministically chosen and evaluated completely, and then the other argument is evaluated completely (these behaviors are the same as those supported by the big-step semantics —Section 3.2).

In many practical situations, the loss of behaviors incurred when switching from fully non-deterministic to non-deterministic choice semantics is acceptable. As already mentioned in Section 3.1, one of the main reasons for which arithmetic language constructs like `+` are allowed to be non-deterministic is because one wants to allow flexibility in how the language is implemented, and not because these constructs are indeed intended to have fully non-deterministic behaviors. In other words, such constructs are in fact deliberately underspecified and one should not rely on their non-deterministic behavior when writing programs. If one is interested in a faithful rewrite logic semantics that captures even those rare and subtle behaviors that are visible under full non-deterministic but not under non-deterministic choice semantics of arithmetic language constructs, then one is referred to the other three rewrite logic embedding of K in this section. An alternative is to attempt to statically reject programs containing expressions that can yield such behaviors, the same way a type checker statically rejects programs that do not obey the intended typing policy.

Exercise* 175. *Define a special safety policy for* IMP++ *in K, following the type system style in Figure 5.3, which rejects as inappropriate programs containing expressions whose value depends upon the particular evaluation strategies of* `+`*,* `/` *and* `<=`*. The K definition of this safety policy should be executable, so that it results into a static analysis tool for this property when executed. For simplicity, the typing policy can be local, that is, should not take into account what other threads can do. Can one define a global policy, where other threads are allowed to potentially interfere, that rejects precisely those programs that violate the desired property and no other programs?*

Once we agree to ignore the loss of behaviors discussed above, we can define an embedding of K into rewrite logic that takes each pair of heating/cooling structural rules of the form

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

into a pair of potentially conditional[1] rewrite rules

$$
\begin{array}{lcll}
a_1 + a_2 & \rightarrow & a_1 \curvearrowright \square + a_2 & \text{when } a_1 \text{ is not a result} \\
a_1 \curvearrowright \square + a_2 & \rightarrow & a_1 + a_2 & \text{when } a_1 \text{ is a result}
\end{array}
$$

The embedding above is easy, mechanical and efficient. Indeed, the rewrite logic definition of IMP obtained by applying the transformation above to the K definition of IMP in Figure 5.1, when executed in Maude (see Section 5.3.2), yields an interpreter which is faster than any of the Maude interpreters of IMP corresponding to the conventional semantic approaches in Chapter 3 (an even faster interpreter is given by our second practical embedding of K into rewrite logic discussed next). However, our embedding transformation discussed above still has two limitations:

1. It modifies the computational granularity of the original K definition, because structural rules that do not count as computational steps in the original K definition now count as computational rewrite logic steps; and

2. It is rather inefficient when used for exhaustive analysis, e.g., for search or model checking, because one ends up having more rewrite rules like above corresponding to heating/cooling structural K rules than actual semantic rules (like those in the right columns of the K definitions of IMP and IMP++ in Figures 5.1 and 5.2), which blow the complexity of the exhaustive analysis tool. Recall from Section 2.7 that rewrite rules are assumed to potentially generate new behaviors, so their application generates the state-space analyzed by such tools, while equations are assumed to not generate new behaviors, so tools apply equations to canonize existing states but not to generate new states.

Second Practical Embedding of K into Rewrite Logic

We next describe our fourth and in our view best embedding of K into rewrite logic. It is a combination of the second impractical and the first practical embeddings above. More precisely, instead of transforming the heating/cooling structural rules into pairs of conditional rules as the first practical embedding above does, it transforms them into pairs of equations. For example, the heating/cooling pair "$a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$" for $+$ above yields the following two equations:

$$
\begin{array}{lcll}
a_1 + a_2 & = & a_1 \curvearrowright \square + a_2 & \text{when } a_1 \text{ is not a result} \\
a_1 \curvearrowright \square + a_2 & = & a_1 + a_2 & \text{when } a_1 \text{ is a result}
\end{array}
$$

From a (model- or proof-)theoretical point of view, since the two equations "$l = r$ when ..." and "$r = l$ when ..." associated to a heating/cooling pair of rules "$l \rightleftharpoons r$" have complementary conditions, they are completely equivalent to only one equation, namely "$l = r$". Therefore, from the same theoretical point of view, the embedding discussed here yields rewrite logic theories that are equivalent to the ones yielded by the second impractical embedding above, which means, in particular, that none of the behaviors of the original K definition is lost.

Like for the other three embeddings of K into rewriting logic discussed above, the problem with this transformation is also more of a practical rather than theoretical nature. Since the equations are expected to be confluent and to terminate when regarded as rewrite rules and since rewrite rules apply modulo equations, they are not considered as possible sources of non-determinism in

[1]One may be able to use subsorting of result and non-result computations into K and thus avoid the conditions.

current rewrite engines or formal analysis tools. That means that the rewrite theories generated by this new embedding, when executed, lose even more behaviors due to non-deterministic evaluation strategies than the previous embedding. Indeed, instead of non-deterministic choice semantics we now have an "arbitrary but fixed order" semantics: an arbitrary evaluation order is chosen, but one cannot explore any other evaluation order. Note that, in theory, the equations of a rewrite theory need not be confluent (nor terminate) when regarded as rewrite rules, but that in practice they are assumed so by the rewrite logic systems, in that their non-determinism is not explored. Current rewrite logic systems apply the equations as rewrite rules anyway when executing them (so from an executability point of view they are "half-equations", same as the K structural rules are intended to be), but, since they are not expected to generate new behaviors, less bookkeeping is needed for them, so they are more efficiently executable than the rewrite rules. Equations should therefore be preferred whenever possible (i.e., whenever they are sound) if efficiency is a concern.

5.3.2 K Semantics and Type System of IMP in Rewrite Logic

We next exemplify the second practical embedding above by completely defining both the K semantics and the type system of IMP (see Sections 5.2.1 and 5.2.3) in rewrite logic. Although the rewrite logic embeddings of K discussed above are conceptually straightforward, there are, however, several technical details that need to be addressed in order to make them work. We only focus on the last embedding above here, because, as mentioned, that is the most practical one.

First, we need to introduce the sort K for computations as a list sort with constructors "$\curvearrowright$" for concatenation of computations and "$\cdot$" for the empty computation, that is, with the notation for algebraic context-free grammars in Section 2.5, "$K ::= \mathbf{List}^{\cdot}_{\curvearrowright}\{K\}$". Further, as already mentioned in Section 5.2, K is a supersort for all the syntactic categories; in our case that means that we need to define the subsorts "$\mathit{AExp}, \mathit{BExp}, \mathit{Stmt}, \mathit{Pgm}, \mathbf{List}\{\mathit{VarId}\} < K$". Also, we need to define all the necessary computation freezers, e.g., "$\square + \_ : K \to K$", "$\_ + \square : K \to K$", etc., so that the heating/cooling equations parse. To state the conditions of the heating/cooling equations, we also need to define our result computations, namely the elements of sort $\mathit{KResult}$, where $\mathit{KResult} < K$. In the case of the K semantics of IMP, the results are the integer and the boolean values, so we add the subsortings "$\mathit{Int}, \mathit{Bool} < \mathit{KResult}$". In the case of the K type system of IMP, the results are the actual types, namely $\mathit{int}, \mathit{bool}, \mathit{stmt}, \mathit{pgm}$, which we define as constants of sort $\mathit{KResult}$.

All these are shown in Figures 5.4 and 5.5. The type system needs more freezers and more heating/cooling equations than the semantics, because the language constructs are strict in more arguments in the type system than in the semantics. Also, note that we took advantage of two rewrite-logic-specific features in the heating/cooling equations, namely: we used membership assertions in the conditions of the "heating" equations , e.g., "$K_1 : \mathit{KResult}$", and we used unconditional "cooling" equations but ones using variables of sort $\mathit{KResult}$. In rewriting logic we can assume that all the sorts of all the terms are dynamically computed and known at any moment (a term can have more than one sort, because of subsorting and operator overloading). The membership assertions allow one to dynamically check whether a term has a desired sort. If one does not want to rely on membership assertions and subsorting, e.g., if one's target engine does not support these, then one can alternatively define one's own membership predicate and make both equations conditional.

The next step is to define the cell-based configurations. Both the K semantics and the type system of IMP admit very simple configurations, consisting of a top cell that contains two subcells, the computation and either the state (in the semantics) or the list of variables (in the type system). These are defined by means of three operations, listed in the first raw of operations in Figures 5.4

sorts:

$K = \mathbf{List}^{\cdot}_{\curvearrowright}\{K\},\ \mathit{KResult},\ \mathit{Cell}$

subsorts:

$\mathit{AExp}, \mathit{BExp}, \mathit{Stmt}, \mathit{Pgm}, \mathbf{List}\{\mathit{VarId}\} < K \qquad \mathit{Int}, \mathit{Bool} < \mathit{KResult} < K$

operations:

$\langle\_\rangle_{\mathsf{T}} : \mathbf{Bag}\{\mathit{Cell}\} \to \mathit{Cell} \qquad \langle\_\rangle_{\mathsf{k}} : K \to \mathit{Cell} \qquad \langle\_\rangle_{\mathsf{state}} : \mathit{State} \to \mathit{Cell}$

$\square\,\texttt{+}\,\_ : K \to K \qquad \_\,\texttt{+}\,\square : K \to K$

$\square\,\texttt{/}\,\_ : K \to K \qquad \_\,\texttt{/}\,\square : K \to K$

$\square\,\texttt{<=}\,\_ : K \to K \qquad \_\,\texttt{<=}\,\square : K \to K$

$\square\,\texttt{and}\,\_ : K \to K$

$\texttt{not}\,\square : \to K$

$\_\,\texttt{:=}\,\square : K \to K$

$\texttt{if}\ \square\ \texttt{then}\,\_\,\texttt{else}\,\_ : K \times K \to K$

strictness equations: // mechanically derived

$K_1 \texttt{+} K_2 = K_1 \curvearrowright \square \texttt{+} K_2$ **if** $\neg_{Bool}(K_1 : \mathit{KResult}) \qquad R_1 \curvearrowright \square \texttt{+} K_2 = R_1 \texttt{+} K_2$

$K_1 \texttt{+} K_2 = K_2 \curvearrowright K_1 \texttt{+}\ \square$ **if** $\neg_{Bool}(K_2 : \mathit{KResult}) \qquad R_2 \curvearrowright K_1 \texttt{+} \square = K_1 \texttt{+} R_2$

$K_1 \texttt{/} K_2 = K_1 \curvearrowright \square \texttt{/} K_2$ **if** $\neg_{Bool}(K_1 : \mathit{KResult}) \qquad R_1 \curvearrowright \square \texttt{/} K_2 = R_1 \texttt{/} K_2$

$K_1 \texttt{/} K_2 = K_2 \curvearrowright K_1 \texttt{/}\ \square$ **if** $\neg_{Bool}(K_2 : \mathit{KResult}) \qquad R_2 \curvearrowright K_1 \texttt{/} \square = K_1 \texttt{/} R_2$

$K_1 \texttt{<=} K_2 = K_1 \curvearrowright \square \texttt{<=} K_2$ **if** $\neg_{Bool}(K_1 : \mathit{KResult}) \qquad R_1 \curvearrowright \square \texttt{<=} K_2 = R_1 \texttt{<=} K_2$

$R_1 \texttt{<=} K_2 = K_2 \curvearrowright R_1 \texttt{<=}\ \square$ **if** $\neg_{Bool}(K_2 : \mathit{KResult}) \qquad R_2 \curvearrowright R_1 \texttt{<=} \square = R_1 \texttt{<=} R_2$

$K_1\,\texttt{and}\,K_2 = K_1 \curvearrowright \square\,\texttt{and}\,K_2$ **if** $\neg_{Bool}(K_1 : \mathit{KResult}) \qquad R_1 \curvearrowright \square\,\texttt{and}\,K_2 = R_1\,\texttt{and}\,K_2$

$\texttt{not}\,K = K \curvearrowright \texttt{not}\ \square$ **if** $\neg_{Bool}(K : \mathit{KResult}) \qquad R \curvearrowright \texttt{not}\,\square = \texttt{not}\,R$

$K_1 \texttt{:=} K_2 = K_2 \curvearrowright K_1 \texttt{:=}\ \square$ **if** $\neg_{Bool}(K_2 : \mathit{KResult}) \qquad R_2 \curvearrowright K_1 \texttt{:=} \square = K_1 \texttt{:=} R_2$

$\texttt{if}\,K\,\texttt{then}\,K_1\,\texttt{else}\,K_2 = K \curvearrowright \texttt{if}\ \square\ \texttt{then}\,K_1\,\texttt{else}\,K_2$ **if** $\neg_{Bool}(K : \mathit{KResult})$

$R \curvearrowright \texttt{if}\ \square\ \texttt{then}\,K_1\,\texttt{else}\,K_2 = \texttt{if}\,K\,\texttt{then}\,K_1\,\texttt{else}\,K_2$

semanitc rules:

$\langle X \curvearrowright \mathit{Rest}\rangle_{\mathsf{k}}\ \langle X \mapsto I\ \&\ \sigma\rangle_{\mathsf{state}} \to \langle I \curvearrowright \mathit{Rest}\rangle_{\mathsf{k}}\ \langle X \mapsto I\ \&\ \sigma\rangle_{\mathsf{state}}$

$I_1 \texttt{+} I_2 \to I_1 +_{\mathit{Int}} I_2$

$I_1 \texttt{/} I_2 \to I_1 /_{\mathit{Int}} I_2$ **if** $I_2 \neq 0$

$I_1 \texttt{<=} I_2 \to I_1 \leq_{\mathit{Int}} I_2$

$\texttt{true and}\,B_2 \to B_2$

$\texttt{false and}\,B_2 \to \texttt{false}$

$\texttt{not true} \to \texttt{false}$

$\texttt{not false} \to \texttt{true}$

$\texttt{skip} \to \cdot$

$\langle X \texttt{:=} I \curvearrowright \mathit{Rest}\rangle_{\mathsf{k}}\ \langle X \mapsto J\ \&\ \sigma\rangle_{\mathsf{state}} \to \langle \mathit{Rest}\rangle_{\mathsf{k}}\ \langle X \mapsto I\ \&\ \sigma\rangle_{\mathsf{state}}$

$S_1 \texttt{;} S_2 \to S_1 \curvearrowright S_2$

$\texttt{if true then}\,S_1\,\texttt{else}\,S_2 \to S_1$

$\texttt{if false then}\,S_1\,\texttt{else}\,S_2 \to S_2$

$\langle\,\texttt{while}\,B\,\texttt{do}\,S \curvearrowright \mathit{Rest}\rangle_{\mathsf{k}} \to \langle\,\texttt{if}\,B\,\texttt{then}\,(S\ \texttt{;}\ \texttt{while}\,B\,\texttt{do}\,S)\,\texttt{else skip} \curvearrowright \mathit{Rest}\rangle_{\mathsf{k}}$

$\langle\,\texttt{vars}\,\mathit{Xl}\ \texttt{;}\ S\rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{state}} \to \langle S\rangle_{\mathsf{k}}\ \langle \mathit{Xl} \mapsto 0\rangle_{\mathsf{state}}$

Figure 5.4: Complete K semantics of IMP in rewrite logic (variables K, K_1, K_2 B, B_2, S, S_1, S_2, *Rest* have *kind* $[K]$, variables R, R_1 and R_2 have sort *KResult*, variable X has sort *VarId*, variable *Xl* has sort $\mathbf{List}\{\mathit{VarId}\}$, variable σ has sort *State*, and variables I, I_1, I_2, J have sort *Int*)

sorts:

$K = \mathbf{List}^{\cdot}_{\curvearrowright}\{K\}$, *KResult*, *Cell*

subsorts:

AExp, *BExp*, *Stmt*, *Pgm*, $\mathbf{List}\{$ *VarId* $\}$ $<$ K

operations:

$int, bool, stmt, pgm\ :\ \rightarrow KResult$ // result constants, corresponding to the types

$\langle\_\rangle_{\mathsf{T}} : \mathbf{Bag}\{Cell\} \rightarrow Cell$ $\quad\langle\_\rangle_{\mathsf{k}} : K \rightarrow Cell$ $\quad\langle\_\rangle_{\mathsf{vars}} : \mathbf{List}\{VarId\} \rightarrow Cell$

// all operations whose names contain $\square$ are mechanically derived from strictness attributes

$\square\texttt{+}\_ : K \rightarrow K$ $\qquad \_\texttt{+}\square : K \rightarrow K$

$\square\texttt{/}\_ : K \rightarrow K$ $\qquad \_\texttt{/}\square : K \rightarrow K$

$\square\texttt{<=}\_ : K \rightarrow K$ $\qquad \_\texttt{<=}\square : K \rightarrow K$

$\square\,\texttt{and}\,\_ : K \rightarrow K$ $\qquad \_\,\texttt{and}\,\square : K \rightarrow K$

$\texttt{not}\,\square : \rightarrow K$

$\_\texttt{:=}\square : K \rightarrow K$ $\qquad \texttt{if}\ \square\ \texttt{then}\,\_\,\texttt{else}\,\_ : K \times K \rightarrow K$

$\texttt{if}\,\_\,\texttt{then}\ \square\ \texttt{else}\,\_ : K \times K \rightarrow K$ $\qquad \texttt{if}\,\_\,\texttt{then}\,\_\,\texttt{else}\,\square : K \times K \rightarrow K$

$\texttt{while}\ \square\ \texttt{do}\,\_ : K \rightarrow K$ $\qquad \texttt{while}\,\_\,\texttt{do}\,\square : K \rightarrow K$

strictness equations:

// all equations below are mechanically derived from the strictness attributes

$K_1 \texttt{+} K_2 = K_1 \curvearrowright \square \texttt{+} K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\qquad R_1 \curvearrowright \square \texttt{+} K_2 = R_1 \texttt{+} K_2$

$K_1 \texttt{+} K_2 = K_2 \curvearrowright K_1 \texttt{+} \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\qquad R_2 \curvearrowright K_1 \texttt{+} \square = K_1 \texttt{+} R_2$

$K_1 \texttt{/} K_2 = K_1 \curvearrowright \square \texttt{/} K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\qquad R_1 \curvearrowright \square \texttt{/} K_2 = R_1 \texttt{/} K_2$

$K_1 \texttt{/} K_2 = K_2 \curvearrowright K_1 \texttt{/} \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\qquad R_2 \curvearrowright K_1 \texttt{/} \square = K_1 \texttt{/} R_2$

$K_1 \texttt{<=} K_2 = K_1 \curvearrowright \square \texttt{<=} K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\qquad R_1 \curvearrowright \square \texttt{<=} K_2 = R_1 \texttt{<=} K_2$

$K_1 \texttt{<=} K_2 = K_2 \curvearrowright K_1 \texttt{<=} \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\qquad R_2 \curvearrowright K_1 \texttt{<=} \square = K_1 \texttt{<=} R_2$

$K_1\,\texttt{and}\,K_2 = K_1 \curvearrowright \square\,\texttt{and}\,K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\qquad R_1 \curvearrowright \square\,\texttt{and}\,K_2 = R_1\,\texttt{and}\,K_2$

$K_1\,\texttt{and}\,K_2 = K_2 \curvearrowright K_1\,\texttt{and}\,\square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\qquad R_2 \curvearrowright K_1\,\texttt{and}\,\square = K_1\,\texttt{and}\,R_2$

$\texttt{not}\,K = K \curvearrowright \texttt{not}\,\square$ **if** $\neg_{Bool}(K : KResult)$ $\qquad R \curvearrowright \texttt{not}\,\square = \texttt{not}\,R$

$K_1 \texttt{:=} K_2 = K_2 \curvearrowright K_1 \texttt{:=} \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\qquad R_2 \curvearrowright K_1 \texttt{:=} \square = K_1 \texttt{:=} R_2$

$K_1 \texttt{;} K_2 = K_1 \curvearrowright \square \texttt{;} K_2$ **if** $\neg_{Bool}(K_1 : KResult)$ $\qquad R_1 \curvearrowright \square \texttt{;} K_2 = R_1 \texttt{;} K_2$

$K_1 \texttt{;} K_2 = K_2 \curvearrowright K_1 \texttt{;} \square$ **if** $\neg_{Bool}(K_2 : KResult)$ $\qquad R_2 \curvearrowright K_1 \texttt{;} \square = K_1 \texttt{;} R_2$

$\texttt{if}\,K\,\texttt{then}\,K_1\,\texttt{else}\,K_2 = K \curvearrowright \texttt{if}\ \square\ \texttt{then}\,K_1\,\texttt{else}\,K_2$ **if** $\neg_{Bool}(K : KResult)$

$R \curvearrowright \texttt{if}\ \square\ \texttt{then}\,K_1\,\texttt{else}\,K_2 = \texttt{if}\,K\,\texttt{then}\,K_1\,\texttt{else}\,K_2$

$\texttt{if}\,K\,\texttt{then}\,K_1\,\texttt{else}\,K_2 = K_1 \curvearrowright \texttt{if}\,K\,\texttt{then}\ \square\ \texttt{else}\,K_2$ **if** $\neg_{Bool}(K_1 : KResult)$

$R_1 \curvearrowright \texttt{if}\,K\,\texttt{then}\ \square\ \texttt{else}\,K_2 = \texttt{if}\,K\,\texttt{then}\,R_1\,\texttt{else}\,K_2$

$\texttt{if}\,K\,\texttt{then}\,K_1\,\texttt{else}\,K_2 = K_2 \curvearrowright \texttt{if}\,K\,\texttt{then}\,K_1\,\texttt{else}\ \square$ **if** $\neg_{Bool}(K_2 : KResult)$

$R_2 \curvearrowright \texttt{if}\,K\,\texttt{then}\,K_1\,\texttt{else}\,\square = \texttt{if}\,K\,\texttt{then}\,K_1\,\texttt{else}\,R_2$

$\texttt{while}\,K_1\,\texttt{do}\,K_2 = K_1 \curvearrowright \texttt{while}\ \square\ \texttt{do}\,K_2$ **if** $\neg_{Bool}(K_1 : KResult)$

$R_1 \curvearrowright \texttt{while}\ \square\ \texttt{do}\,K_2 = \texttt{while}\,R_1\,\texttt{do}\,K_2$

$\texttt{while}\,K_1\,\texttt{do}\,K_2 = K_2 \curvearrowright \texttt{while}\,K_1\,\texttt{do}\ \square$ **if** $\neg_{Bool}(K_2 : KResult)$

$R_2 \curvearrowright \texttt{while}\,K_1\,\texttt{do}\,\square = \texttt{while}\,K_1\,\texttt{do}\,R_2$

Figure 5.5: K computations, configurations, and strictness attributes for the definition of IMP's type system in rewrite logic; the remaining semantic rules and equations are given in Figure 5.6 (variables K, K_1, K_2 have *kind* $[K]$, and variables R, R_1 and R_2 have sort *KResult*)

semanitc rules:

$\langle X \curvearrowright \mathit{Rest}\rangle_{\mathsf{k}}\ \langle \mathit{Xl}, X, \mathit{Xl}'\rangle_{\mathsf{vars}} \to \langle \mathit{int} \curvearrowright \mathit{Rest}\rangle_{\mathsf{k}}\ \langle \mathit{Xl}, X, \mathit{Xl}'\rangle_{\mathsf{vars}}$
$\mathit{int} \mathbin{\texttt{+}} \mathit{int} \to \mathit{int}$
$\mathit{int} \mathbin{\texttt{/}} \mathit{int} \to \mathit{int}$
$\mathit{int} \mathbin{\texttt{<=}} \mathit{int} \to \mathit{int}$
$\mathit{bool}\ \texttt{and}\ \mathit{bool} \to \mathit{bool}$
$\texttt{not}\ \mathit{bool} \to \mathit{bool}$
$\texttt{skip} \to \mathit{stmt}$
$\langle X \mathbin{\texttt{:=}} \mathit{int} \curvearrowright \mathit{Rest}\rangle_{\mathsf{k}}\ \langle \mathit{Xl}, X, \mathit{Xl}'\rangle_{\mathsf{vars}} \to \langle \mathit{stmt} \curvearrowright \mathit{Rest}\rangle_{\mathsf{k}}\ \langle \mathit{Xl}, X, \mathit{Xl}'\rangle_{\mathsf{vars}}$
$\mathit{stmt}\ \texttt{;}\ \mathit{stmt} \to \mathit{stmt}$
$\texttt{if}\ \mathit{bool}\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt} \to \mathit{stmt}$
$\texttt{while}\ \mathit{bool}\ \texttt{do}\ \mathit{stmt} \to \mathit{stmt}$
$\langle\ \texttt{vars}\ \mathit{Xl}\ \texttt{;}\ S\rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{vars}} \to \langle S \curvearrowright \mathit{pgm}\rangle_{\mathsf{k}}\ \langle \mathit{Xl}\rangle_{\mathsf{vars}}$
$\mathit{stmt} \curvearrowright \mathit{pgm} \to \mathit{pgm}$

Figure 5.6: The semantic rules of the K definition of IMP's type system in rewrite logic (variable X has sort *VarId*, variables Xl and Xl' have sort **List{** *VarId* **}**, variable I has sort *Int*, and variables S and *Rest* have *kind* $[K]$)

and 5.5. The semantic rules in Figures 5.4 and 5.6 are straightforward: they are obtained by blindly applying the transformation discussed in the preamble of Section 5.3.1 to the corresponding K rules in Figures 5.1 and 5.3. Note that both the K computational rules and the non-reversible structural rules were translated into rewrite rules. The distinction between the two categories of K rules is therefore "lost in translation"; the resulting rewrite theories have finer-grained computational steps.

Note that the non-result K variables in Figures 5.4, 5.5, and 5.6 actually were assumed to have the *kind* $[K]$ and not the sort K. The reason is that the strictness equations corresponding to the heating structural rules can apply anywhere, including inside arguments of language constructs. When that happens, the respective arguments change their sort from their original language syntactic sort into K. Since the respective language construct expected the original syntactic sort which is subsorted to K and not K, the resulting term will therefore end up having the kind $[K]$. For more on the relationship between sorts, subsorts and kinds, the reader is referred to Section 2.7.

☆ K Semantics and Type System of IMP in Maude

Here we discuss the Maude representations of the rewrite theories above. The only notable difference between the next Maude modules and the rewrite theories above is that the various list and bag sorts, which were simply assumed above, need to be explicitly defined as associative and associative/commutative operations in Maude.

Figure 5.7 shows the Maude definition of the K computations, configurations and strictness attributes corresponding to the K rewrite logic definition of IMP in Figure 5.4. To distinguish the unit or empty computation "·" from other empty or unit constants, we follow our general convention in this book and write it "`.K`" (a dot followed by its sort). We follow the same convention for the empty cell, namely we write it "`.Bag{Cell}`". Note that, for the reason explained above, the variables `K`, `K1` and `K2` are declared to have the kind `[K]`. The Maude modules in Figure 5.7 are admittedly low level and boring to define. Indeed, the user of K-Maude (see Section 5.6) will never

```
mod IMP-K-COMPUTATIONS is including IMP-SYNTAX .
  sorts K KResult .  subsorts AExp BExp Stmt Pgm KResult List{VarId} < K .
  subsorts Int Bool < KResult .
  op .K : -> K .
  op _~>_ : K K -> K [assoc id: .K] .
endm

mod IMP-CONFIGURATION-K is including IMP-K-COMPUTATIONS + STATE .
  sorts Cell Bag{Cell} .  subsort Cell < Bag{Cell} .
  op .Bag{Cell} : -> Bag{Cell} .
  op __ : Bag{Cell} Bag{Cell} -> Bag{Cell} [assoc comm id: .Bag{Cell}] .

  op <T>_</T> : Bag{Cell} -> Cell .
  op <k>_</k> : K -> Cell .
  op <state>_</state> : State -> Cell .
endm

mod IMP-K-STRICTNESS is including IMP-K-COMPUTATIONS .
  var K K1 K2 : [K] .  var R R1 R2 : KResult .

  ops ([]+_) (_+[]) : K -> K .
 ceq K1 + K2 = K1 ~> [] + K2 if notBool(K1 :: KResult) .
  eq R1 ~> [] + K2 = R1 + K2 .
 ceq K1 + K2 = K2 ~> K1 + [] if notBool(K2 :: KResult) .
  eq R2 ~> K1 + [] = K1 + R2 .

  ops ([]/_) (_/[]) : K -> K .
 ceq K1 / K2 = K1 ~> [] / K2 if notBool(K1 :: KResult) .
  eq R1 ~> [] / K2 = R1 / K2 .
 ceq K1 / K2 = K2 ~> K1 / [] if notBool(K2 :: KResult) .
  eq R2 ~> K1 / [] = K1 / R2 .

  ops ([]<=_) (_<=[]) : K -> K .
 ceq K1 <= K2 = K1 ~> [] <= K2 if notBool(K1 :: KResult) .
  eq R1 ~> [] <= K2 = R1 <= K2 .
 ceq R1 <= K2 = K2 ~> R1 <= [] if notBool(K2 :: KResult) .
  eq R2 ~> R1 <= [] = R1 <= R2 .

  op []and_ : K -> K .
 ceq K1 and K2 = K1 ~> [] and K2 if notBool(K1 :: KResult) .
  eq R1 ~> [] and K2 = R1 and K2 .

  op not[] : -> K .
 ceq not K = K ~> not [] if notBool(K :: KResult) .
  eq R ~> not [] = not R .

  op _:=[] : K -> K .
 ceq K1 := K2 = K2 ~> K1 :=[] if notBool(K2 :: KResult) .
  eq R2 ~> K1 :=[] = K1 := R2 .

  op if[]then_else_ : K K -> K .
 ceq if K then K1 else K2 = K ~> if[]then K1 else K2 if notBool(K :: KResult) .
  eq R ~> if[]then K1 else K2 = if R then K1 else K2 .
endm
```

Figure 5.7: K computations, configurations and strictness attributes of IMP in Maude

define them; instead, she only adds strictness attributes to syntactic language constructs, like we did in Figure 5.1. Nevertheless, if one wants to write language definitions using the K semantic technique in plain Maude, without relying on any other tools (the same way we wrote Maude language definitions using various semantic techniques in Chapter 3), then, unfortunately, one has to manually define such low level operations and equations (similarly, recall that one had to manually define the infrastructure for splitting/plugging in RSEC in Section 3.5). When defining the strictness equations, one will most likely use cut-and-paste; one should be careful to replace all the symbols appropriately (e.g., the + into /, etc.), otherwise one's language may have hard to debug errors, such as performing an operation instead of another one (e.g., addition instead of division, etc.).

To test the strictness equations, one can ask Maude to rewrite various programs or fragments of program. For example, the rewrite command

```
Maude> rewrite if 3 <= (2 + x) / 7 then x := 3 / x else x := x / 7 ; y := x .
```

yields the following result:

```
rewrites: 45 in 0ms cpu (0ms real) (0 rewrites/second)
result [K]: x ~> 2 +[] ~> []/ 7 ~> 3 <=[]
                 ~> if[]then x ~> 3 /[] ~> x :=[] else x ~> []/ 7 ~> x :=[] ;
            x ~> y :=[]
```

Note that the strictness (heating) equations were applied everywhere they matched. As a result, the heated term does not have the sort *Stmt* anymore, not even the sort K, but the kind $[K]$. The reason is that the sequential composition construct, "`;`", now takes two terms of sort K instead of two terms of sort *Stmt*, so it cannot yield a well-sorted term. This is also the reason for which all the variables ranging over computations were and will continue to be defined to have kind `[K]` (as opposed to sort `K`), to accommodate the fact that the strictness equations can eagerly apply anywhere transforming syntactic terms into computations.

Once the low-level, mechanical and tedious K infrastructure and strictness equations are defined, the interesting Maude rules corresponding to the semantic K rules are natural and elegant. The module in Figure 5.8 contains all the Maude rewrite rules corresponding to actual K semantic rules in the definition of IMP. Like for all the semantic approaches in Chapter 3, Maude, through its rewriting capabilities, gives us an IMP interpreter by simply executing the semantics discussed above. For example, the Maude rewrite command

```
Maude> rewrite <T> <k> sumPgm </k> <state> .State </state> </T> .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are also irrelevant, so they were replaced by "..."):

```
rewrites: 6566 in ... cpu (... real) (... rewrites/second)
result Cell: <T> <k> .K </k> <state> n |-> 0 & s |-> 5050 </state> </T>
```

The resulting Maude interpreter is faster than any of the similar interpreters discussed in Chapter 3. We believe that the reason for the increased performance stays in the fact that the resulting rewrite logic rules are mostly unconditional, and unconditional rules generally outperform the conditional ones (to apply a conditional rule, the rewrite engine needs to stack the current rewrite context, start a new rewrite session to evaluate the condition, then pop the previous context, etc.). The reason for which the reduction semantics with evaluation contexts interpreters obtained in Section 3.5 are

```
mod IMP-SEMANTICS-K is including IMP-K-STRICTNESS + IMP-CONFIGURATION-K .
  var X : VarId .  var Xl : List{VarId} .  var Sigma : State .
  var I I1 I2 J : Int .  var B B2 S S1 S2 K Rest : [K] .
  rl <k> X ~> Rest </k> <state> X |-> I & Sigma </state>
  => <k> I ~> Rest </k> <state> X |-> I & Sigma </state> .
  rl I1 + I2 => I1 +Int I2 .
 crl I1 / I2 => I1 /Int I2 if I2 =/= 0 .
  rl I1 <= I2 => I1 <=Int I2 .
  rl true and B2 => B2 .
  rl false and B2 => false .
  rl not true => false .
  rl not false => true .
  rl skip => .K .
  rl <k> X := I ~> Rest </k> <state> X |-> J & Sigma </state>
  => <k>          Rest </k> <state> X |-> I & Sigma </state> .
  eq S1 ; S2 = S1 ~> S2 .
  rl if true  then S1 else S2 => S1 .
  rl if false then S1 else S2 => S2 .
  eq <k> (while B do S)                       ~> Rest </k>
   = <k> (if B then S ; while B do S else skip) ~> Rest </k> .
  eq <k> vars Xl ; S </k> <state> .State </state> = <k> S </k> <state> Xl |-> 0 </state> .
endm
```

Figure 5.8: K semantics of IMP in Maude

much slower, in spite of the fact that the third of them also had mostly unconditional semantic rules, is because their splitting/plugging mechanism had to be defined using conditional rules (actually very expensive ones, which enable full search in their conditions).

One can use any of the general-purpose tools provided by Maude on the K semantic definition above. For example, one can exhaustively search for all possible behaviors of a program:

```
Maude> search <T> <k> sumPgm </k> <state> .State </state> </T> =>! Cfg:Cell .
```

As expected, only one behavior will be discovered because our IMP language so far is deterministic.

Exercise 176. *Modify the Maude code in Figures 5.7 and 5.8 so that `/` short-circuits when its numerator evaluates to* 0 *(see also Exercises 54, 58, 60, 65, 70, and 75).*

Exercise 177. *Modify the Maude code in Figures 5.7 and 5.8 so that conjunction is not short-circuited anymore but, instead, is non-deterministically strict in both its arguments (see also Exercises 55, 59, 61, 66, 71, and 76).*

Figures 5.9 and 5.10 are the Maude versions of the rewrite logic theories in Figures 5.5 and 5.6 corresponding to the K type system of IMP. The module including the strictness equations in Figure 5.9 is larger than shown, because it also includes the strictness equations for the constructs `+`, `/`, `and`, `not`, which are identical to the ones in the similar module in Figure 5.7. The typing rules in Figure 5.10 are a blind Maude representation of those in Figure 5.6 and are self-explanatory.

```
mod IMP-K-COMPUTATIONS is including IMP-SYNTAX .
  sorts K KResult .  subsorts AExp BExp Stmt Pgm KResult List{VarId} < K .
  ops int bool stmt pgm : -> KResult .
  op .K : -> K .
  op _~>_ : K K -> K [assoc id: .K] .
endm

mod IMP-CONFIGURATION-K is including IMP-K-COMPUTATIONS + STATE .
  sorts Cell Bag{Cell} .  subsort Cell < Bag{Cell} .
  op .Bag{Cell} : -> Bag{Cell} .
  op __ : Bag{Cell} Bag{Cell} -> Bag{Cell} [assoc comm id: .Bag{Cell}] .

  op <T>_</T> : Bag{Cell} -> Cell .
  op <k>_</k> : K -> Cell .
  op <vars>_</vars> : List{VarId} -> Cell .
endm

mod IMP-K-STRICTNESS is including IMP-K-COMPUTATIONS .
  var K K1 K2 : [K] .  var R R1 R2 : KResult .

---
--- strictness equations for +, /, and, not are the same as for the semantics
---

  ops ([]<=_) (_<=[]) : K -> K .
 ceq K1 <= K2 = K1 ~> [] <= K2 if notBool(K1 :: KResult) .
  eq R1 ~> [] <= K2 = R1 <= K2 .
 ceq K1 <= K2 = K2 ~> K1 <= [] if notBool(K2 :: KResult) .
  eq R2 ~> K1 <= [] = K1 <= R2 .

  ops ([];_) (_;[]) : K -> K .
 ceq K1 ; K2 = K1 ~> [] ; K2 if notBool(K1 :: KResult) .
  eq R1 ~> [] ; K2 = R1 ; K2 .
 ceq K1 ; K2 = K2 ~> K1 ; [] if notBool(K2 :: KResult) .
  eq R2 ~> K1 ; [] = K1 ; R2 .

  ops (if[]then_else_) (if_then[]else_) (if_then_else[]) : K K -> K .
 ceq if K then K1 else K2 = K ~> if[]then K1 else K2 if notBool(K :: KResult) .
  eq R ~> if[]then K1 else K2 = if R then K1 else K2 .
 ceq if K then K1 else K2 = K1 ~> if K then [] else K2 if notBool(K1 :: KResult) .
  eq R1 ~> if K then [] else K2 = if K then R1 else K2 .
 ceq if K then K1 else K2 = K2 ~> if K then K1 else [] if notBool(K2 :: KResult) .
  eq R2 ~> if K then K1 else [] = if K then K1 else R2 .

  ops (while[]do_) (while_do[]) : K -> K .
 ceq while K1 do K2 = K1 ~> while [] do K2 if notBool(K1 :: KResult) .
  eq R1 ~> while [] do K2 = while R1 do K2 .
 ceq while K1 do K2 = K2 ~> while K1 do [] if notBool(K2 :: KResult) .
  eq R2 ~> while K1 do [] = while K1 do R2 .
endm
```

Figure 5.9: K computations, configurations and strictness attributes for the definition of the type system of IMP in Maude; strictness equations for `+`, `/`, `and`, `not` same as in Figure 5.7

```
mod IMP-TYPE-SYSTEM-K is including IMP-K-STRICTNESS + IMP-CONFIGURATION-K .
  var X : VarId .  var Xl Xl' : List{VarId} .  var I : Int .  var S Rest : [K] .
  rl I => int .
  rl <k> X   ~> Rest </k> <vars> Xl, X, Xl' </vars>
  => <k> int ~> Rest </k> <vars> Xl, X, Xl' </vars> .
  rl int + int => int .
  rl int / int => int .
  rl int <= int => bool .
  rl bool and bool => bool .
  rl not bool => bool .
  rl skip => stmt .
  rl <k> X := int ~> Rest </k> <vars> Xl, X, Xl' </vars>
  => <k> stmt     ~> Rest </k> <vars> Xl, X, Xl' </vars> .
  rl stmt ; stmt => stmt .
  rl if bool then stmt else stmt => stmt .
  rl while bool do stmt => stmt .
  eq <k> vars Xl ; S </k> <vars> .List{VarId} </vars>
   = <k> S ~> pgm    </k> <vars> Xl           </vars> .
  rl stmt ~> pgm => pgm .
endm
```

Figure 5.10: K type system of IMP in Maude

5.5 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 its configurations, computations, or semantic rules. Each Q/A box captures the essence of the corresponding subsection *from a user perspective*. They will ease the understanding of how the various components fit together.

Like term rewriting and rewriting logic, the K concurrent rewrite abstract machine (KRAM) discussed in Section 5.4 can be used in various ways in various applications; in other words, the KRAM itself does not tell us *how* to define a programming language or calculus as a K system. In this section we present the *K technique*, which consists of a series of guidelines and notations that turn the KRAM or even plain term rewriting into an effective framework for defining programming languages or calculi. The development of the K technique has been driven by practical needs, and it is the result of our efforts to define various programming languages, paradigms, and calculi as rewrite or K systems. We would like to make two important observations before we proceed:

1. The K technique is *flexible* and *open-ended*. Our current guidelines and notations are convenient enough to define the range of languages, features and calculi that we considered so far. Some readers may, however, prefer different or new notations. As an analogy, recall that there are no rigid rules for how to write a configuration in SOS (see Sections 3.2 and 3.3): one may use the angle-bracket notation $\langle \mathit{code}, \mathit{state}, ...\rangle$, or the square bracket notation $[\mathit{code}, \mathit{state}, ...]$, or even the simple tuple notation $(\mathit{code}, \mathit{state}, ...)$; also, one may use a different (from comma) symbol to separate the various configuration ingredients and, even further, one could use writing conventions (such as the "state" or "exception" conventions in [35]) to simplify the writing of SOS definitions. Even though we believe that our notational conventions discussed in this section should be sufficient for any definitional task, we still encourage our reader to feel free to change our notations or propose new ones if needed to better fit one's needs or style. Nevertheless, our current prototype implementations of K rely on our current notation as described in this section; therefore, to use our tools one needs to obey our notation.

2. The K technique yields a *semantic definitional style*. As an analogy, no matter what notations one uses for configurations and other ingredients in SOS definitions (see item above), or even whether one uses rewriting logic (as we did in Chapter 3) 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 all the other definitional styles discussed in Chapter 3. Similarly, we expect that the K technique can be represented or implemented in various back-end computational frameworks. We prefer KRAM because we believe that it gives us the maximum of concurrency one can hope for in K definitions. However, if one is not sensitive to this true concurrency aspect or if one prefers a certain computational framework over anything else, then one can very well use the K technique in that framework. Indeed, the same way the various conventional language definitional styles become *definitional methodologies or styles* within rewriting logic as shown in Chapter 3, the K technique can also

be cast as a definitional methodology or style within other computational frameworks. In Section 5.3 we show how this can be done for rewriting logic and Maude, for example.

5.5.1 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 5.14, which is a cell term over the simple cell grammar described below. By defining the configuration you shoot three rabbits with one stone:

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

In K definitions, the programming language, calculus or system configuration is represented as a potentially *nested cell* structure. This is similar is spirit to how configurations are represented in chemical abstract machines (CHAMs; Section 3.6) or in membrane systems (P-systems; Section 9.7), 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 5.5.2), 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 (see Section 2.5), such as lists, sets, multisets and maps.

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

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

where *Sort* can be *any sort name*, including arbitrary list ($\mathbf{List\{}\mathit{Sort}\mathbf{\}}$), set ($\mathbf{Set\{}\mathit{Sort}\mathbf{\}}$), bag ($\mathbf{Bag\{}\mathit{Sort}\mathbf{\}}$) or map ($\mathbf{Map\{}\mathit{Sort_1} \mapsto \mathit{Sort_2}\mathbf{\}}$) sorts. Many K definitions share the cell labels $\top$ (which stays for "*top*") and k (which stays for "*computation*"). They are built-in in our implementation of K in Maude (Section 5.6), 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 in Sections 7.8 and 7.9. The cells with starred labels say that there could be multiple instances, or clones, of that cell. This multiplicity information is optional[2], but can be useful for context transforming (Section 5.5.4).

We have seen so far three K configurations, one for IMP (Section 5.2.1), one for IMP++ (Section 5.2.2) and one for their type system (Section 5.2.3); we recall all three of them below:

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

Notice that they all obey the general cell grammar above, that is, they are nested cell structures; the bottom cells only contain a sort and no other cells. As a more complex example, below is the K configuration of CHALLENGE (Section 5.7), an experimental language conceived to challenge and expose the limitations the various language definitional frameworks:

$$\begin{array}{rcl}
\mathit{Configuration}_{\mathrm{CHALLENGE}} & \equiv & \langle \mathit{Agents}_{\mathrm{CHALLENGE}}\ \langle \mathbf{List}\{\mathit{Int}\}\rangle_{\mathsf{output}}\ \mathit{Messages}_{\mathrm{CHALLENGE}}\ \langle \mathit{Nat}\rangle_{\mathsf{nextAgent}}\rangle_{\top} \\
\mathit{Agents}_{\mathrm{CHALLENGE}} & \equiv & \left\langle \left\langle \begin{array}{c} \mathit{Threads}_{\mathrm{CHALLENGE}}\ \langle \mathbf{Map}\{\mathit{Nat}\mapsto \mathit{Val}\}\rangle_{\mathsf{store}} \\ \langle \mathit{Nat}\rangle_{\mathsf{nextLoc}}\ \langle K\rangle_{\mathsf{aspect}}\ \langle \mathbf{Set}\{\mathit{Val}\}\rangle_{\mathsf{busy}} \\ \langle \mathit{Nat}\rangle_{\mathsf{me}}\ \langle \mathit{Nat}\rangle_{\mathsf{parent}}\ \langle \mathbf{Map}\{\mathit{Nat}\mapsto \mathit{Nat}\}\rangle_{\mathsf{ptr}} \end{array}\right\rangle_{\mathsf{agents}*}\right\rangle_{\mathsf{agents}} \\
\mathit{Threads}_{\mathrm{CHALLENGE}} & \equiv & \langle\langle\langle K\rangle_{\mathsf{k}}\ \langle \mathbf{Map}\{\mathit{VarId}\mapsto \mathit{Nat}\}\rangle_{\mathsf{env}}\ \langle \mathbf{Map}\{\mathit{Val}\mapsto \mathit{Nat}\}\rangle_{\mathsf{holds}}\rangle_{\mathsf{thread}*}\rangle_{\mathsf{theread}} \\
\mathit{Messages}_{\mathrm{CHALLENGE}} & \equiv & \langle\langle\langle \mathit{Nat}\rangle_{\mathsf{sender}}\ \langle \mathit{Nat}\rangle_{\mathsf{receiver}}\ \langle \mathit{Val}\rangle_{\mathsf{val}}\rangle_{\mathsf{message}*}\rangle_{\mathsf{messages}}
\end{array}$$

To make it more readable, we introduced some intuitive "macros" above, namely $\mathit{Agents}_{\mathrm{CHALLENGE}}$, $\mathit{Threads}_{\mathrm{CHALLENGE}}$, and $\mathit{Messages}_{\mathrm{CHALLENGE}}$. Figure 5.14 shows a graphical representation of this configuration, which was generated automatically by the K2LATEX component of our current implementation of K in Maude (Section 5.6). Note that the CHALLENGE configurations have six levels of cell-nesting and several cell labels are starred, meaning that there can be multiple instances of those cells. For example, the $\langle\rangle_{\mathsf{agents}}$ 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., a $\langle\rangle_{\mathsf{threads}}$ cell which can contain an arbitrary number of $\langle\rangle_{\mathsf{thread}}$ cells; each thread contains a local computation, a local environment and a number of resources (resources can be acquired multiple times by the same thread, so a map is needed). As one may expect, real life language definitions tend to employ rather complex configurations.

The advantage of representing configurations as nested cell-structures is that, like in MSOS (Section 3.4), 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 (Section 3.3). 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

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

semantic rules, one which can be consulted to quickly find out, for example, what kind of cells are available and where they can be found. Second, the configuration structure is the basis for *context transforming* (Section 5.5.4), which gives more modularity to K rules by allowing them to be reusable in language extensions that require changes in the structure of the configuration.

5.5.2 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 5.5.3), which need to be defined no matter what semantic framework you prefer. By doing so, you implicitly define the basic K computational infrastructure. In many cases you do not need to define any other computation constructs.
Q: *Do I need to understand in depth what computations are in order to use K?*
A: Not really. If you follow a purely syntactic definitional style mimicking reduction semantics with evaluation contexts (see Section 3.8.1) in K, then the only computations that that you will ever see in your rules are abstract syntax terms.
Q: *What is the benefit of using more complex (than abstract syntax) computations?*
A: K at its full strength. Many complex languages are very hard or impossible to define purely syntactically, while they admit elegant and natural definitions using proper K computations. For example, the CHALLENGE language in Section 5.7.

K takes a very abstract view of language syntax and, in theory, it is not concerned *at all* with parsing aspects[3]. More precisely, in K there is only one top-level sort[4] associated to all the language syntax, called K and staying for *computational structures* or *computations*, and terms t of sort K have the abstract syntax tree (AST) representation $l(t_1, ..., t_n)$, where l is some *K label* and t_1,...,t_n are terms of sort K, extended with the list (infix) construct "$\curvearrowright$", read "followed by" or "and then"; for example, if t_1, t_2, ..., t_n are computations then $t_1 \curvearrowright t_2 \curvearrowright \cdots \curvearrowright t_n$ is also a computation, namely the one *sequentializing* t_1, t_2, ..., t_n. All the original language constructs, including constants and program variables, as well as all the freezers (discussed below and also in Section 5.5.3), 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:

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

[3]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.

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

To make the distinction between the two kinds of lists of K terms clear, we explicitly wrote both their constructs and units: $\mathbf{List}^{()}_{\_,\_}\{K\}$ consists of comma-separated lists with "()" as unit, and $\mathbf{List}^{\cdot}_{\_\curvearrowright\_}\{K\}$ consists of $\curvearrowright$-separated lists with "$\cdot$" as unit (this algebraic CFG notation is described in detail in Section 2.5). We call "()" the *empty list of computations* and "$\cdot$" the *empty computation*. Since in algebraic CFG notation we can use parentheses for disambiguation, the following are all well-formed K computations whenever l is a K label and t, t_1, ..., t_n well-formed K terms: $l()$ is a computation consisting of label l applied to an empty list, $l(\cdot)$ is a computation consisting of label l applied to the empty computation "$\cdot$", $l(t)$ is a computation applying l to computation t, and $l(t_1, ..., t_n)$ is a computation consisting of label l applied to computations t_1, ..., t_n.

The $\mathbf{List}^{()}_{\_,\_}\{K\}$ 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:

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

We typically 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 [18] and followed by many other akin languages, where underscores in the name of an operation mark the places of its arguments. In addition to the language syntax, *KLabel* may include labels for semantic reasons; e.g., labels corresponding to semantic domain values which may have not been automatically included in the language syntax. We may call K *constants* those computations of the form $l()$, where l is a label (e.g., those corresponding to constants in the original syntax: $\mathsf{skip}()$, $\mathsf{true}()$, $1()$, $2()$, etc.).

It is convenient in many K definitions to distinguish syntactically between proper computations and computations which are finished. A similar phenomenon is common and well-accepted in the other definitional styles (see Chapter 3), which distinguish between proper expressions and values, for example. To make this distinction smooth, we add the *KResult* syntactic sub-category of K which is constructed using corresponding labels (all labels in *KResultLabel* are also in *KLabel*):

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

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

We take the liberty to write language or calculus syntax either in AST form, like in "$\lambda\_.\_(x, e)$" and "$\mathsf{if\_then\_else\_}(b, s_1, s_2)$", or in more readable mixfix form, "$\lambda x.e$" and "$\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$". In our Maude implementation of K (see Section 5.6), thanks to Maude's builtin support for mixfix notation and corresponding parsing capabilities, we actually write programs using the mixfix notation. Even though theoretically unnecessary, this is actually very convenient in practice, because it makes language definitions more readable and, consequently, less error-prone. Additionally, programs in the defined languages can be regarded as terms the way they are, without any intermediate AST representation for them. In other implementations of K, one may need to use an explicit parser or to get used to reading syntax in AST representation. Either way, from here on we assume that programs, or fragment of programs, parse as computations in K.

The $\mathbf{List}^{\cdot}_{\_\curvearrowright\_}\{K\}$ 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 language semantics 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. The following are examples of computations making use of the $\mathbf{List}^{\cdot}_{\_\curvearrowright\_}\{K\}$ structure of K (parentheses disambiguate):

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

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

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

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 (Section 5.6), all these are generated automatically as constants of sort *KLabel* after a simple analysis of the language syntax.

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

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

Additionally, to increase readability, we take the freedom to generalize the adopted mixfix notation in K and "plug" the remaining computations in the freezer, that is, we write $c[\square, k_1, ..., k_n]$ instead of $c[\square, \_, ..., \_](k_1, ..., k_n)$. For instance, if $\_@\_$ is some binary operation and if, for some reason, in contexts of the form $(e_1@e_2)@(e_3@e_4)$ one wishes to freeze e_1, e_3 and e_4 (in order to, e.g., process e_2), then, when there is no confusion, one may write $(e_1@\square)@(e_3@e_4)$ instead of $((\_@\square)@(\_@\_))(e_1, e_3, e_4)$. This convention is particularly useful when one wants to follow a reduction semantics with evaluation contexts style in K, because one can mechanically associate such a freezer to each context-defining production. For example, the freezer $(\_@\square)@(\_@\_)$ above would be associated to a production of the form "$Cxt ::= (Exp@Cxt)@(Exp@Exp)$"; see Section 9.4 for more details on how reduction semantics with evaluation contexts is faithfully embedded in K.

5.5.3 K Rules: Computational and Structural

Q/A

Q: *How are the K rules different from conventional rewrite rules?*
A: The K framework builds upon the K concurrent rewrite abstract machine (Kram); how the Kram rules differ from standard rules is explained in Section 5.4.
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 (see Section 5.6) desugars the K rules into ordinary rules and equations anyway.

The K technique aims, among other things, at maximizing the potential for concurrency in the defined languages. Similarly, as discussed in Section 5.4, the concurrent rewrite abstract machine (Kram) also aims at maximizing the potential for concurrency, but for rewriting. Therefore, it is natural that the rewrite-based K framework employs the Kram as a rewriting infrastructure.

Recall from Section 5.4 that Kram provides two kinds of rules, computational and structural:

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

They both consist of a local context, or pattern, p, with some of its subterms underlined and rewritten to corresponding subterms underneath the line. The idea is that the underlined subterms represent the "write-only" part of the rule, while the operations in p which are not underlined represent the "read-only" part of the rule and can be shared by concurrent rule instances. The difference between computational and structural rules is that rewrite steps using the latter do not count as computational steps, their role being to rearrange the structure of the term to rewrite so that computational rules can match and apply. In general, there are no rigid requirements on when a K semantic rule should be computational versus structural. While in most cases the distinction between the two is quite natural, there are situations where one needs to subjectively choose one or the other; for example, we chose the rule for variable declarations in the IMP semantics in Figure 5.1 to be structural, but we believe that some language designers may prefer it to be computational.

Recall also from Section 5.4 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 write-only part, namely the entire pattern, and no read-only part). There is not much to say about K rules in addition to what has already been said in Sections 5.2 and 5.4. We would like to only elaborate on the heating/cooling rules and their corresponding strictness attributes.

Heating/Cooling Structural Rules

Q/A

Q: *What is the role of the heating/cooling rules?*
A: These are K's mechanism to define evaluation strategies of language constructs. They allow you to decompose fragments of programs into sequences of smaller computations, and to compose smaller computations back into fragments of programs.
Q: *Do I need to define such heating/cooling rules myself?*
A: Most likely no. It usually suffices to define *strictness attributes*, as discussed below; these are equivalent to defining evaluation contexts in reduction semantics (see Section 3.8.1). 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 5.2.1 that, in IMP, addition is intended to be non-deterministic). As discussed in Section 5.5.2, 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 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 5.1 shows several examples of strictness attributes. For example, the *strict* attribute of $\_ + \_$ is equivalent to the following two heating/cooling pairs of rules in K (a_1 and a_2 range over computations in K):

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

The symbol "$\rightleftharpoons$" is borrowed from the CHAM (Section 3.6), as a shorthand for combinations of a heating rule ("$\rightharpoonup$") and a cooling rule ("$\leftharpoondown$"). Indeed, one can think of the first rule above as follows: to process $a_1 + a_2$, let us first "heat" a_1, applying the rule from left to right; once a_1 is processed (using other rules in the semantics) producing some result, place that result back into context via a "cooling" step, applying the rule from right to left. However, it is important to realize that these heating/cooling rules can be applied at any moment and in any direction, because they are regarded not as computational steps but as structural rearrangements. For example, one can use the heating/cooling rules for "$\_ + \_$" above to pick and pass in front either a_1 or a_2, then rewrite it one step into a_1' using a computational rule, then plug a_1' back into the sum via cooling, then pick and pass in front either a_1' or a_2 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 $c[\square, N_1, ..., N_n]$, where N_1, ..., N_n are the various syntactic categories involved (or non-terminals in the CFG of the language), is to define a *KLabel* freezer $c[\square, \_, ..., \_]$ like discussed in Section 5.5.2 together with a pair of heating/cooling structural rules "$c[k, k_1, ..., k_n] \rightleftharpoons k \curvearrowright c[\square, k_1, ..., k_n]$".

One should be aware that in K "$\square$" is nothing but a symbol that we prefer to use as part of label names. In particular, "$\square$" is *not* a computation (recall that in reduction semantics with evaluation contexts "$\square$" 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:

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

What is wrong in the above "proof" is that one cannot apply the second rule in the second step above, because $\square + a_2$ is nothing but a convenient way to write the frozen computation $\square +\_(a_2)$. 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 aspect, which fortunately is invisible 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. This is called "computation adequacy" and is discussed in detail in Appendix B.1. The idea is to only allow heating and/or cooling of operator arguments that are proper syntactic computations (i.e., terms over the original syntax, i.e., different from "$\cdot$" and containing no "$\curvearrowright$"). With that, for example, the computation class of the expression $x * (y + 2)$ in the context of a language definition with non-deterministically strict binary $+$ and $*$, consists of the terms:

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

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

Strictness Attributes

In K definitions, one typically defines zero, one, or more heating/cooling rules per language construct, depending on its intended evaluation/processing strategy. These rules tend to be straightforward and boring to write, so in K we prefer a higher-level and more compact and intuitive approach: we annotate the language syntax with *strictness attributes*. A language construct annotated as *strict*, such as for example the "$\_ + \_$" in Figure 5.1, is automatically associated a heating/cooling pair of rules as above for each of its subexpressions. If an operator is intended to be strict in only some of its arguments, then the positions of the strict arguments are listed as arguments of the *strict* attribute. For example, note that the strictness attribute of $\mathsf{if\_then\_else\_}$ in Figure 5.1 is *strict*(1); that means that a heating/cooling equation is added only for the first subexpression of the conditional, namely the equation "$\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \rightleftharpoons b \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$".

The two pairs of heating/cooling rules corresponding to the strictness attribute *strict* of $\_ + \_$ above did not enforce any particular order in which the two subexpressions were processed. It is often the case that one wants a deterministic order in which the strict arguments of a language construct are processed, typically from left to right. Such an example is the relational operator $\_\mathsf{<=}\_$ in Figure 5.1, which was declared the strictness attribute *seqstrict*, saying that its subexpressions are processed deterministically, from left to right. The attribute *seqstrict* requires the definition of the syntactic category of result computations *KResult*, as discussed in Section 5.5.2, and it can be desugared automatically as follows: generate a heating/cooling pair of rules for each argument like in the case of *strict*, but requiring that all its previous arguments are in *KResult*. For example, the *seqstrict* attribute of $\_ \leq \_$ desugars into (a_1, a_2 range over K and r_1 over *KResult*):

$$a_1 \leq a_2 \rightleftharpoons a_1 \curvearrowright \square \leq a_2$$
$$r_1 \leq a_2 \rightleftharpoons a_2 \curvearrowright r_1 \leq \square$$

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

Our most general strictness declaration in K, also supported by our current implementation (Section 5.6), is to declare a certain syntactic context (a derived term) strict or sequentially strict in a certain list of arguments. For example, in the K definition of CHALLENGE in Section 5.7, we declare the context $* e := e'$ to be *strict*(e), with the meaning that the assignment statement applied to a pointer needs to first evaluate the pointer expression.

5.5.4 Context Transforming

We next introduce one of the most advanced feature of K, the *context transforming*, which gives K an additional degree of modularity. The process of context transforming is concerned with automatically modifying existing K rules according to the cell structure defined by the desired configuration of a target language. The benefit of context transforming is that it allows us to define semantic rules more abstractly, without worrying about the particular details of the concrete final language configuration. This way, it implicitly enhances the modularity and reuse of language definitions: existing rules do not need to change as the configuration of the language 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 5.5.1) is therefore a necesary step in order to make use of K's context transforming. Assuming that the various cell-labels forming the configuration are distinct, then one can use the structure of the configuration to *automatically* transform abstract rule contexts/patterns, i.e., ones that do not obey the intended cell-structure of the configuration, into concrete ones that are well-formed within the current configuration structure. This rule context transforming process can be thought of as being applied statically, before the K-system is executed.

Consider, for example, the K semantic rule for the output statement in IMP++ (Figure 5.2):

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

This rule says exactly what one wants the semantics of the output statement to be and as abstractly and compactly as possible: if "$\texttt{output}\,(i)$" is the next computational task, then append i to the end of the output buffer and dissolve the output statement. This rule perfectly matched the configuration structure of IMP++, because the IMP++ configuration structure was very simple: a top level cell containing all the other cells inside as simple, non-nested cells. Consider now defining a more complex language, like the CHALLENGE language in Section 5.7 whose configuration is shown in Figure 5.14. The particular cell arrangement in the CHALLENGE configuration makes the rule above directly inapplicable; to be precise, even though the rule context still parses as a *CellContents*-term, it will never match/apply when used in the context of the CHALLENGE configuration.

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

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

Thus, context transforming can be defined as the process of 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, volatile variables are used for the remaining parts of the introduced cells, so that other rule instances concerned with those parts of the cells can apply concurrently with the transformed rule.

The Locality Principle

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The Cell-Cloning Principle

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

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

Since this K rule captures the essence of the intended rendezvous synchronization, we would like to reuse it unchanged in language definitions which are more complex than IMP++, such as the Challenge language in 5.7. Unfortunately, this rule will never match/apply as is on Challenge configurations, because two $\langle\rangle_{\mathsf{k}}$ cells can never appear next to each other. A context transforming operation is therefore necessary, but it is not immediately clear how the rule context should be changed. The *cell-cloning principle* applies when abstract rules refer to two or more cells with the same name, and it states that context transforming should be consistent with the cell cloning, or multiplicity, information provided as part of the configuration definition; this can be done using starred labels, as explained in Section 5.5.1. Note that, for example, the Challenge configuration in Figure 5.14 declares both the agent and the thread cells clonable. Thus, using the cell-cloning principle in combination with the locality principle, the abstract rule above is transformed into the following Challenge-concrete rule:

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

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