University of Illinois at Urbana-Champaign
Department of Computer Science UIUCDCS-R-2007-2926 and College of Engineering UILU-ENG-2007-1827

K: A Rewriting-Based Framework for Computations — Preliminary version —

Grigore Rosu
Department of Computer Science
University of Illinois at Urbana-Champaign

Contents

Abstract

K is a definitional framework based on term rewriting, in which programming languages, calculi, as well as type systems or formal analysis tools can be defined making use of special list and/or set structures, called *cells*, which can be potentially nested. In addition to cells, K definitions contain *equations* capturing structural equivalences that do not count as computational steps, and *rewrite rules* capturing computational steps or irreversible transitions. Rewrite rules in K are *unconditional*, i.e., they need no computational premises (they are rule schemata and may have ordinary side conditions, though), and they are *context-insensitive*, so in K rewrite rules apply concurrently as soon as they match, without any contextual delay or restrictions.

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

K provides special support for list cells that carry "computational meaning", called *computations*. Computations are 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. Computation (structural) equations or *heating/cooling equations*, which technically are ordinary equations but which practically tend to have a special "computational" intuition (reason for which we use the "equality" symbol "$\rightleftharpoons$ " instead of "=" for them), allow to regard computations (finitely) many different, but completely equivalent ways. For example, in a language with an addition operation whose arguments can be evaluated in non-deterministic order, a computation $a_1 + a_2$ may also be regarded 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 $a_2 \curvearrowright a_1 + \square$. 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, so K can be naturally used to define concurrent languages or calculi. Structural equations can rearrange computations, so that rewrite rules can match and apply. Equations and rules can apply anywhere, in particular in the middle of computations, not only at their top. However, rules corresponding to inherently sequential operations (such as reads/writes of variables in the same thread) must be designed with care, to ensure that they are applied only at the top of computations.

K achieves, in one uniform framework, the benefits of both Chams and reduction semantics with evaluation contexts (context reduction), at the same time avoiding what may be called the "rigidity to chemistry" of the former and the "rigidity to syntax" of the latter. Any Cham and any context reduction definition can be captured in K with minimal (in our view *zero*) representational distance. K can support concurrent language definitions with either an interleaving or a true concurrency semantics. K definitions can be efficiently executed on existing rewrite engines, thus providing "interpreters for free" directly from formal language definitions. Additionally, general-purpose formal analysis techniques and tools developed for rewrite logic, such as state space exploration for safety violations or model-checking, give us corresponding techniques and tools for the defined languages, at no additional development cost.

1 Introduction

This paper is an informal introduction to K, a rewriting-based language definitional framework. By saying that K is a "rewriting-based" framework, in contrast to a "reduction-based" one, we mean that in K rules can be applied concurrently and unrestricted by context, following the basic intuitions and operational/semantical strengths of Meseguer's rewriting logic [20]. K was first introduced by the author in an algebraic form in the lecture notes of a programming language design course at the University of Illinois at Urbana-Champaign in Fall 2003 [32], as a means to define executable concurrent languages in rewriting logic using the Maude executable specification system [9]. A more formal, detailed algebraic description of K can be found in [33].

The major purpose of this report is to explain K using a non-algebraic, intuitive and conventional notation based on context-free grammars enriched with structural equivalences and rewrite rules, as opposed to algebraic signatures, algebraic specifications and rewriting logic theories. The reason for doing so is that we strongly believe that K can also be useful to the working programming language or calculi designer who may not like, may not be familiar with, or simply may not want to use the algebraic notation. An additional purpose for stripping K of its algebraic armor is to reflect the important fact that K is more than "a technique to implement interpreters in Maude"; it is a language definitional style, like structural operational semantics (SOS) [31], modular SOS (MSOS) [29], reduction semantics with evaluation contexts (context reduction) [11], or the chemical abstract machine (Cham) [5].

Figure 1 shows definitions of a simple imperative language and of some λ-calculi using both context reduction and K. For a fair comparison on the imperative language, we assumed that both definitions initiate the evaluation of a program p by wrapping it into double square brackets, $[\![p]\!]$; then each definition takes that and initiates the configuration in its own way. The syntax of the simple imperative language is:

$$\begin{array}{rcl}
Name & ::= & \text{standard identifiers} \\
Val & ::= & Int \\
AExp & ::= & Name \mid Val \mid AExp + AExp \\
BExp & ::= & \mathsf{true} \mid \mathsf{false} \mid AExp \leq AExp \mid BExp \ \mathsf{and}\ BExp \mid \mathsf{not}\ BExp \\
Stmt & ::= & Name := AExp \mid Stmt; Stmt \mid \mathsf{if}\ BExp\ \mathsf{then}\ Stmt\ \mathsf{else}\ Stmt \mid \mathsf{while}\ BExp\ \mathsf{do}\ Stmt \mid \mathsf{halt}\ AExp \\
Pgm & ::= & AExp \mid Stmt; AExp
\end{array}$$

To make it more interesting, we assumed that $\_+\_$ is strict but evaluates its arguments non-deterministically (i.e., it may interleave execution steps in its argument expressions), that $\_\leq\_$ is strict from left to right

Context reduction	K
A simple imperative language	
Configuration and initialization $Config ::= Int \mid [\![Pgm]\!] \mid [\![Pgm, State]\!]$ $[\![p]\!] \to [\![p, \emptyset]\!]$ $[\![v, \sigma]\!] \to v$ $Cxt ::= \square \mid [\![Cxt, State]\!]$	$KResult ::= Val$ $K ::= KResult \mid \mathsf{List}^{\cdot}_{\curvearrowright}[K]$ $Config ::= Int \mid [\![K]\!] \mid (\!\vert K\vert\!)_k \mid (\!\vert State\vert\!)_{state} \mid (\!\vert \mathsf{Set}[Config]\vert\!)_\top$ $[\![p]\!] = (\!\vert (\!\vert p\vert\!)_k\ (\!\vert \emptyset\vert\!)_{state}\vert\!)_\top$ $\langle\!\vert (\!\vert v\vert\!)_k \vert\!\rangle_\top = v$
Variable lookup $[\![c, \sigma]\!][x] \to [\![c, \sigma]\!][\sigma[x]]$	$(\!\vert \underset{\sigma[x]}{\underline{\ x\ }} \rangle_k\ (\!\vert \sigma\vert\!)_{state}$
Operators $Cxt ::= ... \mid Cxt + AExp$ $Cxt ::= ... \mid AExp + Cxt$ $i_1 + i_2 \to i_1 +_{Int} i_2$	$AExp + AExp$ $[strict\ extends\ \_ +_{Int} \_]$
$Cxt ::= ... \mid Cxt \leq AExp$ $Cxt ::= ... \mid Int \leq Cxt$ $i_1 \leq i_2 \to i_1 \leq_{Int} i_2$	$AExp \leq AExp$ $[seqstrict\ extends\ \_ \leq_{Int} \_]$
$Cxt ::= ... \mid \mathsf{not}\ Cxt$ $\mathsf{not}\ b \to \mathsf{not}_{Bool}\ b$	$\mathsf{not}\ BExp$ $[strict\ extends\ \mathsf{not}_{Bool}\_]$
$Cxt ::= ... \mid Cxt\ \mathsf{and}\ BExp$ $\mathsf{true}\ \mathsf{and}\ b \to b$ $\mathsf{false}\ \mathsf{and}\ b \to \mathsf{false}$	$BExp\ \mathsf{and}\ BExp$ $[strict(1)]$ $\mathsf{true}\ \mathsf{and}\ b \to b$ $\mathsf{false}\ \mathsf{and}\ b \to \mathsf{false}$
Statements $Stmt ::= ... \mid \mathsf{skip}$ $Cxt ::= ... \mid Name := Cxt$ $[\![c, \sigma]\!][x := v] \to [\![c, \sigma[v/x]]\!][\mathsf{skip}]$	 $Name := AExp$ $[strict(2)]$ $(\!\vert \underset{\cdot}{\underline{x := v}} \rangle_k\ (\!\vert \underset{\sigma[v/x]}{\underline{\ \ \sigma\ \ }} \vert\!)_{state}$
$Cxt ::= ... \mid Cxt; Stmt$ $\mathsf{skip}; s \to s$	$k_1; k_2 \rightleftharpoons k_1 \curvearrowright k_2$
$Cxt ::= ... \mid \mathsf{if}\ Cxt\ \mathsf{then}\ Stmt\ \mathsf{else}\ Stmt$ $\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \to s_1$ $\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \to s_2$	$\mathsf{if}\ BExp\ \mathsf{then}\ Stmt\ \mathsf{else}\ Stmt$ $[strict(1)]$ $\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \to s_1$ $\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \to s_2$
$\mathsf{while}\ b\ \mathsf{do}\ s \to \mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else\ skip}$	$(\!\vert \mathsf{while}\ b\ \mathsf{do}\ s \rangle_k = (\!\vert \mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else}\ \cdot \rangle_k$
$Cxt ::= ... \mid \mathsf{halt}\ Cxt$ $c[\mathsf{halt}\ i] \to i$	$\mathsf{halt}\ AExp$ $[strict]$ $(\!\vert \mathsf{halt}\ i \rangle_k \to (\!\vert i\vert\!)_k$
Programs $Cxt ::= ... \mid Cxt; AExp$ $\mathsf{skip}; a \to a$	
λ-calculus	
$Cxt ::= \lambda Name.Cxt \mid Cxt\ Exp \mid Exp\ Cxt$ $(\lambda x.e)\ e' \to e[e'/x]$	 $(\lambda x.e)\ e' \to e[e'/x]$
Call-by-value λ-calculus	
$Val ::= \lambda Name.Exp$ $Cxt ::= Cxt\ Exp \mid Exp\ Cxt$ $(\lambda x.e)\ v \to e[v/x]$	$Val ::= \lambda Name.Exp$ $Exp\ Exp$ $[strict]$ $(\!\vert (\lambda x.e)\ v \rangle_k \to (\!\vert e[v/x] \rangle_k$
Call-by-name λ-calculus	
$Cxt ::= Cxt\ Exp$ $(\lambda x.e)\ e' \to e[e'/x]$	$Exp\ Exp$ $[strict(1)]$ $(\!\vert (\lambda x.e)\ e' \rangle_k \to (\!\vert e[e'/x] \rangle_k$

Figure 1: A simple imperative language and λ-calculi defined using evaluation contexts and K ($p \in Pgm$, $v \in Val$, $c \in Cxt$, $x \in Name$, $\sigma \in State$, $i, i_1, i_2 \in Int$, $b \in Bool$, $s, s_1, s_2 \in Stmt$, $a \in AExp$; $e, e' \in Exp$)

(it first evaluates its first argument completely then its second), and that $\_\mathsf{and}\_$ shortcuts (it evaluates its first argument and it continues to evaluate the second only if the first is true). Only arithmetic expressions can be assigned, so in particular variables can take only integer values. Programs consist of a statement followed by an arithmetic expression, so they can only evaluate to an integer value. Halt also halts programs only with integer values. Booleans can be introduced in programs only as results of comparisons and can be used in conditionals and loops. This language is admittedly trivial and limited, but it serves our purpose of introducing K by comparison with context reduction. This language will be non-trivially modified and extended in Section 6. Note that even though we had to rewrite the syntactic constructs in the K definition in Figure 1 in order to augment them with strictness attributes[1], the K definition was still more compact than the context reduction one. For the sake of completeness, in both definitions we also included declarations of default items that appear in almost any language definition and so deserve to be implicit to the framework, such as the "$\Box$" in context reduction, and the configuration constructs $(\!|K|\!)_k$ and $(\!|\mathsf{Set}[Config]|\!)_\top$, respectively, in K. In general, K definitions tend to be more compact than their equivalent context reduction definitions; that holds true especially for more complex definitions of languages. As discussed in Section 5.4.6, context reduction definitions can be automatically converted into equivalent K definitions. However, K has several other advantages over context reduction that may make one consider using K directly at its full strength, instead of adopting a methodological fragment of it.

Since K makes use only of standard, context-insensitive term rewriting modulo equations, it can be executed on rewrite engines like Maude almost *as is*. Nevertheless, the relationship between K and Maude is like that between SOS, or any of the above-mentioned styles, and Maude (or ML, or Scheme, or Haskell, etc.): Maude can be used as a means to execute and analyze K language definitions. The interested reader is referred to [36] for a detailed discussion on how the various language definitional styles can be faithfully embedded in rewriting logic and then executed using systems like Maude, or even more conventional programming languages. The basic idea of these faithful embeddings is that a language definition in any of these styles, say $\mathcal{L}$, can be regarded as a rewrite logic theory, say $R_\mathcal{L}$, in such a way that there is a *one-to-one computational equivalence* between reductions using $\mathcal{L}$ and rewrites using $R_\mathcal{L}$. Note that this is significantly stronger than encoding, or implementing, a framework into another framework: *$R_\mathcal{L}$ is $\mathcal{L}$, not an encoding of it*, the only difference between the two being insignificant notational/syntactic conventions. This is totally different from encoding $\mathcal{L}$ on a Turing machine or in a λ-calculus, for example, because such encodings would not preserve the intended computational granularity of $\mathcal{L}$'s reductions (if correct, they would only preserve the "relational behavior" of $\mathcal{L}$: whatever is reduced with $\mathcal{L}$ can also be reduced, in any number of steps, with its encodings).

One can naturally ask the following question then:

> *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. The following important 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.?*

[1]In standalone K definitions, the language syntax is annotated with K attributes when the syntax is defined, not as part of the semantics as we did in Figure 1.

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 features to a language does not require modifying existing definitions of unrelated features? Modularity is crucial for scalability and reuse.*
5. *Is generic, that is, not tied to any particular programming language or paradigm?*
6. *Supports naturally non-determinism* and *concurrency?*
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 formal semantics, so one can also carry out proofs about programs? A fix-point, or an initial model, semantics is necessary if proofs by induction are desired.*

We believe that 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 desired features one at a time, temporarily or permanently declaring the others as "something else". We next list how, in our view, current practices and language definitional styles fail to satisfy the above-mentioned requirements.

1. To formally capture one's intuition about a language feature to be defined, one may use a big-step or a small-step SOS reduction semantics, 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, once one has concurrency and shared memory, one cannot have a big-step SOS definition of almost anything. 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. 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 definitional framework should have all the benefits of the "richer" framework, at no additional notational or logical complexity, yet 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 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. Some popular language definitional frameworks are incapable of defining even existing language features. In our view, 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 almost 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 fail to properly support existing common language features is concurrency: most frameworks enforce an interleaving semantics, which may not necessarily always be desired. Concurrency is further discussed in item 5 below. 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. As Mosses pointed out in [29], SOS is non-modular. Even in the original notes on SOS [31], Plotkin had to modify the definition of simple arithmetic expressions three times as his initial language evolved [29]. Even more frustratingly, to add an innocent abrupt termination statement to a language defined using SOS, say a halt, one would need 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 KOOL (which is a comparatively simple language) as shown in Figure 2. 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 [27], 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 [29], such conventions are not only adhoc and language specific, but may also lead to erroneous definitions. Mosses' Modular SOS [29] (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 5.3, MSOS still inherits all the remaining limitations of SOS.

5. 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.

6. 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 important and 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.

7. 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 [7]) and for reduction semantics with evaluation contexts (the PLT Redex tool, implemented by Findler and his collaborators in Scheme [19]). 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. While executability of language definitions is indispensable when designing non-trivial languages, one needs richer tool support when the language is concurrent. 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. Moreover, 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 analyses are also desirable as intrinsic components of an ideal language definitional framework.

9. 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. Most likely, proofs about programs or programming languages will need induction. In order for induction to be a valid proof principle, either it needs to be hardwired as one or more proof rules in the framework when the framework is proof-theoretical, or the framework must posses a fixed-point or an initial model semantics when it is model-theoretical.

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, sometimes many, of these ideal features (we discuss several such frameworks and their limitations in Section **??**). 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 should 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 was conducted aiming *explicitly* to fulfill all nine requirements discussed above.

K's actual merit is in addressing the first four requirements in our list above, namely it proposes a scalable, modular and comprehensive formal approach to define programming languages and language-related features and analyses. While the first four requirements are specific to programming languages, the remaining five, namely genericity, support for concurrency, executability, formal analysis and formal semantics, transcend the domain of programming languages. Indeed, the latter are desired features of system specification formalisms in general, an area of great interest to many researchers and, consequently, with a variety of approaches, methods, techniques and supporting tools. K's current approach is to avoid proposing a new general purpose system specification formalism, with a new semantics and with tool support developed from scratch (in other words, to avoid "reinventing the wheel"), but instead to build upon the vast existing theoretical and tool infrastructure of rewriting logic [20, 9]. This decision more than paid off. For example, we can carry out proofs about language definitions and corresponding type systems using the existing proof machinery of rewriting logic (see Section **??**). Also, our definition of a polymorphic type inferencer in K using Maude (see Section **??**) significantly outperforms the current type inferencer of SML and slightly that of Haskell, while slightly under-performs that of OCAML. Experiments performed with the K definition of Java 1.4 using Maude's builtin model checker, that translates into a model-checker for Java "for free", outperforms state-of-the art model checkers developed specifically for Java, such as Java PathFinder [10].

In this report, syntax is defined using only conventional and familiar context-free grammars, extended with lists and sets in a natural way. K is formally defined avoiding the algebraic notation, but its intuitions are borrowed from rewriting logic, an inherently algebraic meta-framework.

2 Term Rewriting and Rewriting Logic in a Nutshell

Term rewriting is a standard computational model supported by many systems and languages. Meseguer's rewriting logic [20], not to be confused with term rewriting, organizes term rewriting modulo equations as a logic with a complete proof system and initial model semantics. Meseguer and Rosu's rewriting logic semantics (RLS) [22, 23] aims at making rewriting logic a foundation for programming language semantics and analysis that unifies operational and algebraic denotational semantics.

A term rewrite system (TRS) consists of a set of uninterpreted operations, possibly over different syntactic categories, or sorts, and a set of rewrite rules of the form $l \rightarrow r$, where l and r are terms possibly containing variables with possibly multiple occurrences. Term rewriting is a method of computation that works by progressively changing (rewriting) a term, using a TRS. A rule can apply to the entire term being rewritten or to a subterm of the term. First, a match within the current term is found. This is done by finding a substitution, θ, from variables to terms such that the left-hand side of the rule, l, matches part or all of the current term when the variables in l are replaced according to the substitution. The matched subterm is then replaced by the result of applying the substitution to the right-hand side of the rule, r. Thus, the part of the current term matching $\theta(l)$ is replaced by $\theta(r)$. The rewriting process continues as long as it is possible to find a subterm, rule, and substitution such that $\theta(l)$ matches the subterm. When no matching subterms are found, the rewriting process terminates, with the final term, a normal form, being the result of the computation. One of the distinctive features of term rewriting, implicit in the discussion above, is that, unlike in reduction-based operational semantics, rules apply in a *context insensitive* manner; in other words, the larger context in which a rule is matched does not influence the applicability of the rewrite rule. Rewriting, like other methods of computation, can continue forever. The following is a rewrite definition of a factorial language construct in a hypothetical calculator language:

$$\begin{array}{l} \mathit{Exp} ::= \mathit{Nat} \mid \mathit{Exp} * \mathit{Exp} \mid \mathit{Exp} - \mathit{Exp} \mid ... \mid \mathit{Exp}\,! \\ 0! \rightarrow 1 \\ n! \rightarrow n * (n-1)!, \quad \text{when } n \geq 1 \end{array}$$

There are a large number of term rewriting engines, including ASF [38], Elan [6], Maude [9], OBJ [12], Stratego [39], and others, some of which are capable of executing several million rewrites per second. Rewriting is also a fundamental part of many existing functional languages and theorem provers. One of the major points of this paper is that rewriting logic definitions of languages or language features, if executed on efficient rewrite engines, can be as performant or even outperform implementations of interpreters or other tools in conventional programming languages. This point can be quickly reflected even on the simple calculator language above. For example, in our experiments executing the definition above, the factorial of 50,000, a number of 213,237 digits, was "calculated" in 18.620 seconds by Maude and in 19.280 and 16.770 seconds by the programming languages ML and Scheme, respectively. While for this particular example one may say that the quality of large integers libraries plays the major role, we obtained similar results for many other experiments, some of them mentioned below and in the rest of the paper.

In contrast to term rewriting, which is just a method of computation, rewriting logic is a *computational logic* proposed by Meseguer [20] as a unified logic for (true) concurrency, which builds upon equational logic by extending it with rewrite rules. In equational logic, a number of *sorts* (types) and *equations* are defined, specifying which terms are equal. Equal terms are members of the same equivalence class. In other words, equal terms are regarded as being identical; in particular, equations can be soundly applied either from left-to-right or from right-to-left when reasoning about equational or rewrite logic theories. Rewriting logic adds *rules* to equational logic, thought of as irreversible transitions: a rewrite theory is an equational theory extended with rewrite rules, where equations can be applied in any direction, while rules can only be applied from left-to-right. In other words, a *rewrite logic theory* consists of:

- *Syntax*, consisting of sorts (syntactic categories) and operators on them, that can be used to define well-formed "uninterpreted" terms; as already mentioned, we prefer to use equivalent CFGs to define syntax in this paper;

- *Equations*, defining structural identities on terms: equal terms can be substituted for each other unrestricted anywhere. Intuitively, equal terms are *the same thing represented differently*, the same way π, 3.1459..., or the circumference of a circle of diameter 1 are the same thing; and

- *Rewrite rules*, defining *irreversible transitions* between terms.

In this paper, we define languages and/or language features as rewrite logic theories: the syntax of the rewriting logic theory captures the syntax of the language possibly extended with auxiliary operators needed for semantic reasons, the equations capture structural rearrangements or equivalences of the program configuration of computation structures and thus carry no computational meaning, while rules capture the intended computational steps and can apply in parallel (provided that they do not overlap).

Both equations and rules can be parametric in rewriting logic, in the sense that they can contain variables that act as placeholders for subterms that *match* them. To avoid defining the details of rewriting logic, including substitutions and matching, in this paper by a "parametric" equation or rule we mean a recursively enumerable set of equations or rules, one per each parameter instance that satisfies the side conditions. In other words, equations and rules in this paper are regarded as *schematas* (when executing them on rewrite engines, one may need to capture side conditions using conditions in rules and/or equations). Unlike in reduction semantics with evaluation contexts [11], there are no context restrictions on applications of equations and/or rules (extensions of rewriting logic have been proposed recently allowing some forms of context-sensitivity, but we are not going to use those). This allows for rewriting logic to serve as a very simple and elegant foundation for concurrency, because rules and equations can indeed be applied concurrently. Rewriting logic admits a *complete proof system* and an *initial model semantics* [20] that makes inductive proofs rigorous and valid. The intuition for why rewriting logic is a unified logic for concurrency comes from two observations: (1) the rewriting process is inherently parallel, in the sense that non-overlapping parts of a term can be rewritten concurrently, and thus fits well with current trends in concurrent system architecture; and (2) as shown in [20, 18], many other models of concurrent computation can be represented within rewriting logic, capturing faithfully the intended true or interleaved concurrency of each model.

Rewriting logic is connected to term rewriting in that the latter can be used to "execute" theories in the former. Indeed, most of the equations $l = r$ can be transformed into term rewriting rules $l \to r$, thus providing a means to taking a rewriting logic theory, together with an initial term, and "executing" it using the underlying TRS. Some of the TRS steps therefore correspond to equations, so can be viewed as "structural transformations" rather than "computational steps". Any of the existing rewrite engines can be used for this purpose. It is important, though, not to confuse orientation of equations for executability purposes with rewrite rules in rewriting logic! As mentioned above, the actual semantics of a rewrite logic theory is that rewrite rules can apply concurrently on equivalence classes of terms obtained *modulo* all the equations! Not all equations can be oriented into rewrite rules for execution purposes; for example, an equation stating commutativity (C) of a binary operator cannot be regarded as a rewrite rule, because it would lead to non-termination; similarly, equations for associativity (A) or even ones defining identities, or units, of binary operators (I) may not always be desired to be regarded as (irreversible) rewrite rules. For that reason, some rewrite engines provide support for rewriting modulo special equations such as A, C, I.

An associative binary operator on a sort S can also be regarded as a list construct for S, an AI one as a list with empty-list construct, while an ACI one as a multiset construct; to obtain a set, one also needs to add an explicit idempotency equation in addition to the ACI equations. Since lists and (multi-)sets admit straightforward equational definitions in rewriting logic, from here on in this paper, we assume lists and (multi-)sets over any sort whenever needed; for a sort *Sort*, $\mathsf{List}[\mathit{Sort}]$ denotes comma-separated lists of terms of sort *Sort*, and $\mathsf{Set}[\mathit{Sort}]$ denotes white space separated (multi-)sets of terms of sort *Sort*. For all these structures, we use "$\cdot$" as unit (nil, empty, etc.). If one wants a different list or set separator, then one writes it as a subscript, while if one wants a different unit then one writes it as a superscript. For clarity, we take the liberty to occasionally mention as subscripts and/or superscripts even the default constructs and units for lists and sets. For example, $\mathsf{List}^{\cdot}_{\curvearrowright}[K]$ stays for $\curvearrowright$-separated lists of terms of sort K. We also assume, whenever needed, product sorts; these are written conventionally ($S_1 \times S_2$) and are also easy to define in rewriting logic.

We next show some examples of rewrite theories modulo equations (modulo A and AI, and modulo AC and ACI). First, let us consider two simple examples, showing also how intuitively rewriting logic and concurrency fit together. Suppose that one would like to define (comma-separated) sorted lists of integers. Then all one needs to do is to add the following "bubble sort" rule (to the AI equations for lists):

$$i, j \to j, i \quad \text{when } i > j$$

Here is an example of a nine-step rewriting logic derivation in the one-rule rewrite theory above (underlined subterms are redexes where the rule applies in the corresponding step):

$$\underline{7,6},\underline{5,4},3,\underline{2,1} \rightarrow 6,\underline{7,4},\underline{5,3},1,2 \rightarrow \underline{6,4},\underline{7,3},5,1,2 \rightarrow 4,\underline{6,3},7,\underline{5,1},2 \rightarrow \underline{4,3},6,\underline{7,1},\underline{5,2} \rightarrow$$
$$3,4,\underline{6,1},\underline{7,2},5 \rightarrow 3,\underline{4,1},\underline{6,2},\underline{7,5} \rightarrow \underline{3,1},\underline{4,2},\underline{6,5},7 \rightarrow 1,\underline{3,2},4,5,6,7 \rightarrow 1,2,3,4,5,6,7$$

Note that the rewrite rule is allowed to apply concurrently, but that it is not enforced to apply everywhere it can; for example, the rule was purposely not applied on the subterm "$5,1$" in the third step above. Also, note that many applications of the associativity equation for "$\_,\_$" were necessary in order to structurally change the term so that the rewrite rule could match and apply (sometimes multiple times). Note also that the applications of rules is non-deterministic and that one rule application choice may "disable" one or more other potential rule applications; for example, in the first step above one could have chosen the redex "$6,5$", which would have disabled the redexes "$7,6$" and "$5,4$". This "disabling" is, of course, just an informal way of saying that equations (such as associativity) can be applied many different and sometimes exclusive ways in order for the rules to match. It is obvious that the resulting rewrite system terminates (modulo AI), because the number of misplaced numbers decreases at each application of the rule, concurrent or not. It is also obvious that the normal-form lists are sorted, so the one-rule rewrite system above gives a correct sorting algorithm. What may be less obvious is that, when executed on a parallel rewrite engine with plenty of cores, this trivial rewrite system gives a parallel algorithm that can sort in worst-case linear time.

Consider now a simple game. A group of children get a large bag (an ACI operator, giving a multiset) with black and white balls, as well as arbitrarily many additional balls. Each child can remove two balls from the bag, whenever they want and concurrently, but then immediately put back in the bag a black ball if both extracted balls had the same color, or a white ball if they had different colors. Other children may dynamically join the game. We can define this problem as a rewrite theory as follows (b ranges over $Ball$):

$$\begin{array}{l} Ball ::= \circ \mid \bullet \\ Bag ::= \mathsf{Set}[Ball] \\ \circ\ \bullet \rightarrow \circ \\ b\ b \rightarrow \bullet \qquad \text{where } b \in \{\circ, \bullet\} \end{array}$$

It is obvious that as far as children keep playing the game it will eventually terminate, because at each extraction the number of balls decreases by at least one. The rules above can apply concurrently as far as there are at least two balls in the bag. What is not immediately obvious is that, despite the high degree of parallelism in this ACI rewrite system, it is actually confluent: the parity of the white balls does not change.

Let us now consider an extension of the simple calculator language above with ","-separated lists and with ";"-separated lists of lists (let us not worry about typing such programs for now) and with a permutation construct taking a number n and generating the list of permutation lists over the elements $\{1, 2, ..., n\}$ (here n, m range over natural numbers, p over ","-separated lists of natural numbers, and pl over ";"-separated lists of ","-separated lists of natural numbers):

$$\begin{array}{l} Exp ::= ... \mid \mathsf{List}_{,}[Exp] \mid \mathsf{List}_{;}[Exp] \mid \mathsf{perm}(Exp) \mid \mathsf{insert}(Exp, Exp) \mid \mathsf{mapcons}(Exp, Exp)) \\ \mathsf{perm}(1) \rightarrow 1 \\ \mathsf{perm}(n) \rightarrow \mathsf{insert}(n, \mathsf{perm}(n-1)) \qquad \text{when } n \geq 1 \\ \mathsf{insert}(n, (p; pl)) \rightarrow \mathsf{insert}(n, p); \mathsf{insert}(n, pl) \\ \mathsf{insert}(n, (m, p)) \rightarrow (n, m, p); \mathsf{mapcons}(m, \mathsf{insert}(n, p)) \\ \mathsf{insert}(n, \cdot) \rightarrow n \\ \mathsf{mapcons}(m, (p; pl)) \rightarrow \mathsf{mapcons}(m, p); \mathsf{mapcons}(m, pl) \\ \mathsf{mapcons}(m, p) \rightarrow (m, p) \end{array}$$

Note that the ","-separated lists above have an identity (i.e., it is an AI operator), "$\cdot$", while the ";"-separated ones have no identity (only A). Also, note that the second occurrence of "1" in the first rule can and should be thought of as a degenerated ";"-separated list of ","-separated lists; indeed, thanks to matching modulo identity (I), the fourth rule also applies when p is $\cdot$ (since (m,p) matches 1 with $m = 1$ and $p = \cdot$). In our experiments, the rewrite theory above, when executed in Maude, outperformed best implementations of permutation generation in ML and Scheme: Maude took 61 seconds to "calculate" permutations of 10, while ML and Scheme took 83 and 92 seconds, respectively. None of these systems were able to calculate permutations of 11. These simplistic experiments should by no means be considered conclusive; all we are trying to say is that the pragmatic, semantics-reluctant language designer, can safely regard the semantic programming language and related definitions in this paper also as efficient implementations, in addition to their conciseness and mathematical rigor.

To further exemplify the use of ACI operators and corresponding matching, let us now consider an extension of the simple calculator language above with partial maps (from natural numbers to expressions) defined by their graphs (i.e., as sets of pairs (n, e)), as well as update and lookup constructs:

$$\begin{array}{l}
\mathit{Exp} ::= \dots \mid \mathsf{Set}[\mathit{Nat} \times \mathit{Exp}] \mid \mathit{Exp}[\mathit{Nat}] \mid \mathit{Exp}[\mathit{Exp}/\mathit{Nat}] \\
e\ e = e \\
([n, e]\ g)[n] \to e \\
([n, e]\ g)[e'/n] \to [n, e']\ g \\
g[e'/n] \to [n, e']\ g \qquad \text{when } g \text{ contains no pair } [n, \_]
\end{array}$$

In the above, we assumed white-space-separated sets and the notation $[n, e]$ for pairs in the product sort $\mathit{Nat} \times \mathit{Exp}$. The equation assures that maps are indeed sets, not multi-sets (note that this set requirement is not really necessary in this case, because if one constructs maps only with the provided update interface then one can show that maps contain no duplicate pairs. The first rule shows a first example of ACI matching: note that n appears both in the map and as the lookup index, and that g can also be "$\cdot$". In theory, many applications of associativity and commutativity equations may take place in the background to "bring" the pair $[n, e]$ on the first position in the set. In practice, ACI rewrite engines like Maude implement efficient data-structures and indexing to perform such matches in essentially logarithmic time (in the size of the set). We here did not bother to produce an error if the map was not defined in n; if g has no pair of the form $[n, \_]$, then the term $g[n]$ is "stuck" and thus can be regarded as a "core dump" error message. On rewrite engines providing support for errors, one can define lookup as a potentially error generating operator; for example, in Maude one would define it to return an element of *kind* $[\mathit{Exp}]$ instead of sort *Exp*. The remaining two rules define the update operation; the first uses again ACI matching, while the second has a side condition. As mentioned above, in the theoretical developments of rewriting logic semantics and K in this paper we assume only unconditional rules and equations, though we adopt common conventions in logics and regard rules and equations as schematas, so they are allowed to have side conditions.

The simple calculator language above, with or without its extensions, is rather trivial and therefore cannot be used as an argument that rewriting logic can serve as a solid foundation for programming languages. Indeed, the main problem when defining non-trivial languages is that the applications of reductions needs to be somehow controlled, to ensure the correct evaluation flow in a program, and, moreover, applications of rules may need data that is not available locally, such as, e.g., values associated to locations in a shared, top-level store. Rewriting logic semantics (RLS), proposed by Meseguer and Rosu [22, 23], builds upon the strong belief, supported by extensive empirical evidence, that arbitrarily complex programming languages can, in fact, be easily defined as rewriting logic theories. By doing so, one gets essentially "for free" not only an interpreter and an initial model semantics for the defined language, but also a series of formal analysis tools obtained as instances of existing tools for rewriting logic.

As mentioned above, operationally speaking the major difference between conventional reduction semantics, with [11] or without [31] evaluation contexts, and rewriting logic semantics is that the former typically impose contextual restrictions on applications of reduction steps and the reduction steps happen one at a time, while the latter imposes no such restrictions. To avoid undesired applications of rewrite steps, one has to follow certain techniques and obey certain methodologies when using rewriting logic. In particular, as shown in [36], the more conventional language definitional styles (small-step and big-step SOS, reduction semantics with evaluation contexts, chemical abstract machines, continuation-based semantics, etc.) can be faithfully captured in rewriting logic by appropriate particular uses of its general reduction machinery. Consequently, one can define a language many different ways in rewriting logic, each with its advantages and disadvantages; indeed, the fact that one faithfully captures a small-step SOS definition as a rewrite logic theory does not mean that the resulting rewrite theory will give a true concurrency semantics to the original SOS definition: the adverb "faithfully", justified by theorems proved in [36], ensures that the corresponding rewrite theories have exactly the same strengths and limitations as the original definitions.

In this paper, we discuss the K rewriting logic technique, which was first introduced in the lecture notes of a programming language design course at the University of Illinois at Urbana-Champaign in Fall 2003 [32], as a means to define executable concurrent languages in rewriting logic using the Maude executable specification system [9]. K and its corresponding Maude tool support incrementally improved every year since then, as seen in the series of technical reports [33]. The latest of these reports also contains a more formal and detailed algebraic description of K. K is reminiscent of abstract state machines [13] and continuations [37], and glosses over language-irrelevant rewriting logic details. Analyzing the various faithful translations of language definitional styles into rewriting logic discussed in detail in [36], as well as their advantages and disadvantages when regarded as rewrite logic theories, we confidently believe that K captures and reflects best the strengths of rewriting logic as a semantic framework for programming languages and related features.

3 K in a Nutshell

K is a rewrite-based definitional framework in which programming languages, calculi, as well as type systems or formal analysis tools can be defined making use of special structures that carry "computational meaning", called *computations*. These are special terms "$T_1 \curvearrowright T_2 \curvearrowright \cdots \curvearrowright T_n$" having a nested list structure comprising "$\curvearrowright$"-separated *computational tasks* that are processed sequentially. Computations are typically derived from programs or fragments of programs via a "heating" mechanism; in particular, the original syntax of the programming language is "swallowed" as computation constructors. 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. In addition to computations, K definitions also contain *K-sentences*: *equations* capturing structural equivalences and *rewrite rules* capturing reduction steps (or irreversible transitions). K-sentences are *unconditional*, i.e., they need no computational premises (they are schematas and may have ordinary side conditions, though), and *context-insensitive*. Therefore, K-sentences may apply concurrently as soon as they match, without any delay or restrictions. A term may contain an arbitrary number of computations, so K can be naturally used to define concurrent languages or calculi. Equations can rearrange computations, so that rewrite rules can match and apply. K definitions can be efficiently executed on existing rewrite engines, thus providing "interpreters for free" directly from formal language definitions. Additionally, general-purpose formal analysis techniques and tools developed for rewrite logic, such as state space exploration for safety violations or model-checking, give us corresponding techniques and tools for the defined languages, at no additional development cost.

3.1 K Configurations: Nested Multisets

In K definitions, the program configuration is represented as a potentially nested "soup" (multiset) of *configuration item* terms, also called *configuration cells*, representing the current infrastructure needed to process the remaining program; these may include the current computation (e.g., continuation-like structure), environment, store, remaining input, output, analysis results, bookkeeping information, etc. The set of configuration cells is not fixed and is typically different from definition to definition. K assumes lists, sets and multisets over any sort whenever needed, using the notation discussed in Section 2. As discussed in Section 2, lists and sets admit straightforward equational definitions in rewriting logic (list = associative operator, multiset = associative and commutative operator, etc.). Formally, configurations have the following structure:

$$\begin{array}{rcll} \mathit{ConfigLabel} & ::= & \top \mid k \mid \mathit{env} \mid \mathit{store} \mid \ldots & \text{(descriptive names; first two common, rest differ with language)} \\ \mathit{Config} & ::= & (\!|\mathit{Sort}|\!)_{\mathit{ConfigLabel}} \mid \ldots & \end{array}$$

where *Sort* can be any sort, including set or list sorts such as $\mathsf{Set}[\mathit{Sort}]$ or $\mathsf{List}[\mathit{Sort}]$, or even $\mathsf{Set}[\mathit{Config}]$ or $\mathsf{List}[\mathit{Config}]$. Many K definitions share the configuration labels $\top$ (which stays for "*top*") and k (which

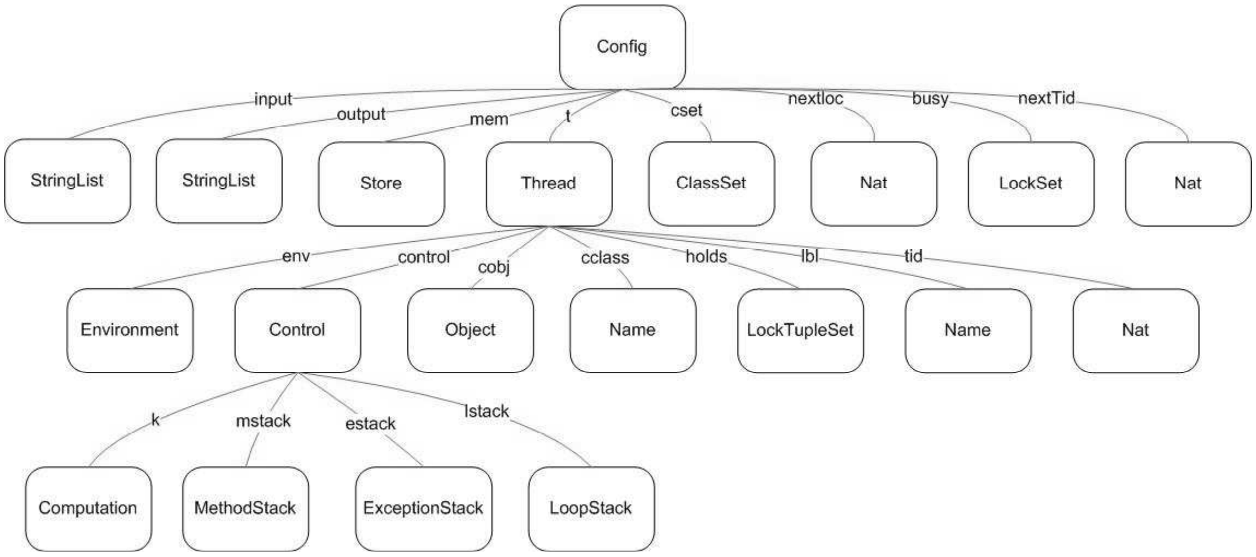


Figure 2: KOOL configuration "soup"

stays for "*computation*"). For that reason, in our implementation of K in Maude (see Appendix E), they are assumed builtin, so that one needs not declare them in each language definition. If in a particular language definition there is only one configuration label, like in the definition of CCS in Section 4.4, then we take the liberty to drop the label name all together. The configuration labels typically differ with the language or analysis technique to be defined in K. A configuration $(\!|c|\!)_l$ may also be called a *configuration item* (or *cell*) *named* (or *labeled*) l; interesting configuration cells are the nested ones, namely those where $c \in \mathsf{Set}[\mathit{Config}]$. One is also allowed to define some language-specific configuration constructs, to more elegantly initialize and terminate computations. A common such additional configuration construct, also declared as builtin in our implementation, is $[\![K]\!]$, which takes the given program to an initial configuration. An equation therefore needs to be given taking such special initializing configuration into an actual configuration cell; in most definitions, an equation identifies a term of the form $[\![p]\!]$ with one of the form $(\!|...(\!|p|\!)_k...|\!)_\top$, for some appropriate configuration items replacing the dots. Sometimes one may want to provide more info than just a program as part of the initial configuration, for example, an input buffer; then one needs to define appropriate configuration constructs, such as $[\![K, \mathit{input}]\!]$, and give corresponding equations, such as $[\![p, l]\!] = (\!|...(\!|p|\!)_k...(\!|l|\!)_{in}...|\!)_\top$, etc. If one's goal is to give a dynamic semantics of a language and if p is some terminating program, then $[\![p]\!]$ eventually produces (after a series of rewrites) the result of evaluating p; if one's goal is to analyze p, for example to type check it, then $[\![p]\!]$ eventually rewrites to the result of the analysis, for example a type when p is well-typed or an error term when p is not well-typed.

Figure 2 shows the nested "soup" configuration of KOOL [14, 15, 8], a medium-complexity experimental object-oriented language defined by Mark Hills using the K technique in Maude. A simplified variant of KOOL, called SKOOL, is discussed in Appendix D. We often display the configuration structure graphically as trees, like above. Non-leaf nodes in the tree represent (multiset) configurations containing as configuration items the incoming sub-configurations, each wrapped with a unique configuration label named as indicated on the corresponding edge. For example, in Figure 2, the top-level configuration contains 8 configuration items: input and output lists of strings, a store, a thread, a class set, a next available location, a set of busy locks, and a next available thread id. Each thread configuration contains 7 other configuration items: an environment, a control item comprising the various control information data (stacks) needed, a current object and class that the thread is executing code of, a set of pairs (lock,counter) containing all the locks that the thread holds together with their order of multiplicity, a label and a thread id. The control configuration item wraps a configuration of 4 additional items, each a stack containing computations possibly augmented with other resources: the first contains the main computation (remaining part of the program, a first-order continuation-like structure), a method stack, an exception stack, and a loop stack (loops can break and continue). Therefore, there are no less than 17 leaf configuration items, each containing important information about the state of the program. Moreover, 10 of them, those within threads, can have multiple instances, depending upon the number of threads that are alive at a given moment. Thus, the total number of leaf configuration items at any given moment is $7+n*10$, where n is the number of threads t that moment.

The advantage of representing configurations as nested "soups", is that, like in MSOS [29], subsequent semantic equations and rules will only need to mention those configuration items that are needed for those particular equations and rules, as opposed to having to mention the entire configuration, whether needed or not, like in conventional SOS. We can add or remove elements from the configuration multiset as we like, only impacting rules that use those particular configuration items. Rules that do not need the changed configurations items do not need to be touched. This is one important aspect of K's modularity.

There are two optional conventions/guidelines regarding the use of configurations that help increase the modularity of K language definitions. By convention, configuration labels must be distinct in the initial configuration, so that the configurations they wrap can be uniquely identified even when the exact paths to them are not completely specified. Also, equations and rules are allowed to add new configuration item instances (a necessary process when, for example, creating threads) only at the same nesting level as originally declared in the nested configuration. These conventions are crucial for K's *context transformation* process; essentially, equations and rules in a K definition are unambiguously and automatically modified by completing partial configuration contexts to "fit" the current configuration. This optional process, explained in detail in [33], gives an additional degree of modularity to K definitions. For example, suppose that initially KOOL was not concurrent and had no statements that can abruptly change the control, such as return of methods, exceptions, break/continue of loops, so there was no need for a multi-layered configuration; in other words, suppose that all configuration items were initially part of the top level configuration. Then the rule for variable lookup (defined also in Appendix D) would match the environment and the store at the same configuration level. Adding threads and control-changing statements to the language would, unfortunately, break the existing definition of variable lookup, because now each of the involved configuration items are located at different levels in the configuration, so the previous rule would not match anymore. With context transformers, no change needs to be done in the existing rule for variable lookup when the structure of the configuration changes. The correct environment will be matched (and not the environment of another thread), because the context transformers are resolved following a "greedy" depth-first strategy.

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

An important configuration item, present in many K definitions and "wrapped" in configurations with the *ConfigLabel* k, is the *computation*, denoted by the syntactic category K. Computations generalize abstract syntax trees (ASTs) by adding a special list construct $\_\curvearrowright\_$:

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

The first construct scheme for K abstractly captures any programming language syntax, under the form of abstract syntax trees, provided that one adds one *KLabel* for each language construct. In addition to the language syntax, one may include in *KLabel* additional labels for semantic reasons and for "heating/cooling" reasons (explained below). If one wants more K syntactic categories, then one can do that, too, but, for simplicity, we prefer to keep only one in this paper. It is often more intuitive to call the language constructs taking no arguments constants and write them as "leaves" in K structures; e.g., skip instead of skip(); we therefore may tacitly assume an additional syntactic subcategory of K called *KConstant*, for constructs such as skip, 0, 1, true, etc. Also, for executability reasons, it may be useful to define another syntactic subcategory of K, called *KResult*, for "finished computations"; this may contain certain constants, such as 0, 1, true, etc., but also computations which are proper terms, such as $\lambda\_.\_(x,e)$ (written in AST form).

We take the liberty to write syntax either in AST form, like in "$\lambda\_.\_(x,e)$" and "if_then_else_(b,s_1,s_2)", or in more readable mixfix form, "$\lambda x.e$" and "if b then s_1 else s_2". In our Maude implementation of K (see Appendix E), thanks to Maude's builtin mixfix notation and corresponding parsing for syntax, 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 as 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 following fragments of program in a hypothetical language are therefore terms of sort K,

$$\begin{array}{l} 1+2 \\ \mathsf{if\ true\ then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else}\ s' \end{array}$$

provided that all the corresponding constructs are defined as constructs for *KLabel* and *KConstant*:

$$\begin{array}{rcl} \mathit{KLabel} & ::= & ... \mid \_+\_ \mid \_;\_ \mid \mathsf{if\_then\_else\_} \mid \mathsf{while\_do\_} \\ \mathit{KConstant} & ::= & ... \mid \mathit{Bool} \mid \mathit{Nat} \end{array}$$

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

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

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

$$\mathit{KLabel} \quad ::= \quad ... \mid \_+\square \mid \square+\_ \mid \mathsf{if}\ \square\ \mathsf{then\_else\_}$$

The $\square$ carries the obvious "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 $\square + a_2$". Needless to say that the user of K is not expected to declare these special labels. We assume them whenever needed. In our implementation of K in Maude (Appendix E), 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 $c[k, k_1, ..., k_n]$ 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 can take the freedom to write $(e_1@\Box)@(e_3@e_4)$ instead of $((\_@\Box)@(\_@\_))(e_1, e_3, e_4)$. 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 $(\_@\Box)@(\_@\_)$ above would be associated to a production of the form "$\mathit{Cxt} ::= (\mathit{Exp}@\mathit{Cxt})@(\mathit{Exp}@\mathit{Exp})$"; see Section 5.4.6 for more details on how reduction semantics with evaluation contexts is captured by K.

3.3 K Heating/Cooling and Strictness Attributes

After defining the desired language syntax so that programs or fragments of programs become terms of sort K, the very first step towards giving a K semantics to a language is to define the *computation (structural) equations*. These allow to regard computations many different, but completely equivalent ways. For example, $a_1 + a_2$ may be regarded also as $a_1 \curvearrowright \Box + a_2$, with the intuition "schedule a_1 for processing and freeze a_2 in freezer $\Box + \_$", but also as $a_2 \curvearrowright a_1 + \Box$ (if, of course, addition is intended to be non-deterministic). As discussed in Section 3.2, the freezers' role is to store the remaining computations for future processing.

Heating/Cooling equations, informally. Computation structural equations know 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* computation structural equations can be generated automatically from K strictness operator attributes as explained below; Figure 1 shows several examples of strictness attributes. For example, the *strict* attribute of $\_ + \_$ is equivalent to the following two computation structural equations in K (a_1 and a_2 range over computations in K):

$$\begin{aligned} a_1 + a_2 &\rightleftharpoons a_1 \curvearrowright \Box + a_2 \\ a_1 + a_2 &\rightleftharpoons a_2 \curvearrowright a_1 + \Box \end{aligned}$$

To distinguish them from the other equations, we use the symbol "$\rightleftharpoons$" instead of "$=$" in computational structural equations. The symbol "$\rightleftharpoons$" is also used in Cham [5], as a shorthand for combinations of a heating rule ("$\rightharpoonup$") and a cooling rule ("$\leftharpoondown$"). Even though in K we use it for equations, which have a model-theoretical semantics that is different from that of pairs of rewrite rules, operationally one can regard an equation as a pair of rewrite rules. Moreover, "$\rightleftharpoons$" in K carries in fact a heating/cooling intuition similar to that of Cham. 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 equation from left to right; once a_1 is processed (using other equations/rules in the semantics) producing some result, place that result back into context via a "cooling" step, applying the equation from right to left. We actually find the conceptual analogy between computation structural equations in K and heating/cooling rules in Cham so insightful, that we take the liberty to borrow both the notation "$\rightleftharpoons$" and the terminology "heating/cooling" from Cham. However, it is important to realize that these heating/cooling equations can be applied at any moment and in any direction, because they are regarded not as computational steps but as structural identities. For example, one can use the two heating/cooling equations for "$\_ + \_$" above to pick and pass in front either a_1 or a_2, then rewrite it one step only using semantic rules (defined later in this section), then plug it back into the sum, then pick and pass in front either a_1 or a_2 again and rewrite it one step only, and so on, thus obtaining the desired non-deterministic operational semantics of $\_ + \_$.

One should always be aware in K definitions that "$\Box$" is nothing but a convenient notation used as part of label names freezing some computations for future use, and *not* get carried away and think of "$\Box$" as a special "hole expression" of sort K. For example, a hasty reader may think that K's approach to strictness is unsound, because one can "prove" wrong equalities as follows:

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

What is wrong in the above "proof" is that one cannot apply the second equation 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 equations are thought of as structural identities, 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 equivalence class. One of K's most subtle technical aspects, which fortunately is transparent to users, is to find the right (i.e., as weak as possible) restrictions on the applications of heating/cooling equations, so that each computational equivalence class contains no more than one fragment of program. This is called "computation adequacy" and is discussed in detail in Appendix **??**. The idea is to only allow heating and/or cooling of operator arguments that are proper syntactic computations (i.e., terms over the original syntax, i.e., different from "$\cdot$" and containing no "$\curvearrowright$"). With that, for example, the computation equivalence class of the expression $x * (y + 2)$ in the context of a language definition with non-deterministically strict binary $+$ and $*$, consists of the terms:

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

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

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

Seqstrict attribute. Recall that the two heating/cooling equations 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 $\_\leq\_$ in Figure 1, which was declared the strictness attribute *seqstrict*, saying that its subexpressions are processed deterministically, from left to right. The attribute *seqstrict* can be used only in definitions with a non-empty *KResult* and can be desugared automatically as follows: generate a heating/cooling equation for each argument like in the case of *strict*, but requiring that all its previous arguments are in *KResult*. For example, the *seqstrict* attribute of $\_\leq\_$ desugars into (a_1, a_2 range over K and r_1 over *KResult*):

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

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

Useful K labels. One can use heating/cooling equations to also define computational structural equivalences on K terms involving auxiliary K labels, not necessarily corresponding to the original language syntax. For example, some language constructs may have an arbitrary number of arguments, not always appearing as a list, such as, for example, a letrec construct with a syntax like "letrec $x_1 = e_1$ and $x_2 = e_2$ and $\cdots$ and $x_n = e_n$ in e". Since the expressions e_1, e_2, ..., e_n will need to be processed at some moment, it is very useful to extract them and schedule them for processing all at once. We assume two special K labels, *strict* and *seqstrict*, designed for such situations and defined as follows ($kl, kl' \in \mathsf{List}[K]$, $rl \in \mathsf{List}[\mathit{KResult}]$, $k \in K$):

$$\begin{aligned}
&\mathit{strict}(\cdot) = \mathit{seqstrict}(\cdot) = \cdot \\
&\mathit{strict}(kl, k, kl') \rightleftharpoons k \curvearrowright \mathit{strict}(kl, \square, kl') \\
&\mathit{seqstrict}(rl, k, kl') \rightleftharpoons k \curvearrowright \mathit{seqstrict}(rl, \square, kl')
\end{aligned}$$

As before, "$\square$" is not a hole but part of the name of a freezer, in our case of "$\mathit{strict}(\_, \square, \_)$" and "$\mathit{seqstrict}(\_, \square, \_)$", respectively, written in mixfix notation to ease reading. Recall that in the theoretical developments of K all sentences are schematas, so the two heating/cooling equations above comprise a recursively enumerable set of ground equations, one for each instance of the variables, including ones with kl, kl', or rl empty lists. In practice, though, the above equations are universally quantified (or parametric) in their variables and their application involves a matching modulo associativity; that is precisely how we implemented them in our Maude implementation of K (see Section E).

General strictness convention. More generally, we allow any K labels to be associated strictness attributes. For example, one may define some label *myEval* as follows:

$$\mathit{KLabel} ::= \cdots \mid \mathit{myEval}\ [\mathit{strict}]$$

That means that *myEval* is strict in all its arguments, no matter how many they are, so that one can schedule $e_1, e_2, \ldots, e_n$ for processing by wrapping them in a term $\mathit{myEval}(e_1, e_2, \ldots, e_n)$. With this general convention, the strictness of $\_ + \_$ and $\_ \leq \_$ can also be defined as follows

$$\mathit{KLabel} ::= \cdots \mid \_ + \_\ [\mathit{strict}] \mid \_ \leq \_\ [\mathit{seqstrict}]$$

while the useful K labels *strict* and *seqstrict* are nothing but two builtin labels defined as follows:

$$\mathit{KLabel} ::= \cdots \mid \mathit{strict}\ [\mathit{strict}] \mid \mathit{seqstrict}\ [\mathit{seqstrict}].$$

3.4 K Semantic Sentences: Equations and Rules

In its full generality as described in [33], K allows three types of sentences: structural (or non-computational), computational and non-deterministic. Regarded from a rewriting logic point of view, the structural sentences are always equations, while the non-deterministic ones are always rewrite rules; the computational sentences can be either equations or rules, depending upon how one uses the K definition: if used exclusively for execution to get an interpreter for the defined language, then it does not matter whether the computational sentences are equations or rules; if used for proofs by induction on computations, then one may want to regard all computational sentences as rules, so that the equational equivalence classes are as small and precise as possible; if used for model-checking a program, then one may regard all the computational sentences as equations, because they add no new behaviors and keeping them equations significantly reduces the state-space to be analyzed by the model checker. However, to keep the presentation simple, in this paper we only consider two types of sentences, like in rewriting logic: equations and rules. Equations are structural in nature and thus carry no computational meaning; they only say which terms should be viewed as identical and their role is to transparently modify the term so that rewrite rules can apply. Rewrite rules are seen as irreversible computational steps and can happen concurrently. All the heating/cooling equations in the previous section were K equations. The following are also examples of equations:

$$\begin{aligned}
&[\![p]\!] = [\![k(p)\ \mathit{state}(\emptyset)]\!] \\
&s_1; s_2 = s_1 \curvearrowright s_2 \\
&\textsf{while}\ b\ \textsf{do}\ s = \textsf{if}\ b\ \textsf{then}\ (s; \textsf{while}\ b\ \textsf{do}\ s)\ \textsf{else}\ \cdot
\end{aligned}$$

The "$\cdot$" in the "else" branch is the unit of K. The following are examples of rewrite rules:

$$\begin{array}{l} i_1 + i_2 \to i \qquad \text{when } i \text{ is the sum of } i_1 \text{ and } i_2 \\ \mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \to s_1 \\ \mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \to s_2 \\ k(x \curvearrowright \mathit{rest})\ \mathit{state}(\sigma) \to k(\sigma[x] \curvearrowright \mathit{rest})\ \mathit{state}(\sigma) \end{array}$$

Structural equations, like the heating and cooling equations, can be applied back and forth, while the rewrite rules can only be applied from left-to-right. For example, the heating/cooling equations of $\_+\_$ can be applied left-to-right to "schedule" the two sub-computations for evaluation and then right-to-left to plug their results back, and then a rewrite rule like the first one above can apply to irreversibly make the computational step. One should be careful when defining language constructs that have "sequential effects" on the configuration, so that those effects are propagated appropriately: these constructs should be processed only when they are placed at the leftmost side of the computation (see, for example, the last rule above for variable lookup).

Figure 3 shows all the equations and rewrite rules corresponding to the K definition of the simple imperative language in Figure 1, that is, its corresponding rewrite logic theory. For clarity, we grouped the equations and rules by the syntactic construct whose semantics they define.

When defining a language in K, one may need to define three things (these are not always all needed):

- A *configuration structure*, serving as a backbone for storing everything we need, such as computations, threads, states, environments, etc. It can have various layers of information, and configuration items are allowed to multiply themselves. We have a flat, two-item configuration in our example above.
- *Computation structural equations*, or *heating/cooling equations*, allowing one to change representations of computations at will. Each fragment of program will have a unique computation equivalence class, which can be derived using the computation structural equations.
- *Rewriting rules* and *equations*. Rules capture the execution steps of the defined language. Equations capture structural rearrangements that do not count as execution steps.

As already explained in Section 3.2, computations have a monoid structure (with unit a central dot "$\cdot$" and with composition $\_ \curvearrowright \_$, called "computation sequencing") and extend the syntax of the language, more precisely, its abstract syntax trees (AST). All the original syntactic categories, such as *AExp*, *BExp*, *Stmt*, *Pgm* in our case, are sinked into computations. Also, all the language constructs become simple K labels. To ease notation and reading, we sometimes use intuitive names, such as a, b, s, or p, to refer to computations originating from *AExp*, *BExp*, *Stmt*, or *Pgm*, respectively (we did that in Figure 1 but not in Figure 3), and use the language constructs in their original, mixfix syntax (we did that both in Figure 1 and in Figure 3). As pointed out in Section 3.3, most computation structural equations can be derived automatically from strictness attributes annotating the language constructs. In K we freely use such strictness attributes to keep our definitions more compact and less boring. Note, however, that there is almost a one-to-one correspondence between the context reduction definition in Figure 1 (left column) and the rewrite logic theory in Figure 3 corresponding to the K definition in 1 (right column). As shown in Section 5.4.6, context reduction can be captured formally, automatically and faithfully in K; by "faithfully" we mean that everything that can be done with the context reduction definition, can be done, in an isomorphic correspondence, in the resulting K definition.

Even though context reduction can be used as a definitional methodology within K, it will often be more convenient to define languages directly in K, using K's full strength and deviating from the strict obedience of syntax. For example, note that in order to give the semantics of assignment and while loops, the context reduction definition in Figure 1 had to "enrich" the syntax with a special "result" statement skip, which was not part of the original language; all statements then evaluate to this special statement skip, and skip also needs to be propagated through the syntax (only "$\mathsf{skip}; s \to s$" in this simple language, but one may need more such propagation rules in more complex languages, e.g., "$\{\mathsf{skip}\} \to \mathsf{skip}$", "$\{\mathit{varDecl}; \mathsf{skip}\} \to \mathsf{skip}$", etc.). If one follows the context reduction style in K, then one would have to do the same thing. However, in Figures 1 and 3 we subtly deviated from the context reduction methodology, because, from a K perspective,

$$\text{Configuration}\ \begin{cases} \mathit{KResult} ::= \mathit{Val} \\ \mathit{ConfigItem} ::= k(K) \mid \mathit{state}(\mathit{State}) \\ \mathit{Config} ::= \mathit{Int} \mid \llbracket K \rrbracket \mid \llbracket \mathsf{Set}[\mathit{ConfigItem}] \rrbracket \end{cases}$$

$$\begin{array}{c}\text{Initialization}\\ \text{Termination}\end{array}\ \begin{cases} \llbracket p \rrbracket = \llbracket k(p)\ \mathit{state}(\emptyset) \rrbracket \\ \llbracket k(v)\ \mathit{restConfig} \rrbracket = v \end{cases}$$

$$\text{Variable lookup}\ \big\{\ k(x \curvearrowright \mathit{rest})\ \mathit{state}(\sigma) \to k(\sigma[x] \curvearrowright \mathit{rest})\ \mathit{state}(\sigma)$$

$$\mathit{AExp} + \mathit{AExp}\ \begin{cases} k_1 + k_2 \rightleftharpoons k_1 \curvearrowright \square + k_2 \\ k_1 + k_2 \rightleftharpoons k_2 \curvearrowright k_1 + \square \\ i_1 + i_2 \to i_1 +_{\mathit{Int}} i_2 \end{cases}$$

$$\mathit{AExp} \leq \mathit{AExp}\ \begin{cases} k_1 \leq k_2 \rightleftharpoons k_1 \curvearrowright \square \leq k_2 \\ i \leq k_2 \rightleftharpoons k_2 \curvearrowright i \leq \square \\ i_1 \leq i_2 \to i_1 \leq_{\mathit{Int}} i_2 \end{cases}$$

$$\mathsf{not}\ \mathit{BExp}\ \begin{cases} \mathsf{not}\ k \rightleftharpoons k \curvearrowright \mathsf{not}\ \square \\ \mathsf{not}\ b \to \mathsf{not}_{\mathit{Bool}}\ b \end{cases}$$

$$\mathit{BExp}\ \mathsf{and}\ \mathit{BExp}\ \begin{cases} k_1\ \mathsf{and}\ k_2 \rightleftharpoons k_1 \curvearrowright \square\ \mathsf{and}\ k_2 \\ \mathsf{true}\ \mathsf{and}\ k \to k \\ \mathsf{false}\ \mathsf{and}\ k \to \mathsf{false} \end{cases}$$

$$\mathit{Name} := \mathit{AExp}\ \begin{cases} x := k \rightleftharpoons k \curvearrowright x := \square \\ k(x := v \curvearrowright \mathit{rest})\ \mathit{state}(\sigma) \to k(\mathit{rest})\ \mathit{state}(\sigma[v/x]) \end{cases}$$

$$\mathit{Stmt}; \mathit{Stmt}\ \big\{\ k_1; k_2 \rightleftharpoons k_1 \curvearrowright k_2$$

$$\mathsf{if}\ \mathit{BExp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{St}\ \begin{cases} \mathsf{if}\ k\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2 \rightleftharpoons k \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2 \\ \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2 \to k_1 \\ \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2 \to k_2 \end{cases}$$

$$\mathsf{while}\ \mathit{BExp}\ \mathsf{do}\ \mathit{Stmt}\ \big\{\ k((\mathsf{while}\ k_1\ \mathsf{do}\ k_2) \curvearrowright \mathit{rest}) = k((\mathsf{if}\ k_1\ \mathsf{then}\ (k_2; \mathsf{while}\ k_1\ \mathsf{do}\ k_2)\ \mathsf{else}\ \cdot) \curvearrowright \mathit{rest})$$

$$\mathsf{halt}\ \mathit{AExp}\ \begin{cases} \mathsf{halt}\ k \rightleftharpoons k \curvearrowright \mathsf{halt}\ \square \\ k((\mathsf{halt}\ i) \curvearrowright \mathit{rest}) = k(i) \end{cases}$$

Figure 3: Automatically desugared rewriting logic variant of the imperative language in Figure 1

introducing the skip statement would be a rather artificial and unnecessary step. Instead, we just "dissolved" the statements whenever they carried no further computational meaning (by identifying them with the empty computation) and captured the semantics of sequential composition with precisely one equation, "$k_1; k_2 = k_1 \curvearrowright k_2$"; when k_1 is finished it becomes the empty computation and then k_2 becomes naturally (with no additional rewrite step) the next task.

To start the rewriting machinery, one needs to create an initial configuration with the given program (first equation in Figure 3). Typically, the program is placed somewhere in the configuration where a computation is expected. If everything is as expected (program terminates, no "runtime errors", etc.), the program will eventually become a result. When that happens, one can dissolve the configuration and keep the result (second equation in Figure 3). Note that both these steps are purely structural, preparing things for evaluation or collecting results, so they have no computational meaning; therefore, they are defined with equations instead of rewrite rules.

3.5 K Specialized Notation: Attributes, Contexts, Angle Brackets

In K definitions, rewrite rules and equations can be given either explicitly as above, or implicitly through a specialized and optimized K-notation explained next. A beginner would probably prefer explicit equations and rules, but an expert can take advantage of K's compact notation and move quickly to the interesting aspects of the definition, letting low-level details to be automatically filled through the implicit desugaring mechanisms explained below. Here is the intuition behind the major K notations:

K Attributes. We have already discussed the strictness attributes *strict* and *seqstrict*, as well as their automatic desugaring into heating/cooling equations in Section 3.3. The definition of the imperative language in Figure 1 made use of another K attribute, *extends*. The attribute *extends* can only be used in combination with a *strict* or *seqstrict* attribute, and corresponds to a *rewrite rule* saying what to do after the strict arguments were evaluated. A construct may have more than one *extends* attribute; a rewrite rule will be generated for each. The *extends* attribute for addition, for example, corresponds to the rule

$$i_1 + i_2 \rightarrow i_1 +_{Int} i_2 \quad \text{where } i_1, i_2 \in Int.$$

There is one more attribute, namely one that can be associated to configuration constructs: *multiply*. The *multiply* attribute is useful when one wants to enhance the modularity of K definitions using K's context transforming capability; it basically says that the corresponding configuration attribute is allowed to multiply as part of the same level configuration soup. For example, since threads can be created or terminated dynamically and since each thread has its own sub-configuration soup (containing a computation, an environment, resources, etc.) in the global program configuration, one may want to associate the attribute *multiply* to the *thread ConfigLabel*. This way, the context transforming procedure knows how to desugar rules or equations mentioning the same type of configuration item twice or more times. For example, the rule for rendez-vous synchronization in Section 6 mentions "$k(\mathsf{rv}\ v...)\ k(\mathsf{rv}\ v...)$", which does not match as given, because each k is part of a different thread; however, since the configuration item label k was not declared to multiply, while *thread* is, the context transformers will automatically place each k inside a *thread*: "$thread(k(\mathsf{rv}\ v...)...)\ thread(k(\mathsf{rv}\ v...)...)$".

K Contexts. To avoid repeating context information that does not change in the left-hand and and the right-hand -sides of equations and rules, we take the liberty to use the two-dimensional notation

$$C[\underset{t'_1}{\underline{t_1}}, \underset{t'_2}{\underline{t_2}}, ..., \underset{t'_n}{\underline{t_n}}]$$

for *rules* $C[t_1, t_2, ..., t_n] \rightarrow C[t'_1, t'_2, ..., t'_n]$, and similarly

$$C[\underset{t'_1}{\underline{t_1}}, \underset{t'_2}{\underline{t_2}}, ..., \underset{t'_n}{\underline{t_n}}] \quad \text{(dotted horizontal lines)}$$

for *equations* $C[t_1, t_2, ..., t_n] = C[t'_1, t'_2, ..., t'_n]$. When using K in ASCII, we typically replace the horizontal lines by $\Rightarrow$, so we write $C[t_1 \Rightarrow t'_1,\ t_2 \Rightarrow t'_2,\ ...,\ t_n \Rightarrow t'_n]$; to distinguish between equations and rules in this latter case, we prefix the sentence with a `rl` or `eq` keyword.

K Underscore Variables and Angle Brackets. The K-context notation above allows for two other notation optimizations. Like in Prolog, we use an underscore "_" as a variable matching anything that we don't care (if we had not used the two-dimensional notation, then we would have had to know the name of each variable, because we would have had to repeat it in the right-hand-side). With the K-context and the underscore variable notations, the rule for assignment in our simple imperative language in Figure 3 becomes

$$\llparenthesis \underset{\cdot}{\underline{x := v}} \curvearrowright \_ \rrparenthesis_k \; \llparenthesis \underset{\sigma[v/x]}{\underline{\quad \sigma \quad}} \rrparenthesis_{state}$$

The even more compact notation in Figure 1 uses the K angle bracket convention $\llparenthesis x := v \rangle_k$ which can be used whenever the cell-enclosed structure is a list or a set. The angle bracket reads "and so on" (to the right here, but it can also be to the left).

Like most notations, the K-notation may look strange, obscure and even artificial at first sight. One is, of course, free to use it or not to use it. K's notational conventions were not designed in an *adhoc* manner, but through a series of iterations and refinements, defining large languages in K and observing that many language or language-related definitions follow similar patterns, in particular that the designer was forced by the previously rigid formalism to repeat large portions of the lhs of a rule into its rhs. Repetitive definitions and redundancy are not only boring and heavy looking, but, more importantly, they are error prone and even may reduce modularity. Consider, as an example, the angle bracket notation above; writing $\llparenthesis x := v \rangle_k$ instead of $\llparenthesis x := v \curvearrowright \_ \rrparenthesis_k$ gives us the freedom to change the name of the list construct "$\_ \curvearrowright \_$" without having to touch most of the existing rules; changing such list construct names is not uncommon — for example, one may want to change "$\_ \curvearrowright \_$" into "$\_ \curvearrowright_{concrete} \_$" when one also wants to add a type system to a language, in which case one may use "$\_ \curvearrowright_{type} \_$" as a construct for computations that rewrite to types. Nevertheless, if one does not want to use our optimized K-notation then one can simply ignore it. The philosophy of K stays not in its specialized notation, but in its approach to define programming languages or language analyses by "swallowing" program syntax into computation structures, which can be heated/cooled and reduced by means of ordinary, context-insensitive term rewriting, at the same time giving a "true concurrency" semantics for the defined language if that is what one wants. The specialized notation is nothing but an additional benefit that the experimented users can take advantage of.

4 Defining "Idealized" Programming Languages in K

In this section we give K definitions, including both syntax and semantics, to a series of common simple languages and calculi. By calling these "simple" we do not intend to minimize the importance of these languages or calculi[2]. On the contrary, we selected these particular languages and calculi because we believe that they are intrinsically interesting and challenging from a mathematical and foundational point of view. However, looking at them through the eye of the designer of language design frameworks, these languages and calculi should be easy to define in an ideal language definitional framework. Whenever necessary, we state results saying that the corresponding K definition has the desired properties of the calculus or the language. The purpose of this section is twofold: on the one hand, it has an illustrory role, showing various uses of K that may give the reader hints on how more complex languages can be defined in K; on the other hand, we are not aware of any language definitional formalism or technique that can define all these languages and/or calculi in a uniform way, so it indirectly shows the strength and versatility of K.

Even though some of the calculi, languages and/or features below are challenging in their own way for some of the language definitional frameworks, one should not get tricked to conclude that they serve as a "definitive benchmark" for evaluating language definitional frameworks. Many language definitional frameworks look reasonable on paper when defining simple calculi or idealized languages or features like the ones in this section, but they tend to violate some or most of the requirements listed in Section 1, most importantly they do not scale. Therefore, we prefer to rather regard the examples in this section as a "minimal benchmark", in the sense that a language definitional framework that struggles to define these may, most likely, not be practical. Also, the fact that a framework or technique can naturally define these calculi, languages and features, should not be taken as conclusive: it only proves that the framework or technique is not entirely impractical in general. A much better benchmark could be the K-CHALLENGE language

[2]Similarly, the fact that groups can be defined as simple three-equation algebraic specifications or first-order theories does not make group theory less important or interesting.

in Section 6, the languages in the appendixes of this paper, or even realistic languages, such as Java and Scheme. All these have been defined in K following the very same uniform definitional approach.

4.1 Defining Turing Machines

Equational encodings of general computation into equational deduction are well-known; for example, [4, 2] show such encodings, where the resulting equational specifications, if regarded as term rewrite systems (TRSs), are confluent and terminate whenever the original computation terminates. Our goal in this section is to discuss a simple definition of (Turing machine) computations in K. While any of the existing encodings of Turing machines as unconditional TRSs can also serve as an example of how to define Turing machines in K (because K includes unconditional TRSs), we find the existing encodings more complex and intricate than needed. Indeed, the encodings in [4, 2] aim at emphasizing the strength of equational reasoning or the general undecidability of termination of term rewriting; they are not intended to be elegant or compact. Since K assumes lists and sets whenever needed and since reduction/rewriting takes place modulo (structural) equations (associativity of lists, etc.), we can give simpler, more intuitive and easier to understand definitions of Turing machines in K. The K rewrite theories associated to Turing machines below can be faithfully used as Turing-complete computational engines, because each rewrite step corresponds to precisely one computation step in the Turing machine; in other words, there are no artificial rewrite steps.

There are many equivalent definitions of Turing machines in the literature. We prefer one adapted from [34], and describe it informally in the sequel. The reader is assumed familiar with basic intuitions of Turing machines. Consider a mechanical device which has associated with it a tape of infinite length in both directions, partitioned in spaces of equal size, called *cells*, which are able to hold either a “0” or an “1” and are rewritable. The device examines exactly one cell at any time, and can, potentially nondeterministically, perform any of the following four operations (or *commands*):

1. Write a “1” in the current cell;
2. Write a “0” in the current cell;
3. Shift one cell to the right;
4. Shift one cell to the left.

The device performs one operation per unit time, called a *step*. Formally, let Q be a finite set of *internal states*, containing a *starting state* q_s and a *halting state* q_h. Let $B = \{0, 1\}$ be a set of *symbols* (or *bits*) and $C = \{0, 1, \rightarrow, \leftarrow\}$ be a set of *commands*. Then a Turing machine is a multivalued function (i.e., a total relation) $M : (Q - \{q_h\}) \times B \rightarrow Q \times C$. If M is an ordinary function then the Turing machine is called *deterministic*. We assume that the tape contains only 0’s before the machine starts performing. A *configuration* of a Turing machine is a 4-tuple consisting of an internal state, a current cell, and two infinite strings (notice that the two infinite strings contain only 0’s starting with a certain cell), standing for the cells on the left and for the cells on the right of the current cell, respectively. We let $(q, L\underline{b}R)$ denote the configuration in which the machine is in state q, with current cell b, left tape L and right tape R. For convenience, we write the left tape L backwards, that is, its head is at its right end; for example, Lb cons a b to the left tape L. Given a configuration $(q, L\underline{b}R)$, the content of the tape is LbR, which is infinite at both ends. We also let $(q, L\underline{b}R) \rightarrow_M (q', L'\underline{b'}R')$ denote a configuration transition under one of the four commands. Given a configuration in which the internal state is q and the examined cell contains b, and if $(q', c) \in M(q, b)$, then exactly one of the following configuration transitions can take place:

1. $(q, L\underline{b}R) \rightarrow_M (q', L\underline{c}R)$ if $c = 0$ or $c = 1$;
2. $(q, L\underline{b}b'R) \rightarrow_M (q', Lb\underline{b'}R)$ if $c = \rightarrow$;
3. $(q, Lb'\underline{b}R) \rightarrow_M (q', L\underline{b'}bR)$ if $c = \leftarrow$.

Note that the relation $\to_M$ on configurations is nondeterministic in general because M can be a multifunction, but is deterministic if the Turing machine is deterministic (i.e., M is a function). The machine starts performing in the internal state q_s. If there is no input, the initial configuration on which the Turing machine is run is $(q_s, 0^\omega \underline{0} 0^\omega)$, where 0^ω is the infinite string of zeros. If the Turing machine is intended to be run on a specific input, say $x = b_1 b_2 \cdots b_n$, its initial configuration is $(q_s, 0^\omega \underline{0} b_1 b_2 \cdots b_n 0^\omega)$. For simplicity, we assume that inputs are always encoded as finite sequences of 1's. We let $\to^\star_M$ denote the transitive and reflexive closure of the binary relation on configurations $\to_M$ above. A Turing machine M terminates when it gets to its halting state, q_h; formally, *M terminates on input $b_1 b_2 \cdots b_n$* iff $(q_s, 0^\omega \underline{0} b_1 b_2 \cdots b_n 0^\omega) \to^\star_M (q_h, L\underline{b}R)$ for some $b \in B$ and some tape instances L and R. Note that a Turing machine cannot get stuck in any state but q_h, because the mapping M is defined everywhere except in q_h. Therefore, a Turing machine carries out successions of steps, which may or may not terminate. Because of its non-determinism, it may be the case that, for the same input, there are both terminating and non-terminating sequences; however, if the machine terminates on the given input then there must be at least a terminating sequence for that input. If the Turing machine is deterministic then the succession of steps is uniquely determined for any input and, obviously, the machine either terminates on the given input or it does not. It is well-known that Turing machines can compute exactly the partial recursive functions [34].

From here on in this section, let us fix a Turing machine M. We next define a K theory, K_M, which captures precisely the Turing machine M, in the sense that any computational step in M corresponds to precisely one rewrite step in K_M. We start by defining the signature, or syntax, of K_M. The syntactic category Q contains all the states of M and may obviously differ from Turing machine to Turing machine; the other three syntactic categories, B, K and *Config* are generic for all Turing machines:

$$\begin{array}{rcl}
Q & ::= & q_s \mid q_h \mid \cdots \text{ the finite number of states of } M \\
B & ::= & 0 \mid 1 \\
K & ::= & B \mid \mathsf{List}^{\cdot}_{\curvearrowright}[K] \\
\mathit{Config} & ::= & \llparenthesis Q \rrparenthesis_{\mathit{state}} \mid \llparenthesis K \rrparenthesis_{\mathit{left}} \mid \llparenthesis B \rrparenthesis_{\mathit{cell}} \mid \llparenthesis K \rrparenthesis_{\mathit{right}} \mid \llparenthesis \mathsf{Set}[\mathit{Config}] \rrparenthesis_\top \mid [\![\mathsf{List}^{\cdot}_{\_\_}[B]]\!] \mid \mathsf{done}
\end{array}$$

The configuration items $\llparenthesis K \rrparenthesis_{\mathit{left}}$ and $\llparenthesis K \rrparenthesis_{\mathit{right}}$ hold the left and the right tapes, $\llparenthesis B \rrparenthesis_{\mathit{cell}}$ holds the current cell, and $\llparenthesis Q \rrparenthesis_{\mathit{state}}$ holds the current state. As usual, the input to the machine, which can be regarded as the initial "computational structure", is given enclosed by double brackets $[\![\_]\!]$, and the initial configuration is wrapped into a $\llparenthesis \_ \rrparenthesis_\top$ soup. The constant configuration done marks that the Turing machine computation terminated. The following two equations are generic for all machines M of start state q_s and halt state q_h:

$$\begin{array}{rcl}
[\![\mathit{input}]\!] & = & \llparenthesis \llparenthesis q_s \rrparenthesis_{\mathit{state}} \; \llparenthesis \cdot \rrparenthesis_{\mathit{left}} \; \llparenthesis 0 \rrparenthesis_{\mathit{cell}} \; \llparenthesis \mathit{input} \rrparenthesis_{\mathit{right}} \rrparenthesis_\top \\
\langle\!| \llparenthesis q_h \rrparenthesis_{\mathit{state}} |\!\rangle_\top & = & \mathsf{done}
\end{array}$$

The first equation initializes the computation by placing the input on the right tape and setting the current state to the start state. The second terminates the computation when the halting state is reached. Note that both sentences above are equations (and not rules), because they carry no computational meaning. Note that the left and right tapes are finite structures in our setting, so they cannot store the infinite sequences of zeros. However, such infinite sequences are only conceptual; all one needs to be able to do is to generate "one more 0" whenever needed. The following two equations, also generic for all Turing machines M can "generate new 0s" on each tape on a by-need basis:

$$\begin{array}{rcl}
\llparenthesis \cdot |\!\rangle_{\mathit{left}} & = & \llparenthesis 0 |\!\rangle_{\mathit{left}} \\
\langle\!| \cdot \rrparenthesis_{\mathit{right}} & = & \langle\!| 0 \rrparenthesis_{\mathit{right}}
\end{array}$$

We are now ready to give the M-specific rules of K_M, where $b, c \in B$ and $q, q' \in Q$:

$$\begin{array}{lllll}
\llparenthesis \underset{q'}{\underline{q}} \rrparenthesis_{\mathit{state}} & \llparenthesis \underset{c}{\underline{b}} \rrparenthesis_{\mathit{cell}} & & & \text{when } (q', c) \in M(q, b) \\
\llparenthesis \underset{q'}{\underline{q}} \rrparenthesis_{\mathit{state}} & \langle\!| \underset{b}{\underline{\cdot}} \rrparenthesis_{\mathit{left}} & \llparenthesis \underset{b'}{\underline{b}} \rrparenthesis_{\mathit{cell}} & \llparenthesis \underset{\cdot}{\underline{b'}} |\!\rangle_{\mathit{right}} & \text{when } (q', \to) \in M(q, b) \\
\llparenthesis \underset{q'}{\underline{q}} \rrparenthesis_{\mathit{state}} & \langle\!| \underset{\cdot}{\underline{b'}} \rrparenthesis_{\mathit{left}} & \llparenthesis \underset{b'}{\underline{b}} \rrparenthesis_{\mathit{cell}} & \llparenthesis \underset{b}{\underline{\cdot}} |\!\rangle_{\mathit{right}} & \text{when } (q', \leftarrow) \in M(q, b)
\end{array}$$

Since all K rewrite rules above overlap on the $(\!|\_|\!)_{state}$ subterm, the rewrite steps in K_M on an initial term of the form $[\![b_1 b_2 \cdots b_n]\!]$ are sequential (non-concurrent); indeed, there is no concurrent rewriting taking place in K_M, which is consistent with the intended Turing machine computation. The following result states that K is not only a Turing-complete framework, but a *faithfully Turing-complete* one, in the sense that any computation in a Turing machine M can be carried out, step-for-step, as a rewriting sequence in K_M. In other words, K_M *is* M, not an encoding of it, the only difference between the two being their different but ultimately irrelevant notational conventions:

Theorem 1. ***(Faithful embedding of Turing machines into K)*** *If M is a Turing machine M then*

1. $(q, L\underline{b}R) \to_M (q', L'\underline{b'}R')$ *iff* $K_M \models (\!|q|\!)_{state}\ (\!|L|\!)_{left}\ (\!|b|\!)_{cell}\ (\!|R|\!)_{right} \to (\!|q'|\!)_{state}\ (\!|L'|\!)_{left}\ (\!|b'|\!)_{cell}\ (\!|R'|\!)_{right}$*;*
2. *M terminates on input $b_1 b_2 \cdots b_n$ iff $K \models [\![b_1 b_2 \cdots b_n]\!] \to^\star$ done; moreover, if that is the case, then done is the only normal form of $[\![b_1 b_2 \cdots b_n]\!]$ in K_M.*

The first item in the theorem above states that each computational step in M corresponds to a rewrite step in K_M, while the second item gives a means to "execute" M using a rewrite engine and K_M: M terminates on some input iff K_M terminates as a rewrite system on that input wrapped in $[\![\_]\!]$.

4.2 Defining Lambda-Calculi in K

Variants of λ-calculus have been already shown in Figure 1. We here discuss them in more detail. For simplicity and direct comparison of K with other formalisms to define λ-calculi, in this section we prefer to use a substitution-based approach. Therefore, we assume given a substitution function "$e[e'/x]$", which replaces each free occurrence of x in e by e', avoiding variable captures via α-conversion. One should not underestimate the complexity of applying such a substitution. Even though we provide a syntax-independent (using a generic notion of "binder") definition of substitution in our implementation of K in Maude (See Appendix E), we do not advocate using it when defining non-toy languages. Instead, we prefer the more practical approach based on environments and closures, as shown in Section 4.9 and in the appendixes.

Standard λ-calculus is one of the most basic calculi. In standard, untyped λ-calculus, there is only one syntactic category of λ-*expressions*, *Exp*; "evaluation" consists of syntactic transformations of the original λ-expression. Constant expressions, such as integers or booleans, may be included in *Exp*, but here we do not do that. The syntax of λ-calculus includes a construct for λ-*abstraction* and one for λ-*application.* No values need to be defined in standard λ-calculus, and no evaluation strategy is imposed (no strictness attributes):

$$Exp ::= \lambda Name.Exp \mid Exp\ Exp.$$

There is only one rewrite rule in our K definition of λ-calculus, which is precisely the β-reduction:

$$(\lambda x.e)\ e' \to e[e'/x], \quad \text{where } x \in Name,\ e, e' \in Exp.$$

Note that, unlike in reduction semantics, with [11] or without [31] evaluation contexts, when using K (or any other context-insensitive rewriting-based framework), one does not need to explicitly give the rewrite rule access to subexpressions. In other words, the K definition of conventional λ-calculus is precisely the standard definition of λ-calculus; no different syntax or notational conventions are necessary, and there is no faithful-embedding result to be proved. However, one should note that, unlike in the conventional λ-calculus where, operationally speaking, β-reduction steps apply non-deterministically but one-at-a-time, the underlying K rewriting machinery allows for the β-reduction steps to apply concurrently; the confluence (or Church-Rosser property) of λ-calculus ensures the safety of the concurrent application of β-reduction steps.

Call-by-value λ-calculus. In call-by-value λ-calculus, applications of β-reduction are not unrestricted anymore as in standard λ-calculus: the two sub-expressions involved in a λ-application are first evaluated, and then β-reduction is applied. Also, evaluation is inhibited in λ-abstractions, which are therefore regarded as values. Here is the complete K definition of call-by-value λ-calculus:

$$\begin{array}{l}
\mathit{Val} ::= \lambda \mathit{Name}.\mathit{Exp} \\
\mathit{Exp} ::= \mathit{Val} \mid \mathit{Exp}\ \mathit{Exp}\ [\mathit{strict}] \\
\mathit{Config} ::= (\!|K|\!)_k \\
\mathit{KResult} ::= \mathit{Val} \\
(\!|(\lambda x.e)\, v|\!)_k \rightarrow (\!|e[v/x]|\!)_k, \quad \text{where } x \in \mathit{Name},\ e \in \mathit{Exp},\ v \in \mathit{Val}.
\end{array}$$

Note that λ-application is defined *strict* and there is only one configuration item, $(\!|K|\!)_k$. Reduction takes place only on the top of the computation and only when the top construct is a λ-application whose two sub-expressions are values. Therefore, the concurrency aspect of K is completely inhibited.

Call-by-name λ-calculus. There are two differences between call-by-value and call-by-name λ-calculi: (1) in the latter, the λ-application is strict only in its first sub-expression and, (consequently) (2) it can apply on any expression as argument, not only on a value:

$$\begin{array}{l}
\mathit{Val} ::= \lambda \mathit{Name}.\mathit{Exp} \\
\mathit{Exp} ::= \mathit{Val} \mid \mathit{Exp}\ \mathit{Exp}\ [\mathit{strict}(1)] \\
\mathit{Config} ::= (\!|K|\!)_k \\
\mathit{KResult} ::= \mathit{Val} \\
(\!|(\lambda x.e)\, e'|\!)_k \rightarrow (\!|e[e'/x]|\!)_k, \quad \text{where } x \in \mathit{Name},\ e, e' \in \mathit{Exp}.
\end{array}$$

4.3 Defining a Simple Imperative Language in K

Figure 4 shows the complete K definition of the simple imperative language discussed in Sections 1 and 3. Let us call this language IMP. Note that the construct ";" for programs does not need to be given an attribute equation like the one for the construct ";" for statements, because the two constructs get identified once the language syntax is sinked into computations. Indeed, from a computational point of view, the two constructs have the *same* semantics: process the first subcomputation, then the second. If one's intention for two language constructs with the same name is to have different computational semantics, then one should give them different names; this can be easily achieved at parse-time.

There is not much to comment on this simple language definition, except perhaps to note that, except for the definition of the K-specific "syntactic" infrastructure (the syntactic categories *KResult*, K, and *Config*), every single equation or rule (explicit or implicit in the notational conventions) in the definition in Figure 4 defines a distinct aspect of the language, so it cannot be avoided. This obvious property is actually violated by many other language definitional formalisms; for example, purely syntactic formalisms, such as SOS or context reduction, even for this simple language have to enrich the syntax with an artificial "skip" statement that acts as a "value" to which statements evaluate. This is unnecessary in K, because the concept of "evaluation" is replaced by "computation processing" in K and computations may place zero, one or more computational items back on the computation structure after they are processed.

4.4 Defining Milner's CCS in K

The Calculus of Communicating Systems (CCS) [25] was introduced by Milner as a process calculus/algebra in which processes can execute concurrently and communicate with each other (send and receive messages) via channels, or communication ports. The grammar of CCS is as follows:

$$\begin{array}{rcll}
\mathit{Ch} & ::= & a \mid b \mid ... & (\textit{channels}; \text{ given as a set of channel names}) \\
L & ::= & \mathit{Ch} \mid \overline{\mathit{Ch}} & (\textit{labels}; \text{ there can be input or output ones}) \\
A & ::= & \tau \mid L & (\textit{actions}; \tau \text{ is the ``internal'' action}) \\
P & ::= & 0 \mid \mathit{Name} \mid A.P \mid P + P \mid P|P \mid P[\varphi] \mid P\backslash L & (\textit{processes})
\end{array}$$

$\boxed{\text{K-Annotated Syntax of IMP}}$

$$\begin{array}{rcll}
\mathit{Int} & ::= & \ldots \text{ all integer numbers} & \\
\mathit{Bool} & ::= & \mathsf{true} \mid \mathsf{false} & \\
\mathit{Name} & ::= & \text{all identifiers; to be used as names of variables} & \\
\mathit{Val} & ::= & \mathit{Int} & \\
\mathit{AExp} & ::= & \mathit{Val} \mid \mathit{Name} & \\
 & \mid & \mathit{AExp} + \mathit{AExp} & [\mathit{strict},\ \mathit{extends}\ +_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
\mathit{BExp} & ::= & \mathit{Bool} & \\
 & \mid & \mathit{AExp} \leq \mathit{AExp} & [\mathit{seqstrict},\ \mathit{extends}\ \leq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathsf{not}\ \mathit{BExp} & [\mathit{strict},\ \mathit{extends}\ \neg_{\mathit{Bool}\to\mathit{Bool}}] \\
 & \mid & \mathit{BExp}\ \mathsf{and}\ \mathit{BExp} & [\mathit{strict}(1)] \\
\mathit{Stmt} & ::= & \mathit{Stmt}; \mathit{Stmt} & [s_1; s_2 = s_1 \curvearrowright s_2] \\
 & \mid & \mathit{Name} := \mathit{AExp} & [\mathit{strict}(2)] \\
 & \mid & \mathsf{if}\ \mathit{BExp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt} & [\mathit{strict}(1)] \\
 & \mid & \mathsf{while}\ \mathit{BExp}\ \mathsf{do}\ \mathit{Stmt} & \\
 & \mid & \mathsf{halt}\ \mathit{AExp} & [\mathit{strict}] \\
\mathit{Pgm} & ::= & \mathit{Stmt}; \mathit{AExp} &
\end{array}$$

$\boxed{\text{K Configuration and Semantics of IMP}}$

$$\begin{array}{rcl}
\mathit{KResult} & ::= & \mathit{Val} \\
K & ::= & \mathit{KResult} \mid \mathsf{List}^{\cdot}_{\curvearrowright}[K] \\
\mathit{Config} & ::= & (\!|K|\!)_k \mid (\!|\mathit{State}|\!)_{\mathit{state}} \\
 & \mid & \mathit{Val} \mid [\![K]\!] \mid (\!|\mathsf{Set}[\mathit{Config}]|\!)_\top
\end{array}$$

$[\![p]\!] = (\!|(\!|p|\!)_k\ (\!|\emptyset|\!)_{\mathit{state}}|\!)_\top$
$\langle\!|(\!|v|\!)_k|\!\rangle_\top = v$

$(\!|\frac{x}{\sigma[x]}\rangle\!|_k\ (\!|\sigma|\!)_{\mathit{state}}$
$\mathsf{true\ and}\ b \to b$
$\mathsf{false\ and}\ b \to \mathsf{false}$
$(\!|\frac{x := v}{\cdot}\rangle\!|_k\ (\!|\frac{\sigma}{\sigma[v/x]}|\!)_{\mathit{state}}$
$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \to s_1$
$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \to s_2$
$(\!|\mathsf{while}\ b\ \mathsf{do}\ s\rangle\!|_k = (\!|\mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else}\ \cdot\rangle\!|_k$
$(\!|\mathsf{halt}\ i\rangle\!|_k = (\!|i|\!)_k$

Figure 4: K definition of IMP.

There can be any number of channels. Labels are either channel names or over-lined channel names, such as $\overline{a}$, $\overline{b}$, etc. The intuition for a label that is a channel name is that a message is being sent on that channel, while the intuition for a label which is an over-lined channel name is that a message is received on that channel. An action can be either a label (send/receive) or a silent action, denoted τ. To simplify technical developments, the over-line notation is extended by convention to actions and the equations

$$\begin{array}{rcll}
\overline{\tau} & = & \tau & \\
\overline{\overline{\alpha}} & = & \alpha & (\text{for any } \alpha \in A)
\end{array}$$

are implicitly assumed.

Processes can be thought of as boxes that can perform actions and can communicate with each other. The process 0, also called "nil" or "deadlock", is the basic process that can perform no action. Processes can be assigned names and then referred to by their name in other processes; this is explained in more detail below. A process of the form $\alpha.p$ where α is an action is one which takes the action α and then continues like p. The process $p + q$ can be thought of as a process non-deterministically choosing between p and q, but the choice is being made only if the chosen process can act; therefore, $p + q$ has the initial capabilities of both p and q, but choosing to perform first an action from either p or q, will prevent the other one from acting any further. In particular, a process $p+0$ can take whatever action p can take, but it cannot choose to non-deterministically transit to 0 and then deadlock. Because of the commutative nature of $+$, many/most developments of CCS or extensions of it assume the following structural equivalence on processes:

$$p + q \equiv q + p \quad \text{(for any } p, q \in P).$$

The process $p|q$ is the process thought of[3] as executing in parallel processes p and q, that is, it can take any action that p or q can take; moreover, parallel processes are allowed to communicate via channels: if p takes the action a then behaves as p', and if q takes the action $\overline{a}$ then behaves like q', then the process $p|q$ takes the silent action τ then behaves like $p'|q'$. Parallel composition is also assumed to be structurally commutative:

$$p|q \equiv q|p \quad \text{(for any } p, q \in P).$$

CCS allows channels to be relabeled and even hidden. For technical simplicity, relabeling functions φ are extended to actions and are required to obey the properties

$$\begin{array}{rcl} \varphi(\tau) & = & \tau \\ \varphi(\overline{\alpha}) & = & \overline{\varphi(\alpha)} \quad \text{(for any } \alpha \in A). \end{array}$$

Then the process $p[\varphi]$ is the process obtained from p by relabeling all its actions according to φ. Finally, $p\backslash l$ is the process that behaves like p except that it cannot export any action over the channel that the label l refers to; in other words, $p\backslash l$ can perform any action p can perform except actions l or $\overline{l}$.

There is one more important aspect of CCS that needs to be discussed before we can give formal definitions of CCS. A CCS definition consists of a set of process definitions, that is, of a set of equalities

$$\mathit{Name} \overset{\text{def}}{=} P$$

where each process name is defined precisely once and where, of course, the process expressions bound to the process names can refer to any of the defined processes through their names. For notational simplicity, we adopt a relatively standard fix-point notation (see e.g., [5]): CCS definitions of the form

$$x_i \overset{\text{def}}{=} p_i \text{ for } i \in I$$

are replaced by a vectorial equality $\vec{x} = \vec{p}$, and each reference to process name x_i in the original process (the one whose behavior one is interested in) is replaced by $\mathsf{fix}_i(\vec{x} = \vec{p})$. The semantics of $\mathsf{fix}_i(\vec{x} = \vec{p})$ can now be given locally using a straightforward fixed point reduction rule, namely

$$\mathsf{fix}_i(\vec{x} = \vec{p}) \rightarrow p_i[\overrightarrow{\mathsf{fix}_i}(\vec{x} = \vec{p})/\vec{x}].$$

Figure 5 shows a conventional SOS definition of CCS (left column), as well as two different K definitions, one following an interleaving semantics (middle column) which is entirely equivalent to the SOS definition, and another one which is truly concurrent (right column). Rewriting logic and K definitions of CCS equivalent to the SOS definition can be mechanically derived as well, applying the faithful embedding procedures discussed in Sections 5.1.1 and 5.1.2. If $\mathit{SOS} \vdash p \overset{\alpha}{\rightarrow} p'$ then we say that process p can take action α and then transit to p'. This labeled transition relation is typically non-deterministic. Note that a significant number of transitions between various subprocesses of p may need to be derived in order to derive $p \overset{\alpha}{\rightarrow} p'$. Also, due to the inherent non-determinism and the fact that some processes may get blocked, a significant amount of search may be needed to derive a particular transition.

The K interleaved definition of CCS (middle column) in Figure 5 starts by defining all the process constructors strict in their first argument. That means that their first argument process can be heated and cooled at discretion. Since "|" and "+" are commutative, they are hereby strict in their both arguments. There is no need to have a correspondent in K for the SOS rule "$\alpha.p \overset{\alpha}{\rightarrow} p$"; that is because one can already

[3]The SOS definition of CCS (see Figure 5) provides no capability for saying that p and q run in parallel, or concurrently.

SOS_{CCS}	$K^{interleaved}_{\text{CCS}}$	$K^{concurrent}_{\text{CCS}}$
	$K ::= P \mid \mathsf{List}^{\cdot}_{\curvearrowright}[K]$ $Config ::= (\!\mid K \mid\!)$ $\_\mid\_,\ \_+\_,\ \_\backslash\_,\ \_[\_]$ $\quad[strict(1)]$	$K ::= P \mid \mathsf{List}^{\cdot}_{\curvearrowright}[K]$ $\quad\mid\ (\!\mid K \mid\!) \mid K \mathbin{[\!]} K \mid K \oplus K$ $\_\backslash\_,\ \_[\_]$ $\quad[strict(1)]$
$\alpha.p \xrightarrow{\alpha} p$		
$\dfrac{p \xrightarrow{\alpha} p'}{p \mid q \xrightarrow{\alpha} p' \mid q}$	$(\!\mid \alpha.p \mid q \mid\!) \rightarrow (\!\mid \alpha.(p \mid q) \mid\!)$	$(\!\mid p \mid q \mid\!) = (\!\mid (\!\mid p \mid\!) \mathbin{[\!]} (\!\mid q \mid\!) \mid\!)$ $(\!\mid \alpha.p \mid\!) \mathbin{[\!]} (\!\mid q \mid\!) \rightarrow \alpha.(p \mid q)$
$\dfrac{p \xrightarrow{\alpha} p' \quad q \xrightarrow{\overline{\alpha}} q'}{p \mid q \xrightarrow{\tau} p' \mid q'}$	$(\!\mid \alpha.p \mid \overline{\alpha}.q \mid\!) \rightarrow (\!\mid \tau.(p \mid q) \mid\!)$	$(\!\mid \alpha.p \mid\!) \mathbin{[\!]} (\!\mid \overline{\alpha}.q \mid\!) \rightarrow \tau.(p \mid q)$
$\dfrac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'}$	$(\!\mid \alpha.p + q \mid\!) \rightarrow (\!\mid \alpha.p \mid\!)$	$(\!\mid p + q \mid\!) = (\!\mid (\!\mid p \mid\!) \oplus (\!\mid q \mid\!) \mid\!)$ $(\!\mid \alpha.p \mid\!) \oplus \_ \rightarrow \alpha.p$
$\dfrac{p \xrightarrow{\alpha} p' \quad \alpha \notin \{l, \overline{l}\}}{p\backslash l \xrightarrow{\alpha} p'\backslash l}$	$(\!\mid \alpha.p\backslash l \mid\!) = (\!\mid \alpha.(p\backslash l) \mid\!)$ if $\alpha \notin \{l, \overline{l}\}$	– same –
$\dfrac{p \xrightarrow{\alpha} p'}{p[\phi] \xrightarrow{\phi(\alpha)} p'[\phi]}$	$(\!\mid \alpha.p[\phi] \mid\!) = (\!\mid \phi(\alpha).(p[\phi]) \mid\!)$	– same –
$\mathsf{fix}_i(\vec{x} = \vec{p}) \rightarrow p_i[\overrightarrow{\mathsf{fix}_i}(\vec{x} = \vec{p})/\vec{x}]$	$(\!\mid \mathsf{fix}_i(\vec{x} = \vec{p}) \mid\!) = (\!\mid p_i[\overrightarrow{\mathsf{fix}_i}(\vec{x} = \vec{p})/\vec{x}] \mid\!)$	– same –

Figure 5: SOS and SOS-inspired K definitions of CCS.

think of a process $\alpha.p$ as one that can transit with α in p. With this intuition in mind, one can easily see how the rule in the K interleaved definition of CCS correspond to those in its SOS definition. For example, the first K rule for parallel composition reads as follows: if process $(\alpha.p)|q$ is allowed to be processed (i.e., it is in the top of the computation structure), then it becomes the process emitting α followed by $p|q$. If one does not like to dually think of $\alpha.p$ as both a process and an action α followed by process p, then one can introduce a different notation for the later, say something like $\alpha \circ p$ or $\alpha \# p$, and modify the rules in Figure 5 accordingly. We prefer not to introduce a different notation. The K concurrent definition of CCS (right column in Figure 5) is similar in spirit to the CHAM definition of CCS in [5] and it differs from the K interleaved definition in that the constructs "|" and "+" are first translated into truly concurrent variants, i.e., ones enabling the computation to concurrently proceed in both argument processes. When one or both of the argument processes rewrite to the expected patterns, the actual irreversible transition is applied. Theorem 2 establishes the relationship between the SOS and the K definitions of CCS in Figure 5:

Theorem 2. *With the CCS definitions in Figure 5, the following properties hold:*

- *for any $p, p' \in P$, $K_{CCS}^{interleaved} \vdash (\!|p|\!) \rightarrow^* (\!|p'|\!)$ iff $K_{CCS}^{concurrent} \vdash (\!|p|\!) \rightarrow^* (\!|p'|\!)$;*
- *for any $p, p' \in P$, if $SOS_{CCS} \vdash p \xrightarrow{\alpha} p'$ then $K_{CCS}^{style} \vdash (\!|p|\!) \rightarrow^* (\!|\alpha.p'|\!)$ for any $style \in \{interleaved, concurrent\}$;*
- *if $K_{CCS}^{style} \vdash (\!|p|\!) \rightarrow^* (\!|\alpha.k|\!)$ for some $style \in \{interleaved, concurrent\}$, $p \in P$, and $k \in K$, then there is some $p' \in P$ such that $SOS_{CCS} \vdash p \xrightarrow{\alpha} p'$ and $K_{CCS}^{style} \vdash (\!|p'|\!) \rightarrow^* (\!|k|\!)$*

Theorem 2 tells us that, from an external point of view, an action α is possible on a process p under the SOS definition of CCS if and only if the same action α is possible on the process $(\!|p|\!)$ on any of the K definitions of CCS in Figure 5, that is, if and only if $K_{\mathrm{CCS}}^{style} \vdash (\!|p|\!) \rightarrow^* (\!|\alpha.k|\!)$ for some $k \in K$. Moreover, it actually says that the same holds true for any sequence of actions: sequence $\alpha_1\alpha_2 \ldots \alpha_n$ possible in SOS_{CCS} on a process p if and only if $\exists k_0, k_1, k_2, ..., k_n \in K$ such that $k_0 = p$ and $K_{\mathrm{CCS}}^{style} \vdash (\!|k_i|\!) \rightarrow^* (\!|\alpha_{i+1}.k_{i+1}|\!)$ for all $i \in \{0, 1, \ldots, n-1\}$.

Both K definitions in Figure 5 are inspired by the SOS definition of CCS. That made them more complicated and less intuitive that needed. Figure 6 shows two K definitions of CCS that follow the actual concurrency intuitions underlying CCS rather that its particular SOS definition. Let us explain this intuition informally: CCS processes are intended to be highly concurrent, in the sense that many sub-computations happen *at the same time* in different places of the system; processes can be put together in *parallel* "soups" as well as in *non-deterministic choice* "soups"; in such soups, processes can evolve *truly concurrently*; processes in parallel soups can also communicate with each other; in non-deterministic choice soups, a process which is found to produce an action can preempt and discard all the other processes in the soup. With this intuition, the definitions in Figure 6 become straightforward. Interestingly, besides giving a truly concurrent semantics to CCS, they are also more compact and simpler than the SOS definition, reducing to essentially one unconditional rewrite rule or equation per CCS construct. The following result states the relationship between the original SOS definition of CCS and the two K definitions in Figure 6. In short, that relationship is: every reduction step in SOS is also possible as a concurrent computation in these equivalent K definitions; conversely, any action that can be produced as a result of a concurrent computation in the K definitions can also be produced by a reduction step in the original SOS definition.

Theorem 3. *The K definitions of CCS in Figure 6 have the following properties ($p, p', q, p_1, p_2 \in P$):*

- *$KCCS_1 \vdash p \rightarrow^* \alpha.q$ iff $KCCS_2 \vdash \hat{p} \rightarrow^* \alpha.\hat{q}$, where $\hat{p}$ replaces each subprocess of the form $p_1|p_2$ by $(\!|p_1\ p_2|\!)_|$ and each process of the form $p_1 + p_2$ by $(\!|p_1\ p_2|\!)_+$ everywhere in p;*
- *If $SOS \vdash p \xrightarrow{\alpha} p'$ then $KCCS_1 \vdash p \rightarrow^* \alpha.p'$; and*
- *If $KCCS_1 \vdash p \rightarrow^* \alpha.q$ then there is some p' such that $SOS \vdash p \xrightarrow{\alpha} p'$ and $KCCS_1 \vdash p' \rightarrow^* q$.*

$K \ ::= \ 0 \mid Name \mid A.K \mid K[\varphi] \mid K\backslash L$ $\quad \mid \ \mathsf{Set}_{\mid}[K] \mid \mathsf{Set}_{+}[K]$	$K \ ::= \ 0 \mid Name \mid A.K \mid K[\varphi] \mid K\backslash L$ $\quad \mid \ (\!\mid \mathsf{Set}[K] \mid\!)_{\mid} \mid (\!\mid \mathsf{Set}[K] \mid\!)_{+}$
$\alpha.p \mid q \rightarrow \alpha.(p \mid q)$ $\alpha.p \mid \overline{\alpha}.q \rightarrow \tau.(p \mid q)$ $\alpha.p + q \rightarrow \alpha.p$ $(\alpha.p)[\varphi] = \varphi(\alpha).(p[\varphi])$ $(\alpha.p)\backslash l = \alpha.(p\backslash l) \quad \text{if } \alpha \notin \{l, \bar{l}\}$	$(\!\mid (\!\mid s \mid\!)_{\mid} \mid\!)_{\mid} = (\!\mid s \mid\!)_{\mid}$ $(\!\mid (\!\mid s \mid\!)_{+} \mid\!)_{+} = (\!\mid s \mid\!)_{+}$ $(\!\mid \alpha.p \mid\!)_{\mid} \rightarrow \alpha.(\!\mid p \mid\!)_{\mid}$ $(\!\mid \alpha.p \ \overline{\alpha}.q \mid\!)_{\mid} \rightarrow \tau.(\!\mid p \ q \mid\!)_{\mid}$ $(\!\mid \alpha.p \mid\!)_{+} \rightarrow \alpha.p$ $(\alpha.p)[\varphi] = \varphi(\alpha).(p[\varphi])$ $(\alpha.p)\backslash l = \alpha.(p\backslash l) \quad \text{if } \alpha \notin \{l, \bar{l}\}$

Figure 6: Truly concurrent K definitions of CCS: $KCCS_1$ (left column) and $KCCS_2$ (right column)

The advantage of $KCCS_1$ over $KCCS_2$ is that it does not change the syntax of the original CCS and it has precisely one unconditional rule or equation per CCS construct. It has two related drawbacks compared to $KCCS_2$: (1) its rules make use of top-level variables that are sets, so matching such rules becomes highly-nondeterministic (exponential matches in the number of processes in the soup) and challenging to implement, and (2) because of the above, if one does not choose the right matching then $KCCS_1$ may lead to many irrelevant small computational steps. For example, there are 2^n possible matches of the first rule on a process "$p_1|p_2|...|p_n|\alpha.p$", all of them leading eventually to "$\alpha.(p_1|p_2|...|p_n|p)$" after multiple applications of the same rule. If one could enforce "greedy matching" on set variables like q in the first and third rules of $KCCS_1$, then we would have no reservation to claim that $KCCS_1$ is the most elegant and concurrent definition of CCS that we are aware of. $KCCS_2$ achieves this desired greedy matching by wrapping soups into corresponding K-cells and then having the rule match the entire cell. However, two additional equations are necessary to dissolve nested cells of the same type (first two equations in $KCCS_2$).

Our K definitions of CCS discussed in this section were aimed at trace-equivalence with respect to the original definition of CCS. If our goal was to obtain tree-equivalent K definitions of CCS, then we would have had to follow an approach similar to that taken in the CHAM definition of CCS in [5], namely to replace our "irreversible" rules for action propagation through parallel composition and for external choice with appropriate "reversible" heating/cooling equations.

4.5 Defining Milner's π-Calculus

π-calculus is a process calculus originally introduced by Milner [26] as a continuation of his CCS. The aim of the π-calculus is to describe concurrent computations where the process configurations' may change during the computation and where processes can replicate and can communicate with each other via channels. Unlike in CCS, communication in π-calculus can result in a channel name being transmitted from one process to the other; the receiving process can use the received channel to send or receive data, and is also allowed to send it to other processes. Figure 7 shows the original definition of π-calculus (extended with the common summation) as well as an equivalent K definition. There are two constructs in π-calculus which bound names, namely the input channel $z(x).p$ and the new channel $\nu x.p$ constructs (here both binding x).

$\begin{array}{l} N ::= x \mid y \mid z \mid \ldots \text{ an infinite set of names} \\ A ::= N(N) \mid \overline{N}\langle N\rangle \mid \tau \\ P ::= M \mid P \vert P \mid \nu N.P \mid !P \\ M ::= 0 \mid A.P \mid M+M \end{array}$	$\begin{array}{l} N ::= x \mid y \mid z \mid \ldots \text{ an infinite set of names} \\ A ::= N(N) \mid \overline{N}\langle N\rangle \mid \tau \\ P ::= 0 \mid \llparenthesis \mathsf{Set}[A.P] \rrparenthesis_{+} \mid \llparenthesis \mathsf{Set}[P] \rrparenthesis_{\vert} \mid \nu N.P \mid !P \end{array}$
$\left\{ \begin{array}{l} 0+m \equiv m \\ m_1+m_2 \equiv m_2+m_1 \\ m_1+(m_2+m_3) \equiv (m_1+m_2)+m_3 \\ 0 \vert p \equiv p \\ p_1 \vert p_2 \equiv p_2 \vert p_1 \\ p_1 \vert (p_2 \vert p_3) \equiv (p_1 \vert p_2) \vert p_3 \end{array} \right\}$ $\begin{array}{l} \alpha\text{-equivalence of bound names} \\ \nu x.(p \vert q) \equiv p \vert \nu x.q \quad \text{when } x \text{ not free in } p \\ \nu x.0 \equiv 0 \\ \nu x.\nu y.p \equiv \nu y.\nu x.p \\ !p \equiv p \vert !p \end{array}$	$\left\{ \begin{array}{l} \llparenthesis \cdot \rrparenthesis_{+} = 0 \\ \llparenthesis \cdot \rrparenthesis_{\vert} = 0 \\ \llparenthesis \llparenthesis s \rrparenthesis_{\vert} \rrparenthesis_{\vert} = \llparenthesis s \rrparenthesis_{\vert} \end{array} \right\}$ $\begin{array}{l} \alpha\text{-equivalence of bound names, regarded as equation} \\ \nu x.(p \vert q) = p \vert \nu x.q \quad \text{when } x \text{ not free in } p \\ \nu x.0 = 0 \\ \nu x.\nu y.p = \nu y.\nu x.p \\ !p = \llparenthesis p \ !p \rrparenthesis_{\vert} \end{array}$
$\tau.p+m \to p$ $(x(y).p+m) \vert (\overline{x}\langle z\rangle.q+n) \to p[z/y] \vert q$ $\dfrac{p \to p'}{p \vert q \to p' \vert q}$ $\dfrac{p \to p'}{\nu x.p \to \nu x.p'}$ $\dfrac{p \to p',\ p \equiv q,\ p' \equiv q'}{q \to q'}$	$\llparenthesis \tau.p \rrparenthesis_{+} \to p$ $\llparenthesis \dfrac{\llparenthesis x(y).p \rrparenthesis_{+}}{p[z/y]} \dfrac{\llparenthesis \overline{x}\langle z\rangle.q \rrparenthesis_{+}}{q} \rrparenthesis_{\vert}$

Figure 7: π-Calculus: Original definition (left) and K definition (right)

4.6 Defining Hoare's CSP

4.7 Defining Cardelli's Ambient Calculus

4.8 Defining Agha's Actors

4.9 Defining Milner's EXP Language

Milner defined and proved the correctness of his $\mathscr{W}$ type inferencer in the context of a simple higher-order language that he called EXP [24]. Recall that $\mathscr{W}$ is the basis for the type checkers of all statically typed functional languages, including ML, OCAML, HASKELL, etc. EXP is a simple expression language containing the following constructs: lambda abstraction and application, conditional, fix point, and "let" and "letrec" binders. Here is the annotated K syntax of EXP (the application is strict in both its subexpressions, while the conditional is strict only in its first subexpression; also, let and letrec are defined as syntactic sugar — note that the desugaring of let is not general-purpose, in the sense that it is only allowed for the semantics of EXP, not for its polymorphic type system discussed in Section 4.11):

$\boxed{\text{K-Annotated Syntax of EXP}}$

$$\begin{array}{rcll}
\mathit{Name} & ::= & \text{standard identifiers} & \\
\mathit{Exp} & ::= & \mathit{Name} \mid \text{... add basic values (Booleans, integers, etc.)} & \\
& \mid & \lambda\, \mathit{Name} \,.\, \mathit{Exp} & \\
& \mid & \mathit{Exp}\ \mathit{Exp} & [\mathit{strict}] \\
& \mid & \mu\, \mathit{Name} \,.\, \mathit{Exp} & \\
& \mid & \textsf{if } \mathit{Exp} \textsf{ then } \mathit{Exp} \textsf{ else } \mathit{Exp} & [\mathit{strict}(1)] \\
& \mid & \textsf{let } \mathit{Name} = \mathit{Exp} \textsf{ in } \mathit{Exp} & [\textsf{let } x = e \textsf{ in } e' = (\lambda x.e')\, e] \\
& \mid & \textsf{letrec } \mathit{Name}\, \mathit{Name} = \mathit{Exp} \textsf{ in } \mathit{Exp} & [\textsf{letrec } f\ x = e \textsf{ in } e' = \textsf{let } f = \mu f.(\lambda x.e) \textsf{ in } e']
\end{array}$$

As usual in our K definitions, the K computations "swallow" the syntax of the language. We next give two K semantic definitions of EXP, one straightforward based on substitution like the definitions of λ-calculi in Section 4.2, and another based on environments and closures. Let us first define computations, values and configurations which are common to both definitions:

$$\begin{array}{rcl}
\mathit{Val} & ::= & \text{(integers, booleans, other basic values)} \\
\mathit{KResult} & ::= & \mathit{Val} \\
K & ::= & \mathit{Exp} \mid \mathit{KResult} \mid \textsf{List}^{\cdot}_{\curvearrowright}[K] \\
\mathit{Config} & ::= & \mathit{Val} \mid [\![K]\!] \mid (\!|K|\!)_k \mid (\!|\textsf{Set}[\mathit{Config}]|\!)_\top
\end{array}$$

The definitions below may extend the above syntax as needed.

Substitution-based definition of EXP. The definition below is straightforward:

$\boxed{K^{subst}_{\text{EXP}}\text{: Substitution-based K Definition of EXP}}$ $e, e_1, e_2 \in \mathit{Exp},\ v \in \mathit{Val},\ x \in \mathit{Name}$

$\mathit{Val} ::= \text{...} \mid \lambda\, \mathit{Name} \,.\, \mathit{Exp}$

$[\![e]\!] = (\!|(\!|e|\!)_k|\!)_\top$
$(\!|(\!|v|\!)_k|\!)_\top = v$

$(\!|(\lambda x.e)\, v\rangle_k \rightarrow (\!|e[v/x]\rangle_k$
$(\!|\mu x.e\rangle_k \rightarrow (\!|e[\mu x.e/x]\rangle_k$
$\textsf{if true then } e_1 \textsf{ else } e_2 \rightarrow e_1$
$\textsf{if false then } e_1 \textsf{ else } e_2 \rightarrow e_2$

To test the semantics, one can now execute programs like the factorial:

$[\![$

```
letrec f x = if x <= 0 then 1 else x * f(x - 1)
in f (f 5)
```

$]\!]$

The definition above yields, when run in Maude, the following result in 12,312 rewrites and about 0.2 seconds:

```
66895029134491270575881180540903725867527463331380298102956713523016335572449629893668741652719849
8130815763789321409055253440858940812185989848111438965000596496052125696000000000000000000000000000
```

Environment-based definition of EXP. The definition below is based on environments and closures:

$\boxed{K^{env}_{\text{EXP}}\text{: Environment-based K Definition of EXP}}$ $\quad e, e_1, e_2 \in \mathit{Exp},\ v \in \mathit{Val},\ x \in \mathit{Name}$

$\mathit{Val} ::= \ldots \mid \mathit{closure}(\mathit{Name}, K, \mathit{Env})$
$K ::= \ldots \mid \mathit{restore}(\mathit{Env})$
$\mathit{Config} ::= \ldots \mid (\!|\mathit{Env}|\!)_{env} \mid (\!|\mathit{Store}|\!)_{store}$

$[\![e]\!] = (\!|(\!|e|\!)_k\ (\!|\cdot|\!)_{env}\ (\!|\cdot|\!)_{store}|\!)_\top$ (initialize the configuration)
$\langle\!|(\!|v|\!)_k|\!\rangle_\top = v$ (dissolve the configuration)
$(\!|v \curvearrowright \underset{\cdot}{\underline{\mathit{restore}(\rho)}}|\!\rangle_k\ (\!|\underset{\rho}{\underline{\_}}|\!)_{env}$ (restore the environment)

$(\!|\underset{\sigma[\rho[x]]}{\underline{\quad x \quad}}|\!\rangle_k\ (\!|\rho|\!)_{env}\ (\!|\sigma|\!)_{store}$

$(\!|\underset{\mathit{closure}(x, e, \rho)}{\underline{\quad \lambda x.e \quad}}|\!\rangle_k\ (\!|\rho|\!)_{env}$

$(\!|\underset{e \curvearrowright \mathit{restore}(\rho')}{\underline{\mathit{closure}(x, e, \rho)\ v}}|\!\rangle_k\ (\!|\underset{\rho[l/x]}{\underline{\quad \rho' \quad}}|\!)_{env}\ (\!|\underset{\sigma[v/l]}{\underline{\quad \sigma \quad}}|\!)_{store}$ (where l is a fresh location)

$(\!|\underset{e \curvearrowright \mathit{restore}(\rho)}{\underline{\quad \mu x.e \quad}}|\!\rangle_k\ (\!|\underset{\rho[l/x]}{\underline{\quad \rho \quad}}|\!)_{env}$ (where l is a fresh location)

if true then e_1 else $e_2 \to e_1$
if false then e_1 else $e_2 \to e_2$

The definition of the auxiliary computation item *restore* is so useful in language definitions, that, in our Maude implementation of K, it is part of the prelude file which is included in all K definitions. These and many others can be found in Appendix F.

There are several things to note and/or prove about the definitions K^{subst}_{EXP} and K^{env}_{EXP} above. First and most importantly, they are confluent and equivalent. Second, the environment-based definition suffers from tail-recursion problems; indeed, a tail recursive program may unboundedly grow the computation structure. Fortunately and interestingly, that has a very simple fix. All one needs to do is to add the following equation to K^{env}_{EXP}, which can actually be proved correct (in other words, adding this equation is like adding a lemma, not an axiom, so it does not change the intended semantics, it is just an optimization):

$$\mathit{restore}(\rho) \curvearrowright \mathit{restore}(\rho') = \mathit{restore}(\rho') \quad \text{for all } \rho, \rho' \in \mathit{Env}.$$

Finally, it is an interesting exercise to prove that the direct definition of letrec as given in Appendix **??** for the functional language FUN is actually semantically equivalent to what we get by desugaring it as we did in this section and then using the semantics of μ.

4.10 Defining Robinson's Unification

Unification is different many places in computer science. Sections **??** and 4.11 show two possible uses of unification, the first in the context of defining a simple logic programming language and the second in the context of defining a type inferencer. Unification is straightforward to define equationally using set matching, in such a way that, when replacing the equations by rewrite rules, one gets an implementation of unification at no additional cost. For simplicity in exposition, we here assume a mono-sorted syntax, that is, there is only one syntactic category for terms, *Term*, that all operation names are distinct, and that given terms are well-formed (so we do not bother parsing/checking them). In a many-sorted setting one can rename overloaded operator names so that one can reduce the many-sorted unification problem to a mono-sorted one. An input to the unification problem consists of a set of parametric equations, which are pairs of terms over variables. We also allow equations over lists of terms, with the meaning that the listed terms must be component-wise equal. Here is the formal definition of terms and equations:

$\boxed{\text{Term and equations syntax}}$

$$\begin{array}{rcl} \mathit{Op} & ::= & \sigma \mid \tau \mid \ldots \text{ (any number of operator names)} \\ \mathit{Var} & ::= & x \mid y \mid \ldots \text{ (any number of variable names)} \\ \mathit{Term} & ::= & \mathit{Var} \mid \mathit{Op}(\mathsf{List}[\mathit{Term}]) \\ \mathit{Eqns} & ::= & \mathsf{List}[\mathit{Term}] \equiv \mathsf{List}[\mathit{Term}] \mid \mathsf{Set}[\mathit{Eqns}] \end{array}$$

The following equational properties are all obviously valid:

$\boxed{\text{Unification}}$ $t, t' \in \mathit{Term}$, $tl, tl' \in \mathsf{List}[\mathit{Term}]$, $\sigma \in \mathit{Op}$, $x \in \mathit{Var}$,

(1) $\underset{\cdot}{\underline{tl \equiv tl}}$

(2) $\underset{tl \equiv tl'}{\underline{\sigma(tl) \equiv \sigma(tl')}}$

(3) $\underset{(t \equiv t')\ (tl \equiv tl')}{\underline{t, tl \equiv t', tl'}}$

(4) $\underset{x \equiv t}{\underline{t \equiv x}}$ when $t \notin \mathit{Var}$

(5) $(x \equiv t)\ (\underset{t}{\underline{x}} \equiv t')$ when $t, t' \neq x$

(6) $(x \equiv t)\ (x' \equiv \underset{t'[t/x]}{\underline{t'}})$ when $x \neq x',\ x \neq t,\ x' \neq t',$ and $x \notin \mathit{vars}(t), x \in \mathit{vars}(t')$

The first equation above eliminates non-informative term constraints. The second and third equations distribute constraints over non-variable terms to constraints over their argument subterms. The fourth swaps the two terms involved in each constraint so that each constraint containing a variable term has the term variables as its lhs. The fifth ensures that, eventually, no two term constraints have the same lhs variable. Finally, the sixth equation canonizes the constrains in order to give a substitution; note that the side conditions of the sixth equation guarantee that it cannot be applied forever, thus leading to non-termination. As expected, the equations above take a set of equational term constraints and eventually produce a most general unifier for them:

Theorem 4. *Let $\gamma \in \mathit{Eqns}$ be a set of equational term constraints and let us regard the six equations above as rewrite rules. Then:*

- *The six-rule rewrite system above terminates (modulo AC); let $\theta \in Eqns$ be the normal form of γ;*
- *γ is unifiable iff θ contains only pairs of the form $x \equiv t$, where $x \notin vars(t)$; if that is the case, then we identify θ with the implicit substitution that it comprises, that is, $\theta(x) = t$ when there is some type equality $x \equiv t$ in θ, and $\theta(x) = x$ when there is no type equality of the form $x \equiv t$ in θ;*
- *If γ is unifiable then θ is idempotent (i.e., $\theta \circ \theta = \theta$) and is a most general unifier of γ.*

Therefore, the six equations above give us a simple rewriting procedure for unification. Experiments using the Maude rewrite engine show that it may also quite efficient (the type inferencer in Section 4.11 is comparable in performance to state of the art implementations of type inference in languages like OCAML). The tests "$x \in vars(t)$" and "$x \in vars(t')$" in the sixth equation of the unification definition and in the second item of the theorem above, respectively, may be a bottle-neck in the performance of an implementation of the unification definition above. If the underlying rewrite engine provides support for memoization, one may memoize such membership tests, so that one pays the price of traversing a term for its variables only once (though one would need to pay the price of memo table lookup; experiments are probably needed for each application to decide whether memoization is useful or not). Moreover, tests "$x \notin vars(t)$" saying whether equational constraints "$x \equiv t$" are circular or not can be avoided on rewrite engines providing support for subsorting and sort membershipping, like Maude: one can subsort once and for all each equational constraint of the form $x \equiv t$ with $x \notin vars(t)$ to a "proper" constraint, and then allow only proper constraints in the sort *Eqns* (the improper ones remain part of the "kind", or "error supersort" [*Eqns*]).

If $\gamma \in Eqns$ is a set of equational constraints and $t \in Term$ is some term, then we let $\gamma[t]$ denote $\theta(t)$; if γ is not unifiable, then $\gamma[t]$ is some error term (in the kind [*Term*] when using a rewrite engine like Maude with subsorting or kinding). A simple way to calculate $\gamma[t]$ using the already existing machinery of unification defined above is:

1. Add an equational constraint $freshVar \equiv t$ to γ, where $freshVar$ is a variable that does not occur in γ; let γ' be the new set of constraints;
2. Let the unification procedure solve the constraints in γ';
3. If γ' is unifiable, let θ' be the resulting substitution, like in the second item in the theorem above;
4. Let θ be the substitution obtained by restricting θ' to all variables except $freshVar$.
5. Let $\gamma[t]$ be the term $\theta'(freshVar)$.

Then the following holds:

Corollary 1. *With the notation above,*

1. *γ' is unifiable iff γ is unifiable;*
2. *If the above holds, then θ is the most general unifier of γ;*
3. *$\gamma[t] = \theta(t)$.*

4.11 Defining Milner's $\mathscr{W}$ Polymorphic Type Inferencer

We next define the $\mathscr{W}$ type inferencer in [24] using the same K approach. The overall idea of our definition of W is that EXP programs are iteratively rewritten, i.e., "evaluated" to types; in particular, the typing process is started by rewriting all integers and booleans to types *int* and *bool*, respectively. The K annotated syntax below changes the conditional to strict in all its arguments, makes let strict in its second argument, and desugars letrec like in the definition of EXP in Section 4.9:

$\boxed{\text{K-Annotated Syntax of EXP for W}}$

$$\begin{array}{rcll}
\mathit{Name} & ::= & \text{standard identifiers} & \\
\mathit{Exp} & ::= & \mathit{Name} \mid \text{... add basic values (Booleans, integers, etc.)} & \\
& \mid & \lambda\, \mathit{Name}\,.\, \mathit{Exp} & \\
& \mid & \mathit{Exp}\ \mathit{Exp} & [\mathit{strict}] \\
& \mid & \mu\, \mathit{Name}\,.\, \mathit{Exp} & \\
& \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Exp}\ \mathsf{else}\ \mathit{Exp} & [\mathit{strict}] \\
& \mid & \mathsf{let}\ \mathit{Name} = \mathit{Exp}\ \mathsf{in}\ \mathit{Exp} & [\mathit{strict}(2)] \\
& \mid & \mathsf{letrec}\ \mathit{Name}\ \mathit{Name} = \mathit{Exp}\ \mathsf{in}\ \mathit{Exp} & [\mathsf{letrec}\ f\ x = e\ \mathsf{in}\ e' = \mathsf{let}\ f = \mu f.(\lambda x.e)\ \mathsf{in}\ e']
\end{array}$$

We consider the usual syntax for (implicitly universal) parametric types:

$$\begin{array}{rcl}
\mathit{TypeVar} & ::= & \text{type variables} \\
\mathit{Type} & ::= & \mathit{TypeVar} \mid \mathit{int} \mid \mathit{bool} \mid \mathit{Type} \to \mathit{Type}
\end{array}$$

At no but notational expense, we can regard types as terms like in Section 4.10, where Var is replaced by $\mathit{TypeVar}$ and where Op contains three operations: int and bool taking zero arguments, and $\_ \to \_$ taking two arguments. This way, we take the liberty to use the unification procedure in Section 4.10 with no modification; moreover, if one wants to extend the results and add new type constructs, then all one needs to do is to add new operator names (type constructs) to Op (no additional equations for unification needed).

We next give two definitions for $\mathscr{W}$, one substitution-based and another environment-based. Both start by "swallowing" the syntax and the types into computations K, and then continue by iteratively rewriting the computation into a type, accumulating and solving the type constraints as needed. For both definitions we need some additional computation infrastructure:

- A computation item $\mathit{let}(\mathit{Type})$ holding a polymorphic type that needs to be concretized when encountered for processing;
- A computation item $K \to K$ extending the function type construct $\mathit{Type} \to \mathit{Type}$ that is strict in its two arguments (only strictness in the second matters, but for the sake of generality and simplicity we keep it strict in both arguments); once the two arguments are processed into types, the resulting computation is, of course, a type (function type);
- A computation item $K \equiv K$ which is also strict in its arguments (like above, only strictness in the second argument is necessary here); once the two arguments are processed, the resulting type equality is added to the set of type equational constraints to be solved.

Here is the syntax of computations and of configurations that is common to both definitions:

$$\begin{array}{rcl}
\mathit{KResult} & ::= & \mathit{Type} \\
K & ::= & \mathit{KResult} \mid \mathit{let}(\mathit{Type}) \mid K \to K\ [\mathit{strict}] \mid K \equiv K\ [\mathit{strict}] \mid \mathsf{List}^{\cdot}_{\curvearrowright}[K] \\
\mathit{Config} & ::= & \mathit{Type} \mid [\![K]\!] \mid (\!|K|\!)_k \mid (\!|\mathit{Eqns}|\!)_{\mathit{eqns}} \mid (\!|\mathsf{Set}[\mathit{Config}]|\!)_\top
\end{array}$$

and here is the K equation needed to move the type equality resulting from processing $K \to K$ from the computation structure into the set of equational constraints:

$$(\!|\underset{\cdot}{\underline{\mathit{eqn}}}\!\!|)_k\ \langle\!|\underset{\mathit{eqn}}{\underline{\ \cdot\ }}|\!\rangle_{\mathit{eqns}} \qquad \begin{array}{l}\text{where } \mathit{eqn} \in \mathit{Eqn} \\ \text{(this general purpose equation is part of prelude)}\end{array}$$

Each of the two definitions of $\mathscr{W}$ discussed in the sequel will enrich computations and/or configurations with specific constructs. We next present the two definitions, the first based on substitutions and the second on environments, by first presenting the rules which are independent upon the particular style (substitution or environment) adopted and, therefore, are common to both. These rules are straightforward and self explanatory. All they do is to initiate the process of rewriting programs into types by rewriting the basic values into their corresponding types, and then propagate the typing policy through the simple constructs accumulating at the same time additional type constraints:

$\boxed{\text{K Definition of W: simple language constructs}}$ $\quad t, t_1, t_2 \in \mathit{Type}$

$i \to \mathit{int},\ \mathit{true} \to \mathit{bool},\ \mathit{false} \to \mathit{bool}$ (and similarly for all the other basic values)

$\llparenthesis \underset{\mathit{int}}{\underline{t_1 + t_2}} \rrparenthesis_k \ \langle\!| \underset{t_1 \equiv \mathit{int},\ t_2 \equiv \mathit{int}}{\underline{\qquad\cdot\qquad}} \rrparenthesis_{\mathit{eqns}}$ (and similarly for all other standard operators)

$\llparenthesis \underset{\mathit{tvar}}{\underline{t_1\ t_2}} \rrparenthesis_k \ \langle\!| \underset{t_1 \equiv t_2 \to \mathit{tvar}}{\underline{\qquad\cdot\qquad}} \rrparenthesis_{\mathit{eqns}}$ where *tvar* is a fresh type variable

$\llparenthesis \underset{t_1}{\underline{\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2}} \rrparenthesis_k \ \langle\!| \underset{t \equiv \mathit{bool},\ t_1 \equiv t_2}{\underline{\qquad\cdot\qquad}} \rrparenthesis_{\mathit{eqns}}$

The rules above assume that the $\llparenthesis ... \rrparenthesis_k$ and $\llparenthesis ... \rrparenthesis_{\mathit{eqns}}$ cells will both be part of the same cell in the configuration. This will be the case in both subsequent definitions. Due to the strictness attributes, in the above we assumed that the corresponding arguments of the language constructs (in which these constructs were defined strict) have already been "evaluated" to their types and the corresponding type constraints have been propagated.

Below is our first K definition of $\mathscr{W}$, based on substitutions. A "path" configuration item is necessary to store the set of type variables corresponding to the variables bound with λ or μ in whose scope the expression that is being currently processed is. The path is easily maintained by adding to it the fresh type variable corresponding to the binding variable whenever a λ or a μ expression is processed, making sure that it is removed once the processing of the λ or μ expression is completed:

$\boxed{\text{Substitution-based K Definition of W}}$ $\quad t, t_1, t_2 \in \mathit{Type}$

$K ::= ... \mid \overline{\mathit{TypeVar}}$
$\mathit{Config} ::= ... \mid \llparenthesis \mathsf{Set}[\mathit{TypeVar}] \rrparenthesis_{\mathit{path}}$

$[\![e]\!] = \llparenthesis \llparenthesis e \rrparenthesis_k\ \llparenthesis \cdot \rrparenthesis_{\mathit{path}}\ \llparenthesis \cdot \rrparenthesis_{\mathit{eqns}} \rrparenthesis_\top$ (initialize the configuration)
$\langle\!| \llparenthesis t \rrparenthesis_k\ \llparenthesis \gamma \rrparenthesis_{\mathit{eqns}} |\!\rangle_\top = \gamma[t]$ (dissolve the configuration when done)
$\llparenthesis t \curvearrowright \underset{\cdot}{\underline{\overline{\mathit{tvar}}}} \rrparenthesis_k\ \langle\!| \underset{\cdot}{\underline{\mathit{tvar}}} |\!\rangle_{\mathit{path}}$ (remove *tvar* from the path)

$\lambda x.e \to (\mathit{tvar} \to e[\mathit{tvar}/x]) \curvearrowright \overline{\mathit{tvar}}$ where *tvar* is a fresh type variable

$\mu x.e \to (\mathit{tvar} \equiv e[\mathit{tvar}/x]) \curvearrowright \overline{\mathit{tvar}}$ where *tvar* is a fresh type variable

$\mathsf{let}\ x = t\ \mathsf{in}\ e \to e[\mathit{let}(t)/x]$

$\llparenthesis \underset{(\gamma[t])[\mathit{tvars}'/\mathit{tvars}]}{\underline{\qquad \mathit{let}(t) \qquad}} \rrparenthesis_k\ \llparenthesis \eta \rrparenthesis_{\mathit{path}}\ \llparenthesis \gamma \rrparenthesis_{\mathit{eqns}}$ where $\mathit{tvars} = \mathit{vars}(\gamma[t]) - \eta$ and tvars' are $|\mathit{tvars}|$ fresh type variables

When the let-bound "*let*" types are encountered, their unconstrained and path-unbound type variables (i.e., their "universal" type variables), are instantiated with fresh type variables. Note that the equational type constraints are required to be solved whenever a let-bound type is encountered, because one needs to know how many fresh type variables need to be generated to replace the variables in which the let-bound type is polymorphic (the last rule above). Other than that, there is no requirement on when or if to solve the type constraints; in particular, they can be solved concurrently and while the program is still being processed. One particularly interesting aspect of the definition above is that we allow λ, μ, and let expressions to be rewritten at any moment and at any place (not necessarily in "context"), provided that one has a mechanism to generate fresh type variables there. In concrete and "pure" implementations of the $\mathscr{W}$ procedure following our technique above, one may want to restrict the applications of these rules to the top of the computation, where one can easily generate a fresh type variable (e.g., one may add a new "next fresh variable" cell at the top level of the configuration), i.e., to replace them with the following:

$(\!|\lambda x.e|\!)_k \rightarrow (\!|(tvar \rightarrow e[tvar/x]) \curvearrowright \overline{tvar}|\!)_k$ where *tvar* is a fresh type variable

$(\!|\mu x.e|\!)_k \rightarrow (\!|(tvar \equiv e[tvar/x]) \curvearrowright \overline{tvar}|\!)_k$ where *tvar* is a fresh type variable

$(\!|\mathsf{let}\ x = t\ \mathsf{in}\ e|\!)_k \rightarrow (\!|e[let(t)/x]|\!)_k$

One can show that the two K theories above are equivalent modulo renaming of type variables (that is because our first one is confluent modulo renaming of variables, so one can reorder the applications of the three rule to apply only at the top of the computation).

We believe that the K definition above of W is as simple, if not simpler, to understand as the original W procedure proposed by Milner in [24]. However, note that the procedure in [24] is an *algorithm*, almost an *implementation*, rather than a formal, logic-based definition!

We next give our second K definition of W. The main difference between our first definition and the second is that the second uses a type environment to map types to binding names instead of a substitution. All we have to do is to replace the $(\!|...|\!)_{path}$ cell above storing a set of type variables with a $(\!|...|\!)_{tenv}$ cell storing a type environment (i.e., set of pairs name/type); the later will already contain the set of type variables visible in the current scope: let $tvars(\eta)$ be the type variables bound to names in the type environment η (the type environment may also contain let-bound types bound to names, which we do not consider when computing $tvars(\eta)$). Then our second definition of $\mathscr{W}$ is as follows:

$\boxed{\text{Environment-based K Definition of W}}$ $\quad t, t_1, t_2 \in Type$

$$\begin{array}{rcl} K & ::= & ...\mid restore(\mathsf{Set}[Name \times Type]) \\ Config & ::= & ...\mid (\!|\mathsf{Set}[Name \times Type]|\!)_{tenv} \end{array}$$

$[\![e]\!] = (\!|(\!|e|\!)_k\ (\!|\cdot|\!)_{tenv}\ (\!|\cdot|\!)_{eqns}|\!)_\top$ (initialize the configuration)

$\langle\!|(\!|t|\!)_k\ (\!|\gamma|\!)_{eqns}|\!\rangle_\top = \gamma[t]$ (dissolve the configuration)

$(\!|t \curvearrowright \underset{\cdot}{\underline{restore(\eta)}}|\!)_k\ (\!|\underset{\eta}{\underline{\_}}|\!)_{tenv}$ (restore the type environment)

$(\!|\underset{(\gamma[t])[tvars'/tvars]}{\underline{\qquad x \qquad}}|\!)_k\ (\!|\eta|\!)_{tenv}\ (\!|\gamma|\!)_{eqns}$ when $\eta[x] = let(t)$, $tvars = vars(\gamma[t]) - vars(\eta)$ and $tvars'$ are $|tvars|$ fresh type variables

$(\!|\underset{\eta[x]}{\underline{x}}|\!)_k\ (\!|\eta|\!)_{tenv}$ when $\eta[x] \neq let(t)$

$(\!|\underset{(tvar \rightarrow e) \curvearrowright restore(\eta)}{\underline{\qquad \lambda x.e \qquad}}|\!)_k\ (\!|\underset{\eta[tvar/x]}{\underline{\quad \eta \quad}}|\!)_{tenv}$ where *tvar* is a fresh type variable

$(\!|\underset{tvar \equiv e \curvearrowright restore(\eta)}{\underline{\qquad \mu x.e \qquad}}|\!)_k\ (\!|\underset{\eta[tvar/x]}{\underline{\quad \eta \quad}}|\!)_{tenv}$ where *tvar* is a fresh type variable

$(\!|\underset{e \curvearrowright restore(\eta)}{\underline{\mathsf{let}\ x = t\ \mathsf{in}\ e}}|\!)_k\ (\!|\underset{\eta[let(t)]/x}{\underline{\quad \eta \quad}}|\!)_{env}$

Both our K definitions above are nothing but ordinary rewrite logic theories, same as the formal definition of EXP itself. That should not, and indeed it does not, mean that our K definitions, when executed, must necessarily be slower than an actual implementation of W. Experiments using the second definition above executed in Maude show that our K definition above of W is comparable or even outperforms state of the art implementations of type inference in conventional functional languages. For example, on our experiments it was faster than the type inferencers of SML and Haskell, and only about twice slower than that of OCAML.

Concretely, the program (which is polymorphic in $2^n + 1$ type variables!)

$$\begin{array}{l}
\mathsf{let}\ f_0 = \lambda x.\lambda y.x\ \mathsf{in} \\
\quad \mathsf{let}\ f_1 = \lambda x.f_0(f_0 x)\ \mathsf{in} \\
\quad\quad \mathsf{let}\ f_2 = \lambda x.f_1(f_1 x)\ \mathsf{in} \\
\quad\quad\quad \ldots \\
\quad\quad\quad\quad \mathsf{let}\ f_n = \lambda x.f_{n-1}(f_{n-1} x)\ \mathsf{in}\ f_n
\end{array}$$

takes the following time/space resources to be type checked using OCAML, Haskell, SML, and our K definition executed in Maude, respectively:

-	n = 10		n = 12		n = 13		n = 14	
OCAML (version 3.09.3)	0.6s	3M	8.1s	5M	31.2s	8M	120.6s	13M
Haskell (ghci version 6.7.20070312)	2.7s	25M	42.4s	31M	207.8s	38 M	1177s	61M
SML (version 110.59)	5s	76M	110.5s	324M	2129.2s	950M	out of M	
W in K/Maude2.2 with memo	1.2s	17M	18.9s	65M	78s	191M	304s	653M
W in K/Maude2.2 without memo	2.2s	10M	22.4s	47M	81s	152M	305s	563M

The experiments above have been conducted on a 3.4GHz/2GB Linux machine. Only the user time has been recorded. Except for SML, the user time was very close to the real time; for SML, the real time was 30% larger than the user time. These ratios appear to scale and be preserved for other programs, too. Moreover, extensions of the type system with lists, products, side effects (through referencing, dereferencing and assignment) and weak polymorphism did not add any noticeable slowdown. Therefore, our K definitions surprisingly yield quite realistic *implementations* of type checkers/inferencers when executed on efficient rewrite engines! While calculating the numbers above, Maude run at an average of 3 million rewrites per second. In Maude, memoization can be enabled by adding the attribute "`[memo]`" to operations whose results one wants memoized; in our case, we only experimented with memoizing the "built-in" operation `vars : Type -> Set{TypeVar}` in `k-prelude.maude`, which extracts the set of type variables that occur in a type. Memoization appears to pay off when the polymorphic types are small, which is typically the case.

Our Maude "implementation" of an extension[4] of the K definition of W above has about 30 lines of code. How is it be possible that a formal definition of a type system that one can write in 30 lines of code can be executed *as is* more efficiently than well-engineered implementations of the same type system in widely used programming languages? We think that the answer to this question involves at least two aspects. On the one hand, Maude, despite its generality, is itself a well-engineered rewrite engine implementing state-of-the-art AC matching and term indexing algorithms. On the other hand, our K definition makes intensive use of what Maude is very good at, namely AC matching. For example, note the fourth rule in our rewrite definition of unification[5] that precedes Theorem 4: the type variable t_v appears twice in the lhs of the rule, once in each of the two type equalities involved. Maude will therefore need to search and then index for two type equalities in the set of type constraints which share the same type variable. Similarly, but even more complicatedly, the fifth rule involves two type equalities, the second containing in its t' some occurrence of the type variable t_v that appears in the first. Without appropriate indexing to avoid rematching of rules, which is what Maude does well, such operations can be very expensive. Moreover, note that our type constraints can be "solved" incrementally (by applying the five unification rewrite rules), as generated, into a most general substitution; incremental solving of the type constraints can have a significant impact on the complexity of unification as we defined it, and Maude indeed does that (one can see it by tracing/logging Maude's rewrite steps).

[4]With conventional arithmetic and boolean operators added for writing and testing our definition on meaningful programs
[5]Type unification is "built-in" in K: it is defined as shown above in `k-prelude.maude`.

4.12 Defining Plotkin's PCF language

5 Comparing K With Other Formalisms

Since the various language definitional formalisms were all designed to solve more or less the same problem, it is not surprising that there are deep relationships between almost any two such formalisms. Some of these relationships have been spelled out in the literature. In this section we briefly discuss some of the more popular language definitional formalisms, and then show how each of them can be captured as a *methodological fragment* of rewriting logic and even of K. Since rewriting logic is a real computational logical meta-framework that does not impose on the language designer any particular definitional style, it captures very naturally, with almost zero representational distance, the various other formalisms. Since K was not intended to be such a generic meta-framework, but rather a particular definitional style that makes unrestricted use of the full strength of (unconditional) rewriting logic when defining programing languages or calculi, it is not surprising that some of the translations into K are less straight-forward. With the exception of the chemical abstract machine, which, like rewriting logic and K explicitly aims at avoiding the obedience of structural operational semantics to syntax and interleaving semantics, the complexity of the subsequent translations of formalisms stays in "inhibiting" the otherwise unrestricted computational mechanism of rewriting logic, so that only one rewrite step is applied at a time, and only in a context that is allowed by the evaluation strategy of the defined language constructs.

For simplicity, in each of the formalisms discussed in this section we limit ourselves to ground instances of rule schematas; in other words, we assume that each rule in each formalism, which may typically be restricted by side conditions, was replaced by all its concrete instances (typically a recursively enumerable set), one per variable instances satisfying the side conditions of the rule. One can also translate side conditions in the original rules into conditions in the resulting rewriting logic rules and/or equations. In most cases, in particular when each rule has precisely one correspondent rule in rewriting logic or K (like in the translation of context reduction into K, for example), the side conditions simply carry over unchanged. In some other cases, such as when a conditional rule in the original formalism translates into more than one rule or equation in the corresponding rewrite logic theory and different transitions in the original condition may refer to the same variable (like in the translation of general-purpose big-step or small-step SOS into K), one must be careful to properly propagate the accumulated substitution. Techniques for systematic elimination of conditional rules in term rewriting, applicable also to rewriting logic, can be used for such propagations of substitutions; these include, for example, the following works (in chronological order) [17, 30, 1, 35].

Each of the embeddings of definitional formalisms into rewriting logic and K discussed in this section is *faithful*, in the sense that any language definition in any of these formalisms, say $\mathcal{L}$, can be regarded as a rewrite logic theory, say $R_{\mathcal{L}}$, in such a way that there is a *one-to-one computational equivalence* between reductions using $\mathcal{L}$ and rewrites using $R_{\mathcal{L}}$. Note that these are significantly stronger results than encoding, or implementing, a framework into another framework: *$R_{\mathcal{L}}$ is $\mathcal{L}$, not an encoding of it*, the only difference between the two being insignificant notational/syntactic conventions. This is totally different from encoding $\mathcal{L}$ on a Turing machine or in a λ-calculus, for example, because such encodings would not preserve the intended computational granularity of $\mathcal{L}$'s reductions (correct encodings would only preserve the "relational behavior" of $\mathcal{L}$: whatever is reduced with $\mathcal{L}$ can also be reduced, in any number of steps, with its encodings). When a definitional framework is faithfully embedded into another one then the former becomes a *methodological fragment* of the latter. Besides showing the generality and versatility of rewriting logic and K, there are two additional reasons for showing that other definitional frameworks fall as (mechanically derived) methodological fragments of rewriting logic:

- Existing efficient tool support for rewriting logic, such as execution engines, theorem provers, state-space analyses (including an LTL model checker and a breath-first search for safety property violations), etc., can be used for language definitions in definitional styles that lack tool support; and
- Comparisons and limitation analysis of language definitional frameworks become possible and rigorous, because there is a uniform (meta-)framework, called "ecumenical" in [36], in which all of them co-exist without loosing any of their particularities (both on the positive and on the negative sides).

It is important to realize that each of these faithful embeddings has all the good properties of the embedded framework, but also all its problems and limitations. In other words, one should not expect that the rewriting logic embedding of a framework magically overcomes its problems. For example, since all the rules and equations in our embeddings of SOS or context reduction apply only at the top of the term to reduce, just like in SOS or context reduction, the true concurrency semantics strength of rewriting logic and K is drastically inhibited and reduced to interleaving semantics, just like in SOS or context reduction. Also, the embedding of SOS is just as non-modular as SOS. Considering the strengths and elegance of K in its full generality, we cannot advocate any of its methodological fragments discussed in this section as ideal language definitional formalisms. These may, nevertheless, be useful when existing definitions are already available and lack tool support; in that case, K, through its generic tool support, provides specialized tool support for the embedded formalisms.

While it is hard to argue against the usefulness of the first item above (derived tool support), we are aware of the fact that proponents of existing definitional frameworks or styles may disagree with our conclusions drawn from the second item (objective comparison and limitation analysis). For example, there are proponents of existing formalisms who would rather attack existing and sometimes desired language features (such as, e.g., call/cc or true concurrency) declaring them "obscure" or "pathological", rather then accept that their favorite definitional formalism cannot support them. Since in this paper we are on a quest for an "ideal" language definitional framework, we shall discuss limitations of existing formalisms without any attachment to any of them, assuming that all existing language features are equally desirable.

5.1 Structural Operational Semantics (SOS)

Introduced by Plotkin in [31], also called transition semantics or reduction semantics, (small-step) structural operational semantics (SOS) semantics captures the notion of one computational step. In a structural operational semantics of a language or calculus, one typically defines configurations and labels, and then gives rules of the form (where $C, C', C_1, C_1', C_2, C_2', \ldots, C_n, C_n'$ configurations, $l, l_1, l_2, \ldots, l_n$ labels):

$$\frac{C_1 \xrightarrow{l_1} C_1',\ C_2 \xrightarrow{l_2} C_2',\ \ldots,\ C_n \xrightarrow{l_n} C_n'}{C \xrightarrow{l} C'}$$

Triples $C \xrightarrow{l} C'$ where C, C' are configurations and l is a label are called "labeled transitions" or "sequents". Configurations can be just terms over the syntax of the language or calculus, but can also be tuples comprising both syntactic terms and semantic components, such as stores, environments, etc. The following is, for example, one of the rules in the SOS definition of CCS (here p, p', q, q' are processes, α and $\overline{\alpha}$ are two related labels, and τ is the label for "silent" transitions; see Section 4.4 for the complete SOS definition of CCS):

$$\frac{p \xrightarrow{\alpha} p', \quad q \xrightarrow{\overline{\alpha}} q'}{p \mid q \xrightarrow{\tau} p' \mid q'}$$

The labeled transitions above the line are called "conditions" or "premises"; if they are missing, then the SOS rule is called "unconditional" and is more simply written as $C \xrightarrow{l} C'$. For example, the following is an unconditional rule in the SOS definition of CCS:

$$\alpha.p \xrightarrow{\alpha} p$$

Labels are not always required; if they are not present then the SOS definition and the corresponding transitions are called "unlabeled". For example, the following is an unlabeled conditional SOS rule defining part of the small-step operational semantics of addition in the context of an imperative language (here a_1, a_1', a_2 are arithmetic expressions and σ, σ' are states – we assumed side effects):

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma'\rangle}{\langle a_1 + a_2, \sigma\rangle \to \langle a_1' + a_2, \sigma'\rangle}$$

For simplicity, from here on we assume labeled rules; if one wants unlabeled rules, one can assume an artificial, transparent label. SOS rules should be seen as derivation or proof rules; they allow us to derive possible labeled transitions. If *SOS* is an SOS definition, with configurations Config_{SOS} and labels Label_{SOS}, then

$$SOS \vdash C \xrightarrow{l} C'$$

denotes the fact that the labeled transition $C \xrightarrow{l} C'$ can be derived with the rules in *SOS*. Intuitively, that means that the defined language or calculus can transit from configuration C to configuration C' in *one step*, where l labels this transition and can be either some meaningful name or some output that is emitted to the environment. Since a derivation in SOS can only capture one step of computation and since typical computations may consist of many such steps, it is common to extend SOS definitions with multiple-step transitions by extending labels to "label paths", which are sequences of labels (suppose that label paths are constructed using semicolon, e.g., $l_1; l_2; \ldots; l_k$, where $l_1, l_2, \ldots, l_k$ are labels), and adding rules of the form:

$$\frac{C \xrightarrow{\pi} C',\ C' \xrightarrow{\pi'} C''}{C \xrightarrow{\pi;\pi'} C''}$$

Strengths. Small-step operational semantics precisely defines the notion of one computational step. It is easily executable so one can quickly get an (inefficient) interpreter for the defined language or analysis, though one may need to pay a cost linear in the size of the current program at each step to "traverse" the program via conditional SOS rules in order to find a redex, that is, a place where the next step can take place. It is easy to trace and debug. In particular, non-termination of programs results in non-termination of searching for a derivation, while erroneous programs (e.g., ones performing division by zero) can be reduced step-by-step until the actual error takes place and then one can be given a meaningful error message. It supports definitions of non-deterministic or parallel languages, obeying the interleaving semantics approach to concurrency.

Weaknesses. It is somehow too low level and explicit, which results into relatively large and boring language definitions; in particular, one needs to explicitly give all the congruence rules for ordinary operators such as addition, multiplication, etc. For that reason, each small step comes at a cost which is worst-case linear to the size of the syntactic term to reduce, which, unfortunately, can grow unbounded. It does not give a "true concurrency" semantics, that is, one has to chose a certain interleaving (no two rules can be applied on the same term at the same time), mainly because reduction is forced to occur only at the top.

One of the main motivations for SOS was that abstract machines are not purely syntactic, that is, they have to introduce new syntactic constructs to decompose the abstract syntax tree; on the other hand, SOS would and should only work by modifying the structure of the program. We argue that this is not entirely accurate: one often needs to include additional "syntax" in an SOS language definition, too, in order to give the semantics of certain language constructs; sometimes, this additional "syntax" is very semantic in nature. For example, one needs to include syntax for values, even though such values were not intended as part of the original language syntax; examples are objects (as nested sets of field values) in object-oriented languages, or closures (as environment enclosing structures) in functional languages. Also, one needs to introduce "stuck" syntactic constructs when defining abrupt termination of exceptions, and even a conditional "if" construct (in case the defined language did not provide one) in order to give an SOS semantics for "while". It is hard and inconvenient to deal with control in SOS —for example, consider adding halt to a simple imperative language. One cannot simply give halt an SOS semantics modularly (i.e., in isolation) like to other ordinary statements: instead, one has to add a corner case (additional rule) to many other unrelated statements, e.g.:

$$\frac{\langle a, \sigma\rangle \to \langle a', \sigma'\rangle}{\langle \mathsf{halt}\ a, \sigma\rangle \to \langle \mathsf{halt}\ a', \sigma'\rangle}$$

$$\langle \mathsf{halt}\ i; s, \sigma\rangle \to \langle \mathsf{halt}\ i, \sigma\rangle$$

$$\langle \mathsf{halt}\ i; a, \sigma\rangle \to \langle \mathsf{skip}; i, \sigma\rangle$$

If expressions could also halt the program, e.g., if one adds functions, then one or more new rules would have to be added to specify the corner case for each arithmetic or boolean construct. Moreover, by having to explicitly propagate the "halt signal" through all the statements and expressions, one fails to capture the intended computation granularity of halt: it should just abruptly terminate the execution (compare that to "propagate the abrupt termination request bottom-up all the way to the top of the program and then terminate the execution", which is what happens in terms of deduction in SOS).

5.1.1 SOS as a Methodological Fragment of Rewriting Logic

As shown in [21, 36], SOS is a methodological fragment of rewriting logic. Indeed, without loosing anything in the translation, any SOS definition is a rewrite logic theory containing one (conditional) rewrite rule per SOS (conditional) rule. We next show a slightly different (from those in [21, 36]) but equivalent way to capture SOS within rewriting logic. To each SOS rule of the form

$$\frac{C_1 \xrightarrow{l_1} C_1',\ C_2 \xrightarrow{l_2} C_2',\ \ldots,\ C_n \xrightarrow{l_n} C_n'}{C \xrightarrow{l} C'}$$

one can associate a rewrite rule of the form:

$$\{C\} \to \{l, C'\} \ \text{ if } \ \{C_1\} \to \{l_1, C_1'\} \ \wedge \ \{C_2\} \to \{l_2, C_2'\} \ \wedge \cdots \wedge \{C_n\} \to \{l_n, C_n'\},$$

where $\{\_\} : \mathit{Config}_{SOS} \to \mathit{Config}_{RLS}$ $\{\_,\_\} : \mathit{Label}_{SOS} \times \mathit{Config}_{SOS} \to \mathit{Config}_{RLS}$ are auxiliary operators constructing rewriting logic semantics variants of configurations — assume Config_{RLS} to be a new sort corresponding to such rewriting logic semantics configurations. These configuration "wrappers" are needed exclusively for embedding reasons. Then the resulting rewriting logic theory, say RLS_{SOS}, has the property:

Theorem 5. ***(Faithful embedding of SOS into rewrite logic)*** *For any SOS definition SOS, any configurations C and C', and any label l, the following holds:*

$$SOS \vdash C \xrightarrow{l} C' \quad \Longleftrightarrow \quad RLS_{SOS} \vdash \{C\} \to \{l, C'\}.$$

The embedding of SOS into rewriting logic presented in [36] is in some sense (the result in [36] considers unlabeled SOS) even more general than the above, because it allows one to mention, as part of the rewriting logic configuration construct (called a "configuration modifier" there), how many small-steps are allowed to be rewritten.

One inherent technicality involved in capturing small-step operational semantics as rewrite theories in a one-to-one notational and computational correspondence is that the rewriting relation is by definition transitive, while the small-step relation is *not* transitive (its transitive closure can be defined a posteriori). Therefore, one needs to devise mechanisms to "inhibit" rewriting logic's transitive and uncontrolled application of rules. Our use of curly brackets as wrappers of SOS configurations into rewriting logic configurations was precisely such a inhibitory mechanism. In the case of unlabeled SOS definitions, one must not get tricked to simply remove the labels in the corresponding rewriting logic definition, thus obtaining rules of the form

$$\{C\} \to \{C'\} \ \text{ if } \ \{C_1\} \to \{C_1'\} \ \wedge \ \{C_2\} \to \{C_2'\} \ \wedge \cdots \wedge \{C_n\} \to \{C_n'\},$$

because then one fails to faithfully capture the original SOS semantics. Indeed, in rewriting logic there is nothing to prevent transitive applications of rules when deriving each of the transitions in conditions, which is in sharp contrast with SOS. One simple solution to circumvent that "problem[**?**, **?**]" in the context of unlabeled SOS definitions is to consider a fake label and then use exactly the transformation above. Different but equivalent ways to embed unlabeled SOS into rewriting logic have been proposed in [21, 36]. The former uses two different types of configuration wrappers, one for the left-hand-side of the transitions ("$\{C\}$") and one for the right-hand-side ("$[C]$"), yielding rules have the form:

$$\{C\} \to [C'] \ \text{ if } \ \{C_1\} \to [C_1'] \ \wedge \ \{C_2\} \to [C_2'] \ \wedge \cdots \wedge \{C_n\} \to [C_n'].$$

The latter proposes a "configuration modifier" operator, say "•", which enables a configuration to be reduced one-step, yielding rules of the form:

$$\bullet C \rightarrow C' \;\; \text{if} \;\; \bullet C_1 \rightarrow C'_1 \;\wedge\; \bullet C_2 \rightarrow C'_2 \;\wedge \cdots \wedge \bullet C_n \rightarrow C'_n.$$

The corresponding embedding theorems are straightforward to formulate and can be found in [21, 36].

Note that all the various faithful embeddings of SOS into rewriting logic above impose the same important restrictions on the resulting rewriting logic theory, namely (1) all matchings, and in particular all rewrite steps, happen *at the top* of the term to rewrite, and because of that, (2) *only one rewrite rule can apply at a time*. Analyzing these necessary restrictions (otherwise the embeddings of SOS would not be faithful) through the prism of rewriting logic, one can easily see how they are the source of two of the main limitations of SOS: lack of modularity and interleaving-only semantics for concurrency. The former follows from the fact that, since rules only apply at the top of a term, each rule needs to mention the complete configuration, so that other rules that may need semantic components from the configuration can find it locally. The latter follows from the fact that there is no way to apply two rules at the same time, because any potentially concurrent matchings of rules overlap, so only one can be applied at a time. Third major limitation of SOS, namely its incapacity of defining complex control-intensive language features such as call-cc, is less obvious from its embeddings in rewriting logic. On the positive side, one can say that SOS, despite its restrictions, has been successfully used to define a variety of language features and calculi, so in some sense it can be seen as a methodological subset of rewriting logic that proved to be quite usable in practice and sufficient in many cases. While one cannot refute these obvious arguments, since we are are a quest for an *ideal* language definitional framework, the limitations of SOS in the context of modern, real-life complex languages, make it a less stimulating candidate.

5.1.2 SOS as a Methodological Fragment of K

Since in an SOS definition we have two high-level semantics categories, namely SOS configurations and SOS labels, we define two configuration items in our formalization of SOS into K, namely $(\!|K|\!)_k$ wrapping SOS configurations and $(\!|\mathsf{List}_;[Label_{\mathrm{SOS}}]|\!)_{path}$ wrapping semicolon-separated sequences, or paths, of labels; these are used in all K definitions derived from SOS definitions. In addition to the SOS configurations, the computations K also contain some special "frozen" computations of the form "$\circ \xrightarrow{l} C'$", where l is a label and C' some SOS configuration (which can be read "whatever configuration was there goes via label l to C'"), and some special constants, each corresponding uniquely to precisely one rule in the SOS definition (which can be read "that rule is being processed"):

$$\begin{array}{rcl}
\mathit{Config} & ::= & (\!|K|\!)_k \mid (\!|\mathsf{List}_;[Label_{\mathrm{SOS}}]|\!)_{path} \\
K & ::= & \mathit{Config}_{\mathrm{SOS}} \mid \circ \xrightarrow{Label_{\mathrm{SOS}}} \mathit{Config}_{\mathrm{SOS}} \mid \mathit{KConstant} \\
\mathit{KConstant} & ::= & \mathit{rule}_1 \mid \mathit{rule}_2 \mid \ldots \text{ (one K constant per SOS rule)}
\end{array}$$

The role of the frozen computations $\circ \xrightarrow{l} C'$ is twofold. On the one hand, the computation C' is indeed "frozen", until the frozen term gets on the top of the computation and is "scheduled" for processing using the equation below; on the other hand, when that happen, the label l is also "frozen", so when C' is scheduled for processing the label l can be collected in the path. The equation below takes care of both "unfreezing" C' and collecting l in the path, and is generic (for any embedding of SOS definitions into K):

$$(\!|\underset{C'}{\underline{\circ \xrightarrow{l} C'}}|\!)_k \;\; \langle\!|\underset{l}{\underline{\cdot}}|\!)_{path}$$

Finally, each SOS rule

$$\mathit{rule}_i : \;\; \frac{C_1 \xrightarrow{l_1} C'_1,\; C_2 \xrightarrow{l_2} C'_2,\; \ldots,\; C_n \xrightarrow{l_n} C'_n}{C \xrightarrow{l} C'}$$

can be converted into one K rule and one K equation as follows:

$$\llparenthesis \underset{strict(C_1, C_2, \ldots, C_n) \curvearrowright rule_i \curvearrowright (\circ \xrightarrow{l} C')}{\underline{\qquad\qquad\qquad C \qquad\qquad\qquad}} \rrparenthesis_k$$

$$\llparenthesis \underset{\cdot}{\underline{strict(\circ \xrightarrow{l_1} C'_1,\ \circ \xrightarrow{l_2} C'_2, \ldots,\ \circ \xrightarrow{l_n} C'_n) \curvearrowright rule_i}} \curvearrowright (\circ \xrightarrow{l} C') \rrparenthesis_k$$

The rule above says that in order to process C when this is at the top of the computation (note that there could be other tasks following C in $\llparenthesis C \rrparenthesis_k$), one should "first" process C_1, C_2, ..., C_n (in any order and possibly non-deterministically) and "then" go with l to C'. The equation above dissolves the processed C_1, C_2, ..., C_n when these reach the expected configurations C'_1, C'_2, ..., C'_n with the expected labels l_1, l_2, ..., l_n, respectively, as stated by the original SOS rule. The constant "$rule_i$" is necessary to connect the rule and the equation above; otherwise, it could be that two SOS rules have the same C and the same number of conditions, the first rule is "scheduled" and each of its condition left-hand configuration reduces to the *other rule's* corresponding condition right-hand configuration with the same label; if that is the case, then the equation above would incorrectly "apply" the SOS rule. In other words, the K definition associated to an SOS definition intuitively simulates a back-and-forth stack mechanism (recall that *strict* iteratively allows any of the wrapped computations to be scheduled for processing for any arbitrary number of steps) that eventually performs the same computational work and task as the original SOS definition. Formally, one can prove the following result:

Theorem 6. ***(Faithful embedding of SOS into K)*** *For any $C, C' \in Config_{\mathrm{SOS}}$ and $l \in Label_{\mathrm{SOS}}$,*

$$SOS \vdash C \xrightarrow{l} C' \iff K_{\mathrm{SOS}} \vdash \llparenthesis C \rrparenthesis_k\ \llparenthesis \cdot \rrparenthesis_{path} \to \llparenthesis C' \rrparenthesis_k\ \llparenthesis l \rrparenthesis_{path}$$

Moreover, for any $C, C' \in Config_{\mathrm{SOS}}$ and $l_1, l_2, \ldots, l_k \in Label_{\mathrm{SOS}}$,

$$SOS \vdash C \xrightarrow{l_1;l_2;\ldots;l_k} C' \iff K_{\mathrm{SOS}} \vdash \llparenthesis C \rrparenthesis_k\ \llparenthesis \cdot \rrparenthesis_{path} \to \llparenthesis C' \rrparenthesis_k\ \llparenthesis l_1; l_2; \ldots; l_k \rrparenthesis_{path}.$$

Therefore, SOS can be regarded as a methodological fragment of K, in the sense that for any SOS definition of a language or calculus, one can mechanically derive a corresponding K definition that faithfully captures both the one-step relation and the transition systems associated to the original SOS definition. In particular, one can use this embedding to derive tool support for SOS from tool support for K. For example, with our K implementation in Maude (Section E), one can use the search or the model-checking capabilities of Maude to formally analyze SOS definitions; in particular, given a configuration C, one can inquire for one, two, ..., or even all the paths via which a configuration (or a pattern of configurations) C' can be reached, as well as for one, two, ..., or all the configurations that can be reached with a given path (or pattern of paths). However, as mentioned at the beginning of this section for all the embeddings, this faithful embedding also comes with all the limitations of SOS: no true concurrency semantics (just interleaving), non-modularity, and incapacity of defining complex features (such as call/cc). Therefore, SOS is far from being an ideal definitional framework, with or without an embedding in K. Theorem 6 shows formally that K is no worse than SOS. The rest of this paper shows that K actually smoothly overcomes the limitations of SOS.

5.2 Big-Step Operational Semantics

Introduced as natural semantics in [16], also named relational semantics in [27], or evaluation semantics, big-step semantics is in some sense "the most denotational" of the operational semantics. One can view big-step definitions as definitions of functions or relations, interpreting each language construct in an appropriate domain. A big-step rule has the form

$$\frac{C_1 \Downarrow R_1,\ C_2 \Downarrow R_2,\ \ldots,\ C_n \Downarrow R_n}{C \Downarrow R}$$

where like in SOS C, C_1, C_2, ..., C_n are configurations holding fragments of program together with all the needed semantic components, and R, R_1, R_2, ..., R_n are *result configurations*, i.e., configurations which cannot be advanced anymore. For example, here is the big-step rule defining addition in a language whose expression evaluation has no side-effects (a_1, a_2 are arithmetic expressions, σ is a state, and i_1, i_2 are integers):

$$\frac{\langle a_1, \sigma\rangle \Downarrow i_1,\ \langle a_2, \sigma\rangle \Downarrow i_2}{\langle a_1 + a_2, \sigma\rangle \Downarrow i} \quad \text{where } i \text{ is the sum of } i_1 \text{ and } i_2$$

Typically the result configurations are just a value like above or an abstract value (e.g., a type), but when the language under consideration has side effects, result configurations may need also to contain some semantic components. We assume that the down-arrow symbol $\Downarrow$ is not decorated with any labels; this can be easily achieved by moving such labels as components into the result configurations. Interestingly, any big-step operational semantic definition is a special case of a (small-step) SOS definition like in Section 5.1: all what needs to do is to replace "$\rightarrow$" by $\Downarrow$". The crucial technical difference between the two approaches is that in big-step semantics the result configurations are not reducible anymore: one derives directly the final result of processing a configuration (if any). Therefore, all the embeddings of SOS into rewrite logic and K in Section 5.1 are also applicable for big-step operational semantics and their faithfulness theorems still hold. However, thanks to the particular nature of big-step semantics, we next give simpler embeddings.

Strengths. When it can be given to a language, big-step operational semantics is easy to understand, since it relates syntactic entities directly to their expected results. Big-step operational semantics, due to its straightforward recursive nature, can be easily and efficiently interpreted in any recursive, functional or logical framework. Many proponents of big-step operational semantics regard it "almost as an interpreter" for the defined language. It is more abstract than many other definitional styles, so one can in principle more easily define and prove properties about programs (in languages that can be given a reasonable big-step semantics). It is particularly useful for defining type systems and side-effect free languages or language features.

Weaknesses. Due to its monolithic, single-step evaluation, it is hard to debug or trace big-step semantic definitions. If the program is wrong, no information is given about where the failure occurred.

Divergence or non-termination is not observable in the specified evaluation relation. In particular, one cannot define at all non-terminating processes or systems, such as reactive system. Consider, for example, the task of giving CCS (see Figure 5 for a small-step SOS definition) a big-step operational semantics; even if one (unreasonably) assumes that a CCS process terminates, then its big step semantics should "evaluate" it to all its possible behaviors; even making abstraction (again) of the various semantic choices, such as trace-based versus tree-based semantics, the major problem is that a potential big-step semantics of concurrent processes $p|q$ is not compositional, that is, it is not possible to simply combine the semantics of p and q in order to obtain the semantics of $p|q$. A technique for capturing an interleaving semantics for concurrent systems using big-step operational semantics is sketched by Mosses in [29] (using the modular features of his MSOS), but he did not provide details because he does not encourage the use of the big-step semantics style for concurrency[6]. It is, in fact, well known and broadly accepted that big-step operational semantics is simply unsuitable for concurrency.

Big-step semantics is not modular. For example, to add side effects to expressions, one must redefine the rules to allow expressions to evaluate to pairs (value-state) and to appropriately propagate side effects; e.g.:

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1, \sigma'\rangle,\ \langle a_2, \sigma'\rangle \Downarrow \langle i_2, \sigma''\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i, \sigma''\rangle} \quad \text{where } i \text{ is the sum of } i_1 \text{ and } i_2.$$

Therefore, in order to add side-effects to one's language, one needs to revisit the entire existing definition to take explicit care of propagating the side effects. Additionally it is very inconvenient (and non-modular) to define complex control statements; consider, for example, adding a halt statement to a hypothetical simple

[6]Personal communication.

and side-effect-free imperative language —one needs to add a special configuration *halting*(i), and rules like the following:

$$\frac{\langle a, \sigma\rangle \Downarrow i}{\langle \mathsf{halt}\ a, \sigma\rangle \Downarrow \mathit{halting}(i)}$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow \mathit{halting}(i)}{\langle s_1; s_2, \sigma\rangle \Downarrow \mathit{halting}(i)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \mathsf{true},\ \langle s, \sigma\rangle \Downarrow \mathit{halting}(i)}{\langle \mathsf{while}\ b\ \mathsf{do}\ s, \sigma\rangle \Downarrow \mathit{halting}(i)}$$

$$\frac{\langle s, \sigma\rangle \Downarrow \mathit{halting}(i)}{\langle s; a, \sigma\rangle \Downarrow i}$$

Therefore, one needs to rethink all the language constructs and add big-step rules that propagate the halting signal through all the constructs that may halt the program. If each fragment of program may potentially halt the program, which is typically the case, then one needs to add a new halt rule for each argument of each language construct. The situation can easily become unmanageable when the language under consideration is non-trivial; for example, a language may have exceptions, functions with return, loops with break and continue, etc., the addition of each of these features requiring the language designer to potentially more than double the number of existing rules in the language, not to mention and subtle interactions between the various control-intensive language features (for example, a return from a function crosses the boundaries of loops, but a break/continue of a loop does not cross the boundaries of a function invocation, etc.).

We hope that all the arguments above make it clear that big-step operational semantics is far from being an ideal language definition framework. While it is nevertheless useful to define certain aspects of a language, such as a type system for it, big-step operational semantics has too many limitations to serve as a uniform and unique definitional framework for languages, language features and/or language analyzers.

5.2.1 Big-Step Semantics as a Methodological Fragment of Rewriting Logic

Due to its straightforward evaluation-like recursive nature, big-step semantics is relatively easy to define in any other formalism as well as to translate into an interpreter for the defined language into any programming language. (The difficulty with big-step SOS is to actually give semantics to complex constructs.) It is therefore not surprising that one can replace each big-step rule with one conditional rewrite rule and thus obtain a rewrite logic theory that faithfully captures the big-step definition. For simplicity, we here assume that configurations and result configurations are disjoint; in case one needs the result configurations to mention the same semantic components as the other configurations, then we assume that one uses a different configuration wrapper notation. For example, one can write $\langle a, \sigma\rangle$ for a reducible configuration containing arithmetic expression a and state σ, and $[v, \sigma]$ for a result configuration containing value v and state σ. Also, we make the reasonable assumption that configurations are not nested. Then one can associate to each big-step semantic rule as above a rewrite rule as follows:

$$C \to R\ \ \mathsf{if}\ \ C_1 \to R_1\ \wedge C_2 \to R_2\ \wedge \cdots \wedge C_n \to R_n.$$

A first thing to notice is that, since all these rewrite rules involve configurations, rewriting can only occur at the top, thus the general application of term rewriting under contexts is disabled by the definitional style. Another thing to notice here is that all configurations in the right hand sides are normal forms, thus the transitivity rule for rewriting logic also becomes inapplicable. Then the following result can be proved:

Theorem 7. ***(Faithful embedding of big-step operational semantics into rewrite logic)*** *For any big-step operational semantics definition BigStep, any configuration C and any result configuration R,*

$$\mathit{BigStep}\ \vdash C \Downarrow R \iff \mathit{RLS}_{\mathit{BigStep}} \vdash C \to R,$$

where $\mathit{RLS}_{\mathit{BigStep}}$ is the rewrite logic semantic definition obtained from BigStep as above.

The only apparent difference between *BigStep* and $\mathrm{RLS}_{\mathit{BigStep}}$ is the different notational conventions they use ("$\rightarrow$" instead of "$\Downarrow$" and conditional rewrite rules instead of conditional deduction rules). However, as the above theorem shows, there is a one-to-one correspondence also between their corresponding "computations" (or executions, or derivations). Therefore, $\mathrm{RLS}_{\mathit{BigStep}}$ actually *is* the big-step operational semantics *BigStep*, not an "encoding" of it. Note that, in order to be faithfully equivalent to *BigStep* computationally, $\mathrm{RLS}_{\mathit{BigStep}}$ lacks the main strength of rewriting logic that makes it an appropriate formalism for concurrency, namely, that rewrite rules can apply under any context and in parallel (here all rules are syntactically constrained so that they can only apply at the top, sequentially).

5.2.2 Big-Step Semantics as a Methodological Fragment of K

Our embedding of big-step semantics into K is very similar to, but simpler than the embedding of SOS into K. We consider no paths here (these may be methodologically included as part of the result configurations if one wants to, though labels typically come with concurrent system definitions, for which big-step is notoriously inappropriate) and two types of computations, one corresponding to big-step configurations and one corresponding to big-step result configurations:

$$\begin{array}{rcl} \mathit{Config} & ::= & (\!|K|\!)_k \\ K & ::= & \mathit{Config}_{\mathrm{BigStep}} \mid \mathit{ResultConfig}_{\mathrm{BigStep}} \mid \mathit{KConstant} \\ \mathit{KConstant} & ::= & \mathit{rule}_1 \mid \mathit{rule}_2 \mid \ldots \ \text{(one K constant per big-step rule)} \end{array}$$

Then each big-step rule

$$\mathit{rule}_i: \ \frac{C_1 \Downarrow R_1,\ C_2 \Downarrow R_2,\ \ldots,\ C_n \Downarrow R_n}{C \Downarrow R}$$

can be converted into one K rule and one K equation as follows:

$$(\!|\frac{C}{\mathit{strict}(C_1, C_2, \ldots, C_n) \curvearrowright \mathit{rule}_i \curvearrowright R}|\!)_k$$

$$(\!|\underset{\cdot}{\underline{\mathit{strict}(R_1, R_2, \ldots, R_n) \curvearrowright \mathit{rule}_i}} \curvearrowright R|\!)_k$$

The following expected result can be proved now:

Theorem 8. ***(Faithful embedding of big-step operational semantics into K)*** *For any big-step operational semantics definition BigStep, any configuration C and any result configuration R,*

$$\mathit{BigStep} \vdash C \Downarrow R \iff K_{\mathit{BigStep}} \vdash (\!|C|\!)_k \rightarrow (\!|R|\!)_k,$$

where K_{BigStep} is the K semantic definition obtained from BigStep as above.

5.3 Modular Structural Operational Semantics (MSOS)

Modular structural operational semantics (MSOS) was introduced by Peter Mosses in [28] and then discussed in detail in [29], to deal with the non-modularity issues of small-step and big-step SOS. The solution proposed in MSOS involves moving the non-syntactic state components to the labels on transitions (as provided by SOS), plus a discipline for only selecting needed attributes from the states. There are both big-step and small-step variants of MSOS, but we discuss only small-step MSOS here. As discussed in Section 5.2, big-step SOS has too many limitations in addition to its non-modularity; even though big-step MSOS partially improves the modularity of big-step SOS, it is far from being an ideal general purpose language definitional framework, so we do not discuss it anymore in this paper.

Before we get into the technicalities of MSOS, one natural question to address is why we need modularity of language definitions. One may argue that defining a programming language is a major initiative that is being done once and for all, so having to go through the defining rules many times is, after all, not such a bad idea, because it gives one the chance to find and fix potential errors in them. Here are several reasons why modularity is desirable in language definitions:

1. Having to modify many or all rules whenever a new rule is added that modifies the structure of the configuration is actually more error prone than it may seem, because rules become heavier to read and debug; for example, one can write σ instead of σ' in a right-hand-side of a rule and a different or wrong language is being defined.

2. When designing a new language, as opposed to an existing well-understood language, one needs to experiment with features and combinations of features; having to do lots of unrelated changes whenever a new feature is added to or removed from the language burdens the language designer with boring tasks taking considerable time that could have been otherwise spent on actual interesting language design issues.

3. There is a plethora of domain-specific languages these days, generated by the need to abstract away from low-level programming language details to important, domain-specific aspects of the application. Therefore, there is a need for rapid language design and experimentation for various domains. Moreover, domain-specific languages tend to be very dynamic, being added or removed features frequently as the domain knowledge evolves. It would be very nice to have the possibility to "drag-and-drop" language features in one's language, such as functions, exceptions, objects, etc.; however, in order for that to be possible, modularity of language feature definitions is crucial.

Whether MSOS gives us such a desired framework for modular language design is still open and debatable, but it is certainly the first framework explicitly aiming at modular language design.

A transition in MSOS is of the form $P \xrightarrow{\mathcal{X}} P'$ or the form $P \rightarrow P'$, where P and P' are programs or fragments of programs and $\mathcal{X}$ is a label describing the structure of the remaining configuration both before and after the transition. If X is missing, then the remaining part of the configuration is assumed to stay unchanged. Specifically, $\mathcal{X}$ is a record containing fields denoting the semantic components of the configuration; the preferred notation in MSOS for saying that in label $\mathcal{X}$ the semantic component associated to the field name σ (e.g., a state or a store name) is σ_0 (e.g., a function associating values to variables) is $\mathcal{X} = \{\sigma = \sigma_0, ...\}$. Modularity in MSOS is achieved by two important notational conventions: (1) the record comprehension notation "…" which indicates that more fields could follow but that they are not of interest for this transition, in particular that they stay unchanged after the application of the transition, and (2) by not mentioning labels at all when nothing changes on the labels (this is a recent feature of MSOS introduced by Peter Mosses in a December 2007 talk at the University of Illinois at Urbana-Champaign). If record comprehension is used in both the condition and the conclusion of an MSOS rule, then all the occurrences of "…" stand for the same fields with the same semantic components.

If no labels are mentioned on any of the transitions in the condition and the conclusion of the rule, then it means that all the transitions have the same label (labels do not change). This allows for most of the MSOS rules in a language or calculus definition to look very natural, without unnecessary decorations of the transition relations. For example, Figure 8 shows the small-step MSOS rules for advancing the expressions that are assigned to variables, bound by a let, or written to the output; note that these rules have no conceptual connection with the actual process of assigning or binding a value to a variable, or of writing it to the output (their role is to prepare the terrain for the actual assignment, binding, or output writing; these are defined shortly), so no labels needed by other rules need to be artificially carried by MSOS transitions. For comparison, we also give the K correspondent to each MSOS rule, assuming more or less the same conceptual definitional style and configuration structuring. For now, the reader can check the K corresponding definitions for illustrory purposes; at the end of this section we compare MSOS with K and, as part of that comparison, we shall discuss the K definitions in more detail. Fields of an MSOS label can be read-write, read-only, or write-only. Read-write fields come in pairs, having the same field name, except that the "write" field name is primed. They are used for transitions modifying existing configuration fields. For example, a store field σ can be read and written, as illustrated by the MSOS rule in Figure 9 for assignment in a language definition based on a split of state into an environment (ρ) and a store (σ). The rule in Figure 9 says that, if before the transition the environment was ρ_0 and the store was σ_0, after the transition the store will become $\sigma_0[\rho_0[x] \leftarrow i]$, updating the location of x in the store to i. (We adopted the common

$MSOS$	Language syntax	K syntax annotation
$\dfrac{e \longrightarrow e'}{x := e \longrightarrow x := e'}$	$Exp ::= \ldots \mid Name := Exp$	$strict(2)$
$\dfrac{e_1 \longrightarrow e_1'}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \longrightarrow \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2}$	$Exp ::= \ldots \mid \mathsf{let}\ Name = Exp\ \mathsf{in}\ Exp$	$strict(2)$
$\dfrac{a \longrightarrow a'}{\mathsf{write}(a) \longrightarrow \mathsf{write}(a')}$	$Stmt ::= \ldots \mid \mathsf{write}(Exp)$	$strict$

Figure 8: Syntax traversing rules in MSOS and corresponding K strictness attributes

$MSOS$	K
$x := v \xrightarrow{\{\rho=\rho_0,\sigma=\sigma_0,\sigma'=\sigma_0[v/\rho_0[x]],\ldots\}} \mathsf{skip}$	$⦇\dfrac{x := v}{\cdot}⦈_k\ ⦇\rho_0⦈_\rho\ ⦇\dfrac{\sigma_0}{\sigma_0[v/\rho_0[x]]}⦈_\sigma$

Figure 9: Semantics of assignment in MSOS and K

SOS convention that unconditional rules are written as just their conclusion transition, omitting their empty condition.)

Read-only fields are only inspected by the rule, but not modified. For example, when writing (as above) or reading the location of a variable in an environment, the environment is not modified. Also, when temporarily switching to a different environment to process a subexpression, a new environment is created for the subtask, but no different environment is ever needed to be produced as a result of applying a transition; for example, Figure 10 shows the remaining MSOS environment-based definition of the let construct.

$MSOS$	K
$\dfrac{e_2 \xrightarrow{\{\rho=\rho_0[l/x],\sigma=\sigma_0[v_1/l],\ldots\}} e_2'}{\mathsf{let}\ x = v_1\ \mathsf{in}\ e_2 \xrightarrow{\{\rho=\rho_0,\sigma=\sigma_0,\ldots\}} \mathsf{let}\ x = v_1\ \mathsf{in}\ e_2'}$	$⦇\dfrac{\mathsf{let}\ x = v_1\ \mathsf{in}\ e_2}{e_2 \curvearrowright restore(\rho_0)}⦈_k\ ⦇\dfrac{\rho_0}{\rho_0[l/x]}⦈_\rho\ ⦇\dfrac{\sigma_0}{\sigma_0[v_1/l]}⦈_\sigma$
$\mathsf{let}\ x = v_1\ \mathsf{in}\ v_2 \longrightarrow v_2$	— no similar K rule needed —

Figure 10: Semantics of let in MSOS and K

Note that unlike in the case of the write-read label σ for the store, the label ρ does not need a primed version; indeed, following the definitional style above, an environment is never changed at the end of a small-step, while a store may be changed. The first MSOS rule above advances e_2 a small-step in a correctly modified environment and state. It is interesting to note that, even though the environment and the store in which e_2 is evaluated appear to change in a similar way, there is actually a crucial difference between the two, precisely because the former is read-only and the latter is write-read: the fact that the label σ' is not mentioned it means that the condition and the conclusion transitions have the same σ', i.e., the conclusion transition carries over the "side effects" propagated by the condition, while the fact that there is no label ρ' it means that the environment seen by the let construct does not change as the subexpression e_2 is evaluated (in its properly updated environment).

Write-only fields are used to record things not analyzable during the execution of the program, such as the output or the trace. Their names are always primed and they have a free monoid semantics –everything written on them is actually added at the end. A good example of the usage of write-only fields would be a rule for defining a print/write/output language construct, like the one in Figure 11. Note that, since the

$MSOS$	K
$\mathsf{write}(i) \xrightarrow{\{out'=i,\ldots\}} \mathsf{skip}$	$(\!\vert \underset{\cdot}{\underline{\mathsf{write}(i)}} \vert\!\rangle_k \ \langle\!\vert \underset{i}{\underline{\cdot}} \vert\!)_{out}$

Figure 11: Semantics of write in MSOS and K

"accumulated" write-only field semantic components are not allowed to be modified or used by rules, one only mentions the new items being written to a write field, without ever having to mention the existing value of the field; in other words, one writes "$out' = i$" instead of "$out' = o, i$" where o is the existing value of the out write field.

The idea in MSOS is therefore that one can use the labels to maintain important semantic information about the "execution" of a program. The implicit notational conventions of MSOS allow the rules to propagate this semantic information in the "back-scenes". This apparently simple trick is actually crucial to increasing the modularity of language definitions, because new features needing new semantic components in the configuration can add and modify those without having to change any of the existing rules to propagate the new semantic components, as an explicit-label framework like SOS would have to. The advantage of MSOS over SOS can perhaps be seen best when defining language features that change abruptly the execution flow of the program. Recall that in order to define a halt statement in SOS one needs to possibly more than double the number of rules, because each language construct needs to be changed its semantics to propagate the "halting signal" produced by any of its subexpressions. As Figure 12 shows, in MSOS one can define halt modularly, adding a write-only label that holds the halting value, if any generated. A top level construct,

$MSOS$	K
$\dfrac{p \xrightarrow{halt'=()} p'}{\mathsf{program}\ p \xrightarrow{halt'=()} \mathsf{program}\ p'}$	— no similar K rule needed —
$\dfrac{p \xrightarrow{halt'=v} p'}{\mathsf{program}\ p \xrightarrow{halt'=v} v}$	— no similar K rule needed —
$\dfrac{e \to e'}{\mathsf{halt}\ e \to \mathsf{halt}\ e'}$	$Stmt ::= \ldots \mid \mathsf{halt}\ Exp \qquad [strict]$
$\mathsf{halt}\ v \xrightarrow{halt'=v} \mathsf{stuck}$	$(\!\vert \mathsf{halt}\ v \vert\!\rangle_k \to (\!\vert v \vert\!)_k$

Figure 12: Semantics of halt in MSOS and K

"program p", wrapping a program needs to be added to the syntax in order to know where to "catch" the halting signal. The first rule advances the program as before if it generates no halting signal, while the second catches the halting value and stops the program with that value.

MSOS is given a semantics based on category theory in [29], but, as the examples above show, one does not need to understand category theory in order to understand or use MSOS. The simplest way to explain MSOS is perhaps through its automatic translation into conventional SOS: each MSOS rule can be desugared into a rule that explicitly mentions all the fields with all their values on all the labels; this way, any MSOS definition becomes a conventional SOS definition with labels on transitions. It is important to realize that the desugared SOS definition does not enjoy the modularity characteristics of the original MSOS definition (otherwise one could use MSOS as an "SOS style"); for example, if one decides to add a new label in order to accommodate a new language feature, then one needs to change all the existing SOS rules. Nevertheless, once the MSOS-to-SOS mechanical translation is applied, one can further use the embeddings of SOS into rewriting logic and K described in Sections 5.1 (small-step SOS) and 5.2 (big-step SOS), and thus give an embedding of MSOS into rewriting logic and K, respectively. An embedding of MSOS in rewriting logic has also been proposed in [21]; also, the MSOS tool [7], which can execute MSOS definitions, is implemented using rewriting logic in Maude. While such embeddings are interesting and instructive in their own way, and they also indirectly give an intuition for the definitional capability of the various formalisms involved, we believe that they should not be used as an absolute measure. Indeed, the whole point of MSOS is to provide a framework facilitating one to write methodologically better SOS definitions. MSOS was, unlike K, not intended to be an alternative to SOS; the proponents of SOS will find little or no difficulty in using MSOS, and there is no doubt that MSOS brings a significant degree of modularity to SOS.

Therefore, a systematic derivation of a computationally faithful K definition from an MSOS definition is possible, which allows one to use the machinery of K (rewriting and formal analysis mechanisms) for MSOS. However, as mentioned above, such a translation would not be "conceptually faithful", because it would not capture the inherent modularity of MSOS over SOS; that modularity aspect would be "lost in translation". That being said, in the remaining of this section we focus on conceptual similarities and differences between MSOS and K, irrespective of any particular translation of one into the other.

- Perhaps the main difference between the two is that MSOS is still heavily based on conditional rules, that is, on rules with computational premises (in addition to the conventional side conditions), while K only uses unconditional rules (possibly using side conditions). The hasty reader could obviously say that conditional is more general than unconditional because any unconditional rule is a special case of a conditional rule, so therefore MSOS and even SOS are more general than K. First, as discussed above and shown in detail in Sections 5.2 and 5.1, thanks to K having equations in addition to rules, the "conditionality" of rules in SOS-like formalisms does not have any computational or definitional advantage over unconditional rules in K, simply because one could use equations to embed a "scheduler" for derivations using conditional rules on top of K's derivation infrastructure based on unconditional rules. Second, conditional rules in SOS-like definitional frameworks, including MSOS, are typically used to reduce the semantics of a language construct to that of its sub-constructs; in particular, the evaluation context of a program or fragment of program is captured by a proof, or derivation context. This is a laudable goal which proved to be particularly useful in inductive proofs. However, it comes with a major drawback: the evaluation context is lost during the computation/evaluation, in the sense that it is *not* being made available to rules. Thus, it is very hard to give SOS semantics to control-intensive language constructs that non-trivially alter the evaluation context. In particular, we are not aware of any natural/elegant way to give an SOS or MSOS semantics to a language construct like call/cc[7].

- We do not regard the use of strictness attributes, such as those in the definitions of assignment, let and write above, as a major difference between MSOS and K. Indeed, MSOS or mostly any other formalism can adopt K's strictness attributes and desugar them into its corresponding rules. However, the way these are desugared in K, namely into a heating/cooling unconditional rule, is of course specific to K.

- Both MSOS and K make intensive use of labels: MSOS carries labeled semantic components on its syntax-to-syntax transitions, while in K everything is in some label, including the syntax itself. Therefore, unlike in MSOS, syntax has no preferential role in K. Moreover, labels in K can be "nested" and are "dynamic", that is, the semantic or syntactic information associated to each label can be itself subject to decomposition and grouping into labels, and labels can be created and terminated dynamically, as the program evolves. Labels in MSOS are statically desugared, cannot be dynamically created or terminated, and can only be "top-level" (not nested) and must be tagged with a type: read-only, write-only, read-write. This tagging is crucial in MSOS, being intimately used in its generation

[7]Some proponents of SOS-like formalisms claim that constructs like call/cc are obscure and hard to use anyway, so being or not being able to define them formally is irrelevant. Our position is that a designer of a language design framework should refrain from commenting on the usefulness of particular language constructs; this is the responsibility of the one using the framework, the language designer. In our (admittedly subjective) view, a language definitional framework worth its salt should simply support the definition of any language feature; the fact that a definitional framework cannot define a particular feature in current use (call/cc is used in Scheme, SML, Haskell, etc.) shows a serious limitation of that framework.

of labeled transition systems. The semantics of K is a "true concurrency" one, based on unlabeled concurrent transitions (in other words, "the necessary labeling is encoded in the state") implicit in the underlying rewriting mechanism. Unlike in MSOS, in K there is no special step corresponding to the "desugaring" of labels: due to the way associative and associative/commutative matching works, one simply mentions in each rule only the configuration components of interest; this simplicity compared to MSOS is largely facilitated by the fact that rules in K are unconditional, otherwise some MSOS-like mechanism to synchronize labels in conditions with their counterparts in hypotheses would be unavoidable. Note that the nesting and dynamicity of labels is a crucial characteristic of K; for example, as seen in Section 6, each thread in our definitions corresponds to a label containing inside a computation, various execution stacks, the resources that particular thread holds, etc. One could argue that all this information can be flattened in some way or another at the top-level; however, that is neither necessary not appealing in K. One could also argue that nested labels could be added to MSOS as well; however, since the use of nested labels in K tends to be most effective when the syntax itself appears nested in the configuration structure and since syntax plays a special role in MSOS, it is not clear whether it is possible or even beneficial to do that in MSOS.

It may be worth discussing in more depth the use of labels in K versus in MSOS. The MSOS definitions above have all been given K variants which show how one makes use of similar labels in K. While this may give the reader some intuition about the relationships and differences between the two, one should be aware that this is not the only way to define these language constructs in K. The above K definitions follow a common K style in which special environment labels are being maintained as part of the configuration, where expressions that modify the environment are "responsible" for cleaning up the environment after they execute (using a *restore* computation item like above). An alternative would be to instead tag each expression that can potentially be scheduled for evaluation with its evaluating environment; this way, one would never need to worry about recovering environments. This was the approach that was first used in K [32] and is similar in spirit to the one followed by the CK abstract machine [] (see Section **??**). Assuming expressions e paired with their evaluation environment ρ using the pairing construct $\langle e, \rho\rangle$, the K rules for assignment and let above become, respectively:

$$(\!|\underset{\cdot}{\underline{\langle x := v, \rho_0\rangle}}|\!)_k \;\; (\!|\underset{\sigma_0[v/\rho_0[x]]}{\underline{\sigma_0}}|\!)_\sigma$$

$$(\!|\langle \underset{e_2}{\underline{\mathsf{let}\ x = v_1\ \mathsf{in}\ e_2}}, \underset{\rho_0[l/x]}{\underline{\rho_0}}\rangle|\!)_k \;\; (\!|\underset{\sigma_0[v_1/l]}{\underline{\sigma_0}}|\!)_\sigma$$

As explained in Section **??**, this style has the advantage that nothing needs to be done for tail-recursive functions: they do not increase the size of the computation structure by stacking useless *restore* items. However, this style of defining a language has the drawback that it enforces an environment-based approach, so language definitions are less modular. Also, strictness attributes should either be dropped or their automatic desugaring changed into corresponding heating/cooling rules where the environment is being silently passed to the "heated" subexpressions. Since tail recursion can be easily handled also when expressions are not tagged with their environments (see Section **??**), we prefer to avoid tagging expressions with their evaluation environments in most K definitions.

Figure 9 shows well the relationship between MSOS and K in what regards their use of labels. In MSOS, several semantic components corresponding to labels may be changed by each transition. For example, is the store before the assignment transition (σ) is σ_0 and the environment is ρ_0, then the store after the transition (σ') becomes $\sigma_0[v/\rho_0[x]]$. In K, this change of store is captured by underlying the existing store, σ_0, and writing below the line the modified store, $\sigma_0[v/\rho_0[x]]$; however, in K, unlike in MSOS, the "syntax" change is captured exactly the same way: the current statement is underlined, and below the line a "dot" signifies that the assignment is simply erased from the computation structure. Both in MSOS and K the current environment (ρ) is accessed but let unchanged. As mentioned in the paragraph above, other semantic styles within K are also possible.

Figure 10 shows the conceptually major difference between MSOS and K. MSOS inherits from SOS its definitional style based on the use of conditional rules, while K builds upon the belief that conditional rules are unnecessary in programming language definitions. In order to prepare the right semantic content for the condition transition, in MSOS one is expected to explicitly state which label contents change; the semantic contents of every other label stays unchanged. While in K one is also expected to explicitly change the contents of the very same labels using its unique mechanism to "change" (underline what changes and write the new contents underneath the line), one does that using one unconditional rule. The next subexpression to evaluate is "scheduled" on the computation structure, without worrying about ever "coming back" to the original let construct. If additional work needs to be done after processing the subexpression, such as restoring the environment in K definition in Figure 10, appropriate computation items need to be *a priori* placed on the computation structure. Depending upon the style adopted, more, less, or no post-processing is required; for example, the style adopted in the paragraph above requires no environment restoration, because each expression carries its evaluation environment with it. Because of K's "no coming back" approach, no rules are necessary to state what to do once the subexpression is completely processed; in particular, no K rule corresponding to the second MSOS rule in Figure 10 is necessary.

Let us next discuss the similarities and differences between MSOS and K in the context of defining control-intensive statements, such as the halt statement in Figure 12. Because of its conditional-rule-based nature, MSOS needs to declare a top level construct, "program p", in addition to the actual halt construct; this way, one can either let the computation advance normally when no halting signal was recorded in the write-only *halt'* label (first rule), or stop it at the top level appropriately when a halting signal was recorded in the label (second rule). Since in K the complete computation is explicit in the configuration structure, one needs not add any artificial construct (e.g., program) in order to define another (e.g., halt) and consequently no rules deciding whether the artificial construct should let the computation go or stop it. All one needs to do when a "halt v" is encountered is to discard the existing computation and replace it by v. This puts the same little load on the language designer as a context reduction definition of halt (see Section **??**).

To better compare MSOS and K in the context of control-intensive statements, Figure 13 shows the definition of exceptions in both formalisms. Figure 13 also shows that there are cases where labels are needed in MSOS definitions where corresponding labels are not needed in corresponding K definitions, and vice-versa. Like for halt, a special write-only label, *throw'* in this case, is necessary in the MSOS definition.

MSOS	*K*
$\dfrac{e_1 \xrightarrow{\textit{throw'}=()} e_1'}{\mathsf{try}\ e_1\ \mathsf{catch}(x)e_2 \xrightarrow{\textit{throw'}=()} \mathsf{try}\ e_1'\ \mathsf{catch}(x)e_2}$	$\llparenthesis \underset{e_1 \curvearrowright \textit{popx}}{\underline{\mathsf{try}\ e_1\ \mathsf{catch}(x)e_2 \curvearrowright r}} \rrparenthesis_k\ \llparenthesis \underset{\langle r, \rho_0\rangle}{\underline{\quad\cdot\quad}} \rrparenthesis_\chi\ \llparenthesis \rho_0 \rrparenthesis_\rho$
$\mathsf{try}\ v\ \mathsf{catch}(x)e_2 \rightarrow v$	$\llparenthesis v \curvearrowright \underset{r}{\underline{\textit{popx}}} \rrparenthesis_k\ \llparenthesis \underset{\cdot}{\underline{\langle r, \_\rangle}} \rrparenthesis_\chi$
$\dfrac{e_1 \xrightarrow{\textit{throw'}=v} e_1'}{\mathsf{try}\ e_1\ \mathsf{catch}(x)e_2 \xrightarrow{\textit{throw'}=()} (\lambda x.e_2)v}$	$\llparenthesis \underset{(\lambda x.e_2)v \curvearrowright r}{\underline{\mathsf{throw}\ v \curvearrowright \_}} \rrparenthesis_k\ \llparenthesis \underset{\cdot}{\underline{\langle r, \rho_0\rangle}} \rrparenthesis_\chi\ \llparenthesis \underset{\rho_0}{\underline{\_}} \rrparenthesis_\rho$
$\dfrac{e \rightarrow e'}{\mathsf{throw}\ e \rightarrow \mathsf{throw}\ e'}$	$\textit{Stmt} ::= \ldots \mid \mathsf{throw}\ \textit{Exp} \qquad [\textit{strict}]$
$\mathsf{throw}\ v \xrightarrow{\textit{throw'}=v} \mathsf{stuck}$	— no similar K rule needed —

Figure 13: Semantics of parametric exceptions in MSOS and K

The MSOS rules are straightforward, following the same intuition as for the definition of halt. The first MSOS rule for throw states nothing but the fact that it is strict, so its K correspondent is obvious: a strictness attribute. K needs overall fewer rules to complete the semantics of exceptions, namely three instead of four. An exception stack is added to the configuration, here referred to using the label χ; the current environment and computation are stacked when the try/catch is initiated, and are then properly recovered when the tried block terminates (normally or abnormally). Note that it is straightforward to modify the K semantics of exceptions above so that once an exception is thrown the program is terminated after executing the catch block. All we need to do is to omit adding r after $(\lambda x.e_2)v$ in the third K rule; to achieve the same modification in MSOS one would need to implement a top-level catching mechanism like the one for halt in Figure 12. The exception stack in our K definition of exceptions could be avoided, though we prefer not to. For example, one could place special markers in the computation structure when exception blocks are initiated, and then, when exceptions are thrown, discard computation items until corresponding markers are reached. This is easy to define but it is less efficient when executed. When one can execute and formally analyze programming language definitions (e.g., model checking them), efficiency is an important aspect of a language definition.

Strengths. MSOS has all the strengths of SOS. Moreover, MSOS offers the language designer a framework where modular language definitions can be developed. In other words, if one uses MSOS properly, one needs not revisit definitions of unrelated language features when adding new features to a language.

Weaknesses. With respect to definitional strength, MSOS does not improve over SOS. Any MSOS definition can be completely desugared into an SOS definition, which also gives the labeled-transition system semantics underlying MSOS. In other words, if the semantics of a language or language feature cannot be defined using SOS, then it cannot be defined using MSOS either. In particular, one can only give interleaving semantics to concurrent languages/systems using MSOS[8]. Another important limitation of MSOS is that it does not have an explicit representation and control over evaluation contexts, thus having inherent difficulty to define control-intensive language constructs. For example, at our knowledge, there is still no MSOS definition of constructs like call/cc. In a discussion with Peter Mosses in December 2007, it turned out that there could be some systematic way to provide a handle on evaluation contexts in MSOS. The idea would be to use a "builtin" label that "automatically" collects the evaluation context as the proof tree of a derivation is being build. Since in SOS-like formalisms, including MSOS, the evaluation context is captured as a proof tree, in order for the above to work it is important to understand how a proof tree can be obtained back from an evaluation context. Also, having an explicit representation of the evaluation context in one of its labels may make the use of conditional rules and SOS-like derivation trees redundant.

5.4 Context Reduction: Reduction Semantics with Evaluation Contexts

Reduction semantics with evaluation contexts, called *context reduction* for simplicity in this paper, has been introduced by Felleisen and his collaborators (see, e.g., [11, 40]) as a variant SOS where the *evaluation context is explicit* in the term being reduced.

5.4.1 Evaluation Contexts

In a context reduction language definition one starts by defining the syntax of *evaluation contexts*, or simply just *contexts*, which is typically done by means of a context-free grammar (CFG). A context is a program or fragment of program with a "hole", the hole being a placeholder where the next computational step takes place. Figure 14 shows the CFG of evaluation contexts defined for a simple imperative language like that in Figure 1; as in the previous section, we also show the corresponding K definition (the right column) for

[8]One can, again, argue that interleaving semantics is all that matters anyway. Our position is, again, that "interleaving versus true concurrency" is an important choice that needs to be addressed by the language designer not by the designer of language design frameworks. From a definitional framework design perspective, the very fact that there are language designers who prefer a true concurrency semantics for their language (and there are many) is overwhelming evidence that it should be simply supported by the framework.

a direct comparison. Note how the intended evaluation strategies of the various language constructs are

Context syntax	Language syntax	K Syntax annotation
$Cxt ::= \square$		
$\mid Cxt + AExp \mid AExp + Cxt$	$AExp ::= AExp + AExp$	*strict*
$\mid Cxt \leq AExp \mid Int \leq Cxt$	$\mid AExp \leq AExp$	*seqstrict*
$\mid \mathsf{not}\ Cxt$	$BExp ::= \mathsf{not}\ BExp$	*strict*
$\mid Cxt\ \mathsf{and} BExp$	$\mid BExp\ \mathsf{and}\ BExp$	*strict*(1)
$\mid Name := Cxt$	$Stmt ::= Name := AExp$	*strict*(2)
$\mid Cxt; Stmt$	$\mid Stmt; Stmt$	
$\mid \mathsf{if}\ Cxt\ \mathsf{then}\ Stmt\ \mathsf{else}\ Stmt$	$\mid \mathsf{if}\ BExp\ \mathsf{then}\ Stmt\ \mathsf{else}\ Stmt$	*strict*(1)
$\mid \mathsf{halt}\ Cxt$	$\mid \mathsf{halt}\ AExp$	*strict*
$\mid Cxt; AExp$	$\mid Stmt; AExp$	

Figure 14: Evaluation contexts and their K correspondent strictness attributes

reflected in the definition of evaluation contexts: "+" is non-deterministically strict (so the evaluation hole "$\square$" can go to either of its subexpressions), "$\leq$" is sequentially strict ("$\square$" goes to the first subexpression until evaluated to an *Int*, then to the second subexpression), "and" is intended to be shortcut (via an appropriate rule) so only its first argument needs to be evaluated, ":=" is strict only its second argument while sequential compositions (both of them) ";" are strict in their first argument, and the conditional is only strict in its first argument. Note that there is no production "$Cxt ::= ... \mid \mathsf{while}\ Cxt\ \mathsf{do}\ Stmt$" as one may (wrongly) expect. That is because such a production would allow the evaluation of the boolean expression in the while loop to a boolean value; supposing that that value is true, then, unless one modifies the syntax in some rather awkward way, there is no chance to recover the original boolean expression to evaluate it after the evaluation of the statement. The preferred solution to handle loops remains the same as in SOS, namely to explicitly unroll them into conditional statements. To avoid ambiguous parsing, from here on we assume left associativity for binary constructs and also take a liberty to use parentheses for grouping even though they were not defined explicitly as part of the CFG.

Here are some examples of correct evaluation contexts:

$\square$

$3 + \square$

$3 \leq \square + 7$

$5 \leq \square + (3 + a)$, where a is any well-formed arithmetic expression

$\mathsf{if}\ \square\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$, where s_1 and s_2 are any well-formed statements

$\square; x := 5$, where x is any variable

$\square; x + y$, where x and y are any variables

Here are some examples of incorrect evaluation contexts:

$\square + (2 + \square)$ — a context can have only one hole

$x + y$ — a context must contain a hole

$x + y; \square$ — one cannot place a semicolon after an expression in our grammar

$\square := 7$ — the grammar of contexts disallows this; holes are placeholders where computations can take place

$\mathsf{if\ true\ then}\ \square\ \mathsf{else\ skip}$ — a hole (or a computational step) can only appear in the condition of a conditional; one must first eliminate the conditional, using a rule skip; $\square$ — one cannot evaluate the second statement in a sequential composition; skip must be discarded first using a rule

Context reduction operational semantics is based on a tacitly assumed parsing-like mechanism that takes a program or a fragment of program p and decomposes it into a context c and a subprogram or fragment of program e, such that $p = c[e]$. Here are some examples of such decompositions:

$$7 = (\square)[7]$$
$$3 + x = (3 + \square)[x] = (\square + x)[3] = (\square)[3 + x]$$
$$3 \leq (2 + x) + 7 = (3 \leq \square + 7)[2 + x] = (\square \leq (2 + x) + 7)[3] = ...$$

If one picks the empty context $c = \square$ in the decomposition of p as $c[e]$ as we did in the first example above, then e is p and thus the characteristic rule of context reduction defined below is useless. Choosing good strategies to search for splits of terms into contextual representations can be a key factor in implementations of context reduction engines; in our experiments with embeddings of context reduction into rewriting logic (see Section 5.4.7), this was the factor that influenced the performance of the resulting reduction engine the most; in our experiments, for example, context reduction was most effective when a depth-first-search strategy for contextual representations was employed. For simplicity, we considered only one type of context in Figure 14, but in general one can have various types, depending upon the types of their "holes" and of their "result".

5.4.2 The Characteristic Context Reduction Rule

The *characteristic reduction rule underlying context reduction* is

$$\frac{e \rightarrow e'}{c[e] \rightarrow c[e']},$$

where c is any appropriate evaluation context (i.e., such that $c[e]$ and $c[e']$ are well-formed programs or fragments of program), capturing the intuition that reductions are allowed to take place only in appropriate evaluation contexts. Context reduction tends to be a purely syntactic definitional framework (following the slogan "everything is syntax"). If semantic components are necessary in a particular definition, then they are typically "swallowed by the syntax". For example, if one needs a state as part of the configuration for a particular language definition, then one adds the context production

$$\mathit{Cxt} ::= \cdots \mid \langle \mathit{Cxt}, \mathit{State} \rangle,$$

where the *State*, an inherently semantic component of the semantics, becomes part of the evaluation context. If one does not like mixing syntactic and semantic components of a definition and prefers instead to work with configuration tuples like in SOS, then one can[9] make use of the contextual representation notation only on the syntactic component of configurations, case in which the characteristic reduction rule becomes

$$\frac{\langle e, \gamma \rangle \rightarrow \langle e', \gamma' \rangle}{\langle c[e], \gamma \rangle \rightarrow \langle c[e'], \gamma' \rangle},$$

where γ and γ' consist of configuration semantic components that are necessary to evaluate e and e', respectively, such as stores, stacks, locks, etc. The first type of characteristic reduction rule above can be seen as an instance of the second when γ is empty. Advantages and disadvantages of these two styles are discussed in Section 5.4.4.

When characteristic reduction rules like above are applied, we say that *e reduces to e' in context c*. Interestingly, the characteristic rule of context reduction tends to be the only "conditional" rule, in the sense that the remaining rules take no reduction premises (though they may still require side conditions). The

[9]We are not aware of such uses of context reduction in the literature.

characteristic rules above are essentially a convenience to the designer, so that one can write semantic rules more compactly like "$\mathsf{true\ and}\ b \to b$" instead of "$c[\mathsf{true\ and}\ b] \to c[b]$". The essence of context reduction is not its characteristic reduction rule, but its approach in defining evaluation contexts as a grammar and then using them as an explicit part of languages or calculi definitions. For example, if for some reasons one does not want to have an "implicit" characteristic rule, say because one's executional framework is unconditional, then one can simply replace each rule "$l \to r$" with a rule "$c[l] \to c[r]$", stating "explicitly" that the reduction from l to r takes place in context. The characteristic rule of context reduction can therefore be regarded as "syntactic sugar", or convenience to the designer. In our theoretical developments in Sections 5.4.5 and Section 5.4.6 translating context reduction definitions automatically into rewriting logic and K definitions, respectively, we are going to assume the "desugared" version of context reduction rules.

5.4.3 From One to Many Steps

The reduction relation $\to$ above captures precisely one computational step, so it is commonly called a *one-step* or a *small-step* reduction relation. If Γ is a context reduction definition, then we write $\Gamma \vdash t \to t'$ whenever $t \to t'$ can be derived as a small-step reduction using the rules in Γ (including the characteristic rule(s), if any). The one step reduction relation captures the computational capability of the defined system and is particularly useful in formal proofs. However, to "execute" context reduction definitions one needs to apply the one-step reduction relation repetitively. The following two derivation rules define the transitive closure of the one-step reduction relation:

$$\frac{t \to t'}{t \twoheadrightarrow t'} \quad \text{and} \quad \frac{t \to u,\ u \twoheadrightarrow t''}{t \twoheadrightarrow t'}$$

As usual, we write $\Gamma \vdash t \twoheadrightarrow t'$ whenever $t \twoheadrightarrow t'$ is derivable using all the rules in Γ plus the two rules above.

5.4.4 An Example

Thanks to the characteristic reduction rule of context reduction above which is implicit in context reduction language definitions, unlike in SOS one only needs to worry about giving rules capturing basic computation steps and not how computations are propagated along the syntax. With this in mind, context reduction definitions of the simple imperative language in Figure 1 can be easily derived from a conventional SOS definition of the language. Figure 15 shows two complete context reduction semantic definitions for the simple imperative language in Figure 1, one in which the state is encapsulated as part of the evaluation context (Version 1), and another one where the state is added as part of the configuration (Version 2).

Version 1 has the advantage that reduction rules look simpler and more natural (syntactic), without carrying the state as part of the configuration everywhere just because one or two rules may need it, like in SOS. The disadvantage of this approach is that what used to be called "syntax" now contains rather semantic components, such as the state of the program. Note that once one adds syntax to evaluation contexts that does not correspond to constructs in the syntax of the language, such as the pairing of a context and a state, one needs to also extend the original syntax with corresponding constructs, so that the parsing-like mechanism decomposing a syntactic term into a context and a redex can be applied. For example, the production "$\mathit{Cxt} ::= \langle \mathit{Cxt}, \mathit{State}\rangle$" suggests that a pairing "configuration" construct of the form "$\langle \mathit{AExp}, \mathit{State}\rangle$" is also available (actually one for each syntactic category that can be reduced, not only for AExp), just like in Version 2. In that case, the top-level reductions take place between such configuration structures, just like in Version 2. The rule for halt in Version 1 could have also been "$c[\mathsf{halt}\ i] \to i$", though it would yield some perhaps undesired top-level nondeterminism: a configuration parsed as $\langle c[\mathsf{halt}\ i], \sigma\rangle$ could indeed reduce to $\langle i, \sigma\rangle$ by applying the characteristic rule of context reduction in the context $\langle \Box, \sigma\rangle$, but it could also reduce directly to i because an alternative parsing of it is "$\langle c, \sigma\rangle[\mathsf{halt}\ i]$", thus allowing top-level reductions between terms of different syntactic categories.

The advantage of Version 2 is that syntactic and semantic components of a language definition are kept separate; also, rules of language constructs that need state information for their semantics, such as variable lookup and assignment, need not mention the rest of the evaluation context. Both Version 1 and Version 2

Version 1	Version 2	K
$Cxt ::= \cdots \mid \langle Cxt, State\rangle$		
$\langle c,\sigma\rangle[x] \rightarrow \langle c,\sigma\rangle[\sigma[x]]$ or $\langle c[x],\sigma\rangle \rightarrow \langle c[\sigma[x]],\sigma\rangle$	$\langle x,\sigma\rangle \rightarrow \langle \sigma[x],\sigma\rangle$	$\llparenthesis \frac{\;x\;}{\sigma[x]} \rrparenthesis_k \ \llparenthesis \sigma \rrparenthesis_{state}$
$i_1 + i_2 \rightarrow i_1 +_{Int} i_2$	$\langle i_1 + i_2,\sigma\rangle \rightarrow \langle i_1 +_{Int} i_2,\sigma\rangle$	$i_1 + i_2 \rightarrow i_1 +_{Int} i_2$
$i_1 \leq i_2 \rightarrow i_1 \leq_{Int} i_2$	$\langle i_1 \leq i_2,\sigma\rangle \rightarrow \langle i_1 \leq_{Int} i_2,\sigma\rangle$	$i_1 \leq i_2 \rightarrow i_1 \leq_{Int} i_2$
$\mathsf{not}\ b \rightarrow \mathsf{not}_{Bool}\ b$	$\langle \mathsf{not}\ b,\sigma\rangle \rightarrow \langle \mathsf{not}_{Bool}\ b,\sigma\rangle$	$\mathsf{not}\ b \rightarrow \mathsf{not}_{Bool}\ b$
$\mathsf{true\ and}\ b \rightarrow b$	$\langle \mathsf{true\ and}\ b,\sigma\rangle \rightarrow \langle b,\sigma\rangle$	$\mathsf{true\ and}\ b \rightarrow b$
$\mathsf{false\ and}\ b \rightarrow \mathsf{false}$	$\langle \mathsf{false\ and}\ b,\sigma\rangle \rightarrow \langle \mathsf{false},\sigma\rangle$	$\mathsf{false\ and}\ b \rightarrow \mathsf{false}$
$\langle c,\sigma\rangle[x := v] \rightarrow \langle c,\sigma[v/x]\rangle[\mathsf{skip}]$ or $\langle c[x := v],\sigma\rangle \rightarrow \langle c[\mathsf{skip}],\sigma[v/x]\rangle$	$\langle x := v,\sigma\rangle \rightarrow \langle \mathsf{skip},\sigma[v/x]\rangle$	$\llparenthesis \frac{x := v}{\cdot} \rrparenthesis_k \ \llparenthesis \frac{\;\sigma\;}{\sigma[v/x]} \rrparenthesis_{state}$
$\mathsf{skip}; s \rightarrow s$	$\langle \mathsf{skip}; s,\sigma\rangle \rightarrow \langle s,\sigma\rangle$	$k_1; k_2 \rightleftharpoons k_1 \curvearrowright k_2$
$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_1$	$\langle \mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2,\sigma\rangle \rightarrow \langle s_1,\sigma\rangle$	$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_1$
$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_2$	$\langle \mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2,\sigma\rangle \rightarrow \langle s_2,\sigma\rangle$	$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_2$
$\mathsf{while}\, b\, \mathsf{do}\, s \rightarrow$ $\mathsf{if}\, b\ \mathsf{then}\ (s; \mathsf{while}\, b\, \mathsf{do}\, s)\ \mathsf{else\ skip}$	$\langle \mathsf{while}\ b\ \mathsf{do}\ s,\sigma\rangle \rightarrow$ $\langle \mathsf{if}\, b\ \mathsf{then}\ (s; \mathsf{while}\, b\, \mathsf{do}\ s)\ \mathsf{else\ skip},\sigma\rangle$	$\llparenthesis \frac{\mathsf{while}\ b\ \mathsf{do}\ s}{\mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else}\ \cdot} \rrparenthesis_k$
$\langle c,\sigma\rangle[\mathsf{halt}\ i] \rightarrow \langle i,\sigma\rangle$ or $\langle c[\mathsf{halt}\ i],\sigma\rangle \rightarrow \langle i,\sigma\rangle$	$\langle c[\mathsf{halt}\ i],\sigma\rangle \rightarrow \langle i,\sigma\rangle$	$\llparenthesis \mathsf{halt}\ i \rrparenthesis_k \rightarrow \llparenthesis i \rrparenthesis_k$
$\mathsf{skip}; a \rightarrow a$	$\langle \mathsf{skip}; a,\sigma\rangle \rightarrow \langle a,\sigma\rangle$	—no K rule needed—

Figure 15: A language definition in context reduction (left and middle columns) and K (right column)

therefore employ top-level reductions between configurations containing a program (or an arithmetic expression) and a state. If one wants to devise a top-level reduction relation that takes programs (with no states) to integers, then one needs to add rules for configuration initialization and termination, like in Figure 1.

Context reduction is an inherently small-step structural operational semantic approach: for each of the two variants discussed, the rules above define precisely one step of computation (after the potential applications of the characteristic rule of context reduction, which propagates the computational step to the top level). Therefore, in order to completely evaluate a program one needs to transitively apply the one-step transition relation. Let us next discuss an example of reduction using the first context reduction variant above. Our goal is to derive

$$[\![x := 1;\ y := 2;\ \mathsf{if\ not}(x \leq y)\ \mathsf{then}\ x := 0\ \mathsf{else}\ y := 0;\ x + y,\ \emptyset]\!] \twoheadrightarrow \langle 1,\sigma\rangle$$

for some state σ. From here on we use the following (standard) notation for instantiated contexts: the redex is placed in a box replacing the hole of the context. For example, the fact that expression $2 \leq (x{+}x){+}y$ is seen as a context instance $(2 \leq \square{+}y)[x{+}x]$ is written compactly and intuitively as follows: $2 \leq \boxed{x + x}{+}y$. With this notation, one gets the sequence of reductions in Figure 16, where the label above the arrow represents the implicit and obvious name of the rule that was applied; boxed labels symbolize that the corresponding reduction step was applied in context, using implicitly the characteristic context reduction rule. To avoid writing parentheses for disambiguation, we assume sequential composition to be left-associative. Note that the evaluation context changes almost at each step during the reduction sequence in Figure 16.

Strengths. Context reduction improves over SOS in at least two ways:

1. Since reductions can only happen in proper evaluation contexts and propagation of computation is done once and for all using the characteristic rule of context reduction, one needs not worry about propagating local computations up-wards along the syntax; this way, definitions in context reduction

$$\begin{array}{ll}
 & \langle \boxed{x := 1}; y := 2;\ \mathsf{if\ not}(x \leq y)\ \mathsf{then}\ x := 0\ \mathsf{else}\ y := 0;\ x + y, \emptyset \rangle \\
\xrightarrow{:=} & \langle \boxed{\mathsf{skip};\ y := 2};\ \mathsf{if\ not}\ x \leq y\ \mathsf{then}\ x := 0\ \mathsf{else}\ y := 0;\ x + y,\ (x \mapsto 1) \rangle \\
\xrightarrow{\boxed{\mathsf{skip}}} & \langle \boxed{y := 2};\ \mathsf{if\ not}\ x \leq y\ \mathsf{then}\ x := 0\ \mathsf{else}\ y := 0;\ x + y,\ (x \mapsto 1) \rangle \\
\xrightarrow{:=} & \langle \boxed{\mathsf{skip};\ \mathsf{if\ not}\ x \leq y\ \mathsf{then}\ x := 0\ \mathsf{else}\ y := 0};\ x + y,\ (x \mapsto 1, y \mapsto 2) \rangle \\
\xrightarrow{\boxed{\mathsf{skip}}} & \langle \mathsf{if\ not}\ \boxed{x} \leq y\ \mathsf{then}\ x := 0\ \mathsf{else}\ y := 0; x + y,\ (x \mapsto 1, y \mapsto 2) \rangle \\
\xrightarrow{\mathrm{lookup}} & \langle \mathsf{if\ not}\ 1 \leq \boxed{y}\ \mathsf{then}\ x := 0\ \mathsf{else}\ y := 0;\ x + y,\ (x \mapsto 1, y \mapsto 2)) \rangle \\
\xrightarrow{\mathrm{lookup}} & \langle \mathsf{if\ not}\ \boxed{1 \leq 2}\ \mathsf{then}\ x := 0\ \mathsf{else}\ y := 0;\ x + y,\ (x \mapsto 1, y \mapsto 2) \rangle \\
\xrightarrow{\boxed{\leq}} & \langle \mathsf{if}\ \boxed{\mathsf{not\ false}}\ \mathsf{then}\ x := 0\ \mathsf{else}\ y := 0;\ x + y,\ (x \mapsto 1, y \mapsto 2) \rangle \\
\xrightarrow{\boxed{\mathsf{not\_false}}} & \langle \boxed{\mathsf{if\ true\ then}\ x := 0\ \mathsf{else}\ y := 0};\ x + y,\ (x \mapsto 1, y \mapsto 2) \rangle \\
\xrightarrow{\boxed{\mathsf{if}_{\mathsf{false}}}} & \langle \boxed{y := 0};\ x + y,\ (x \mapsto 1, y \mapsto 2) \rangle \\
\xrightarrow{:=} & \langle \boxed{\mathsf{skip}; x + y},\ (x \mapsto 1, y \mapsto 0) \rangle \\
\xrightarrow{\boxed{\mathsf{skip}}} & \langle x + \boxed{y},\ (x \mapsto 1, y \mapsto 0) \rangle \\
\xrightarrow{\mathrm{lookup}} & \langle \boxed{x} + 0,\ (x \mapsto 1, y \mapsto 0) \rangle \\
\xrightarrow{\mathrm{lookup}} & \langle \boxed{1 + 0},\ (x \mapsto 1, y \mapsto 0) \rangle \\
\xrightarrow{\boxed{+}} & \langle 1, (x \mapsto 1, y \mapsto 0) \rangle
\end{array}$$

Figure 16: Sample reduction sequence using the context reduction definition in Figure 15

become *more compact* than SOS definitions. In simple terms, context reduction yields more compact definitions than ordinary SOS because it uses parsing to find the next redex rather than towered small-step rules.

2. It provides the possibility of also *altering the evaluation context* in which a reduction occurs. More precisely, unlike SOS, context reduction allows the possibility to handle evaluation contexts like any other entities in the language, in particular to pass them to other contexts or to store them. This way one can easily and elegantly define the semantics of control-intensive language features such as return of functions, break/continue of loops, exceptions, and even callcc.

As shown shortly, context reduction can be faithfully translated (i.e., nothing is lost in translation) into rewriting logic and even in a small fragment of K, replacing each context reduction rule by a corresponding rewrite logic or K rule. That implies that language definitions in K are provably at least as powerful and compact as those using context reduction. However, when using the unleashed computation handling capabilities of K by disobeying what was called in [5] "the rigidity of syntax", one can often write even more compact, and in our (admittedly subjective) view more natural, language definitions.

Weaknesses. The advantages of context reduction over SOS-like approaches, namely the compactness of definitions and the explicit control over evaluation contexts, come at a price. We next discuss the drawbacks of context reduction, first by comparison with SOS and then in general.

The discussion above may imply that context reduction is strictly superior to SOS or other syntactic approaches. That is, unfortunately, not true. SOS' conditional rules have an important advantage, which cannot be recovered in context reduction: they easily and naturally create execution environments just for the sub-expressions (or other fragments of program) to be processed as part of the condition, without having to worry about recovering the execution environment when the processing of the sub-expression is finished. For example, recall the MSOS definition (one can obtain an equivalent SOS definition by desugaring MSOS into SOS [29]) of let in Figures 8 and 10,

$$\frac{e_1 \longrightarrow e_1'}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \longrightarrow \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2}$$

$$\frac{e_2 \xrightarrow{env=\rho[v_1/x]} e_2'}{\mathsf{let}\ x = v_1\ \mathsf{in}\ e_2 \xrightarrow{env=\rho} \mathsf{let}\ x = v_1\ \mathsf{in}\ e_2'}$$

$$\mathsf{let}\ x = v_1\ \mathsf{in}\ v_2 \longrightarrow v_2$$

These rules say that the binding expression e_1 must be first evaluated to a value v_1, which is then bound to x in the environment while the body of the let, e_2, is being reduced; the value v_2 that e_2 hereby eventually reduces to is the result of the evaluation of the let. The important thing to notice here is the second rule: e_2 is reduced in one step to e_2' in a modified environment ($\rho[v_1/x]$), while let $x = v_1$ in e_2 reduces to let $x = v_1$ in e_2' under the original environment (ρ). When using SOS or MSOS, the language designer is therefore not concerned with "recovering" the environment after the reduction of e_2. A similar scenario happens when giving an SOS semantics of function invocation in an environment-based definition. Unfortunately, neither of these is possible in context reduction. Even though the third rule above could be used as is in a hypothetical context reduction definition and the first rule can be replaced by an evaluation context production giving evaluation access to the binding expression,

$$\mathit{Cxt} ::= ...\ |\ \mathsf{let}\ \mathit{Name} = \mathit{Cxt}\ \mathsf{in}\ \mathit{Exp},$$

there is no way to find an equivalent for the second rule in context reduction. What one would like to say is, using an unconditional rule, that in an expression of the form let $x = v_1$ in e_2 evaluation access is given to e_2, *but* in the modified environment $\rho[v_1/x]$; then, once e_2 is completely reduced, the environment changes back to ρ. This is simply not possible in context reduction, unless one modifies the syntax in some rather aggressive way (introducing, e.g., explicit environment restore statements), which would lack generality and be against the "syntactic" spirit of context reduction.

Also, one should not be fooled thinking that context reduction is a generalization of SOS, so one can use conditional SOS-like rules as an escape whenever a particular task cannot be accomplished using unconditional context reduction rules. For example, one cannot use a rule like the second one above, simply because one "loses the context" in the condition; indeed, if e_2 reduces to a call/cc during the processing of the condition, then one cannot provide it with the whole evaluation context. How does K solve these problems, considering that it is also an unconditional-rule framework? The answer is that K, by its nature, explicitly disobeys the purity of syntax; in the case of let, one uses the computation structure to perform the desired tasks in the desired order (that is the precise role of the computation structure): bind x to v_1, schedule e_2 for processing, then recover the original environment. Similar environment recovering steps need to be taken when defining control-changing statements such as return of functions, exceptions, or call/cc. Neither of these can be defined in context reduction following an environment-based definitional style.

The designated approach in context reduction to avoid the complications discussed in the paragraph above is to discard the environment-based style all together and follow instead a substitution-based style. Substitution is the basis of λ-calculus' β-reduction rule and can be used to define other common language constructs, such as let and letrec. However, some language constructs that have straightforward environment-based definitions, such as references, objects, threads, etc., require quite intricate and tricky substitution-based definitions, if any can be found. While substitution-based semantic techniques are interesting in their own way and are worth considering in some practical cases, we believe that a language designer should simply not be forced by the underlying framework to follow a substitution-based style. Moreover, an executable definitional framework should not disregard efficiency, and substitution is expensive. An executable language definition in which each computational step has a complexity linear in the size of the program is simply impractical (wrt executability); not to mention that the size of the program to reduce can grow unbounded in a substitution-based framework. As seen in Section 4, K also allows substitution-based definitions which are as simple and natural, if not simpler and more natural, than their corresponding context reduction definitions. However, K does not enforce, encourage or discourage substitution-based definitions. We believe that that should be the choice of the language designer and not imposed by the framework.

We next discuss several other weaknesses of context reduction, not necessarily by comparison with SOS.

- Context reduction is based on an *implicit mechanism to split a program* or a fragment of program p into an evaluation context c and a fragment of program, also called *redex*, e, so that $p = c[e]$. This split operation is somehow assumed atomic and non-computational, so it elegantly captures and hides all the small-step propagation rules of SOS. However, this decomposition is not obvious and may require significant computational resources, sometimes linear in the size of the program or worse. Moreover, since a new evaluation context needs to be recalculated at almost any reduction step, it can easily become the major bottleneck in the efficiency of an interpreter implemented following the context reduction approach. While such an interpreter would be no worse than one implemented following the small-step SOS approach, it can still be a lot slower than it needs to be, thus making context reduction unattractive as an executable language definitional framework (see the "ideal framework" requirements in Section 1). Techniques such as *refocusing* [**?**] have been proposed to incrementally compute the next evaluation context, but these techniques appear to work only when the decomposition of the program into an evaluation context and a redex is deterministic.

- It is inconvenient to define concurrent languages using context reduction. That is because concurrent processes typically communicate with each other from different places in the configuration, so one may need contexts with more than one hole, preferably with an arbitrary number of them. While some attempts to define and handle multi-contexts have been proposed (see, e.g., [**?**]), computing and matching such contexts on complex configurations becomes a significantly non-trivial issue with potentially serious impact on performance.

- Context reduction is still far from serving as a solid foundation for concurrency, because, just like SOS, it only captures an interleaving semantics of a defined concurrent system. In particular, context reduction can only be used to give an interleaving semantics of CCS (middle column in Figure 5), but not a truly concurrent one (right column in Figure 5).

- It still says nothing about models, being essentially a purely syntactic reduction technique.

K has all the strengths of context reduction, at the same time avoiding its limitations. Indeed, K's matching is comparatively cheap and rather standard in term rewriting; existing advanced indexing techniques in term rewriting can be and are being used (by running K in Maude, for example) to efficiently execute K definitions. K is particularly useful to define concurrent languages, because, if used properly, can give a truly concurrent semantics to the defined languages at the same time also allowing an interleaving semantics — K's approach is to let the language designer, rather than the limitations of the framework, decide what semantics for concurrency is desired. Also, K has a model-theoretical semantics, for the time being borrowed via its desugaring into rewriting logic.

5.4.5 Context Reduction as a Methodological Fragment of Rewriting Logic

In this section we show how one can automatically embed context reduction into rewrite logic. We first give a straightforward embedding, which is easy to prove correct but which does not take advantage of performance-improving techniques currently supported by rewrite engines, so consequently it is relatively slow when executed or formally analyzed. We then discuss simple optimizations which increases the performance of the resulting rewrite definitions an order of magnitude or more. All the embeddings of context reduction into rewriting logic discussed in this section are simple enough that one can use a rewrite engine directly to write, execute and formally analyze context reduction definitions without a need to implement a translator. In this paper we only consider evaluation contexts which can be defined by means of a context-free grammar (CFG). However, the CFG defining evaluation contexts can be non-deterministic, in the sense that a term is allowed to be split many different ways as a context and a reducible redex.

All our embeddings are based on an explicit machinery to non-deterministically split a term into a context and a subterm, as well as to plug a subterm into a context and thus obtain a larger term. Figure 17 shows a general and automatic procedure to generate a rewriting logic specification from any evaluation context grammar, followed by an example instantiating this general procedure on the evaluation context grammar in Figure 14. For simplicity of both theoretical (embedding transformations and their correctness proofs) and practical (execution and formal analysis) developments, we prefer to collapse all the syntactic categories into one category, *Syntax*, and all the context categories into another one, *Context*. When using order-sorted term rewrite settings (e.g., Maude), that can be elegantly done by adding the two syntactic categories as new sorts and then subsorting the syntactic and context sorts to them, respectively; this approach has the advantage that the rewrite framework may reject programs which are not well-formed without a need for an external parser. If subsorting is not available in one's setting, then one can simply replace each syntactic category by either *Syntax* or *Context*. Our construction also works without collapsing of non-terminals, but it is more technical, requires more operations, rules and equations, and it is likely not worth the effort without a real motivation to use it in term rewrite settings without support for subsorting. We have made experiments with both approaches and found no penalty on performance when collapsing syntactic categories.

The implicit context reduction notation "*context*[*term*]" for contextual representations of terms, as well as its implicitly assumed "split" of syntax and "plug" into contexts, are defined explicitly using the new syntactic categories. A term $c[t]$ can be thought of as a contextual representation of some term, where c is an evaluation context and t is the sub-term in the hole. The *plug* operation is only defined on terms in contextual representation. One generic (i.e., independent upon the particular context reduction definition being embedded) rule and one generic equation are added: "$\mathit{split}(N) \to \square[N]$" initiates the process of splitting a term into a contextual representation and "$\mathit{plug}(\square[N]) = N$" terminates the process of plugging a term into a context. It is important that the first be a rewrite rule (because it can lead to non-determinism; this is explained below), while the second can safely be an equation.

Each evaluation context production translates into one equation and one conditional rewrite rule. The equation tells how contexts having that production at the top are instantiated (or the plug operation, following the terminology of context reduction), while the conditional rule tells how that production can be used to split a term into a context and a subterm. The equations defining *plug* are straightforward: for each production in the original CFG of evaluation contexts, iteratively plug the subterm in the smaller context; when the hole is reached, replace it by the subterm via the generic equation. The conditional rules for *split* also look straightforward, but how and why they work is more subtle. For any context production, if the term to split matches the pattern of the production, then first split the subterm corresponding to the position of the subcontext and then use that contextual representation of the subterm to construct the contextual representation of the original term; at any moment, one has the option to stop splitting thanks to the generic rule "$\mathit{split}(N') \to \square[N']$". The reason for which we use a conditional rule instead of a conditional equation is that splitting, unlike plugging, can be non-deterministic. Recall that the use of an arrow/transition in the condition of a rule has an existential nature and that, in Maude, that is executed by performing an exhaustive search, or reachability analysis of all the zero-, one- or more-step rewrites of the condition lhs ($\mathit{split}(N)$ in our case) into the condition rhs ($C[R]$ in our case). Indeed, if one uses Maude's search command one can list all possible splits, or "parsings", of a term into a contextual representation.

(1) Context reduction syntax $\rightsquigarrow$ Rewriting logic syntax + 1 rule + 1 equation

Syntactic nonterminals N_1, N_2, ...	$\rightsquigarrow$	One non-terminal (or sort), *Syntax*
Context nonterminals Cxt_1, Cxt_2, ...	$\rightsquigarrow$	One non-terminal (or sort), *Context*
Implicit notation "*context*[*term*]" and implicit "split" of syntax and "plug" into context	$\rightsquigarrow$	$\left\{\begin{array}{l} Syntax ::= ... \mid Context[Syntax] \mid split(Syntax) \mid plug(Syntax) \\ split(N) \to \square[N] \\ plug(\square[N]) = N \end{array}\right.$

(2) Context production $\rightsquigarrow$ Rewriting logic equation and conditional rule

$$Cxt' ::= \pi(N_1, ..., N_n, Cxt) \quad \rightsquigarrow \quad \begin{array}{l} split(\pi(N_1, ..., N_n, N)) \to \pi(N_1, ..., N_n, C)[R] \ \texttt{if} \ split(N) \to C[R] \\ plug(\pi(N_1, ..., N_n, C)[R]) = \pi(N_1, ..., N_n, plug(C[R])) \end{array}$$

where

$Cxt' ::= \pi(N_1, ..., N_n, Cxt)$ is an evaluation context production with $N_1, ..., N_n$ all its syntactic nonterminals and Cxt its contextual nonterminal. In other words, the implicit context reduction "parsing", as well as its assumed "split" and "plug" operations, are replaced by explicit rewrite logic sentences.

Evaluation context grammar	Corresponding rewriting logic rules and equations
$Cxt ::= \square$	$split(p) \to \square[p]$ $plug(\square[p]) = p$
$Cxt ::= ... \mid Cxt + AExp$	$split(a_1 + a_2) \to (c + a_2)[r]$ `if` $split(a_1) \to c[r]$ $plug((c + a_2)[r]) = plug(c[r]) + a_2$
$Cxt ::= ... \mid AExp + Cxt$	$split(a_1 + a_2) \to (a_1 + c)[r]$ `if` $split(a_2) \to c[r]$ $plug((a_1 + c)[r]) = a_1 + plug(c[r])$
$Cxt ::= ... \mid Cxt \leq AExp$	$split(a_1 \leq a_2) \to (c \leq a_2)[r]$ `if` $split(a_1) \to c[r]$ $plug((c \leq a_2)[r]) = plug(c[r]) \leq a_2$
$Cxt ::= ... \mid Int \leq Cxt$	$split(i_1 \leq a_2) \to (i_1 \leq c)[r]$ `if` $split(a_2) \to c[r]$ $plug((i_1 \leq c)[r]) = i_1 \leq plug(c[r])$
$Cxt ::= ... \mid \mathsf{not}\ Cxt$	$split(\mathsf{not}\ b) \to (\mathsf{not}\ c)[r]$ `if` $split(b) \to c[r]$ $plug((\mathsf{not}\ c)[r]) = \mathsf{not}\ plug(c[r])$
$Cxt ::= ... \mid Cxt\ \mathsf{and}\ BExp$	$split(b_1\ \mathsf{and}\ b_2) \to (c\ \mathsf{and}\ b_2)[r]$ `if` $split(b_1) \to c[r]$ $plug((c\ \mathsf{and}\ b_2)[r]) = plug(c[r])\ \mathsf{and}\ b_2$
$Cxt ::= ... \mid Name := Cxt$	$split(x := a) \to (x := c)[r]$ `if` $split(a) \to c[r]$ $plug((x := c)[r]) = x := plug(c[r])$
$Cxt ::= ... \mid Cxt; Stmt$	$split(s_1; s_2) \to (c; s_2)[r]$ `if` $split(s_1) \to c[r]$ $plug((c; s_2)[r]) = plug(c[r]); s_2$
$Cxt ::= ... \mid \mathsf{if}\ Cxt\ \mathsf{then}\ Stmt\ \mathsf{else}\ Stmt$	$split(\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2) \to (\mathsf{if}\ c\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)[r]$ `if` $split(b) \to c[r]$ $plug((\mathsf{if}\ c\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)[r]) = \mathsf{if}\ plug(c[r])\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$
$Cxt ::= ... \mid \mathsf{halt}\ Cxt$	$split(\mathsf{halt}\ a) \to (\mathsf{halt}\ c)[r]$ `if` $split(a) \to c[r]$ $plug((\mathsf{halt}\ c)[r]) = \mathsf{halt}\ plug(c[r])$
$Cxt ::= ... \mid Cxt; AExp$	$split(s; a) \to (c; a)[r]$ `if` $split(s) \to c[r]$ $plug((c; a)[r]) = plug(c[r]); a$

Figure 17: Embedding an evaluation context grammar into rewriting logic ($\Gamma \rightsquigarrow \mathcal{R}_\Gamma^\square$) and example

Context reduction rule	$\rightsquigarrow$	Rewriting logic conditional rule
$rule_i: \quad l(c_1[l_1], ..., c_n[l_n]) \to r(c'_1[r_1], ..., c'_{n'}[r_{n'}])$	$\rightsquigarrow$	$rule_i: \quad \{l(N_1, ..., N_n)\} \to \{r(plug(c'_1[r_1]), ..., plug(c'_n[r_{n'}]))\}$ $\texttt{if}\ split(N_1) \to c_1[l_1] \wedge \cdots \wedge split(N_n) \to c_n[l_n]$

where

"$Syntax$ ::= ... | {$Syntax$}" is an additional "wrapper", with the same intuition as the homonymous wrapper in the embedding of SOS in Section 5.1.1: $\{t\}$ inhibits "uncontrolled" rewrites in t, all rewrites take place at the top.

Context reduction rules	Corresponding rewriting logic rules
$\langle c, \sigma\rangle[x] \to \langle c, \sigma\rangle[\sigma[x]]$ or	$\{p\} \to \{plug(\langle c, \sigma\rangle[\sigma[x]])\}$ $\texttt{if}$ $split(p) \to \langle c, \sigma\rangle[x]$ or
$\langle c[x], \sigma\rangle \to \langle c[\sigma[x]], \sigma\rangle$	$\{\langle p, \sigma\rangle\} \to \{\langle plug(c[\sigma[x]]), \sigma\rangle\}$ $\texttt{if}$ $split(p) \to c[x]$
$i_1 + i_2 \to i_1 +_{Int} i_2$	$\{p\} \to \{plug(c[i_1 +_{Int} i_2])\}$ $\texttt{if}$ $split(p) \to c[i_1 + i_2]$
$i_1 \leq i_2 \to i_1 \leq_{Int} i_2$	$\{p\} \to \{plug(c[i_1 \leq_{Int} i_2])\}$ $\texttt{if}$ $split(p) \to c[i_1 \leq i_2]$
$\mathsf{not}\ b \to \mathsf{not}_{Bool}\ b$	$\{p\} \to \{plug(c[\mathsf{not}_{Bool}\ b])\}$ $\texttt{if}$ $split(p) \to c[\mathsf{not}\ b]$
$\mathsf{true\ and}\ b \to b$	$\{p\} \to \{plug(c[b])\}$ $\texttt{if}$ $split(p) \to c[\mathsf{true\ and}\ b]$
$\mathsf{false\ and}\ b \to \mathsf{false}$	$\{p\} \to \{plug(c[\mathsf{false}])\}$ $\texttt{if}$ $split(p) \to c[\mathsf{false\ and}\ b]$
$\langle c, \sigma\rangle[x := v] \to \langle c, \sigma[v/x]\rangle[\mathsf{skip}]$ or	$\{p\} \to \{plug(\langle c, \sigma[v/x]\rangle[\mathsf{skip}])\}$ $\texttt{if}$ $split(p) \to \langle c, \sigma\rangle[x := v]$ or
$\langle c[x := v], \sigma\rangle \to \langle c[\mathsf{skip}], \sigma[v/x]\rangle$	$\{\langle p, \sigma\rangle\} \to \{\langle plug(c[\mathsf{skip}]), \sigma[v/x]\rangle\}$ $\texttt{if}$ $split(p) \to c[x := v]$
$\mathsf{skip}; s \to s$	$\{p\} \to \{plug(c[s])\}$ $\texttt{if}$ $split(p) \to c[\mathsf{skip}; s]$
$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \to s_1$	$\{p\} \to \{plug(c[s_1])\}$ $\texttt{if}$ $split(p) \to c[\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2]$
$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \to s_2$	$\{p\} \to \{plug(c[s_2])\}$ $\texttt{if}$ $split(p) \to c[\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2]$
$\mathsf{while}\, b\, \mathsf{do}\, s \to \mathsf{if}\, b\ \mathsf{then}\, (s; \mathsf{while}\, b\, \mathsf{do}\, s)\ \mathsf{else\ skip}$	$\{p\} \to \{plug(c[\mathsf{if}\, b\ \mathsf{then}\, (s; \mathsf{while}\, b\, \mathsf{do}\, s)\ \mathsf{else\ skip}])\}$ $\texttt{if}$ $split(p) \to c[\mathsf{while}\, b\, \mathsf{do}\, s]$
$\langle c, \sigma\rangle[\mathsf{halt}\ i] \to \langle i, \sigma\rangle$ or	$\{p\} \to \{\langle i, \sigma\rangle\}$ $\texttt{if}$ $split(p) \to \langle c, \sigma\rangle[\mathsf{halt}\ i]$ or
$\langle c[\mathsf{halt}\ i], \sigma\rangle \to \langle i, \sigma\rangle$	$\{\langle p, \sigma\rangle\} \to \{\langle i, \sigma\rangle\}$ $\texttt{if}$ $split(p) \to c[\mathsf{halt}\ i]$
$\mathsf{skip}; a \to a$	$\{p\} \to \{plug(c[a])\}$ $\texttt{if}$ $split(p) \to c[\mathsf{skip}; a]$

Figure 18: First embedding of context reduction into rewriting logic ($\Gamma \rightsquigarrow \mathcal{R}^1_\Gamma$) and example

Theorem 9. ***(Embedding splitting/plugging into rewriting logic)*** *Let Γ be any context reduction definition (we are only interested in its evaluation context syntax here), and let $\mathcal{R}^\square_\Gamma$ be the rewrite logic theory associated to Γ using the embeddings of syntax (1) and contexts (2) in Figure 17. Then the following are equivalent for any $t \in Syntax$:*

- *t can be split or "parsed" as $c[r]$ in Γ;*
- *$\mathcal{R}^\square_\Gamma \vdash split(t) \to c[r]$;*
- *$\mathcal{R}^\square_\Gamma \vdash plug(c[r]) = t$.*

The theorem above says that the process of splitting a term t into a context and a redex in context reduction, which can be non-deterministic, reduces to reachability in the corresponding rewrite logic theory of a contextual representation pattern $c[r]$ of the original term marked for splitting, $split(t)$. Rewrite engines such as Maude provide a search command that does precisely that.

Figure 18 shows our first embedding of context reduction into rewriting logic. Each context reduction semantic rule translates into one rewrite logic conditional rule. We allow context reduction rules to have in their lhs and rhs an arbitrary number of subterms in contextual representation. For example, if the lhs l of a context reduction rule has n such subterms, say $c_1[l_1]$, ..., $c_n[l_n]$, then we write it $l(c_1[l_1], ..., c_n[l_n])$ (we avoid the notation $l[c_1[l_1], ..., c_n[l_n]]$ because the use of square brackets for both object- and meta-notation may be confusing). Moreover, note that we allow contexts to have any pattern, not only context variables; for example, a variable lookup context reduction rule can be either "$\langle c, \sigma\rangle[x] \to \langle c, \sigma\rangle[\sigma(x)]$" or "$\langle c[x], \sigma\rangle \to \langle c[\sigma[x]], \sigma\rangle$". A rule in context reduction operates as follows: (1) match the lhs pattern at the top of the term to reduce, making sure that each of the subterms corresponding to the contextual representation sub-patterns can indeed be split as indicated; and (2) then reduce the original term to the rhs pattern instantiated accordingly, plugging all the subterms appearing in contextual representations in the rhs. The above does not exclude matching contexts in the lhs or storing them in the rhs, as needed for example to define constructs that change the control abruptly, such as call/cc; all is required here is that contextual representations appearing in the lhs have the meaning of a split, while those in the rhs have the meaning of a plug. Our corresponding conditional rewrite rule does precisely that: the {_} in the lhs guarantees that the reduction takes place at the top of the original term, the condition exhaustively searches for all the splits, and the rhs plugs all the contextual representations. The only difference between the original context reduction rule and its corresponding rewrite logic conditional rule is that the rewrite logic rule makes explicit the splits and plugs that are implicit in the context reduction rule.

Theorem 10. ***(First faithful embedding of context reduction into rewriting logic)*** *Let Γ be any context reduction definition and let $\mathcal{R}^1_\Gamma$ be the rewrite logic theory associated to Γ using the embedding procedure in Figure 18. Then*

1. ***(step-for-step correspondence)*** $\Gamma \vdash t \to t'$ *using rule$_i$ iff* $\mathcal{R}^1_\Gamma \vdash \{t\} \to^1 \{t'\}$ *using rule$_i$; moreover, rule$_i$ applies similarly (same contexts, same substitution; all modulo correspondence in Theorem 9.);*

2. ***(computational correspondence)*** $\Gamma \vdash t \twoheadrightarrow t'$ *iff* $\mathcal{R}^1_\Gamma \vdash \{t\} \to \{t'\}$.

The first item above says that the resulting rewrite logic theory captures faithfully the small-step reduction relation of the original context reduction definition. The faithfulness of this embedding (i.e., there is precisely one top-level application of a rewrite rule that corresponds to a context reduction rule), comes from the fact that the consistent use of the "{_}" wrapper inhibits any other application of any other rule on the wrapped term. Therefore, a small-step reduction in context reduction also reduces to reachability analysis in the corresponding rewrite theory; one can also use the search capability of a system like Maude to find all the next terms that a given term evaluates to (Maude provides the capability to search for the first n terms that match a given pattern using up to m rules, where n and m are user-provided parameters). Note that this step-for-step correspondence is stronger (and better) than the strong bisimilarity of the two definitions; for example, if *rule$_i$* can be applied two different ways in Γ, then its corresponding rule can also be applied two different ways in $\mathcal{R}^1_\Gamma$. The second item above says that the resulting rewrite theory can be used to perform any computation possible in the original context reduction theory, and vice-versa (the step-for-step correspondence is guaranteed in combination with the first item above). Therefore, there is absolutely no difference between computations using Γ and computations using $\mathcal{R}^1_\Gamma$, except for irrelevant syntactic conventions/notations. This strong correspondence between reductions in Γ and rewrites in $\mathcal{R}^1_\Gamma$ tells that $\mathcal{R}^1_\Gamma$ *is precisely* Γ, *not an encoding of it.* In other words, context reduction can be faithfully regarded as a methodological fragment of rewriting logic, same like SOS.

The discussion above implies that, from a theoretical perspective, the embedding in Figure 18 is as good as one can hope. However, its simplicity cames at a price in performance, which unfortunately tends to be at its worst precisely in the most common cases. Consider, for example, the context reduction rules for "+" and "≤" in the language definition in Figure 15, which after desugaring of the characteristic context reduction rule (as explained at the end of Section 5.4.2) become:

$$\begin{array}{c} c[i_1 + i_2] \to c[i_1 +_{Int} i_2] \\ c[i_1 \leq i_2] \to c[i_1 \leq_{Int} i_2]. \end{array}$$

With the embedding in Figure 18, these translate into the conditional rewrite rules

$$\begin{array}{c} \{p\} \to plug(c[i_1 +_{Int} i_2]) \ \texttt{if} \ split(p) \to c[i_1 + i_2] \\ \{p\} \to plug(c[i_1 \leq_{Int} i_2]) \ \texttt{if} \ split(p) \to c[i_1 \leq i_2], \end{array}$$

where p ranges over *Syntax*. In fact, as seen in the example in Figure 18, all the lhs'es of all the rewrite rules corresponding to the context reduction rules in the first definition in Figure 15 will have the form $\{p\}$. The reason the lhs'es of the rewrite rules are the same and lack any structure is because the contextual representations in the lhs'es of the reduction rules appear at the top, with no structure above them, which is the most common type of context reduction rule encountered. The $\lambda\mu$-calculus defined in Figure **??** also has only rules of this type.

To apply a conditional rewrite rule as above, a rewrite engine would first match the lhs and then would perform the expensive (exhaustive) search in the condition. In other words, the structure of the lhs acts as a cheap "guard" for the expensive search. Unfortunately, since the lhs has no structure, it will always match. That means that the expensive searches in the conditions of all the rewrite rules will be, in the worst case, executed one after the other until a split is eventually found (if any). If one thinks in terms of implementing context reduction in general, then this is what a naive implementation would do. If one thinks in terms of executing term rewrite systems, then this fails to take advantage of some important performance-increasing advances in term rewriting, such as *indexing* [**?**]. In short, indexing techniques use the structure of the rules' lhs'es to augment the term structure to be rewritten with information about which rule can potentially be applied in which nodes. This information is dynamically updated, as the term is rewritten. If the rules' lhs'es do not significantly overlap, it is generally assumed that it takes constant time to find a matching rewrite rule. This is similar in spirit to hashing, where the access time into a hash table is generally assumed to take constant time when there are no or little key collisions. Thinking intuitively in terms of hashing, from an indexing perspective a rewrite system with rules having the same lhs'es is as bad/useless as a hash table in which all accesses are collisions.

Ideally, in an efficient implementation of context reduction one would like to adapt/modify indexing techniques, which currently work for context-insensitive term rewriting, or to invent new techniques serving the same purpose. This seems highly non-trivial and tedious, though. An alternative is to device an embedding transformation of context reduction into term rewriting that takes better or full advantage of existing, context-insensitive indexing. Without context-sensitive indexing hardwired in the reduction engine, due to the inherent non-determinism in parsing/splitting syntax into contextual representations it can be shown that in the worst case one needs to search the entire term to find a legal position where a reduction can take place. What we would like to achieve though is a quick test for which rule applies on a particular redex once a split is found. Such a quick test can be achieved for free on rewrite systems making use of indexing (e.g., Maude) if one slightly modifies the embedding translation of context reduction into rewriting logic as shown in Figure 19. The main idea is to keep the structure of the lhs of the reduction rules in the lhs of the corresponding rewriting rules. This structure is crucial for indexing. To allow it, one needs to do the necessary splitting as a separate step. The first rewrite rule associated to a context reduction rule enables the splitting process on the corresponding contextual representations in the lhs of the original reduction rule. An immediate effect of these rules is that, unlike in the first embedding, a term can now be split at multiple positions, not only at its top. Note that the splitting "task" needs to be propagated at each step (achieved prepending the *split* operation on top of the rhs).

Theorem 11. ***(Second faithful embedding of context reduction into rewriting logic)*** *Let Γ be any context reduction definition and let $\mathcal{R}^2_\Gamma$ be the complete rewrite logic theory associated to Γ using the embedding procedure in Figure 19. Then*

1. ***(matching becomes deduction)*** *A term t matches $rule_i$ in Γ whose lhs is $l(c_1[l_1], ..., c_n[l_n])$ iff $\mathcal{R}^2_\Gamma \vdash split(t) \overset{split}{\rightarrow^\star} l(c_1[l_1], ..., c_n[l_n])$, where $\overset{split}{\rightarrow^\star}$ is the rewrite relation restricted to split rules;*

2. ***(step-for-step correspondence)*** *$\Gamma \vdash t \to t'$ using $rule_i$ iff $\mathcal{R}^2_\Gamma \vdash split(t) \overset{split}{\rightarrow^\star} u$ and $\mathcal{R}^2_\Gamma \vdash \{u\} \to^1 \{split(t')\}$ using $rule_i$; moreover, $rule_i$ applies similarly (same contexts and same substitution);*

3. ***(computational correspondence)*** *$\Gamma \vdash t \twoheadrightarrow t'$ iff $\mathcal{R}^2_\Gamma \vdash \{split(t)\} \to \{\Box[t']\}$.*

Like for the first embedding, one can now use a context-insensitive rewrite engine, in particular its search capabilities, to explore computations in the original context reduction definition. For example, if one is

Context reduction rule	$\leadsto$	Two rewriting logic unconditional rules
rule_i : $l(c_1[l_1], ..., c_n[l_n]) \to r(c'_1[r_1], ..., c'_{n'}[r_{n'}])$	$\leadsto$	$\begin{cases} \text{if } l \text{ is proper (i.e., not a variable) then add the rule} \\ \quad \mathit{split}(l(N_1, ..., N_n)) \to l(\mathit{split}(N_1), ..., \mathit{split}(N_n)) \\ \\ \text{add the rule} \\ \mathit{rule}_i : \\ \quad \{l(c_1[l_1], ..., c_n[l_n])\} \to \{\mathit{split}(r(\mathit{plug}(c'_1[r_1]), ..., \mathit{plug}(c'_n[r_{n'}])))\} \end{cases}$

Context reduction rules	Corresponding rewriting logic rules
$\langle c, \sigma\rangle[x] \to \langle c, \sigma\rangle[\sigma[x]]$ or	$\{\langle c, \sigma\rangle[x]\} \to \{\mathit{split}(\mathit{plug}(\langle c, \sigma\rangle[\sigma[x]]))\}$ or
$\langle c[x], \sigma\rangle \to \langle c[\sigma[x]], \sigma\rangle$	$\begin{cases} \mathit{split}(\langle p, \sigma\rangle) \to \langle \mathit{split}(p), \sigma\rangle \\ \{\langle c[x], \sigma\rangle\} \to \{\mathit{split}(\langle \mathit{plug}(c[\sigma[x]]), \sigma\rangle)\} \end{cases}$
$i_1 + i_2 \to i_1 +_{\mathit{Int}} i_2$	$\{c[i_1 + i_2]\} \to \{\mathit{split}(\mathit{plug}(c[i_1 +_{\mathit{Int}} i_2]))\}$
$i_1 \le i_2 \to i_1 \le_{\mathit{Int}} i_2$	$\{c[i_1 \le i_2]\} \to \{\mathit{split}(\mathit{plug}(c[i_1 \le_{\mathit{Int}} i_2]))\}$
$\mathsf{not}\ b \to \mathsf{not}_{\mathit{Bool}}\ b$	$\{c[\mathsf{not}\ b]\} \to \{\mathit{split}(\mathit{plug}(c[\mathsf{not}_{\mathit{Bool}}\ b]))\}$
$\mathsf{true\ and}\ b \to b$	$\{c[\mathsf{true\ and}\ b]\} \to \{\mathit{split}(\mathit{plug}(c[b]))\}$
$\mathsf{false\ and}\ b \to \mathsf{false}$	$\{c[\mathsf{false\ and}\ b]\} \to \{\mathit{split}(\mathit{plug}(c[\mathsf{false}]))\}$
$\langle c, \sigma\rangle[x := v] \to \langle c, \sigma[v/x]\rangle[\mathsf{skip}]$ or	$\{\langle c, \sigma\rangle[x := v]\} \to \{\mathit{split}(\mathit{plug}(\langle c, \sigma[v/x]\rangle[\mathsf{skip}]))\}$ or
$\langle c[x := v], \sigma\rangle \to \langle c[\mathsf{skip}], \sigma[v/x]\rangle$	$\begin{cases} \mathit{split}(\langle p, \sigma\rangle) \to \langle \mathit{split}(p), \sigma\rangle \quad \text{(may be redundant)} \\ \{\langle c[x := v], \sigma\rangle\} \to \{\mathit{split}(\langle \mathit{plug}(c[\mathsf{skip}]), \sigma[v/x]\rangle)\} \end{cases}$
$\mathsf{skip}; s \to s$	$\{c[\mathsf{skip}; s]\} \to \{\mathit{split}(\mathit{plug}(c[s]))\}$
$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \to s_1$	$\{c[\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2]\} \to \{\mathit{split}(\mathit{plug}(c[s_1]))\}$
$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \to s_2$	$\{c[\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2]\} \to \{\mathit{split}(\mathit{plug}(c[s_2]))\}$
$\mathsf{while}\ b\ \mathsf{do}\ s \to \mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else\ skip}$	$\{c[\mathsf{while}\ b\ \mathsf{do}\ s]\} \to \{\mathit{split}(\mathit{plug}(c[\mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else\ skip}]))\}$
$\langle c, \sigma\rangle[\mathsf{halt}\ i] \to \langle i, \sigma\rangle$ or	$\{\langle c, \sigma\rangle[\mathsf{halt}\ i]\} \to \{\mathit{split}(\langle i, \sigma\rangle)\}$ or
$\langle c[\mathsf{halt}\ i], \sigma\rangle \to \langle i, \sigma\rangle$	$\begin{cases} \mathit{split}(\langle p, \sigma\rangle) \to \langle \mathit{split}(p), \sigma\rangle \quad \text{(may be redundant)} \\ \{\langle c[\mathsf{halt}\ i], \sigma\rangle\} \to \{\mathit{split}(\langle i, \sigma\rangle)\} \end{cases}$
$\mathsf{skip}; a \to a$	$\{c[\mathsf{skip}; a]\} \to \{\mathit{split}(\mathit{plug}(c[a]))\}$

Figure 19: Second embedding of context reduction into rewriting logic ($\Gamma \leadsto \mathcal{R}^2_\Gamma$) and example

interested in listing all terms that are reachable from a given term t using the original context reduction definition, then all one needs to do is to invoke a search procedure on the term $\{\mathit{split}(t)\}$ and limit its responses to those matching the pattern "$\{\square[\mathit{Syntax}]\}$". Moreover, if one is only interested in the values that a program t can reduce to, then, assuming that one defined a syntactic category *Value*, all one needs to do is to filter the results to those matching the pattern "$\{\square[\mathit{Value}]\}$". Even more interestingly, if one is interested in all those reachable configurations in which $x = 0$ and $y = z$, then, assuming that *State* is defined as a comma-separated set of pairs "*variable* $\mapsto$ *value*", all one needs to do is to limit the results of the search to those terms matching the pattern "$\{\square[\langle \mathit{Syntax}, (x \mapsto 0, y \mapsto v, z \mapsto v, \mathit{State})\rangle]\}$". However, unlike for the first embedding, with this second embedding one cannot blindly let the rewrite engine execute context reduction definitions. That is because the rewrite engine may choose to stop the rewriting of the term $\{\mathit{split}(t)\}$ after each application of a rule of the form rule_i, by choosing to apply the rule $\mathit{split}(N) \to \square[N]$ (see Figure 18).

What makes our first embedding ($\Gamma \rightsquigarrow \mathcal{R}^1_\Gamma$) appropriate for executing Γ by letting a rewrite engine to blindly execute $\mathcal{R}^1_\Gamma$ is that its computational correspondence property (3 in Theorem 10) lets t' in the same rewriting context as t, so the rewriting process can continue, and that all the "erroneous" context representation attempts (i.e., ones that cannot be applied a reduction rule) are eventually hidden away by failing conditions. To achieve a similar effect, all we need to do with our second embedding in Figure 19 is to separate the propagation of splitting after each rule application from the actual rules.

Figure 20 shows our third and best direct embedding of context reduction into rewriting logic. An additional wrapper of syntax is introduced, "$\bullet$", which is responsible for executing precisely one reduction step in the original context reduction definition, and then consuming itself. To execute multiple reduction steps sequentially, we need to transitively close this one-step relation. This can be easily achieved with the conditional rule in Figure 20. One should not get tricked and drop the top wrapper, because then there is nothing to stop the applications of rewrite rules at any places in the term to rewrite, potentially including places which are not allowed to be evaluated yet, such as, for example, in the branches of a conditional. Moreover, such applications of rules could happen concurrently, which is strictly disallowed by context reduction. The role of the two wrappers, $\bullet$ and $\{\_\}$, is precisely to inhibit the otherwise unrestricted potential to apply rewrite rules everywhere and concurrently: rules are now applied sequentially and only at the top of the original term, exactly like in context reduction.

Theorem 12. ***(Third faithful embedding of context reduction into rewriting logic)*** *Let Γ be any context reduction definition and let $\mathcal{R}^3_\Gamma$ be the rewrite logic theory associated to Γ using the embedding procedure in Figure 20. Then*

1. ***(matching becomes deduction)*** *A term t matches rule_i in Γ whose lhs is $l(c_1[l_1], ..., c_n[l_n])$ iff $\mathcal{R}^3_\Gamma \vdash \mathit{split}(t) \overset{\mathit{split}}{\to^\star} l(c_1[l_1], ..., c_n[l_n])$, where $\overset{\mathit{split}}{\to^\star}$ is the rewrite relation restricted to split rules;*

2. ***(step-for-step correspondence)*** *$\Gamma \vdash t \to t'$ using rule_i iff $\mathcal{R}^3_\Gamma \vdash \mathit{split}(t) \overset{\mathit{split}}{\to^\star} u$ and $\mathcal{R}^3_\Gamma \vdash \bullet(u) \to^1 t'$ using rule_i; moreover, rule_i applies similarly (same contexts and same substitution);*

3. ***(computational correspondence)*** *$\Gamma \vdash t \twoheadrightarrow t'$ iff $\mathcal{R}^3_\Gamma \vdash \{t\} \to \{t'\}$.*

The faithful embeddings of context reduction into rewriting logic above can be used at least two different ways. On the one hand, they can be used as compilation steps transforming a context-sensitive reduction system into an equivalent context-insensitive term rewrite system, which can further be interpreted or compiled using conventional rewrite techniques and existing rewrite engines. In particular, as suggested by the numbers in Section 5.4.7 comparing Maude running the resulting rewrite theory against PLT-Redex running a direct brute-force implementation of the original context reduction definition, it may serve as a means towards getting a more efficient implementation of context reduction executable engines. On the other hand, the embeddings above are so simple, that one can simply use them manually and thus "think context reduction" in rewriting logic.

Context reduction rule	$\rightsquigarrow$	Two rewriting logic unconditional rules
$rule_i$: $l(c_1[l_1],...,c_n[l_n]) \to r(c'_1[r_1],...,c'_{n'}[r_{n'}])$	$\rightsquigarrow$	$\begin{cases} \text{if } l \text{ is proper (i.e., not a variable) then add the rule} \\ \quad split(l(N_1,...,N_n)) \to l(split(N_1),...,split(N_n)) \\ \\ \text{add the rule} \\ rule_i : \\ \quad \bullet(l(c_1[l_1],...,c_n[l_n])) \to r(plug(c'_1[r_1]),...,plug(c'_n[r_{n'}])) \end{cases}$
	where	

"*Syntax* ::= ... | $\bullet$(*Syntax*)" is an additional "wrapper", with the same intuition as the homonymous wrapper in the embedding of SOS in Section 5.1.1: $\bullet(t)$ reduces t one step (the intuition for "step" comes from Γ).

Transitive closure	$\rightsquigarrow$	Rewriting logic conditional rule
$\dfrac{t \to t'}{t \twoheadrightarrow t'}$ and $\dfrac{t \to u,\ u \twoheadrightarrow t''}{t \twoheadrightarrow t'}$	$\rightsquigarrow$	$\{t\} \to \{t'\}$ `if` $\bullet(split(t)) \to t'$

Context reduction rules	Corresponding rewriting logic rules
	$\{p\} \to \{p'\}$ `if` $\bullet(split(p)) \to p'$
$\langle c,\sigma\rangle[x] \to \langle c,\sigma\rangle[\sigma[x]]$ or	$\bullet(\langle c,\sigma\rangle[x]) \to plug(\langle c,\sigma\rangle[\sigma[x]])$ or
$\langle c[x],\sigma\rangle \to \langle c[\sigma[x]],\sigma\rangle$	$\left\{\begin{array}{l} split(\langle p,\sigma\rangle) \to \langle split(p),\sigma\rangle \\ \bullet(\langle c[x],\sigma\rangle) \to \langle plug(c[\sigma[x]]),\sigma\rangle \end{array}\right.$
$i_1 + i_2 \to i_1 +_{Int} i_2$	$\bullet(c[i_1 + i_2]) \to plug(c[i_1 +_{Int} i_2]))$
$i_1 \leq i_2 \to i_1 \leq_{Int} i_2$	$\bullet(c[i_1 \leq i_2]) \to plug(c[i_1 \leq_{Int} i_2])$
$\mathsf{not}\ b \to \mathsf{not}_{Bool}\ b$	$\bullet(c[\mathsf{not}\ b]) \to plug(c[\mathsf{not}_{Bool}\ b])$
$\mathsf{true\ and}\ b \to b$	$\bullet(c[\mathsf{true\ and}\ b]) \to plug(c[b])$
$\mathsf{false\ and}\ b \to \mathsf{false}$	$\bullet(c[\mathsf{false\ and}\ b]) \to plug(c[\mathsf{false}])$
$\langle c,\sigma\rangle[x := v] \to \langle c,\sigma[v/x]\rangle[\mathsf{skip}]$ or	$\bullet(\langle c,\sigma\rangle[x := v]) \to plug(\langle c,\sigma[v/x]\rangle[\mathsf{skip}])$ or
$\langle c[x := v],\sigma\rangle \to \langle c[\mathsf{skip}],\sigma[v/x]\rangle$	$\left\{\begin{array}{l} split(\langle p,\sigma\rangle) \to \langle split(p),\sigma\rangle \quad \text{(may be redundant)} \\ \bullet(\langle c[x := v],\sigma\rangle) \to \langle plug(c[\mathsf{skip}]),\sigma[v/x]\rangle \end{array}\right.$
$\mathsf{skip}; s \to s$	$\bullet(c[\mathsf{skip}; s]) \to plug(c[s])$
$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \to s_1$	$\bullet(c[\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2]) \to plug(c[s_1])$
$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \to s_2$	$\bullet(c[\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2]) \to plug(c[s_2])$
$\mathsf{while}\, b\, \mathsf{do}\, s \to \mathsf{if}\, b\ \mathsf{then}\ (s; \mathsf{while}\, b\, \mathsf{do}\, s)\ \mathsf{else\ skip}$	$\bullet(c[\mathsf{while}\, b\, \mathsf{do}\, s]) \to plug(c[\mathsf{if}\, b\ \mathsf{then}\ (s; \mathsf{while}\, b\, \mathsf{do}\, s)\ \mathsf{else\ skip}])$
$\langle c,\sigma\rangle[\mathsf{halt}\ i] \to \langle i,\sigma\rangle$ or	$\bullet(\langle c,\sigma\rangle[\mathsf{halt}\ i]) \to \langle i,\sigma\rangle$ or
$\langle c[\mathsf{halt}\ i],\sigma\rangle \to \langle i,\sigma\rangle$	$\left\{\begin{array}{l} split(\langle p,\sigma\rangle) \to \langle split(p),\sigma\rangle \quad \text{(may be redundant)} \\ \bullet(\langle c[\mathsf{halt}\ i],\sigma\rangle) \to \langle i,\sigma\rangle \end{array}\right.$
$\mathsf{skip}; a \to a$	$\bullet(c[\mathsf{skip}; a]) \to plug(c[a])$

Figure 20: Third embedding of context reduction into rewriting logic ($\Gamma \rightsquigarrow \mathcal{R}^3_\Gamma$) and example

5.4.6 Context Reduction as a Methodological Fragment of K

In this section we discuss an automatic embedding procedure of any context reduction definition Γ into a K definition $\mathcal{K}_\Gamma$. This embedding is also faithful to the computational granularity of the original context reduction definition, in the sense that any computational step in Γ has a precise one-step correspondent in $\mathcal{K}_\Gamma$ and vice-versa. Additionally, $\mathcal{K}_\Gamma$ is as compact as Γ: it adds one heating/cooling rule per context evaluation production and one rewrite rule per context reduction rule. Therefore, if one wants to use context reduction within K then one is free to do so, but, of course, one cannot circumvent the limitations of context reduction. As shown by the languages defined in the appendixes of this paper in K, such as imperative, functional, object-oriented and logic programming languages, one may be better off using the full strength of K, unlimited by restricted methodological fragments of K, such as context reduction.

Figure 21 shows our faithful embedding procedure together with an example applying it on our running simple imperative language. The result below shows the faithfulness of this embedding:

Theorem 13. ***(Faithful embedding of context reduction into K)*** *Let Γ be any context reduction definition and let $\mathcal{K}_\Gamma$ be the K theory associated to Γ using the embedding procedure in Figure 21. Then*

1. ***(embedding splitting/plugging into K deduction)*** *A term t can be split or "parsed" as $c[r]$ in Γ iff $\mathcal{K}_\Gamma \vdash t \rightleftharpoons^\star \overline{c[r]}$;*

2. ***(step-for-step correspondence)*** *$\Gamma \vdash t \to t'$ using $rule_i$ iff $\mathcal{K}_\Gamma \vdash t \rightleftharpoons^\star u$, $\mathcal{K}_\Gamma \vdash t' \rightleftharpoons^\star u'$, and $\mathcal{K}_\Gamma \vdash (\!|u|\!) \to^1 (\!|u'|\!)$ using $rule_i$; moreover, $rule_i$ applies similarly (same contexts and same substitution, all modulo the correspondence in 1.);*

3. ***(computational correspondence)*** *$\Gamma \vdash t \twoheadrightarrow t'$ iff $\mathcal{K}_\Gamma \vdash (\!|t|\!) \to (\!|t'|\!)$.*

5.4.7 Experiments

5.5 Abstract State Machines and the SECD Machine

5.6 The Chemical Abstract Machine

Berry and Boudol's *chemical abstract machine*, or *CHAM* [**?**], is both a model of concurrency and a specific style of giving operational semantics definitions. Berry and Boudol identify a number of limitations inherent in SOS, particularly its lack of true concurrency, and what they called SOS's "rigidity to syntax" [**?**]. They then present the CHAM as an *alternative* to SOS. In fact, as pointed out in [**?**], what the CHAM is, is a particular definitional style *within* RLS. That is, every CHAM *is*, by definition, a specific kind of rewrite theory; and CHAM computation is precisely concurrent rewriting computation; that is, proof in rewriting logic.

The basic metaphor giving its name to the Cham is inspired by Banâtre and Le Mètayer's GAMMA language [3]. It views a distributed state as a "solution" in which many "molecules" float, and understands concurrent transitions as "reactions". It is possible to define a variety of chemical abstract machines. Each of them corresponds to a rewrite theory satisfying certain common conditions.

There is an interesting analogy between CHAM and K. If one regards computations as "chemical solutions", then what we call "computation structural equation" above can be thought of as a pair of rules "heating/cooling" in CHAM. Recall that the role of "heating" in CHAM is to rearrange the solution so that "reactions" can take place. Once reactions take place, "cooling" rules insert the result of the reaction back into the solution. Therefore, reactions in CHAM take place "modulo" heating and cooling. Of course, multiple reactions can take place at the same time, which is what makes CHAM an elegant model for true concurrency. Similarly, rewrites in K definitions take place "modulo" computation structural equations and concurrently.

We find the conceptual analogy between K and CHAM so insightful, that we take the liberty to occasionally replace the equality symbol "=" in our computation structural equations with the symbol "$\rightleftharpoons$", which is used in CHAM for heating/cooling.

Context reduction syntax	$\leadsto$	K syntax
Syntax nonterminals N_1, N_2, ...	$\leadsto$	One top non-terminal (or sort), K
Context nonterminals Cxt_1, Cxt_2, ...	$\leadsto$	Nothing needed
Implicit notation "*context*[*term*]" and implicit "split" and "plug"	$\leadsto$	An associative binary operator $\_ \curvearrowright \_$ on K, i.e., $K ::= ... \mid \mathsf{List}^{\cdot}_{\curvearrowright}[K]$

Evaluation context production	$\leadsto$	K heating/cooling rule
$Cxt' ::= \pi(N_1, ..., N_n, Cxt)$	$\leadsto$	$\pi(N_1, ..., N_n, K) \rightleftharpoons K \curvearrowright \pi(N_1, ..., N_n, \square)$

Recall from Section 3 that $\pi(N_1, ..., N_n, \square)$ is syntactic sugar for a (new) operation called $\pi(\_, ..., \_, \square)$ (underscores "$\_$" serve as argument placeholders) applied to N_1, ..., N_n.

Context reduction rule	$\leadsto$	K rule
$l \rightarrow r$	$\leadsto$	$\llparenthesis \overline{l} \rrparenthesis \rightarrow \llparenthesis \overline{r} \rrparenthesis$

$$\begin{array}{rcl}
 & \text{where} & \\
K & ::= & ... \mid \llparenthesis K \rrparenthesis \quad \text{is a "top-level" wrapper for } K \\
 & \text{and} & \\
\overline{\gamma[t]} & = & \overline{t} \curvearrowright \overline{\gamma} \\
\overline{\pi(t_1, ..., t_n, \gamma)} & = & \overline{\gamma} \curvearrowright \pi(\overline{t_1}, ..., \overline{t_n}, \square) \\
\overline{\pi(t_1, ..., t_n)} & = & \pi(\overline{t_1}, ..., \overline{t_n}) \\
\overline{z} & = & z
\end{array}$$

Here t, t_1, ..., t_n range over non-contextual terms (i.e., ones which are not evaluation contexts, but can potentially contain instantiated contexts), γ ranges over evaluation contexts, and z is any variable, contextual or not. In other words, the evaluation contexts are flattened into K computation structures by iteratively "extracting subcontexts and placing them in front", until the redex becomes the first computational task.

Context reduction definition	Corresponding K definition
$\langle c, \sigma \rangle[x] \rightarrow \langle c, \sigma \rangle[\sigma[x]]$ or	$\llparenthesis x \curvearrowright c \curvearrowright \langle \square, \sigma \rangle \rrparenthesis \rightarrow \llparenthesis \sigma[x] \curvearrowright c \curvearrowright \langle \square, \sigma \rangle \rrparenthesis$ or
$\langle c[x], \sigma \rangle \rightarrow \langle c[\sigma[x]], \sigma \rangle$	$\llparenthesis \langle x \curvearrowright c, \sigma \rangle \rrparenthesis \rightarrow \llparenthesis \langle \sigma[x] \curvearrowright c, \sigma \rangle \rrparenthesis$
$i_1 + i_2 \rightarrow i_1 +_{Int} i_2$	$\llparenthesis i_1 + i_2 \rrparenthesis \rightarrow \llparenthesis i_1 +_{Int} i_2 \rrparenthesis$
$i_1 \leq i_2 \rightarrow i_1 \leq_{Int} i_2$	$\llparenthesis i_1 \leq i_2 \rrparenthesis \rightarrow \llparenthesis i_1 \leq_{Int} i_2 \rrparenthesis$
$\mathsf{not}\ b \rightarrow \mathsf{not}_{Bool}\ b$	$\llparenthesis \mathsf{not}\ b \rrparenthesis \rightarrow \llparenthesis \mathsf{not}_{Bool}\ b \rrparenthesis$
$\mathsf{true\ and}\ b \rightarrow b$	$\llparenthesis \mathsf{true\ and}\ b \rrparenthesis \rightarrow \llparenthesis b \rrparenthesis$
$\mathsf{false\ and}\ b \rightarrow \mathsf{false}$	$\llparenthesis \mathsf{false\ and}\ b \rrparenthesis \rightarrow \llparenthesis \mathsf{false} \rrparenthesis$
$\langle c, \sigma \rangle[x := v] \rightarrow \langle c, \sigma[v/x] \rangle[\mathsf{skip}]$ or	$\llparenthesis x := v \curvearrowright c \curvearrowright \langle \square, \sigma \rangle \rrparenthesis \rightarrow \llparenthesis \mathsf{skip} \curvearrowright c \curvearrowright \langle \square, \sigma[v/x] \rangle \rrparenthesis$ or
$\langle c[x := v], \sigma \rangle \rightarrow \langle c[\mathsf{skip}], \sigma[v/x] \rangle$	$\llparenthesis \langle x := v \curvearrowright c, \sigma \rangle \rrparenthesis \rightarrow \llparenthesis \langle \mathsf{skip} \curvearrowright c, \sigma[v/x] \rangle \rrparenthesis$
$\mathsf{skip}; s \rightarrow s$	$\llparenthesis \mathsf{skip}; s \rrparenthesis \rightarrow \llparenthesis s \rrparenthesis$
$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_1$	$\llparenthesis \mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rrparenthesis \rightarrow \llparenthesis s_1 \rrparenthesis$
$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_2$	$\llparenthesis \mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rrparenthesis \rightarrow \llparenthesis s_2 \rrparenthesis$
$\mathsf{while}\, b\, \mathsf{do}\, s \rightarrow \mathsf{if}\, b\ \mathsf{then}\ (s; \mathsf{while}\, b\, \mathsf{do}\, s)\ \mathsf{else\ skip}$	$\llparenthesis \mathsf{while}\, b\, \mathsf{do}\, s \rrparenthesis \rightarrow \llparenthesis \mathsf{if}\, b\ \mathsf{then}\ (s; \mathsf{while}\, b\, \mathsf{do}\, s)\ \mathsf{else\ skip} \rrparenthesis$
$\langle c, \sigma \rangle[\mathsf{halt}\ i] \rightarrow \langle i, \sigma \rangle$ or	$\llparenthesis \mathsf{halt}\ i \curvearrowright c \curvearrowright \langle \square, \sigma \rangle \rrparenthesis \rightarrow \llparenthesis \langle i, \sigma \rangle \rrparenthesis$ or
$\langle c[\mathsf{halt}\ i], \sigma \rangle \rightarrow \langle i, \sigma \rangle$	$\llparenthesis \langle \mathsf{halt}\ i \curvearrowright c, \sigma \rangle \rrparenthesis \rightarrow \llparenthesis \langle i, \sigma \rangle \rrparenthesis$
$\mathsf{skip}; a \rightarrow a$	$\llparenthesis \mathsf{skip}; a \rrparenthesis \rightarrow \llparenthesis a \rrparenthesis$

Figure 21: Embedding of context reduction into K ($\Gamma \leadsto \mathcal{K}_\Gamma$) and example

However, CHAM is not only a special case of rewriting logic, but also a special case of K. What makes this possible is the fact that both CHAM and K make use of unconditional rules. The major distinction between CHAM and K, in addition to dropping all the chemical intuitions, is that rules in K can also match and apply across "molecules", grabbing from each molecule what is needed without any need for an "airlock" operation. Here is how a CHAM can be transformed mechanically in a K definition, say K_{CHAM} (for simplicity, we here assume only one type of molecule, say M):

- Add automatically a "solution" syntactic category as a comma-separated multiset, say S, as well as the CHAM special wrapper of solutions into molecules, $\{\!|\_|\!\}$:

  $M ::= \ldots \mid \{\!|S|\!\}$ ("..." stay for "same as in the CHAM")
  $S ::= \mathsf{Set}_{,}[M]$ solutions of molecules of type M

- Add an "airlock" operation and equation:

  $S ::= \ldots \mid M \triangleleft \{\!|S|\!\}$ (*airlock* added as solution construct)

  $\{\!|m, s|\!\} = \{\!|m \triangleleft \{\!|s|\!\}\,|\!\}$ $(m \in M,\ s \in S_M)$

- For each CHAM rule, i.e., a rule of the form

  $$m_1, m_2, \ldots, m_k \rightarrow m'_1, m'_2, \ldots, m'_l$$

  add a K rule of the form

  $$\{\!|m_1, m_2, \ldots, m_k|\!\} \rightarrow \{\!|m'_1, m'_2, \ldots, m'_l|\!\}$$

- For each heating/cooling rule "$t \rightleftharpoons t'$" in CHAM add either one equation $t = t'$ or two rules $t \rightarrow t'$ and $t' \rightarrow t$; separate heating or cooling CHAM rules are just the same rules in K.

It is easy now easy to see that $s \rightarrow^* s'$ in CHAM iff $K_{\text{CHAM}} \vdash s \rightarrow^* s'$.

University of Illinois at Urbana-Champaign
Department of Computer Science UIUCDCS-R-2007-2926 and College of Engineering UILU-ENG-2007-1827

6 The K-CHALLENGE Language

We next propose a language design scenario in which a hypothetical language designer starts with the simple imperative language in Figure 4 and extends it with various non-trivial language features. The purpose of this section is not to propose any novel interesting programming language (though one could write quite interesting and tricky K-CHALLENGE programs). The goal of this section is twofold:

1. To challenge the existing language definitional frameworks with a non-trivial language design task, at the same time revealing some of their inherent limitations, and
2. To show how K avoids those limitations and how it can support the proposed design scenario, requiring the designer to do minimal changes on the existing design when adding each new feature.

A major goal of an ideal language definitional framework is, of course, to support arbitrarily complex language designs with minimal burden on the user. We argue that, at least for the proposed non-trivial language design scenario, K indeed requires minimal changes on existing definitions when adding new features. We also discuss how other definitional frameworks fail to have this property. To make this language design scenario as realistic as possible, at each moment we pretend that the newly added feature is the last one to be added to the language. In other words, a feature that can be potentially added in the future cannot be used to justify a particular definitional choice at the current moment. For example, if one knew upfront that one wanted to eventually add references with explicit variable address extraction to one's language, then one may choose upfront to split the state into an environment and a store. However, we want to point out that if such a "radical" structural change requires one to revisit all or most of the existing definitions, then the underlying framework is far from ideal.

This section may indirectly also suggest that modularity of language definitions can perhaps be achieved only within a particular and well-defined universe. For example, one may devise a modular definitional methodology (e.g., a purely "syntactic" substitution-based one), where say each additional language feature is added without touching any of definitions of the previous features, in a purely functional universe known *a priori* not to allow for side effects or concurrency. If complex side effects are allowed in the universe then one may need to develop a different definitional methodology, which may also change when one adds concurrency. Moreover, the addition of new features may make previous features "obsolete" of even conflicting. For example, adding references with explicit variable address access may make the language designer prefer an assignment construct that takes a location and a value instead of a variable and a value as before; the two cannot be both kept in the language because of conflicting semantics (if x is a variable holding a location l then one may choose $x := l'$ either to write l' in x, or to write value l' at location l). Finally, any definitional methodology developed for imperative, functional or object-oriented languages may need to be radically changed or even dropped all together when one defines a logic programming language.

Variant 1 — increment. Let us add an increment construct, ++*Name*, taking a variable, incrementing its value, and then returning the new, incremented value. In K one can do it easily as follows:

$$AExp ::= \ldots \mid \texttt{++}\,Name$$
$$\llparenthesis \underset{i}{\underline{\texttt{++}\,x}} \rrparenthesis_k \ \llparenthesis \underset{\sigma[i/x]}{\underline{\ \ \sigma\ \ }} \rrparenthesis_{state} \qquad \text{where } i = \sigma[x] + 1$$

No change is required on the existing definition in Figure 4; all what needs to do is to add the syntax and the semantics of the new language construct above. Since expressions now have side effects, a big step definition of the previous variant of the language would need to be radically changed. Indeed, if expressions have no side effects, then a big-step definition can associate configurations $\langle e, \sigma\rangle$ to values v using, for example, sequents of the form $\langle e, \sigma\rangle \Downarrow v$ with the meaning "expression e evaluates in state σ to value v". Once expressions have side effects, one needs to change all the existing rules to use sequents of the form $\langle e, \sigma\rangle \Downarrow \langle v, \sigma'\rangle$ (σ' is the state obtained after evaluating e in state σ to value v).

The other definitional styles discussed in this paper, namely (small-step) SOS, MSOS and context reduction can define increment as easily and modularly as K.

Variant 2 — merging expressions. One annoying aspect of the current language is that one can only write/read integer values in the state, meaning also that one can only assign integer type expressions to variables. We would naturally like to eliminate this limitation and add arbitrary expressions to the language, and allow them to be assigned to variables. That means, in particular that we would like to merge the different categories of expressions, currently *AExp* and *BExp*, into only one syntactic category, say *Exp*; a type checker can be also easily defined in K to ensure that expressions are used properly, but we do not do it here. Also, we take the opportunity to add one more type of expressions, namely floats, which should, of course, enjoy the same first-class citizen rights of the other expressions. One more change to the language design is also in place here. In the original design, a program was chosen to be a statement followed by an expression. Inspired by languages such as BC, suppose that we prefer at this stage to extend the expressions with a construct "$Exp ::= ... \mid Stmt; Exp$" and to eliminate programs all together, because we can replace them with expressions. There are quite some changes above; however, they can all be done very easily in K:

- Replace *AExp*, *BExp* and *Pgm* by *Exp* everywhere. This can be done either by editing the existing definitions mechanically (a tedious, but automatic process), or better, by using a conventional *rename* module composition operator if the underlying implementation of K offers support for module composition. For example, with our Maude implementation of K, one can simply import the module "$\mathsf{Variant1} * (\mathsf{sort}\ AExp\ \mathsf{to}\ Exp,\ \mathsf{sort}\ BExp\ \mathsf{to}\ Exp,\ \mathsf{sort}\ Pgm\ \mathsf{to}\ Exp)$". For example, the syntax of $\leq$ becomes "$Exp ::= ... \mid Exp \leq Exp$" and that of halt becomes "$Stmt ::= ... \mid \mathsf{halt} Exp$". All these are purely syntactic changes, with no influence on the K semantics. In fact, when using our current implementation of K in Maude (see Section E), none of the existing equations or rules needs to change.
- Add boolean and float numbers to values, that is, add "$Val ::= ... \mid Bool \mid Float$", together with additional attributes (grayed below) for language constructs intended to also work with floats, namely $\_+\_$ and $\_\leq\_$:

$$\begin{array}{ll} Exp + Exp & [strict,\ extends\ \_+_{Int}\_,\ \ extends\ \_+_{Float}\_\,] \\ Exp \leq Exp & [seqstrict,\ extends\ \_\leq_{Int}\_,\ \ extends\ \_\leq_{Float}\_\,] \end{array}$$

The changes mentioned above are simple and necessary: the first merges the three syntactic categories into one, the second extends the strict addition to floats, and the third extends the sequentially strict less-then to floats. Any definitional style or framework must, in one way or another, do at least the above; some may need more changes than necessary. Regarding the first change, note that renaming is a well-understood module composition operator in algebraic specification supported by most algebraic specification engines, so one needs no justification for it in K. In other formalisms, including big-step/small-step SOS (modular or not) and context reduction, one may need to resort on less elegant manual (though admittedly mechanical) changes of syntactic category names, as well as of corresponding side conditions of rules. Moreover, since so far the expressions evaluated only to integer values and since less-than is *sequentially strict*, the (small-step) SOS definition would most likely contain a rule

$$\frac{\langle a, \sigma\rangle \to \langle a', \sigma'\rangle}{\langle i \leq a, \sigma\rangle \to \langle i \leq a', \sigma'\rangle},\quad \text{where } i \in Int,\ a, a' \in AExp,\ \sigma, \sigma' \in State.$$

Unless one envisioned such possible language extensions upfront and defined the rule above for any values $v \in Val$ instead of for any integer $i \in Int$, similar rules need to be added for each extension, in particular

$$\frac{\langle a, \sigma\rangle \to \langle a', \sigma'\rangle}{\langle f \leq a, \sigma\rangle \to \langle f \leq a', \sigma'\rangle},\quad \text{where } f \in Float,\ a, a' \in AExp,\ \sigma, \sigma' \in State.$$

Similarly, if in a context reduction definition of this language the sequential strictness of "$\leq$" was defined using a production "$Cxt ::= ... \mid Int \leq Cxt$" like in Figure 1, then one needs to add one more production to evaluation contexts, namely "$Cxt ::= ... \mid Float \leq Cxt$".

Note that adding such artifact rules or declarations is not necessary in K, because the *seqstrict* attribute is defined in terms of computations (K) and result computations (*KResult*), and that values were already defined to be computation results (a necessary step when using the *seqstrict* attribute). In fact, our major reason for introducing the strictness attributes was precisely to eliminate the need to introduce or revisit such low-level and uninteresting rules. The user of K needs to think and define the intended evaluation strategy of each language construct once and for all (right after defining its syntax). Thinking and designing a language in terms of computations and computation transformations, as opposed to particular syntactic categories and syntactic transformations, brings in our view a level of abstraction that enhances the modularity of the definitional framework.

Variation 3 – output. Let us now add output to our language, that is, a statement "output *Exp*" taking an expression, evaluating it, and then outputting its value into a buffer (i.e., list) that collects all the output. In addition to the strict language construct "output *Exp*", the configuration also needs to incorporate an output buffer, say wrapped by configuration item label *output*. Since values can be now collected in the output buffer, there is no need for a program to evaluate or to halt to a value; in other words, one can pass to the "$[\![\_]\!]$" operation a statement computation and modify the configuration initialization and termination, as well as the syntax and semantics of halt, accordingly (halt takes no arguments now, so we remove its previous syntax/semantics and add the following instead):

$$\begin{array}{l} Stmt ::= \ldots \mid \mathsf{output\_} \quad [strict] \mid \mathsf{halt} \\ Config ::= \ldots \mid \mathsf{List}[\,Val] \mid (\!|\mathsf{List}[\,Val]|\!)_{output} \end{array}$$

$$\begin{array}{l} [\![s]\!] = (\!|(\!|s|\!)_k \ (\!|\cdot|\!)_{state} \ (\!|\cdot|\!)_{output}|\!)_\top \\ \langle\!|(\!|\cdot|\!)_k \ (\!|vl|\!)_{output}|\!\rangle_\top = vl \\ (\!|\mathsf{halt}|\!\rangle_k \to (\!|\cdot|\!)_k \end{array}$$

$$(\!|\underbrace{\mathsf{output}\ v}_{\cdot}|\!\rangle_k \ \langle\!|\underbrace{\cdot}_{v}|\!)_{output}$$

We claim that the K definition above is minimal. Indeed, the two declarations for halt (new syntax and rule) were necessary because we decided for a completely different halt statement. Also, each of the remaining six declarations above states a different and necessary part of the semantics of output: the first declares its syntax and evaluation strategy, the second declares the buffer in which the output values are collected, the third defines the new structure of the configuration that accommodates evaluations of statements and lists of values as results, the fourth and the fifth define the initialization and the termination of the computation using the new configuration structure, while the sixth gives the actual semantics of output.

Adding output to a language defined using conventional big-step or small-step SOS is devastating: one needs to change every single rule to accommodate the new configuration containing the output buffer in addition to the syntax and the state. Like K, MSOS elegantly avoids doing that; all what needs to do in MSOS is to add an output label on transitions that is only used in the semantics of output. Context reduction is more modular than SOS, but one's skill plays a more important role in achieving overall modularity. For example, the immediate way to add output "modularly" in a context reduction definition is to change the context production "$Cxt ::= \ldots \mid [\![Cxt, State]\!]$" into "$Cxt ::= \ldots \mid [\![Cxt, State, Output]\!]$". None of the other productions need to change and so do all the existing reduction rules that do not mention the configuration construct, e.g., "if true then s_1 else s_2", etc. Unfortunately, as an artifact of how matching works in context reduction, rules that use the configuration need to change to accommodate the new configuration. For example, the rule "$[\![c,\sigma]\!][x] \to [\![c,\sigma]\!][\sigma[x]]$" needs to change to "$[\![c,\sigma,o]\!][x] \to [\![c,\sigma,o]\!][\sigma[x]]$", even though the new addition to the configuration, the output, plays no role in the semantics of variable lookup. The "user's skill" we mentioned above to make context reduction also modular in this case, is to envision possible configuration changes and thus define a configuration as a list structure containing various configuration items, for example:

$$\begin{array}{rcl} ConfigItem & ::= & State \mid Output \mid \ldots \\ Cxt & ::= & [\![Cxt, ConfigItem^*]\!] \end{array}$$

Then one can write the reduction rules to match in the list of configuration items only those items of interest, for example ($\gamma, \gamma' \in \mathit{ConfigItem}^*$, $c \in \mathit{Cxt}$, $\sigma \in \mathit{State}$, $x \in \mathit{Name}$, $o \in \mathit{Output}$, $v \in \mathit{Val}$):

$$[\![c, (\gamma, \sigma, \gamma')]\!][x] \rightarrow [\![c, (\gamma, \sigma, \gamma')]\!][\sigma[x]]$$
$$[\![c, (\gamma, o, \gamma')]\!][\mathsf{output}\ v] \rightarrow [\![c, (\gamma, (o, v), \gamma')]\!][\mathsf{skip}]$$

Even though one still needs to mention irrelevant variables just for matching reasons, such as the lists of configuration items γ and γ', the context reduction definition becomes with this change much more modular than before. From a K perspective, a slight inconvenience in both MSOS and context reduction is that one needs to introduce (if not already in the language) the "value" statement skip and then, using an additional reduction step, discard it; in K one just dissolves a statement once finished, thus capturing the intended computation granularity of the statement construct. Introducing skip and changing the computation granularity of statements is, however, generally accepted by purely syntactic definitional approaches.

Variation 4(a) – λ with substitution. The language defined so far has no capabilities to group code in functions or procedures. We next enrich our language with λ-expressions; more precisely, we add λ-abstraction and λ-application as new expression constructs and then give them a substitution-based semantics. Any of the variants of λ-calculus discussed in Section 4.2 can be considered. For the sake of concreteness we, however, choose the call-by-value parameter passing style here:

$$\mathit{Val} ::= \ldots \mid \lambda \mathit{Name}.\mathit{Exp}$$
$$\mathit{Exp} ::= \ldots \mid \mathit{Exp}\,\mathit{Exp} \quad [\mathit{strict}]$$

$$(\!|\underline{(\lambda x.e)\ v}_{e[v/x]}\rangle_k \quad \text{where } x \in \mathit{Name},\ e \in \mathit{Exp},\ v \in \mathit{Val}.$$

One (rather standard) problem with definitions like the above in the context of λ-calculus with imperative features, is that one cannot assign the bound variable (x) in the body of the λ-abstraction (e), so one may need a static analysis to discard programs that do it. A (rather standard) alternative to this is to instead introduce a fresh variable, say y, bind it to the argument value (v) in the state, and then replace the bound variable x by y. We can do all these in one step in K, replacing the rule above with the following one:

$$(\!|\underline{(\lambda x.e)\ v}_{e[y/x]}\rangle_k \ (\!|\underline{\ \sigma\ }_{\sigma[v/y]}|\!)_{state} \quad \text{where } y \in \mathit{Name} \text{ is a fresh name.}$$

The reason we need a fresh name (y) and an α-conversion to replace the bound name (x) in the rule above instead of just binding x to v in the state, is because x may already be used outside the scope of the λ-abstraction and we obviously do not want to affect its value there.

Context reduction definitions take a similar approach to add the features above. When adopting a substitution-based style as above (which may not necessarily be always possible or the best, e.g., when adding references and concurrency to the language), a slight advantage of SOS approaches is that they do not need a fresh name: instead, they just initiate in the condition of the rule a reduction of e in the *modified* state $\sigma[v/x]$, and then continue the reduction in the conclusion of the rule using the original state, σ.

Now that binding is available, the language designer may want to disallow reads and writes of variables that were not explicitly bound. From here on, we therefore assume that the original expressions/program to evaluate is closed (this can be easily achieved automatically by wrapping the entire program in a series of λ abstractions binding all the originally free variables and then calling it on initial values for those variables).

Variation 4(b) – λ with closures

Suppose that, for several reasons, our language designer decides to switch to an environment-based definition of her language. As usual, we assume an infinite number of possible locations and that we can get fresh locations whenever needed. A major structural change in the configuration is required, namely the state is split into an environment and a store. This structural change inevitably affects the existing rules that relied on the state (i.e., explicitly matched it), but, fortunately, the other rules need not be touched. Below we only list those rules that changed, mentioning that we take a liberty here to use the *restore* computation item also used in Section 4.9 which is builtin in our implementation of K (see Appendix F):

$$\mathit{Config} ::= (\!|K|\!)_k \mid (\!|\mathit{Env}|\!)_{env} \mid (\!|\mathit{Store}|\!)_{store} \mid (\!|\mathsf{List}[\mathit{Val}]|\!)_{output} \mid (\!|\mathsf{Set}[\mathit{Config}]|\!)_\top$$
$$[\![e]\!] = (\!|(\!|e|\!)_k\ (\!|\cdot|\!)_{env}\ (\!|\cdot|\!)_{store}\ (\!|\cdot|\!)_{output}|\!)_\top$$

$$(\!|\underset{\sigma[\rho[x]]}{\underline{\ x\ }}\rangle_k\ (\!|\rho|\!)_{env}\ (\!|\sigma|\!)_{store}$$

$$(\!|\underset{\cdot}{\underline{x := v}}\rangle_k\ (\!|\rho|\!)_{env}\ (\!|\underset{\sigma[v/\rho[x]]}{\underline{\ \sigma\ }}|\!)_{store}$$

$$(\!|\underset{i}{\underline{\texttt{++}\,x}}\rangle_k\ (\!|\rho|\!)_{env}\ (\!|\underset{\sigma[i/\rho[x]]}{\underline{\ \sigma\ }}|\!)_{store} \quad \text{where } i \text{ is } \sigma[\rho[x]]+1$$

$$\mathit{Val} ::= \ldots \mid \mathit{closure}(\mathit{Name}, \mathit{Exp}, \mathit{Env})$$
$$\mathit{Exp} ::= \ldots \mid \lambda\mathit{Name}.\mathit{Exp} \mid \mathit{Exp}\,\mathit{Exp}\ [\mathit{strict}]$$

$$(\!|\underset{\mathit{closure}(x,e,\rho)}{\underline{\ \lambda x.e\ }}\rangle_k\ (\!|\rho|\!)_{env}$$

$$(\!|\underset{e \curvearrowright \mathit{restore}(\rho')}{\underline{\mathit{closure}(x,e,\rho)\ v}}\rangle_k\ (\!|\underset{\rho[l/x]}{\underline{\ \rho'\ }}|\!)_{env}\ (\!|\underset{\sigma[v/l]}{\underline{\ \sigma\ }}|\!)_{store} \quad \text{where } l \text{ is a fresh location}$$

Variant 5 – recursion. Let us next add an explicit construct for recursion, namely μ:

$$\mathit{Exp} ::= \ldots \mid \mu\mathit{Name}.\mathit{Exp}$$
$$(\!|\mu x.e\rangle_k = (\!|(\lambda x.e)\,(\mu x.e)\rangle_k$$

One could also give μ a direct semantics, not relying on λ, but one would have to add a closure value like we did for the semantics of λ. Note that we need not worry about affecting the intended computation granularity of μ with our definition above, because we used an equation, not a rule. Other formalisms may be forced to give μ a direct semantics if unaffected computation granularity is important to the designer.

Variant 6 – references. Let us next add references to the language, together with dereferencing and explicit address extraction for names. These suggest one major change in the design of the language: the assignment statement can be changed to assign values to locations, instead of to variables. To do so, one also needs to extend the values with locations and re-refine the assignment to be strict in both its arguments. Here are all the changes needed to the existing language in order to incorporate the above (one also needs to remove the annotated syntax for assignment and its rule from the previous definition):

$Val ::= \ldots \mid Loc$
$Exp ::= \ldots \mid \mathsf{ref}\, Exp\ [strict] \mid *\, Exp\ [strict] \mid \&\, Name$
$Stmt := \ldots \mid Exp := Exp\ [strict]$

$(\!| \underset{l}{\underline{\mathsf{ref}\, v}} |\!)_k\ (\!| \underset{\sigma[v/l]}{\underline{\quad \sigma \quad}} |\!)_{store}$ where l is a fresh location

$(\!| \underset{\sigma[l]}{\underline{\ *\, l\ }} |\!)_k\ (\!| \sigma |\!)_{store}$

$(\!| \underset{\rho[x]}{\underline{\&\, x}} |\!)_k\ (\!| \rho |\!)_{env}$

$(\!| \underset{\cdot}{\underline{l := v}} |\!)_k\ (\!| \underset{\sigma[v/l]}{\underline{\quad \sigma \quad}} |\!)_{store}$

Variant 7* – *call with current continuation. To add call/cc to the language, all one needs to do is to add the following without changing anything to the existing definitions:

$Exp ::= \ldots \mid \mathsf{callcc}\, Exp\ [strict]$
$Val ::= \ldots \mid cc(K, Env)$

$(\!| \underset{v\ cc(k,\rho)}{\underline{\ \mathsf{callcc}\, v\ }} \curvearrowright k |\!)_k\ (\!| \rho |\!)_{env}$

$(\!| \underset{v \curvearrowright k}{\underline{cc(k,\rho)\ v \curvearrowright \_}} |\!)_k\ (\!| \underset{\rho}{\underline{\_}} |\!)_{env}$

Variant 8* – *nondeterminism. Recall that one of our major goals was to design a language definitional framework that is executable. Nondeterminism is trivial to add to all non-functional/non-denotational definitional approaches. Here is our definition of a trivial non-deterministic random boolean choice:

$Exp ::= \ldots \mid \mathsf{randomBool}$
$\mathsf{randomBool} \to \mathsf{true}$
$\mathsf{randomBool} \to \mathsf{false}$

While functional or denotational approaches also give support for nondeterminism, these approaches have a different goal than ours: to capture *all* possible behaviors of a nondeterministic program as a value. Needless to say that that operation is very expensive and, sometimes impossible: for example when a program has an infinite number of behaviors. Our goal is to get executable semantics, so that programs with infinite behaviors can still be executed, with the possibility to *also* obtain all possible behaviors if one is interested in that (using, e.g., a search command in our Maude implementation of K — Appendix E).

Variant 9* – *aspects. There are many approaches to aspects and aspect-oriented programming these days. We, of course, do not intend to cover all those here. We only want to show how easily and modularly one can add aspects to a language formally defined in K. To illustrate this point, we consider only one language construct, "$\mathsf{aspect}\ s$", taking a statement and executing it whenever a function is being called from from there on. The aspect statement is executed "as is", in the sense that aspects come with no static scoping for the variables that they may access or change. It is easy to modify our definition to assume statically scoped aspects, but that is not our purpose here. We allow the aspect to be dynamically changed during the execution of the program. To achieve that, we define one more configuration cell that keeps the aspect:

$$
\begin{aligned}
&\mathit{Stmt} ::= \ldots \mid \mathsf{aspect}\ \mathit{Stmt}\\
&\mathit{Config} ::= \ldots \mid \llparenthesis K \rrparenthesis_{aspect}
\end{aligned}
$$

$$[\![s]\!] = \llparenthesis \llparenthesis s \rrparenthesis_k\ \llparenthesis \cdot \rrparenthesis_{env}\ \llparenthesis \cdot \rrparenthesis_{store}\ \llparenthesis \cdot \rrparenthesis_{output}\ \llparenthesis \cdot \rrparenthesis_{aspect} \rrparenthesis_{\top}$$

$$\llparenthesis \underset{\cdot}{\underline{\mathsf{aspect}\ s}} \rrparenthesis_k\ \llparenthesis \underset{s}{\underline{\_}} \rrparenthesis_{aspect}$$

There are several semantic choices possible, aspect-oriented frameworks opting for one or more of them. For example, one possibility is to grab the aspect available at function declaration time, another is grab it at function application time. Also, for any of these two possibilities, the aspect can be executed in caller's context or in callee's context. Any of these can be easily defined in K. For example, here is a definition in which the aspect is grabbed at function declaration time and later executed in callee's context:

$$\llparenthesis \underset{\mathit{closure}(x, (s \curvearrowright e), \rho)}{\underline{\lambda x.e}} \rrparenthesis_k\ \llparenthesis \rho \rrparenthesis_{env}\ \llparenthesis s \rrparenthesis_{aspect}$$

and here is a definition in which the aspect is executed in callee's context at function invocation time:

$$\llparenthesis \underset{s \curvearrowright e \curvearrowright \mathit{restore}(\rho')}{\underline{\mathit{closure}(x, e, \rho)\ v}} \rrparenthesis_k\ \llparenthesis \underset{\rho[l/x]}{\underline{\rho'}} \rrparenthesis_{env}\ \llparenthesis \underset{\sigma[v/l]}{\underline{\sigma}} \rrparenthesis_{store}\ \llparenthesis s \rrparenthesis_{aspect} \qquad \text{(where } l \text{ is a fresh location)}$$

Variant 10 – concurrency with lock synchronization. Let us next extend this language with dynamic threads that can run concurrently. Whether multi-threading with shared memory is how concurrency should be supported in languages is an interesting subject open to debate (where we have our personal and therefore subjective opinions, but here we refrain from commenting on this subject). Wearing the hat of the designer of a language design framework (and not that of a programming language designer), our position here is that a powerful language design framework should support all existing approaches to concurrency; at the time this paper was written (end of 2007) multi-threading with shared memory was still the most practical approach to concurrency.

We consider the following additional syntax:

$$\mathit{Stmt} ::= \ldots \mid \mathsf{spawn}\ \mathit{Stmt} \mid \mathsf{acquire}\ \mathit{Exp}\ [\mathit{strict}] \mid \mathsf{release}\ \mathit{Exp}\ [\mathit{strict}]$$

spawn s spawns a new thread executing statement s concurrently with the rest of the threads. The newly created thread inherits the environment of its parent at creation time, which is the means by which memory is shared, and dissolves itself when the statement s is completely processed. Threads synchronize through acquiring and releasing locks. Any values can be used as locks, including functions, and two locks are considered equal if and only if they are *identical* as value terms (not even α-equivalent). Locks can be acquired and released multiple times by the same thread and only one thread can hold a lock at any given time. A lock is effectively released by a thread only when the number of releases matches the number of acquires. If a lock is not available (because it is taken by another thread), then the thread attempting to acquire it waits until the lock is released.

We modify the configuration so that the information pertaining to each thread (i.e., computation, environment and lock/counter pairs) is held into one cell of type *thread*. An additional top-level cell is needed to hold all the busy locks. The semantics of thread termination is to dissolve its corresponding thread cell, releasing all its resources. Therefore, the execution of a program is terminated when there is no thread cell left. Here is the new configuration structure, its initialization and its termination:

$$
\begin{aligned}
&\mathit{Config} ::= \ldots \mid \llparenthesis \mathsf{Set}[\mathit{Val} \times \mathit{Nat}] \rrparenthesis_{holds} \mid \llparenthesis \mathsf{Set}[\mathit{Config}] \rrparenthesis_{thread} \mid \llparenthesis \mathsf{Set}[\mathit{Val}] \rrparenthesis_{busy}\\
&[\![s]\!] = \llparenthesis \llparenthesis \llparenthesis s \rrparenthesis_k\ \llparenthesis \cdot \rrparenthesis_{env}\ \llparenthesis \cdot \rrparenthesis_{holds} \rrparenthesis_{thread}\ \llparenthesis \cdot \rrparenthesis_{store}\ \llparenthesis \cdot \rrparenthesis_{output}\ \llparenthesis \cdot \rrparenthesis_{aspect}\ \llparenthesis \cdot \rrparenthesis_{busy} \rrparenthesis_{\top}\\
&\llparenthesis \llparenthesis \_ \rrparenthesis_{store}\ \llparenthesis vl \rrparenthesis_{output}\ \llparenthesis \_ \rrparenthesis_{aspect}\ \llparenthesis \_ \rrparenthesis_{busy} \rrparenthesis_{\top} = vl
\end{aligned}
$$

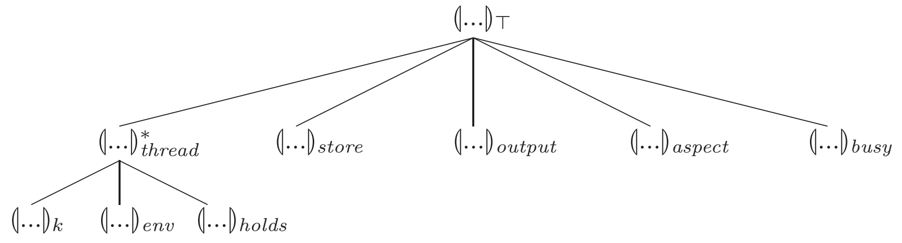
Before we attempt to give the semantics of the concurrency-related language constructs, some explanations are needed with regards to the change of the configuration structure. A major problem when changing the structure of the configuration is that some of the previously defined rules and equations may not match anymore against the new configuration. Indeed, consider for example the rule for variable lookup:

$$\llparenthesis \frac{x}{\sigma[\rho[x]]} \rrparenthesis_k \; \llparenthesis \rho \rrparenthesis_{env} \; \llparenthesis \sigma \rrparenthesis_{store}$$

Since in the new configuration the store is located on a level different from that of the computation and the environment, this rule will never match with the new configuration structure. To fix this problem, one would apparently have to revisit the existing definitions and modify them to account for the new configuration structure, for example:

$$\llangle \llparenthesis \frac{x}{\sigma[\rho[x]]} \rrparenthesis_k \; \llparenthesis \rho \rrparenthesis_{env} \rrangle_{thread} \; \llparenthesis \sigma \rrparenthesis_{store}$$

This is of course very inconvenient and violates one of our major requirements for an ideal language definitional framework: modularity. How can we derive the second rule above from the first? K provides a mechanism called *context transforming*, which is responsible for automatically completing "partially" defined terms. This is explained in detail in [33]. In short, cell wrappers can be defined as "structural" and possibly "repetitive" in K; this additional information can be then used to automatically and unambiguously[10] transform terms in general, and K rules and equations in particular, by completing them with potentially missing structural information. For example, all the cells in this language definition are defined as "structural" and $\llparenthesis ... \rrparenthesis_{thread}$ is also defined as "repetitive". When using the algebraic notation for K as in [33], the language designer can add this information rigorously, using operation attributes to the cell wrapper constructs. We have not yet devised any rigorous non-algebraic way to introduce this configuration structural information; for the remaining of this section we just draw a picture like the one below showing the configuration structure together with the repetitiveness information (a star "$*$" as a superscript of a cell means that the cell is allowed to repeat) and let the reader mentally replay the context transforming process:


$\llparenthesis ... \rrparenthesis_\top$
$\llparenthesis ... \rrparenthesis^*_{thread}$
$\llparenthesis ... \rrparenthesis_{store}$
$\llparenthesis ... \rrparenthesis_{output}$
$\llparenthesis ... \rrparenthesis_{aspect}$
$\llparenthesis ... \rrparenthesis_{busy}$
$\llparenthesis ... \rrparenthesis_{k}$
$\llparenthesis ... \rrparenthesis_{env}$
$\llparenthesis ... \rrparenthesis_{holds}$

The context transforming process is not only necessary in order to achieve the much desired modularity, but it also allows the designer to write semantic rules and equations more succinctly and conceptually: one only needs to focus on *what* is needed from the configuration in order to give the semantics of a construct, rather than *where* that information is located in the (continuously evolving) configuration structure. Thanks to context transforming, we need to make *no change* to the existing rules or equations when adding threads to our language. All we need to do is to add the definitions of the new constructs.

Here is the semantics of thread creation and termination:

$$\llparenthesis \frac{\mathsf{spawn}\ s}{\cdot} \rrparenthesis_k \; \llparenthesis \rho \rrparenthesis_{env} \; \frac{\cdot}{\llparenthesis \llparenthesis s \rrparenthesis_k \; \llparenthesis \rho \rrparenthesis_{env} \; \llparenthesis \cdot \rrparenthesis_{holds} \rrparenthesis_{thread}}$$

$$\frac{\llangle \llparenthesis \cdot \rrparenthesis_k \; \llparenthesis lc \rrparenthesis_{holds} \rrangle_{thread}}{\cdot} \; \llparenthesis \frac{ls}{ls - lc} \rrparenthesis_{busy}$$

Here is the semantics of lock acquire:

$$\llparenthesis \frac{\mathsf{acquire}\ v}{\cdot} \rrparenthesis_k \; \llangle (v, \frac{n}{s(n)}) \rrangle_{holds}$$

$$\llparenthesis \frac{\mathsf{acquire}\ v}{\cdot} \rrparenthesis_k \; \llangle \frac{\cdot}{(v,0)} \rrangle_{holds} \; \llparenthesis \frac{ls}{ls\ v} \rrparenthesis_{busy} \; \texttt{when}\ v \notin ls$$

[10]In case of ambiguity, the most "local" grouping is always chosen, following a depth-first traversal of the term.

Finally, here is the semantics of lock release:

$$(\!|\underset{\cdot}{\underline{\mathsf{release}\ v}}\,\rangle_k\ \langle\!|(v, \underset{n}{\underline{s(n)}})|\!\rangle_{holds}$$
$$(\!|\underset{\cdot}{\underline{\mathsf{release}\ v}}\,\rangle_k\ \langle\!|\underset{\cdot}{\underline{(v,0)}}|\!\rangle_{holds}\ \langle\!|\underset{\cdot}{\underline{v}}|\!\rangle_{busy}$$

An interesting question is what is the resulting semantics of halt in this multi-threaded extension of our language, that is, whether it halts only the current thread or the entire program. Since its previous semantics was to just dissolve the computation that generated it, the answer is that it only halts the current thread. This is quite an acceptable semantics. Supposing that one wants a semantics that stops the entire multi-threaded program, then one has to remove the current rule for halt and add the following instead:

$$\langle\!|(\!|\mathsf{halt}\rangle_k\ (\!|vl|\!)_{output}|\!\rangle_{\top} \rightarrow vl$$

Variant 11 – rendez-vous synchronization. Let us next define a rendez-vous synchronization construct, say rv v, taking also a "lock", or better say a "barrier" argument. The thread executing this command is blocked until another thread executes a similar rendez-vous command with the same value v. When that happens, the two threads discard their rv v statements and continue their executions concurrently. Here is the semantics of rendez-vous synchronization in K:

$$\mathit{Stmt} ::= \ldots \mid \mathsf{rv}\ \mathit{Exp}\ [\mathit{strict}]$$
$$(\!|\underset{\cdot}{\underline{\mathsf{rv}\ v}}\rangle_k\ (\!|\underset{\cdot}{\underline{\mathsf{rv}\ v}}\rangle_k$$

Note that, according to the structural and repetitiveness information about the configuration, the only way for the context transforming process to complete the rule above to match the configuration is as follows:

$$\langle\!|(\!|\underset{\cdot}{\underline{\mathsf{rv}\ v}}\rangle_k|\!\rangle_{thread}\ \langle\!|(\!|\underset{\cdot}{\underline{\mathsf{rv}\ v}}\rangle_k|\!\rangle_{thread}$$

Variant 12 – distributed agents with message communication. Let us next add distributed agents to the language, where each agent encapsulates a multi-threaded program with a locally shared store and where agents communicate with each other by both asynchronous and synchronous messages. Agents can create other agents and messages can contain any value, including other agent names to be used for sending or receiving messages. Here is the additional syntax:

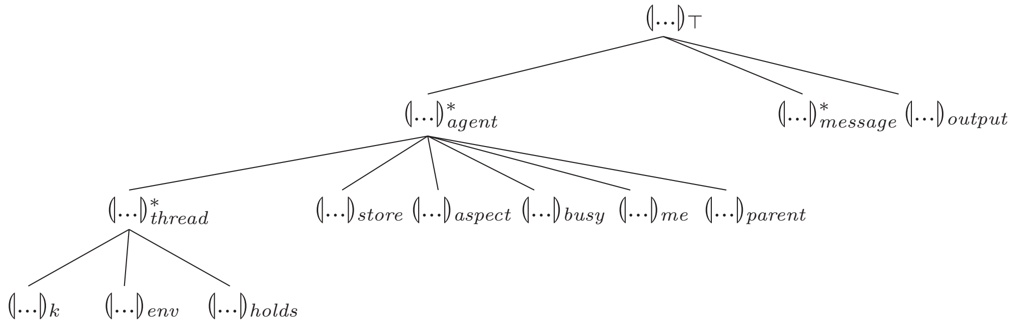
Agent ::= agent identifiers or names
Val ::= ... | *Agent*
Exp ::= ... | new-agent *Stmt* | receive-from *Exp* [*strict*] | receive | me | parent
Stmt ::= ... | send-asynch *Exp Exp* [*strict*] | send-synch *Exp Exp* [*strict*]

Agent is therefore a set set of agent names, which are regarded as any other values in the language, so that they can be passed and returned by functions, assigned to variables, send and received by messages, etc. new-agent s creates a new agent which will execute statement s concurrently with the rest of the agents (and the multiple threads inside those). Unlike in π-calculus (see Section **??**), we here disallow nested agents: all agents are located at the top. Communication between agents is exclusively via messages, which can be send asynchronously or synchronously. Messages contain a sender name, a receiver name, and a value. receive-from e first evaluates e to an agent name, say a, and then the receiving thread blocks until a message from a is received; when that happens, the thread is unblocked and the received value is passed to it. receive is similar, except that the receiving thread will accept any message sent to its enclosing agent. The special expression constructs me and parent evaluate to the current agent's name and to its parent's, respectively; this way, each agent is given access to two basic communication capabilities (access to more agents can be granted dynamically by other agents, by sending agent names via messages). send-asynch e_1 e_2 evaluates e_1

to an agent, say a, and e_2 to a value, say v, sends a message to a containing a, and then dissolves letting the execution to continue normally. send-synch is similar but it blocks the current thread until (one of the threads in) the destination agent receives the message.

An agent terminates when it has no thread left, and a program terminates when it has no agent left. Here is the new configuration syntax, together with its structural/repetition information, as well as the initialization and termination equations (when no agents left, remaining messages are discarded):

$$\begin{array}{l} \mathit{Config} ::= \ldots \mid (\!|\mathsf{Set}[\mathit{Config}]|\!)_{agent} \mid (\!|\mathit{Agent}, \mathit{Agent}, \mathit{Val}|\!)_{message} \\ [\![s]\!] = (\!|(\!|(\!|(\!|s|\!)_k\ (\!|\cdot|\!)_{env}\ (\!|\cdot|\!)_{holds}|\!)_{thread}\ (\!|\cdot|\!)_{store}\ \ (\!|\cdot|\!)_{aspect}\ (\!|\cdot|\!)_{busy}\ (\!|n|\!)_{me}\ (\!|n|\!)_{parent}|\!)_{agent}\ (\!|\cdot|\!)_{output}|\!)_{\top} \quad \text{where } n \in \mathit{Agent} \text{ fresh} \\ (\!|(\!|vl|\!)_{output}\ M|\!)_{\top} \to vl \quad \text{when } M \text{ contains only messages (zero or more)} \end{array}$$



Note that the output is shared by all agents and that there can be multiple agents and messages floating in the top-level soup, same way multiple threads can float in each agent. The following K rules are straightforward; they do formally precisely what was described informally above. Recall that context transforming is assumed to complete the partial configuration structural information in these rules:

$$(\!|\underset{m}{\underline{\textsf{new-agent}\ s}}|\!\rangle_k\ (\!|n|\!)_{me}\ \frac{\cdot}{(\!|(\!|(\!|s|\!)_k\ (\!|\cdot|\!)_{env}\ (\!|\cdot|\!)_{holds}|\!)_{thread}\ (\!|\cdot|\!)_{store}\ \ (\!|\cdot|\!)_{aspect}\ (\!|\cdot|\!)_{busy}\ (\!|m|\!)_{me}\ (\!|n|\!)_{parent}|\!)_{agent}} \quad \text{where } m \in \mathit{Agent} \text{ fresh}$$

$$(\!|A|\!)_{agent} \to \cdot \quad \text{when } A \text{ contains no thread}$$

$$(\!|\underset{n}{\underline{\textsf{me}}}|\!\rangle_k\ (\!|n|\!)_{me}$$

$$(\!|\underset{n}{\underline{\textsf{parent}}}|\!\rangle_k\ (\!|n|\!)_{parent}$$

$$(\!|\underset{\cdot}{\underline{\textsf{send-asynch}\ m\ v}}|\!\rangle_k\ (\!|n|\!)_{me}\ \frac{\cdot}{(\!|n, m, v|\!)_{message}}$$

$$(\!|\underset{v}{\underline{\textsf{receive-from}\ n}}|\!\rangle_k\ (\!|m|\!)_{me}\ \underset{\cdot}{\underline{(\!|n, m, v|\!)_{message}}}$$

$$(\!|\underset{v}{\underline{\textsf{receive}}}|\!\rangle_k\ (\!|m|\!)_{me}\ \underset{\cdot}{\underline{(\!|\_, m, v|\!)_{message}}}$$

$$\langle\!(\!|\underset{\cdot}{\underline{\textsf{send-synch}\ m\ v}}|\!\rangle_k\ (\!|n|\!)_{me}|\!\rangle_{agent}\ \langle\!(\!|\underset{v}{\underline{\textsf{receive-from}\ n}}|\!\rangle_k\ (\!|m|\!)_{me}|\!\rangle_{agent}$$

$$(\!|\underset{\cdot}{\underline{\textsf{send-synch}\ m\ v}}|\!\rangle_k\ \langle\!(\!|\underset{v}{\underline{\textsf{receive}}}|\!\rangle_k\ (\!|m|\!)_{me}|\!\rangle_{agent}$$

None of the definitions of the existing features needs to change. It is interesting, again, to analyze the resulting semantics of halt. If halt was defined as in “Variant 3” (extension with output), then it would just dissolve the remaining computation in the current thread in the current agent. If halt was defined like in the alternative rule proposed at the end of “Variant 10” (extension with concurrency), then it would halt the entire program. Therefore, the two semantic alternatives for halt have extreme behaviors. For this language, it may make sense to have a halt statement that halts only the issuing agent, dissolving all its threads but allowing the remaining agents to continue their executions. If that is what one wants, then one needs to remove the previous rule of halt and add the following rule instead:

$$\langle\!|\,(\!|\mathsf{halt}|\!)_k\,|\!\rangle_{agent} \to \cdot$$

Variant 13 – self-generation of code. There is an increasingly broad interest in generative programming these days. We next add support for dynamic code generation to our language.

$$\begin{array}{l}
\mathit{Exp} ::= \ldots \mid \mathsf{quote}\ \mathit{Exp} \mid \mathsf{unquote}\ \mathit{Exp} \mid \mathsf{eval}\ \mathit{Exp}\ [\mathit{strict}] \\
\mathit{Val} ::= \ldots \mid \mathit{code}(K) \\
K ::= \ldots \mid \mathit{quote}(\mathit{Nat}, \mathsf{List}[K]) \mid \mathit{code}(\mathsf{List}[K]) \mid K \boxed{\curvearrowright} K\ [\mathit{strict}] \mid \boxed{\mathit{Id}}(\mathsf{List}[K])\ [\mathit{strict}(2)] \mid K \boxed{,} K\ [\mathit{strict}]
\end{array}$$

$$\begin{array}{ll}
(\!|\mathsf{quote}(k)|\!)_k = (\!|\mathit{quote}(0, k)|\!)_k & \\
\mathit{quote}(n, k_1 \curvearrowright k_2) = \mathit{quote}(n, k_1) \boxed{\curvearrowright} \mathit{quote}(n, k_2) & \mathit{code}(k_1) \boxed{\curvearrowright} \mathit{code}(k_2) = \mathit{code}(k_1 \curvearrowright k_2) \\
\mathit{quote}(n, f(\mathit{kl})) = \boxed{f}(\mathit{quote}(n, \mathit{kl}))\ \text{if}\ f \neq \mathsf{quote}, \mathsf{unquote} & \boxed{f}(\mathit{code}(\mathit{kl})) = \mathit{code}(f(\mathit{kl})) \\
\mathit{quote}(n, \mathsf{quote}(k)) = \boxed{\mathsf{quote}}(\mathit{quote}(s(n), k)) & \\
\mathit{quote}(0, \mathsf{unquote}(k)) = k & \\
\mathit{quote}(s(n), \mathsf{unquote}(k)) = \boxed{\mathsf{unquote}}(\mathit{quote}(n, k)) & \\
& \\
\mathit{quote}(n, (k, \mathit{kl})) = \mathit{quote}(n, k) \boxed{,}\ \mathit{quote}(n, \mathit{kl})\ \text{if}\ \mathit{kl} \neq \cdot & \mathit{code}(k) \boxed{,}\ \mathit{code}(\mathit{kl}) = \mathit{code}(k, \mathit{kl}) \\
\mathit{quote}(n, k) = \mathit{code}(k)\ \text{if}\ k \in \mathit{Val} \cup \mathit{Name} & \mathsf{eval}\ \mathit{code}(k) = k
\end{array}$$

References

[1] Sergio Antoy, Bernd Brassel, and Michael Hanus. Conditional narrowing without conditions. In *PPDP*, pages 20–31, 2003.

[2] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[3] Jean-Pierre Banâtre and Daniel Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15(1):55–77, 1990.

[4] Jan Bergstra and J. V. Tucker. Equational specifications, complete term rewriting systems, and computable and semicomputable algebras. *Journal of the Association for Computing Machinery*, 42(6):1194–1230, 1995.

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

[6] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. *ENTCS*, 15, 1998.

[7] Fabricio Chalub and Christiano Braga. Maude MSOS tool. In Grit Denker and Carolyn Talcott, editors, *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 133–146. Elsevier, 2007.

[8] Feng Chen, Mark Hills, and Grigore Rosu. A Rewrite Logic Approach to Semantic Definition, Design and Analysis of Object-Oriented Languages. Technical Report UIUCDCS-R-2006-2702, Department of Computer Science, University of Illinois at Urbana-Champaign, March 2006.

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

[10] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Rosu. Formal analysis of Java programs in JavaFAN. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.

[11] Matthias Felleisen and Robert Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[12] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: algebraic specification in action.* Kluwer, 2000.

[13] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and validation methods*, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[14] Mark Hills and Grigore Rosu. KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis. In *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA'07)*, volume 4533 of *LNCS*, pages 246–256. Springer, 2007.

[15] Mark Hills and Grigore Rosu. On Formal Analysis of OO Languages using Rewriting Logic: Designing for Performance. In *Proceedings of the 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *LNCS*, pages 107–121. Springer-Verlag, 2007.

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

[17] Massimo Marchiori. On deterministic conditional rewriting. Computation Structures Group, Memo 405, MIT Laboratory for Computer Science, 1997.

[18] Narciso Martí-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

[19] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proceedings of 15th International Conference on Rewriting Techniques and Applications, (RTA'04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 301–311, 2004.

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

[21] José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, volume 3116 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2004.

[22] José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In *IJCAR '04*, pages 1–44, 2004.

[23] José Meseguer and Grigore Rosu. The rewriting logic semantics project. *J. TCS*, 373(3):213–237, 2007. Also appeared in *SOS '05*, volume 156(1) of *ENTCS*, pages 27–56, 2006.

[24] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[25] Robin Milner. *Communication and Concurrency*. 1989.

[26] Robin Milner. Functions as processes. In Mike Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 167–180. Springer, 1990.

[27] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, 1997.

[28] Peter D. Mosses. Foundations of modular sos. In Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *MFCS*, volume 1672 of *Lecture Notes in Computer Science*, pages 70–80. Springer, 1999.

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

[30] Enno Ohlebusch. Transforming conditional rewrite systems with extra variables into unconditional systems. In *LPAR'99*, pages 111–130, 1999.

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

[32] Grigore Rosu. Cs322, fall 2003 - programming language design: Lecture notes. Technical Report UIUCDCS-R-2003-2897, University of Illinois at Urbana-Champaign, Department of Computer Science, 2003.

[33] Grigore Rosu. K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation. Technical Report UIUCDCS-R-2006-2802, Computer Science Department, University of Illinois at Urbana-Champaign, 2006. A previous version of this work has been published as technical report UIUCDCS-R-2005-2672 in 2005. K was first introduced in 2003, in the technical report UIUCDCS-R-2003-2897: lecture notes of CS322 (programming language design).

[34] Hartley Rogers Jr. *Theory of Recursive Functions and Effective Computability.* MIT press, Cambridge, MA, 1987.

[35] Traian Florin Serbănută and Grigore Rosu. Computationally equivalent elimination of conditions - extended abstract. In *Proceedings of Rewriting Techniques and Applications (RTA'06)*, volume 4098 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2006. also appeared as Technical Report UIUCDCS-R-2006-2693, February 2006.

[36] Traian Florin Serbanuta, Grigore Rosu, and Jose Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 2007. to appear.

[37] Christopher Strachey and Christopher P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. *Higher-Order and Symb. Computation*, 13(1/2):135–152, 2000.

[38] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.

[39] Eelco Visser. Program Transf. with Stratego/XT: Rules, Strategies, Tools, and Systems. In *Domain-Specific Program Generation*, pages 216–238, 2003.

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

A On K Computation Adequacy

Some subtleties. Let us consider a division operator, $\_/\_$, intended to also be *strict*, that is, to evaluate its arguments following the same non-deterministic strategy as $\_+\_$. Its heating/cooling equations are:

$$a_1/a_2 \rightleftharpoons a_1 \curvearrowright \square/a_2$$
$$a_1/a_2 \rightleftharpoons a_2 \curvearrowright a_1/\square$$

To keep the intuitive discussions simple, so far we purposely avoided imposing any restrictions, or side conditions, on the heated sub-computations a_1 and a_2, respectively. In other words, one could in principle do reasoning using the equations above in any direction and for any $a_1, a_2 \in K$. We next show that that is not entirely correct, so in heating/cooling equations like above we'll need to impose some (acceptable) side conditions on the "heated" sub-computations. There is no problem with using these equations from left-to-right, but there are two problems with using them from right-to-left, namely when the heated sub-computations (a_1 in the first and a_2 in the second) are empty or do not correspond to some actual fragment of program. Both problems are artifacts of the desired AI matching for the K constructor $\curvearrowright$. Let us first discuss an example reflecting the first problem (we use parentheses to avoid parsing confusion):

$$\begin{array}{rcll}
a_1/a_2 & = & a_1 \curvearrowright (\square/a_2) & \text{(by the first equation above applied left-to-right)}\\
& = & a_1 \curvearrowright (\cdot \curvearrowright (\square/a_2)) & \text{(by the AI equations for } \curvearrowright)\\
& = & a_1 \curvearrowright (\cdot/a_2) & \text{(by the first equation above applied right-to-left)}\\
& = & a_1 \curvearrowright a_2 \curvearrowright (\cdot/\square) & \text{(by the second equation above applied left-to-right)}\\
& = & a_1 \curvearrowright a_2 \curvearrowright (\cdot \curvearrowright (\cdot/\square)) & \text{(by the AI equations for } \curvearrowright)\\
& = & a_1 \curvearrowright a_2 \curvearrowright (\cdot/\cdot) & \text{(by the second equation above applied right-to-left)}\\
& = & a_1 \curvearrowright a_2 \curvearrowright (\cdot \curvearrowright (\square/\cdot)) & \text{(by the first equation above applied left-to-right)}\\
& = & a_1 \curvearrowright a_2 \curvearrowright (\square/\cdot) & \text{(by the AI equations for } \curvearrowright)\\
& = & a_1 \curvearrowright (a_2/\cdot) & \text{(by the first equation above applied right-to-left)}\\
& = & a_1 \curvearrowright (\cdot \curvearrowright (a_2/\square)) & \text{(by the second equation above applied left-to-right)}\\
& = & a_1 \curvearrowright (a_2/\square) & \text{(by the AI equations for } \curvearrowright)\\
& = & a_2/a_1 & \text{(by the second equation above applied right-to-left)}
\end{array}$$

Therefore, we derived an obviously undesired equality about division using only its heating/cooling equations, so the heating/cooling equations as given above are not sound in their full generality (i.e., for all $a_1, a_2 \in K$). One may think that matching modulo the identity "$\cdot$" is the only problem. The equational proof below shows that matching modulo associativity is also problematic:

$$\begin{array}{rcl}
(a_1/a_2)/(a_3/a_4) & = & (a_1/a_2) \curvearrowright (\square/(a_3/a_4))\\
& = & a_1 \curvearrowright (\square/a_2) \curvearrowright (\square/(a_3/a_4))\\
& = & a_1 \curvearrowright ((\square/a_2)/(a_3/a_4))\\
& = & a_1 \curvearrowright (a_3/a_4) \curvearrowright ((\square/a_2)/\square)\\
& = & a_1 \curvearrowright a_3 \curvearrowright (\square/a_4) \curvearrowright ((\square/a_2)/\square)\\
& = & a_1 \curvearrowright a_3 \curvearrowright ((\square/a_2)/(\square/a_4))\\
& = & a_1 \curvearrowright a_3 \curvearrowright (\square/a_2) \curvearrowright (\square/(\square/a_4))\\
& = & a_1 \curvearrowright (a_3/a_2) \curvearrowright (\square/(\square/a_4))\\
& = & a_1 \curvearrowright ((a_3/a_2)/(\square/a_4))\\
& = & a_1 \curvearrowright (\square/a_4) \curvearrowright ((a_3/a_2)/\square)\\
& = & (a_1/a_4) \curvearrowright ((a_3/a_2)/\square)\\
& = & (a_3/a_2)/(a_1/a_4)
\end{array}$$

Formal definitions and computation adequacy. We next give the formal definition of heating/cooling equations and state their computation (structural) adequacy, saying that no two different "syntactic computations" in K can be proved equal to each other using only the heating/cooling equations; syntactic computations are defined below as ones corresponding to AST's of programs or fragments of programs in the language under consideration (formally, they contain only non-freezer labels and contain no computation sequencing construct "$\_ \curvearrowright \_$").

Definition 1. *Let us consider the following syntactic categories:*

$$\begin{array}{rcll} \mathit{Syntax} & \subseteq & \mathit{KLabel} & \textit{(labels corresponding to the language syntax),} \\ \mathit{KSyntax} & ::= & \mathit{Syntax}(\mathsf{List}[\mathit{KSyntax}]) & \textit{(computations corresponding to programs or fragments of programs),} \\ K^{\circ} & ::= & \mathit{KSyntax} \curvearrowright K & \textit{(computations starting with a program or a fragment of program).} \end{array}$$

The K labels in Syntax are called ***syntactic labels****, or simply* ***syntax****, and the remaining ones (i.e., those in KLabel\Syntax) are called* ***freezers****. Computations in KSyntax correspond to (AST's of fragments of) programs and are called* ***syntactic computations****. Computations in K° correspond to potentially "heated" syntactic computations and we call them* ***weak syntactic computations****; note that $\mathit{KSyntax} \subseteq K^{\circ}$.*

A ***heating/cooling****, or* ***computation structural*** *equation is one of the form (note the side condition)*

$$c[k, k_2, ..., k_n] \rightleftharpoons k \curvearrowright c[\square, k_2, ..., kn] \qquad \textit{when } k \in K^{\circ} \textit{ and } k_2, ..., k_n \in K,$$

where c is some KSyntax multi-context. If HC is a set of heating/cooling equations, then we let $\equiv$ be the equational derivability relation induced by HC on K, i.e., for any $k, k' \in K$, we write $k \equiv k'$ iff $HC \models k = k'$. For $k \in K$, we let $\hat{k}$ denote its equivalence class wrt $\equiv$, called its ***computational equivalence class****.*

The definition above deserves some discussion. *Syntax* contains those labels intended to be the syntax of the language under consideration. The syntactic computations in *KSyntax* are those computations in K corresponding to programs or fragments of programs over *Syntax*, and the weak syntactic computations in K° are those starting with a syntactic computation. The intuition for the weak syntactic computations is that they can be transformed into ones in *KSyntax* after a series of applications of "cooling" steps (heating/cooling equations applied from right-to-left). Both *KSyntax* and K° may contain more computations than intended; for example, the former may contain programs or fragments of programs which are not well-formed, while the latter may contain computations which cannot be necessarily transformed into ones in *KSyntax* by applications of heating/cooling equations. Also, there could be several heating/cooling equations over the same multi-context c; for example, we above had two heating/cooling equations for addition (+). Nevertheless, with all these "inconveniences", the following holds:

Proposition 1. *For any $k \in K$, its computation equivalence class $\hat{k}$ is a finite set.*

When defining languages or language analyses in K, we are only concerned with $\equiv$-equivalence classes of well-formed programs or fragments of programs. For example, in the context of a language definition with binary $+, * \in$ *Syntax* having the heating/cooling equations corresponding to their usual (non-deterministic) strictness but augmented with side conditions as in Definition 1, if $p = x * (y + 2)$ then $\hat{p}$ is the set:

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

Note that there is only one syntactic computation in the computation equivalence class of p above, namely p itself. This is a crucial desired property of K, and is formally stated shortly. Without the side condition in the definition of heating/cooling equations that the scheduled computation must be in K°, the following computations could have also been derivable equationally:

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

While one could say that these alternative computations still make sense, especially the first two, we prefer to regard them as *junk computations*, because, as seen in the examples before Definition 1, they can be the root of undesired problems in which two distinct fragments of program get collapsed as identical via equational reasoning. The following result states that the heating/cooling equations with the reasonable and easy to check side conditions in Definition 1 are indeed adequate:

Theorem 14. ***Computation (structural) adequacy.*** *For any $p, p' \in$ KSyntax, if $\hat{p} = \hat{p'}$ then $p = p'$.*

One can show that precisely the same computation structural equivalence classes could be obtained if one restricted the side conditions of the heating/cooling equations even further, for example requiring $k_2, ..., k_n$ to be in K° like k, or even requiring all the involved computations, namely $k, k_2, ..., k_n$, to be in *KSyntax*! However, we prefer to keep the side conditions of the heating/cooling equations as weak as possible, so that one is given more flexibility in applying them (when executing or when verifying K definitions). For example, when executing K definitions on rewrite engines, the heating/cooling equations are first applied as heating rewrite rules (these are defined shortly; for now, just recall that most equations are applied as rewrite rules when rewrite theories are executed), and then applied as cooling rewrite rules only when the unplugged computation becomes a result. If we required that all the computations involved in the side conditions of heating/cooling rules be in *KSyntax*, then we could only apply the resulting heating rewrite rules following an outermost rewrite strategy. Most rewrite engines provide support for user-configurable strategies, in particular for outermost rewriting, but they typically slow down the rewriting process and, more importantly, their use would inhibit the potential parallelism inherent in the heating/cooling rules of a K definition. Therefore, we prefer to keep the side conditions as general as possible.

On orienting the heating/cooling equations into rewrite rules. Heating/cooling equations therefore give computation equivalence classes for programs or fragments of programs. Subsequent semantic rules and equations can apply by matching computations in such equivalence classes. From a theoretical perspective, there is no need to define "representative" or "canonical" computations in computation equivalence classes; from a practical point of view though, it is quite useful to have techniques that can systematically generate the elements of such computation equivalence classes, or even to automatically calculate representative computations in these equivalence classes. Let us next focus on the first part, namely on how to systematically reach all the computations in a computation equivalence class.

Definition 2. *For each heating/cooling equation*

$$c[k, k_2, ..., k_n] \rightleftharpoons k \curvearrowright c[\square, k_2, ..., kn] \qquad \text{when } k \in K^\circ \text{ and } k_2, ..., k_n \in K,$$

we associate a ***heating (rewrite) rule*** *(note that there is no side condition)*

$$c[k, k_2, ..., k_n] \rightharpoonup k \curvearrowright c[\square, k_2, ..., kn]$$

and a ***cooling (rewrite) rule*** *(note that it has the same side condition as the heating/cooling equation)*

$$k \curvearrowright c[\square, k_2, ..., kn] \rightharpoondown c[k, k_2, ..., k_n] \qquad \text{when } k \in K^\circ \text{ and } k_2, ..., k_n \in K.$$

We used the arrows "$\rightharpoonup$" and "$\rightharpoondown$" for heating and cooling rules, respectively, to easily distinguish them from other rules. As usual, we let $\rightharpoonup^*$ and $\rightharpoondown^*$ denote the transitive and reflexive closures of $\rightharpoonup$ and $\rightharpoondown$.

The side condition "$k \in K^\circ$" has been drooped for heating rules, because one can easily show that the left-hand-sides of heating rules can match a term in a computation equivalence class $\hat{p}$ only when $k \in K^\circ$. Like for the side conditions of heating/cooling equations, we prefer to keep the side conditions as general as possible. The heating rules can be iteratively applied scheduling for processing any allowed subexpressions; once processed, they can be plugged back into context using the cooling rules. Since each heating rule decreases the depth of the heated term (freezers and $\curvearrowright$ do not count in the depth measure, because they cannot match the top of a heating rule), the heating rules terminate as a rewrite system. Moreover, the heating rules are sufficient to reach the entire computational equivalence class of a program or fragment of program, and their effect can be "undone" with cooling rules:

Proposition 2. *If $p \in KSyntax$ and $k \in K$, then $k \in \hat{p}$ iff $p \rightharpoonup^* k$ iff $k \rightharpoondown^* p$.*

Hence, there is a simple and systematic way to generate the computational equivalence class of any program or fragment of program: apply exhaustively all the heating rules and collect all computations that are encountered this way (there is only a finite number of such reachable computations). Also, for any $k \in K$, there is a simple and systematic way to obtain its corresponding syntactic computation: apply all the cooling rules until a normal form is obtained; since the cooling rules form a canonical rewrite system (it terminates because the number of freezers decreases by each application of cooling rules, and is confluent because it is orthogonal) and since each computation equivalence class admits precisely one syntactic computation which is a normal form, it follows that the normal forms obtained by applications of cooling rules are the corresponding syntactic computations. If one wants to generate the complete computation equivalence class corresponding to some $k \in \hat{p}$, then all one needs to do is to apply cooling rules on k until one obtains the normal form p, and then generate $\hat{p}$ by exhaustively applying heating rules on p. In other words, the relation $\rightharpoondown^*; \rightharpoonup^*$ connects any two computations in $\hat{p}$ with $p \in KSyntax$, that is, $\rightharpoondown^*; \rightharpoonup^* = \hat{p} \times \hat{p}$.

An analogy in terms of evaluation contexts may be worthwhile here. The computation equivalence class $\hat{p}$ obtained systematically by applying the heating rules on a program or fragment of program p captures all possible ways in which p, as well as any of its sub-programs, can be split into an "evaluation context" and a "redex". For example, if $k_1 \curvearrowright k_2 \in \hat{p}$ then k_1 can be regarded as the redex and k_2 as the evaluation context; to be more precise, $\hat{k_1}$ is the redex and $\hat{k_2}$ is the evaluation context. This observation is the basis for our faithful translation of context reduction into K discussed in Section 5.4.6. Therefore, it is not surprising that the computation equivalence class approach above is suitable for theoretical developments of K and in particular for proving properties about K definitions, but less suitable for efficient executability. The reason is simple: thinking "modulo" computation equivalence classes is an elegant abstraction, same way as thinking "modulo" splitting a term into an evaluation context and a redex is an elegant abstraction in context reduction, but calculating these equivalence classes at each step before applying a semantic equation or rule is very expensive, same way as calculating the actual decompositions of a term into evaluation contexts and redexes before applying each reduction step in context reduction is very expensive.

In context reduction, unfortunately, there is little to do in general to avoid considering all the splits of a term into an evaluation context and a redex. Refocusing [**?**] may help compute incrementally "the next context/redex split" after each reduction, but it works only in very special cases when there is a unique such split at any moment. Unfortunately, this is not the case when defining complex concurrent languages (if one considers only one split regardless of the desired non-determinism, then one may, obviously, lose behaviors, thus making the definition incorrect). There are two inherent problems in reduction semantics with evaluation contexts that, together, make considering all the expensive context/redex splits unavoidable in the presence of concurrency: (1) its monolithic "everything is syntax" approach, and (2) its interleaving approach to concurrency. Because of (1), there is only one way to get access to a subterm t to reduce it: to find a top-level evaluation context, say c, such that the global, potentially huge syntactic term can be regarded as $c[t]$. In practice, c may contain items that were not part of the original syntax of the language under consideration, such as stores, environments, other resources, etc., but in context reduction these are all "swallowed" by syntax. Because of (2), concurrency is modeled by its resulting non-determinism in reduction semantics, which can only be captured via non-deterministic decomposition/parsing of terms into evaluation contexts and redexes. Therefore, in context reduction one must be ready to pick any possible decomposition of a term into an evaluation context and a redex, so in our terminology, one should have the complete computation equivalence class available at each reduction step.

There is no way to avoid the very same problem in K if one follows the context reduction style of defining languages (explained in Section 5.4.6). However, K promotes a different definitional style, namely one in which the "syntax is kept minimal", by migrating as many items as possible into the configuration "soup". For example, threads or other concurrent processes are kept as special configuration structures, each wrapping, among many other things, a computation. The rewriting logic underlying operational and model-theoretical mechanism will ensure that threads or processes get their expected true concurrency semantics, without ever having to define any evaluation contexts or heating/cooling rules for that purpose. The actual computations, which correspond exclusively to the original syntax of the language under consideration with no artificial enrichments, can be then kept in canonical forms as explained below.

One problem with the current general heating/cooling rules is that they are inverse to each other. Therefore, if one puts them together in one rewrite system then that rewrite system will not terminate; in particular, it cannot be freely used to execute language definitions and thus get an interpreter for the defined languages. Motivated by purely pragmatic reasons, we next describe a subset of heating and cooling rules, more precisely *proper heating* and *minimal cooling* rules, which turned out to be sufficient for many concrete language definitions (in fact, for all we encountered so far), and which can be used together in one rewrite system, because they are not inverse to each other.

Definition 3. *For each heating/cooling equation in a K definition where KResult $\neq \emptyset$*

$$c[k, k_2, ..., k_n] \rightleftharpoons k \curvearrowright c[\square, k_2, ..., kn] \quad \textit{when } k \in K^\circ \textit{ and } k_2, ..., k_n \in K,$$

we associate a ***proper*** *heating (rewrite) rule*

$$c[k, k_2, ..., k_n] \rightharpoonup k \curvearrowright c[\square, k_2, ..., kn] \quad \textit{when } k \in K \backslash KResult \textit{ and } k_2, ..., k_n \in K$$

and a ***minimal*** *cooling (rewrite) rule*

$$r \curvearrowright c[\square, k_2, ..., kn] \rightharpoondown c[r, k_2, ..., k_n] \quad \textit{when } r \in KResult \textit{ and } k_2, ..., k_n \in K.$$

To avoid inventing new arrows, we ambiguously used the same arrows "$\rightharpoonup$" and "$\rightharpoondown$" as for unrestricted heating and cooling rules; unless otherwise specified, when we write these arrows from here on we understand proper heating and/or minimal cooling.

First of all, note that these special heating/cooling rules make sense only when *KResult* is non-empty. Many K definitions do not need to define *KResult*s, especially if one is not interested in executing them. However, when the results of processing programs or fragments of program are known and defined as part of the K definition, then typically the role of the heating rules is to schedule subcomputations for complete processing, while the role of the cooling rules is to plug their results back into context after processing. This is precisely what the above restricted variants of heating and cooling rules capture. Note that there is no risk to violate soundness in considering restricted heating/cooling rules as above, in the sense that one cannot obtain rewriting sequences corresponding to undesired program behaviors when the heating/cooling rules are too restricted; nevertheless, one may not be able to get all the desired behaviors if one's rules are too restricted, in particular, the rewriting process may get "stuck" with a computation on which the existing semantic equations or rewrite rules cannot apply anymore. However, that tends to be a problem more of a theoretical than practical nature. As already mentioned, all our K definitions so far, including all those discussed in this paper, work well with this restricted type of heating/cooling rules when executed.

Since the proper heating and minimal cooling rules are now complementary to each other (rather than inverse to each other), one needs not work with computation equivalence classes anymore. The process of rewriting a program or a fragment of program with heating rules to a normal form can also be regarded as a kind of non-deterministic compilation (static or dynamic, depending upon how/when one applies the heating rules) of the program or fragment of program. The source of non-determinism is the potential non-confluence of the heating rules. For example, recall that the addition (+) was defined to be strict but non-deterministic in our examples above; thus, depending upon which of the two corresponding heating rules one chooses at a certain moment during the evaluation of a program, one may get two different behaviors of the program. Using our implementation of K in Maude (see Section **??**), for example, one can now either get interpreters for defined languages by executing their K definitions as are, without worrying about rewriting computation equivalence classes, or even search the state-space for all the behaviors when the language has concurrency or non-determinism. There is a catch, though, in the latter case: if one's language has strict but non-deterministic operators such as the + above and one does not want to give it a thread-like semantics (explained later in the paper), then one still has to consider computation equivalence classes! The reason is the following: Maude's search command tries all possible matches of rules, but once a rule is picked and applied, it is never undone; for example, if one has the non-deterministic heating rules

$$a_1/a_2 \rightharpoonup a_1 \curvearrowright \square/a_2$$
$$a_1/a_2 \rightharpoonup a_2 \curvearrowright a_1/\square,$$

then during its rewriting or search process Maude non-deterministically picks one of these, in which case the other will never be picked anymore on that expression unless appropriate cooling rules are available. In other words, in our Maude implementation, the two heating rules above define a "non-deterministic choice" evaluation strategy for + (i.e., non-deterministically pick a_1 or a_2 and evaluate it all the way, as opposed to pick any one of them and evaluate it one step, then pick again any one of them and evaluate one step, etc.). The morale here is the following: if one is interested in non-deterministic strict operators in ones language and one wants to explore all possible behaviors of a program, then one should use all the cooling rules instead of the minimal ones; one should be aware then that the heating and cooling rules are inverse to each other, so one should not attempt to use such a definition as an interpreter, because its execution may not terminate; instead, if an interpreter is desired, then one should use minimal cooling rules.

Strictness inference. Chucky Ellison, a PhD student at the University of Illinois, observed that in almost all existing K definitions, both of languages and of analyses such as type checkers or inferencers, the strictness attributes of language constructs can be *automatically inferred* from the rest of the K semantics in a rather straightforward way: if a language construct admits some K semantic sentence (discussed below) in which one of its arguments is a result, then it means that that language construct was intended to be strict in that argument. This strictness inference procedure/convention is simple to implement. There appear to be a few minor issues that need to be well understood before strictness inference is adopted: (1) in some rare cases, one may not want an operation using a value on one of its arguments in the semantics to become strict in that argument (for example, in standard λ-calculus, one may want to apply β-reduction anywhere without obeying any rewriting strategy like in call-by-value or call-by-name); (2) K definitions will be harder to debug and test incrementally, because the strictness information can be inferred only after all the semantics is given; (3) (due to Traian Florin Serbanuta, also a PhD student at the University of Illinois) it would not be possible to distinguish between non-deterministically strict and sequentially strict attributes, unless another annotation/attribute is adopted. Additionally, one may want to specify strictness anyway, just for clarity.

B Defining SILF in K

SILF is a simple imperative language with functions. It is a C-like language, but a lot simpler in many aspects. For example, SILF has no structures and no pointers. A program consists of a ";"-separated sequence of global variable declarations, followed by a sequence of function declarations, one of them expected to be called main. Nested function declarations are not accepted. The program is executed by calling the function main(). Scoping is static, or lexical, and functions see each other and also the global variables. In particular, a function can recursively call itself. Statements can be grouped in blocks with local variable declarations. The names of the variables that appear in a sequence of declarations are expected to be different, and so are the function names. Additionally, the function names are expected to be different from the global variable names and not to be used for anything else but function invocations. Finally, each function must return explicitly, using a "return v" statement, for some value v. These conditions can be easily checked statically.

An untyped version of SILF is presented in Appendix B.1, that is, one in which variables and functions are are not declared any types. Values in SILF can have two basic types, integer and boolean; in our Maude implementation of K definitions, both boolean and integer values are, by convention, prefixed with a "#" to keep language arithmetic and boolean constructs distinct from Maude builtin operators that happen to have the same name. Even in its untyped version, SILF is still partly dynamically typed, in the sense that, when executed, a program "gets stuck" if certain inappropriate operations are encountered during its execution, such as adding an integer with a boolean, or calling a function with a wrong number of arguments, and so on. By a stuck program we mean one that cannot be advanced anymore due to the fact that no rewrite rule matches. No advanced dynamic type checking is performed by the untyped SILF; in particular, a variable can be assigned values of different types during its lifetime, and no type checking is performed when functions are invoked, because no typing information is available for them. Appendix B.2 discusses an actual dynamically typed variant of SILF, in which more advanced dynamic typing is performed. Note that, for uniformity reasons, we prefer to store the declared functions as lambda-like expressions in the store; that is not necessary (functions could have been stored in a function environment), but it eases the definition of name lookup; indeed, in our current definitions of SILF discussed in this section, function names and variable names are looked up similarly. To give the reader a better feel for the untyped version of the SILF language, here is a heap sort program that can be executed by our semantics (written in SILF by Pat Meredith):

```
var numbers[100] ;

function siftDown(root, bottom) {
   var done ; var maxChild ; var temp ;
   done = false ;
   while (root * 2 <= bottom and not done) do {
     if (root * 2 == bottom) then {
       maxChild = root * 2
     }
     else if (numbers[root * 2] > numbers[root * 2 + 1]) then { maxChild = root * 2 }
          else { maxChild = root * 2 + 1 }
     if (numbers[root] < numbers[maxChild]) then {
       temp = numbers[root] ;
       numbers[root] = numbers[maxChild] ;
       numbers[maxChild] = temp ;
       root = maxChild
     }
     else { done = true }
   }
   return 0
}

function heapSort(size) {
  var temp ; var i ;
  i = (size / 2) - 1 ;
  while (i >= 0) do {
     call siftDown(i, size - 1) ;
     i = i - 1
```

```
    }
    i = size - 1 ;
    while (i >= 1 ) do {
     temp = numbers[0] ;
       numbers[0] = numbers[i] ;
       numbers[i] = temp ;
       call siftDown(0, i - 1) ;
       i = i - 1
    }
    return 0
  }

  function main() {
       var x ;
       x = read() ;
       for i = 0 to (x - 1) do {
         numbers[i] = read()
       }
       call heapSort(x) ;
       for i = 0 to (x - 1) do {
         write(numbers[i])
       }
       return 0
  }
```

Appendix B.2 adds type declarations to SILF, for both variables and functions. Type declarations are regarded as rigid "contracts", or "specifications", so they cannot be violated and cannot be changed at runtime. The definition in Appendix B.2 modifies the one in Appendix B.1 to check type declarations dynamically. Only the executed path is checked, so there could be other executions possible (under a different input, for example) that violate the typing policy. Values are not tagged with their types in the store; we assume instead a function *typeOf* on integer and boolean values returning their corresponding type. The declared type of a variable is stored in the environment, together with the location of that variable. In the case of arrays, the declared length of the array is incorporated as part of its type, so that runtime array accesses can be checked to fall within the specified boundaries; languages like Java also perform such boundary checks dynamically. Dynamic checking of types, safety policies, contracts, or specifications is common and instructive in its own way. The purpose of Appendix B.2 is to show that runtime verification of safety policies/properties can be very easily achieved in K, by slightly adjusting the existing definition.

Appendix B.3 defines a static type checker for SILF, following the same K framework as the other definitions. The major difference is that the domain of interpretation for the language syntax is now one of types (instead of concrete values), that is, programs or fragments of program, as well as computations in general, now evolve into types; types are therefore the results of computations. Most operations are now strict in all their arguments, including the conditionals and the loops. The program is first traversed and all the global variable and function declarations are collected in the global environment, while the function bodies are scheduled for typing in *toType*. This step is sub-linear in complexity, because the bodies of the functions are not traversed. Then each function body is processed separately, traversing all its code; in the worst case, assuming no concurrent capabilities of the underlying rewrite engine, the complexity of checking each function is linear in its size. Complexity-wise, our approach to type checking using K is therefore as good as, or better, than defining/implementing type checking algorithms. It has the additional benefit of being formal and using the same logical framework as the definition of the concrete semantics, so it should facilitate formal verification and proofs of correctness (such as, for example, type preservation and progress).

When reading the subsequent sections, recall that the "K-annotated syntax" of a language defines more than the actual syntax of the language: it defines the syntax of that language's computations, including their structural identities (using the "strict" or default attributes; structurally equivalent computations are identical entities), as well as some straightforward reduction rules (using the "extends" attribute). Even though we use variable names such as d for "declarations" or s for "statements", in K definitions there is only one syntactic category, K, of computations. In fact, the different equations for sequential composition ";" are given only for clarity; one would suffice. The K annotations to syntax capture local, syntax driven, typically uninteresting and rather trivial and boring parts of a language definition. The remaining rules and equations are grouped into the "K configuration and semantics" part of the definition.

B.1 Untyped SILF

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\boxed{\text{K-Annotated Syntax of Untyped SILF}}} \\
\mathit{Nat} & ::= & 0 \mid 1 \mid 2 \mid \ldots \text{ all natural numbers} \\
\mathit{Int} & ::= & \ldots \text{ all integer numbers} \\
\mathit{Bool} & ::= & \mathsf{true} \mid \mathsf{false} \\
\mathit{Name} & ::= & \text{all identifiers; to be used as names of variables and functions} \\
\mathit{Exp} & ::= & \mathit{Int} \mid \mathit{Bool} \mid \mathit{Name} \\
 & \mid & \mathit{Name}[\mathit{Exp}]\ [\mathit{strict}(2)] \\
 & \mid & \mathsf{read}() \\
 & \mid & \mathit{Name}(\mathsf{List}[\mathit{Exp}])\ [\mathit{strict}(1),\quad f(el_1, e, el_2) \rightleftharpoons e \curvearrowright f(el_1, \square, el_2)] \\
 & \mid & \mathit{Exp} + \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ +_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} - \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ -_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} * \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ *_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} \,/\, \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \mathit{quotient}_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp}\,\%\mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \mathit{remainder}_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & -\mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ -_{\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} < \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ <_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} <= \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \leq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} > \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ >_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} >= \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \geq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} == \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ =_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ != \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \neq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ \mathsf{and}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \wedge_{\mathit{Bool}\times\mathit{Bool}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ \mathsf{or}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \vee_{\mathit{Bool}\times\mathit{Bool}\to\mathit{Bool}}] \\
 & \mid & \mathsf{not}\ \mathit{Exp}\ [\mathit{strict},\ \mathit{extends}\ \neg_{\mathit{Bool}\to\mathit{Bool}}] \\
\mathit{Decl} & ::= & \mathsf{var}\ \mathit{Name} \mid \mathsf{var}\ \mathit{Name}[\mathit{Nat}] \\
 & \mid & \mathit{Decl}; \mathit{Decl}\ [d_1; d_2 = d_1 \curvearrowright d_2] \\
\mathit{Stmt} & ::= & \{\}\ [\{\} = \cdot] \\
 & \mid & \{\mathit{Stmt}\}\ [\{s\} = s] \\
 & \mid & \{\mathit{Decl}; \mathit{Stmt}\} \\
 & \mid & \mathit{Stmt}; \mathit{Stmt}\ [s_1; s_2 = s_1 \curvearrowright s_2] \\
 & \mid & \mathsf{write}(\mathit{Exp})\ [\mathit{strict}] \\
 & \mid & \mathit{Name} = \mathit{Exp}\ [\mathit{strict}(2)] \\
 & \mid & \mathit{Name}[\mathit{Exp}] = \mathit{Exp}\ [\mathit{strict}(2,3)] \\
 & \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt}\ [\mathit{strict}(1)] \\
 & \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Stmt}\ [\mathsf{if}\ e\ \mathsf{then}\ s = \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ \cdot] \\
 & \mid & \mathsf{while}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Stmt} \\
 & \mid & \mathsf{for}\ \mathit{Name} = \mathit{Exp}\ \mathsf{to}\ \mathit{Exp}\ \mathsf{do}\ \mathit{Stmt} \\
 & & \quad [\mathsf{for}\ i = e_1\ \mathsf{to}\ e_2\ \mathsf{do}\ s = \{\mathsf{var}\ i;\ i = e_1;\ \mathsf{while}\ (i <= e_2)\ \mathsf{do}\ \{s; i = i + 1\}\}] \\
 & \mid & \mathsf{call}\ \mathit{Exp}\ [\mathit{strict}] \\
 & \mid & \mathsf{return}\ \mathit{Exp}\ [\mathit{strict}] \\
\mathit{FunDecl} & ::= & \mathsf{function}\ \mathit{Name}(\mathsf{List}[\mathit{Name}])\ \mathit{Stmt} \\
 & \mid & \mathit{FunDecl}\,\mathit{FunDecl}\ [\mathit{fd}_1\ \mathit{fd}_2 = \mathit{fd}_1 \curvearrowright \mathit{fd}_2] \\
\mathit{Pgm} & ::= & \mathit{FunDecl} \\
 & \mid & \mathit{Decl}; \mathit{FunDecl}\ [d; \mathit{fd} = d \curvearrowright \mathit{fd}]
\end{array}
$$

$\boxed{\text{K Configuration and Semantics of Untyped SILF}}$

$$\begin{array}{rcl}
Val & ::= & Int \mid Bool \mid \lambda\mathsf{List}[Name].K \\
KResult & ::= & Val \\
Env & ::= & Map[Name, Loc] \\
Store & ::= & Map[Loc, Val] \\
FStack & ::= & \mathsf{List}[Env \times K] \\
ConfigItem & ::= & k(K) \mid fstack(FStack) \mid env(Env) \mid genv(Env) \\
 & \mid & in(\mathsf{List}[Int]) \mid out(\mathsf{List}[Int]) \mid store(Store) \mid nextLoc(Loc) \\
Config & ::= & \mathsf{List}[Int] \mid [\![K, \mathsf{List}[Int]]\!] \mid [\![\mathsf{Set}[ConfigItem]]\!] \\
 & \mid & [\![K]\!]\ [[\![k]\!] = [\![k, \cdot]\!]] \\
K & ::= & \ldots \mid run \mid restore(Env)
\end{array}$$

$[\![p, il]\!] = [\![k(p \curvearrowright run)\ fstack(\cdot)\ env(\cdot)\ genv(\cdot)\ in(il)\ out(\cdot)\ store(\cdot)\ nextLoc(loc(0))]\!]$

$\langle\!| k(\cdot)\ out(il) |\!\rangle_\top = il$

$(\!| \underset{\mathsf{call\ main()}}{\underline{\ run\ }} |\!\rangle_k\ env(\rho)\ genv(\underset{\rho}{\underline{\cdot}})$

$(\!| \underset{\sigma[\rho[x]]}{\underline{\ x\ }} |\!\rangle_k\ env(\rho)\ store(\sigma)$

$(\!| \underset{\sigma[\rho[x] +_{Loc} n]}{\underline{\ x[n]\ }} |\!\rangle_k\ env(\rho)\ store(\sigma)$

$(\!| \underset{i}{\underline{\mathsf{read()}}} |\!\rangle_k\ (\!| \underset{\cdot}{\underline{i}} |\!\rangle_{in}$

$k(\underset{s}{\underline{(\lambda xl.s)(vl) \curvearrowright k}})\ (\!| \underset{(\rho, k)}{\underline{\ \cdot\ }} |\!\rangle_{fstack}\ env(\underset{\rho'[ll'/xl]}{\underline{\ \rho\ }})\ genv(\rho')\ store(\underset{\sigma[vl/ll']}{\underline{\ \sigma\ }})\ nextLoc(\underset{l'}{\underline{l}})$ where l' is $l +_{Loc} |xl|$, and ll' is $l \ldots (l' +_{Loc} -1)$

$(\!| \underset{\cdot}{\underline{\mathsf{var}\ x}} |\!\rangle_k\ env(\underset{\rho[l/x]}{\underline{\ \rho\ }})\ nextLoc(\underset{l +_{Loc} 1}{\underline{\ l\ }})$

$(\!| \underset{\cdot}{\underline{\mathsf{var}\ x[n]}} |\!\rangle_k\ env(\underset{\rho[l/x]}{\underline{\ \rho\ }})\ nextLoc(\underset{l +_{Loc} n}{\underline{\ l\ }})$

$(\!| \underset{d \curvearrowright s \curvearrowright restore(\rho)}{\underline{\ \{d; s\}\ }} |\!\rangle_k\ env(\rho)$

$(\!| \underset{\cdot}{\underline{restore(\rho)}} |\!\rangle_k\ env(\underset{\rho}{\underline{\_}})$

$(\!| \underset{\cdot}{\underline{\mathsf{write}\ i}} |\!\rangle_k\ \langle\!| \underset{i}{\underline{\cdot}} |\!\rangle_{out}$

$(\!| \underset{\cdot}{\underline{x = v}} |\!\rangle_k\ env(\rho)\ store(\underset{\sigma[v/\rho[x]]}{\underline{\ \sigma\ }})$

$(\!| \underset{\cdot}{\underline{x[n] = v}} |\!\rangle_k\ env(\rho)\ store(\underset{\sigma[v/\rho[x] +_{Loc} n]}{\underline{\ \sigma\ }})$

$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_1$

$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_2$

$(\!| \mathsf{while}\ b\ \mathsf{do}\ s |\!\rangle_k = (\!| \mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s) |\!\rangle_k$

$\mathsf{call}\ v \rightarrow .$

$k(\underset{v \curvearrowright k}{\underline{\mathsf{return}\ v \curvearrowright \_}})\ (\!| \underset{\cdot}{\underline{(\rho, k)}} |\!\rangle_{fstack}\ env(\underset{\rho}{\underline{\_}})$

$(\!| \underset{\cdot}{\underline{\mathsf{function}\ f(xl)\ s}} |\!\rangle_k\ env(\underset{\rho[l/f]}{\underline{\ \rho\ }})\ store(\underset{\sigma[\lambda xl.s/l]}{\underline{\ \sigma\ }})\ nextLoc(\underset{l +_{Loc} 1}{\underline{\ l\ }})$

B.2 Type Checking SILF Dynamically

K-Annotated Syntax of Dynamically Type-Checked SILF

$$
\begin{array}{rcl}
\mathit{Nat} & ::= & 0 \mid 1 \mid 2 \mid \ldots \text{ all natural numbers} \\
\mathit{Int} & ::= & \ldots \text{ all integer numbers} \\
\mathit{Bool} & ::= & \mathsf{true} \mid \mathsf{false} \\
\mathit{Name} & ::= & \text{all identifiers; to be used as names of variables and functions} \\
\mathit{Type} & ::= & \mathit{int} \mid \mathit{bool} \;\; \text{(one may add more types if one extends the language)} \\
\mathit{Exp} & ::= & \mathit{Int} \mid \mathit{Bool} \\
 & \mid & \mathit{Name} \\
 & \mid & \mathit{Name}[\mathit{Exp}] \; [\mathit{strict}(2)] \\
 & \mid & \mathsf{read}() \\
 & \mid & \mathit{Name}(\mathsf{List}[\mathit{Exp}]) \; [\mathit{strict}(1), \;\; f(\mathit{el}_1, e, \mathit{el}_2) \rightleftharpoons e \curvearrowright f(\mathit{el}_1, \square, \mathit{el}_2)] \\
 & \mid & \mathit{Exp} + \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; +_{\mathit{Int} \times \mathit{Int} \to \mathit{Int}}] \\
 & \mid & \mathit{Exp} - \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; -_{\mathit{Int} \times \mathit{Int} \to \mathit{Int}}] \\
 & \mid & \mathit{Exp} * \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; *_{\mathit{Int} \times \mathit{Int} \to \mathit{Int}}] \\
 & \mid & \mathit{Exp} \,/\, \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; \mathit{quotient}_{\mathit{Int} \times \mathit{Int} \to \mathit{Int}}] \\
 & \mid & \mathit{Exp} \% \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; \mathit{remainder}_{\mathit{Int} \times \mathit{Int} \to \mathit{Int}}] \\
 & \mid & -\mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; -_{\mathit{Int} \to \mathit{Int}}] \\
 & \mid & \mathit{Exp} < \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; <_{\mathit{Int} \times \mathit{Int} \to \mathit{Bool}}] \\
 & \mid & \mathit{Exp} <= \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; \leq_{\mathit{Int} \times \mathit{Int} \to \mathit{Bool}}] \\
 & \mid & \mathit{Exp} > \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; >_{\mathit{Int} \times \mathit{Int} \to \mathit{Bool}}] \\
 & \mid & \mathit{Exp} >= \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; \geq_{\mathit{Int} \times \mathit{Int} \to \mathit{Bool}}] \\
 & \mid & \mathit{Exp} == \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; =_{\mathit{Int} \times \mathit{Int} \to \mathit{Bool}}] \\
 & \mid & \mathit{Exp} \,!\!= \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; \neq_{\mathit{Int} \times \mathit{Int} \to \mathit{Bool}}] \\
 & \mid & \mathit{Exp} \; \mathsf{and} \; \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; \wedge_{\mathit{Bool} \times \mathit{Bool} \to \mathit{Bool}}] \\
 & \mid & \mathit{Exp} \; \mathsf{or} \; \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; \vee_{\mathit{Bool} \times \mathit{Bool} \to \mathit{Bool}}] \\
 & \mid & \mathsf{not} \; \mathit{Exp} \; [\mathit{strict}, \;\; \mathit{extends} \; \neg_{\mathit{Bool} \to \mathit{Bool}}] \\
\mathit{Decl} & ::= & \mathsf{var} \; \mathit{Type} \; \mathit{Name} \mid \mathsf{var} \; \mathit{Type} \; \mathit{Name}[\mathit{Nat}] \\
 & \mid & \mathit{Decl}; \mathit{Decl} \; [d_1; d_2 = d_1 \curvearrowright d_2] \\
\mathit{Stmt} & ::= & \{\} \; [\{\} = \cdot] \\
 & \mid & \{\mathit{Stmt}\} \; [\{s\} = s] \\
 & \mid & \{\mathit{Decl}; \mathit{Stmt}\} \\
 & \mid & \mathit{Stmt}; \mathit{Stmt} \; [s_1; s_2 = s_1 \curvearrowright s_2] \\
 & \mid & \mathsf{write}(\mathit{Exp}) \; [\mathit{strict}] \\
 & \mid & \mathit{Name} = \mathit{Exp} \; [\mathit{strict}(2)] \\
 & \mid & \mathit{Name}[\mathit{Exp}] = \mathit{Exp} \; [\mathit{strict}(2,3)] \\
 & \mid & \mathsf{if} \; \mathit{Exp} \; \mathsf{then} \; \mathit{Stmt} \; \mathsf{else} \; \mathit{Stmt} \; [\mathit{strict}(1)] \\
 & \mid & \mathsf{if} \; \mathit{Exp} \; \mathsf{then} \; \mathit{Stmt} \; [\mathsf{if} \, e \; \mathsf{then} \, s = \mathsf{if} \, e \; \mathsf{then} \, s \; \mathsf{else} \, \cdot] \\
 & \mid & \mathsf{while} \; \mathit{Exp} \; \mathsf{do} \; \mathit{Stmt} \\
 & \mid & \mathsf{for} \; \mathit{Name} = \mathit{Exp} \; \mathsf{to} \; \mathit{Exp} \; \mathsf{do} \; \mathit{Stmt} \\
 & & \quad [\mathsf{for} \; i = e_1 \; \mathsf{to} \; e_2 \; \mathsf{do} \; s = \{\mathsf{var} \; \mathit{int} \; i; \;\; i = e_1; \;\; \mathsf{while} \; (i <= e_2) \; \mathsf{do} \; \{s; i = i + 1\}\}] \\
 & \mid & \mathsf{call} \; \mathit{Exp} \; [\mathit{strict}] \\
 & \mid & \mathsf{return} \; \mathit{Exp} \; [\mathit{strict}] \\
\mathit{FunDecl} & ::= & \mathsf{function} \; \mathit{Type} \; \mathit{Name}(\mathsf{List}[\mathit{Type}] \; \mathsf{List}[\mathit{Name}]) \; \mathit{Stmt} \\
 & \mid & \mathit{FunDecl} \, \mathit{FunDecl} \; [\mathit{fd}_1 \; \mathit{fd}_2 = \mathit{fd}_1 \curvearrowright \mathit{fd}_2] \\
\mathit{Pgm} & ::= & \mathit{FunDecl} \\
 & \mid & \mathit{Decl}; \mathit{FunDecl} \; [d; \mathit{fd} = d \curvearrowright \mathit{fd}]
\end{array}
$$

$\boxed{\text{K Configuration and Semantics of Dynamically Type-Checked SILF}}$

$$\begin{array}{rcl}
Val & ::= & Int \mid Bool \mid Type\ \lambda\mathsf{List}[Type]\,\mathsf{List}[Name].K \\
KResult & ::= & Val \\
Env & ::= & Map[Name, Loc \times Type] \quad \text{(let } \rho[x] \text{ be } (loc(\rho[x]), type(\rho[x])) \text{ for each } \rho \in Env, x \in Name\text{)} \\
Store & ::= & Map[Loc, Val] \\
FStack & ::= & \mathsf{List}[Env \times K \times Type] \\
ConfigItem & ::= & k(K) \mid fstack(FStack) \mid env(Env) \mid genv(Env) \\
 & \mid & in(\mathsf{List}[Int]) \mid out(\mathsf{List}[Int]) \mid return(Type) \mid store(Store) \mid nextLoc(Loc) \\
Config & ::= & \mathsf{List}[Int] \mid [\![K, \mathsf{List}[Int]]\!] \mid [\![\mathsf{Set}[ConfigItem]]\!] \\
 & \mid & [\![K]\!]\ [[\![k]\!] = [\![k, \cdot]\!]] \\
K & ::= & \ldots \mid run \mid restore(Env) \\
Type & ::= & \ldots \mid ? \mid Type[Nat]
\end{array}$$

$[\![p, il]\!] = [\![k(p \curvearrowright run)\ fstack(\cdot)\ env(\cdot)\ genv(\cdot)\ in(il)\ out(\cdot)\ return(?)\ store(\cdot)\ nextLoc(loc(0))]\!]$

$\langle\!|k(\cdot)\ out(il)|\!\rangle_\top = il$

$k(\underset{\mathsf{call\ main()}}{\underline{run}})\ env(\rho)\ genv(\underset{\rho}{\underline{\cdot}})$

$(\!|\underset{\sigma[loc(\rho[x])]}{\underline{x}}|\!)_k\ env(\rho)\ store(\sigma)$

$(\!|\underset{\sigma[loc(\rho[x]) +_{Loc} n]}{\underline{x[n]}}|\!)_k\ env(\rho)\ store(\sigma)$ where $type(\rho[x]) = t[n']$ with $n < n'$

$(\!|\underset{i}{\underline{\mathsf{read()}}}|\!)_k\ \langle\!|\underset{\cdot}{\underline{i}}|\!)_{in}$

$k(\underset{s}{\underline{(t\ \lambda tl\,xl.s)(vl) \curvearrowright k}})\ (\!|\underset{(\rho,k,t')}{\underline{\cdot}}|\!)_{fstack}\ env(\underset{\rho'[(ll',tl)/xl]}{\underline{\rho}})\ genv(\rho')\ return(\underset{t}{\underline{t'}})\ store(\underset{\sigma[vl/ll']}{\underline{\sigma}})\ nextLoc(\underset{l'}{\underline{l}})$

where l' is $l +_{Loc} |xl|$, ll' is $l \ldots (l +_{Loc} -1)$, and $typeOf(vl){=}tl$

$(\!|\underset{\cdot}{\underline{\mathsf{var}\ t\,x}}|\!)_k\ env(\underset{\rho[(l,t)/x]}{\underline{\rho}})\ nextLoc(\underset{l +_{Loc} 1}{\underline{l}})$

$(\!|\underset{\cdot}{\underline{\mathsf{var}\ t\,x[n]}}|\!)_k\ env(\underset{\rho[(l,t[n])/x]}{\underline{\rho}})\ nextLoc(\underset{l +_{Loc} n}{\underline{l}})$

$(\!|\underset{d \curvearrowright s \curvearrowright restore(\rho)}{\underline{\{d; s\}}}|\!)_k\ env(\rho)$

$(\!|\underset{\cdot}{\underline{restore(\rho)}}|\!)_k\ env(\underset{\rho}{\underline{\_}})$

$(\!|\underset{\cdot}{\underline{\mathsf{write}\ i}}|\!)_k\ \langle\!|\underset{i}{\underline{\cdot}}|\!)_{out}$

$(\!|\underset{\cdot}{\underline{x = v}}|\!)_k\ env(\rho)\ store(\underset{\sigma[v/loc(\rho[x])]}{\underline{\sigma}})$ where $type(\rho[x]) = typeOf(v)$

$(\!|\underset{\cdot}{\underline{x[n] = v}}|\!)_k\ env(\rho)\ store(\underset{\sigma[v/loc(\rho[x]) +_{Loc} n]}{\underline{\sigma}})$ where $type(\rho[x]) = typeOf(v)[n']$ with $n < n'$

$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_1$

$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_2$

$(\!|\mathsf{while}\ b\ \mathsf{do}\ s|\!)_k = (\!|\mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)|\!)_k$

$\mathsf{call}\ v \rightarrow .$

$k(\underset{v \curvearrowright k}{\underline{\mathsf{return}\ v \curvearrowright \_}})\ (\!|\underset{\cdot}{\underline{(\rho,k,t)}}|\!)_{fstack}\ env(\underset{\rho}{\underline{\_}})\ return(\underset{t}{\underline{t'}})$ if $typeOf(v) = t'$

$(\!|\underset{\cdot}{\underline{\mathsf{function}\ t\ f(tl\ xl)\ s}}|\!)_k\ env(\underset{\rho[(l,?)/f]}{\underline{\rho}})\ store(\underset{\sigma[t\ \lambda tl\ xl.s/l]}{\underline{\sigma}})\ nextLoc(\underset{l +_{Loc} 1}{\underline{l}})$

B.3 Type Checking SILF Statically

$\boxed{\text{K-Annotated Syntax of a Static Type Checker for SILF}}$

$$
\begin{array}{rcl}
\mathit{Nat} & ::= & 0 \mid 1 \mid 2 \mid \ldots \text{ all natural numbers} \\
\mathit{Int} & ::= & \ldots \text{ all integer numbers} \\
\mathit{Bool} & ::= & \mathsf{true} \mid \mathsf{false} \\
\mathit{Name} & ::= & \text{all identifiers; to be used as names of variables and functions} \\
\mathit{Type} & ::= & \mathit{int} \mid \mathit{bool} \;\; \text{(one may add more types if one extends the language)} \\
\mathit{Exp} & ::= & \mathit{Int} \mid \mathit{Bool} \\
& \mid & \mathit{Name} \\
& \mid & \mathit{Name}[\mathit{Exp}] \; [\mathit{strict}] \\
& \mid & \mathsf{read}() \\
& \mid & \mathit{Name}(\mathsf{List}[\mathit{Exp}]) \; [\mathit{strict}(1), \quad f(el_1, e, el_2) \rightleftharpoons e \curvearrowright f(el_1, \square, el_2)] \\
& \mid & \mathit{Exp} + \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Exp} - \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Exp} * \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Exp} \,/\, \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Exp} \% \mathit{Exp} \; [\mathit{strict}] \\
& \mid & -\mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Exp} < \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Exp} <= \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Exp} > \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Exp} >= \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Exp} == \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Exp} \, != \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Exp} \; \mathsf{and} \; \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Exp} \; \mathsf{or} \; \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathsf{not} \; \mathit{Exp} \; [\mathit{strict}] \\
\mathit{Decl} & ::= & \mathsf{var} \; \mathit{Type} \; \mathit{Name} \mid \mathsf{var} \; \mathit{Type} \; \mathit{Name}[\mathit{Nat}] \\
& \mid & \mathit{Decl}; \mathit{Decl} \; [\mathit{strict}] \\
\mathit{Stmt} & ::= & \{\} \quad \text{(typing of blocks counts as reduction steps, so they are defined in the semantics part)} \\
& \mid & \{\mathit{Stmt}\} \; [\mathit{strict}] \\
& \mid & \{\mathit{Decl}; \mathit{Stmt}\} \\
& \mid & \mathit{Stmt}; \mathit{Stmt} \; [\mathit{strict}] \\
& \mid & \mathsf{write}(\mathit{Exp}) \; [\mathit{strict}] \\
& \mid & \mathit{Name} = \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathit{Name}[\mathit{Exp}] = \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathsf{if} \; \mathit{Exp} \; \mathsf{then} \; \mathit{Stmt} \; \mathsf{else} \; \mathit{Stmt} \; [\mathit{strict}] \\
& \mid & \mathsf{if} \; \mathit{Exp} \; \mathsf{then} \; \mathit{Stmt} \; [\mathit{strict}] \\
& \mid & \mathsf{while} \; \mathit{Exp} \; \mathsf{do} \; \mathit{Stmt} \; [\mathit{strict}] \\
& \mid & \mathsf{for} \; \mathit{Name} = \mathit{Exp} \; \mathsf{to} \; \mathit{Exp} \; \mathsf{do} \; \mathit{Stmt} \\
& & \quad [\mathsf{for} \; i = e_1 \; \mathsf{to} \; e_2 \; \mathsf{do} \; s = \{\mathsf{var} \; \mathit{int} \; i; \;\; i = e_1; \;\; \mathsf{while} \; (i <= e_2) \; \mathsf{do} \; \{s; i = i + 1\}\}] \\
& \mid & \mathsf{call} \; \mathit{Exp} \; [\mathit{strict}] \\
& \mid & \mathsf{return} \; \mathit{Exp} \; [\mathit{strict}] \\
\mathit{FunDecl} & ::= & \mathsf{function} \; \mathit{Type} \; \mathit{Name}(\mathsf{List}[\mathit{Type}] \; \mathsf{List}[\mathit{Name}]) \; \mathit{Stmt} \\
& \mid & \mathit{FunDecl} \, \mathit{FunDecl} \; [\mathit{strict}] \\
\mathit{Pgm} & ::= & \mathit{FunDecl} \\
& \mid & \mathit{Decl}; \mathit{FunDecl} \; [\mathit{strict}]
\end{array}
$$

$\boxed{\text{K Configuration and Semantics of a Static Type Checker for SILF}}$

$$\begin{array}{rcl}
\mathit{Type} & ::= & \ldots \mid ? \mid \mathit{decl} \mid \mathit{stmt} \mid \mathit{fundecl} \mid \mathit{pgm} \mid \mathit{Type}[] \mid \mathsf{List}[\mathit{Type}] \to \mathit{Type} \\
\mathit{KResult} & ::= & \mathit{Type} \\
\mathit{TEnv} & ::= & \mathit{Map}[\mathit{Name}, \mathit{Type}] \\
\mathit{ConfigItem} & ::= & k(K) \mid \mathit{tenv}(\mathit{TEnv}) \mid \mathit{gtenv}(\mathit{TEnv}) \mid \mathit{return}(\mathit{Type}) \mid \mathit{toType}(\cdot) \\
\mathit{Config} & ::= & \mathit{done} \mid [\![K]\!] \mid [\![\mathsf{Set}[\mathit{ConfigItem}]]\!] \\
K & ::= & \ldots \mid \mathit{restore}(\mathit{TEnv}) \mid \mathit{Type}\ \lambda\mathsf{List}[\mathit{Type}]\ \mathsf{List}[\mathit{Name}].K
\end{array}$$

$[\![p]\!] = [\![k(p)\ \mathit{tenv}(\cdot)\ \mathit{gtenv}(\cdot)\ \mathit{return}(?)\ \mathit{toType}(\cdot)]\!]$
$\langle\!|k(\cdot)\ \mathit{toType}(\cdot)|\!\rangle_{\top} = \mathit{done}$

$i \to \mathit{int},\ b \to \mathit{bool}$
$(\!|\underset{\rho[x]}{\underline{\ x\ }}|\!)_k\ \mathit{tenv}(\rho)$
$(t[])[\mathit{int}] \to t$
$\mathsf{read}() \to \mathit{int}$
$(\mathit{tl} \to t)(\mathit{tl}) \to t$
$\mathit{int} + \mathit{int} \to \mathit{int},\ \mathit{int} - \mathit{int} \to \mathit{int},\ \mathit{int}\ *\ \mathit{int} \to \mathit{int},\ \mathit{int}\ /\ \mathit{int} \to \mathit{int},\ \mathit{int}\ \%\ \mathit{int} \to \mathit{int},\ -\mathit{int}\ \to \mathit{int}$
$\mathit{int} < \mathit{int} \to \mathit{bool},\ \mathit{int} <= \mathit{int} \to \mathit{bool},\ \mathit{int} > \mathit{int} \to \mathit{bool},\ \mathit{int} >= \mathit{int} \to \mathit{bool},\ \mathit{int} == \mathit{int} \to \mathit{bool},\ \mathit{int}\ != \mathit{int} \to \mathit{bool}$
$\mathit{bool}\ \mathsf{and}\ \mathit{bool} \to \mathit{bool},\ \mathit{bool}\ \mathsf{or}\ \mathit{bool} \to \mathit{bool},\ \mathsf{not}\ \mathit{bool} \to \mathit{bool}$
$(\!|\underset{\mathit{decl}}{\underline{\mathsf{var}\ t\ x}}|\!)_k\ \mathit{tenv}(\underset{\rho[t/x]}{\underline{\ \ \rho\ \ }})$
$(\!|\underset{\mathit{decl}}{\underline{\mathsf{var}\ t\ x[\mathit{int}]}}|\!)_k\ \mathit{tenv}(\underset{\rho[t[]/x]}{\underline{\ \ \rho\ \ }})$
$\mathit{decl}; \mathit{decl} \to \mathit{decl}$
$\{\} \to \mathit{stmt}$
$\{\mathit{stmt}\} \to \mathit{stmt}$
$(\!|\underset{d;\, s\ \curvearrowright\ \mathit{restore}(\rho)}{\underline{\ \ \{d; s\}\ \ }}|\!)_k\ \mathit{tenv}(\rho)$ (this and the two above are rules now, because they count as reduction steps)
$\mathit{decl}; \mathit{stmt} \to \mathit{stmt}$
$(\!|t \curvearrowright \underset{\cdot}{\underline{\mathit{restore}(\rho)}}|\!)_k\ \mathit{tenv}(\underset{\rho}{\underline{\ \_\ }})$
$\mathsf{write}\ \mathit{int} \to \mathit{stmt}$
$(t = t) \to \mathit{stmt}$
$((t[])[\mathit{int}] = t) \to \mathit{stmt}$
$\mathsf{if}\ \mathit{bool}\ \mathsf{then}\ \mathit{stmt}\ \mathsf{else}\ \mathit{stmt} \to \mathit{stmt}$
$\mathsf{if}\ \mathit{bool}\ \mathsf{then}\ \mathit{stmt}\ \to \mathit{stmt}$
$\mathsf{while}\ \mathit{bool}\ \mathsf{do}\ \mathit{stmt} \to \mathit{stmt}$
$\mathsf{call}\ t \to \mathit{stmt}$
$k(\underset{\cdot}{\underline{\mathsf{return}\ t \curvearrowright \_}})\ \mathit{return}(\underset{?}{\underline{t}})$
$(\!|\underset{\mathit{fundecl}}{\underline{\mathsf{function}\ t\ f(\mathit{tl}\ \mathit{xl})\ s}}|\!)_k\ \mathit{tenv}(\underset{\rho[(\mathit{tl} \to t)/f]}{\underline{\ \ \rho\ \ }})\ \langle\!|\underset{t\,\lambda \mathit{tl}\,\mathit{xl}.s}{\underline{\ \ \cdot\ \ }}|\!)_{\mathit{toType}}$
$\mathit{fundecl}\ \mathit{fundecl} \to \mathit{fundecl}$
$\mathit{decl}; \mathit{fundecl} \to \mathit{pgm}$
$k(\underset{\cdot}{\underline{t}})\ \mathit{tenv}(\rho)\ \mathit{gtenv}(\underset{\rho}{\underline{\ \cdot\ }})$ if t is pgm or $\mathit{fundecl}$
$k(\underset{s}{\underline{\ \cdot\ }})\ \mathit{tenv}(\underset{\rho'[\mathit{xl} \leftarrow \mathit{tl}]}{\underline{\ \ \rho\ \ }})\ \mathit{gtenv}(\rho')\ \mathit{return}(\underset{t}{\underline{t'}})\ (\!|\underset{\cdot}{\underline{t\ \lambda \mathit{tl}\ \mathit{xl}.s}}|\!\rangle_{\mathit{toType}}$

C Defining FUN in K

This section is concerned with the definition of FUN, a simple call-by-value functional language, and of extensions of FUN with different parameter-passing styles (call-by-name, call-by-need), as well as with the definitions of a polymorphic type checker and of a polymorphic type inferencer for FUN. Generally speaking, FUN is much simpler than most existing functional languages. For example, it has no user defined data-types, no modularization constructs such as functors, and no object-oriented or other paradigm features. It does allow, however, a series of interesting functional language features, such as: functions defined via currying as well as tuple arguments (however, tuples are not values, they can only be used in function definitions and function invocations); let and letrec bindings; lists together with basic operations on them; referencing ("ref e"), dereferencing of references ("$* \ r$") and of variables ("& x"), and side effects through assignment to references ("$r := e$"). The following are valid FUN programs that can be executed in our semantics:

```
-----
P1:
-----
  let f x = x := (* x) + 1
  and x = 7
  in list(x, (f(& x) ; x), (f(& x) ; x))

-----
P2:
-----
  letrec max l (x, y) = if (* x) != y then -1
                        else if null?(cdr l) then (car l)
                             else let x = max (cdr l) ((x := (* x) + 1 ; x), y + 1)
                                  in if (x <= car l) then (car l)
                                     else x
     and map f l = if null? l then list()
                   else cons (f (car l)) (map f (cdr l))
     and factorial x = if x <= 0 then 1
                       else x * factorial(x - 1)
  in max (map factorial list(1, 2, 3, 4, 5, factorial 5)) (ref 1, 1)
```

When executing these program with our Maude formalization of the K definition of FUN discussed below, we got the following results in about 2 seconds:

```
-----
P1:
-----
list(int(7),int(8),int(9))

-----
P2:
-----
668950291344912705758811805409037258675274633313802981029567135230163355724496298936687416527198498
1308157637893214090552534408589408121859898481114389650005964960521256960000000000000000000000000000
```

C.1 Untyped FUN

K-Annotated Syntax of Untyped FUN

Int	::=	… all integer numbers	
$Bool$	::=	true $\mid$ false	
$Name$	::=	all identifiers; to be used as names of variables and functions	
Exp	::=	$Int \mid Bool \mid Name$	
	$\mid$	$Exp + Exp$	$[strict,\ extends\ +_{Int\times Int\to Int}]$
	$\mid$	$Exp - Exp$	$[strict,\ extends\ -_{Int\times Int\to Int}]$
	$\mid$	$Exp * Exp$	$[strict,\ extends\ *_{Int\times Int\to Int}]$
	$\mid$	$Exp\ /\ Exp$	$[strict,\ extends\ quotient_{Int\times Int\to Int}]$
	$\mid$	$Exp\ \%\ Exp$	$[strict,\ extends\ remainder_{Int\times Int\to Int}]$
	$\mid$	$-Exp$	$[strict,\ extends\ -_{Int\to Int}]$
	$\mid$	$Exp < Exp$	$[strict,\ extends\ <_{Int\times Int\to Bool}]$
	$\mid$	$Exp <= Exp$	$[strict,\ extends\ \leq_{Int\times Int\to Bool}]$
	$\mid$	$Exp > Exp$	$[strict,\ extends\ >_{Int\times Int\to Bool}]$
	$\mid$	$Exp >= Exp$	$[strict,\ extends\ \geq_{Int\times Int\to Bool}]$
	$\mid$	$Exp == Exp$	$[strict,\ extends\ =_{Int\times Int\to Bool}]$
	$\mid$	$Exp\ != Exp$	$[strict,\ extends\ \neq_{Int\times Int\to Bool}]$
	$\mid$	Exp and Exp	$[strict,\ extends\ \wedge_{Bool\times Bool\to Bool}]$
	$\mid$	Exp or Exp	$[strict,\ extends\ \vee_{Bool\times Bool\to Bool}]$
	$\mid$	not Exp	$[strict,\ extends\ \neg_{Bool\to Bool}]$
	$\mid$	fun $ExpList$ -> Exp	[fun $e\ el$ -> e' = fun e -> fun el -> e']
	$\mid$	$Exp\ ExpList$	$[strict]$
	$\mid$	let $Binding$ in Exp	
	$\mid$	letrec $Binding$ in Exp	
	$\mid$	if Exp then Exp else Exp	$[strict(1)]$
	$\mid$	ref Exp	$[strict]$
	$\mid$	& $Name$	
	$\mid$	$*\ Exp$	$[strict]$
	$\mid$	$Exp := Exp$	$[strict]$
	$\mid$	list $ExpList$	$[strict]$
	$\mid$	car Exp	$[strict]$
	$\mid$	cdr Exp	$[strict]$
	$\mid$	null? Exp	$[strict]$
	$\mid$	cons $Exp\ Exp$	$[strict]$
	$\mid$	$Exp\ ;\ Exp$	$[strict]$
$ExpList$	::=	$\mathsf{List}^{()}_{,}[Exp]$	
$Binding$	::=	$ExpList = ExpList$	[$(e\ el = e') = (e =$ fun el -> $e')$]
	$\mid$	$\mathsf{List}_{\_\mathsf{and}\_}[Binding]$	[$(el = el_1$ and $el' = el'_1) = (el, el' = el_1, el'_1)$]

$\boxed{\text{K Configuration and Semantics of Untyped FUN}}$

$$\begin{array}{rcl}
\mathit{Val} & ::= & \mathit{Int} \mid \mathit{Bool} \mid \mathit{unit} \mid \mathit{closure}(\mathit{Name}, K, \mathit{Env}) \\
\mathit{KResult} & ::= & \mathit{Val} \\
\mathit{Env} & ::= & \mathit{Map}[\mathit{Name}, \mathit{Loc}] \\
\mathit{Store} & ::= & \mathit{Map}[\mathit{Loc}, \mathit{Val}] \\
\mathit{ConfigItem} & ::= & k(K) \mid \mathit{env}(\mathit{Env}) \mid \mathit{store}(\mathit{Store}) \mid \mathit{nextLoc}(\mathit{Loc}) \\
\mathit{Config} & ::= & \mathit{Val} \mid [\![K]\!] \mid [\![\mathsf{Set}[\mathit{ConfigItem}]]\!]
\end{array}$$

$[\![e]\!] = [\![k(e)\ \mathit{env}(\cdot)\ \mathit{store}(\cdot)\ \mathit{nextLoc}(\mathit{loc}(0))]\!]$
$\langle\!| k(v) |\!\rangle_{\top} = v$

$(\!|\frac{x}{\sigma[\rho[x]]}|\!\rangle_k\ \mathit{env}(\rho)\ \mathit{store}(\sigma)$

$(\!|\frac{\mathsf{let}\ xl = el\ \mathsf{in}\ e}{\mathit{strict}(el) \curvearrowright \mathit{bindTo}(xl) \curvearrowright e \curvearrowright \mathit{restore}(\rho)}|\!\rangle_k\ \mathit{env}(\rho)$

$(\!|\frac{\mathsf{letrec}\ xl = el\ \mathsf{in}\ e}{\mathit{bind}(xl) \curvearrowright \mathit{strict}(el) \curvearrowright \mathit{writeTo}(xl) \curvearrowright e \curvearrowright \mathit{restore}(\rho)}|\!\rangle_k\ \mathit{env}(\rho)$

$(\!|\frac{\mathsf{fun}\ xl\ \texttt{->}\ e}{\mathit{closure}(xl, e, \rho)}|\!\rangle_k\ \mathit{env}(\rho)$

$(\!|\frac{\mathit{closure}(xl, e, \rho)\ vl}{\mathit{results}(vl) \curvearrowright \mathit{bindTo}(xl) \curvearrowright e \curvearrowright \mathit{restore}(\rho')}|\!\rangle_k\ \mathit{env}(\frac{\rho'}{\rho})$

$(\!|\frac{\mathsf{ref}\ v}{l}|\!\rangle_k\ \mathit{store}(\frac{\sigma}{\sigma[v/l]})\ \mathit{nextLoc}(\frac{l}{l +_{\mathit{Int}} 1})$
$(\!|\frac{\&\ x}{\rho[x]}|\!\rangle_k\ \mathit{env}(\rho)$
$(\!|\frac{*\ l}{\sigma[l]}|\!\rangle_k\ \mathit{store}(\sigma)$
$(\!|\frac{l := v}{\mathit{unit}}|\!\rangle_k\ \mathit{store}(\frac{\sigma}{\sigma[v/l]})$

$\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \rightarrow e_1$
$\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \rightarrow e_2$

$\mathsf{car\ list}(v, vl) \rightarrow v$
$\mathsf{cdr\ list}(v, vl) \rightarrow \mathsf{list}(vl)$
$\mathsf{null?\ list}() \rightarrow \mathsf{true}$
$\mathsf{null?\ list}(v, vl) \rightarrow \mathsf{false}$
$\mathsf{cons}\ v\ \mathsf{list}(vl) \rightarrow \mathsf{list}(v, vl)$

$\mathit{unit}; v \rightarrow v$

C.2 Untyped FUN Extended with Call-By-Name and Call-By-Need

$$\boxed{\text{K-Annotated Syntax of Untyped FUN Extended with Call-By-Name and Call-By-Need}}$$

$$
\begin{array}{rcll}
\mathit{Int} & ::= & \ldots \text{ all integer numbers} & \\
\mathit{Bool} & ::= & \mathsf{true} \mid \mathsf{false} & \\
\mathit{Name} & ::= & \text{all identifiers; to be used as names of variables and functions} & \\
\mathit{Exp} & ::= & \mathit{Int} \mid \mathit{Bool} \mid \mathit{Name} & \\
 & \mid & \mathit{Exp} + \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ +_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} - \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ -_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} * \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ *_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} \,/\, \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ \mathit{quotient}_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} \% \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ \mathit{remainder}_{\mathit{Int}\times\mathit{Int}\to\mathit{Int}}] \\
 & \mid & -\mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ -_{\mathit{Int}\to\mathit{Int}}] \\
 & \mid & \mathit{Exp} < \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ <_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} <= \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ \leq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} > \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ >_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} >= \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ \geq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp} == \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ =_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ != \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ \neq_{\mathit{Int}\times\mathit{Int}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ \mathsf{and}\ \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ \wedge_{\mathit{Bool}\times\mathit{Bool}\to\mathit{Bool}}] \\
 & \mid & \mathit{Exp}\ \mathsf{or}\ \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ \vee_{\mathit{Bool}\times\mathit{Bool}\to\mathit{Bool}}] \\
 & \mid & \mathsf{not}\ \mathit{Exp} & [\mathit{strict},\ \mathit{extends}\ \neg_{\mathit{Bool}\to\mathit{Bool}}] \\
 & \mid & \mathsf{fun}\ \mathit{Params}\ \texttt{->}\ \mathit{Exp} & [\mathsf{fun}\ ps\ p\ \texttt{->}\ e = \mathsf{fun}\ ps\ \texttt{->}\ \mathsf{fun}\ p\ \texttt{->}\ e] \\
 & \mid & \mathit{Exp}\ \mathit{ExpList} & [\mathit{strict}(1)] \\
 & \mid & \mathsf{let}\ \mathit{Binding}\ \mathsf{in}\ \mathit{Exp} & \\
 & \mid & \mathsf{letrec}\ \mathit{Binding}\ \mathsf{in}\ \mathit{Exp} & \\
 & \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Exp}\ \mathsf{else}\ \mathit{Exp} & [\mathit{strict}(1)] \\
 & \mid & \mathsf{ref}\ \mathit{Exp} & [\mathit{strict}] \\
 & \mid & \&\mathit{Name} & \\
 & \mid & *\mathit{Exp} & [\mathit{strict}] \\
 & \mid & \mathit{Exp} := \mathit{Exp} & [\mathit{strict}] \\
 & \mid & \mathsf{list}\ \mathit{ExpList} & [\mathit{strict}] \\
 & \mid & \mathsf{car}\ \mathit{Exp} & [\mathit{strict}] \\
 & \mid & \mathsf{cdr}\ \mathit{Exp} & [\mathit{strict}] \\
 & \mid & \mathsf{null?}\ \mathit{Exp} & [\mathit{strict}] \\
 & \mid & \mathsf{cons}\ \mathit{Exp}\ \mathit{Exp} & [\mathit{strict}] \\
 & \mid & \mathit{Exp}\ ;\mathit{Exp} & [\mathit{strict}] \\
\mathit{ExpList} & ::= & \mathsf{List}^{()}_{,}[\mathit{Exp}] & \\
\mathit{Params} & ::= & \mathit{Name}\{\mathit{ParamPassStyle}\} & \\
\mathit{ParamPassStyle} & ::= & \mathsf{val} \mid \mathsf{name} \mid \mathsf{need} & \\
\mathit{Params} & ::= & \mathsf{List}^{()}_{,}[\mathit{Params}] & \\
\mathit{ParamsSeq} & ::= & \mathit{ParamsSeq}\ \mathit{Params} & \\
\mathit{Binding} & ::= & \mathit{Params} = \mathit{ExpList} & [(ps\ p = e) = (ps = \mathsf{fun}\ p\ \texttt{->}\ e)] \\
 & \mid & \mathsf{List}_{\mathsf{_and_}}[\mathit{Binding}] & [(p = el\ \mathsf{and}\ p' = el') = (p, p' = el, el')]
\end{array}
$$

$\boxed{\text{K Configuration and Semantics of Untyped FUN Extended with Call-By-Name and Call-By-Need}}$

$$
\begin{array}{rcl}
Val & ::= & Int \mid Bool \mid Loc \mid unit \\
 & \mid & closure(\mathsf{List}[K], K, Env) \mid frozen(K, Env) \mid unfreeze(K, Env) \mid \mathsf{list}(\mathsf{List}[Val]) \\
KResult & ::= & Val \\
KProper & ::= & \ddagger\mathsf{List}[K]\ddagger\mathsf{List}[K]\ddagger\mathsf{List}[Name]\ddagger\mathsf{List}[K]\ddagger \\
 & \mid & mkFrozenVal(K) \mid mkUnfreezeVal(K) \\
KLabel & ::= & @let \mid @letrec \mid @closure \\
Env & ::= & Map[Name, Loc] \\
Store & ::= & Map[Loc, Val] \\
ConfigItem & ::= & k(K) \mid env(Env) \mid store(Store) \mid nextLoc(Loc) \\
Config & ::= & Val \mid [\![K]\!] \mid [\![\mathsf{Set}[ConfigItem]]\!]
\end{array}
$$

$[\![e]\!] = [\![k(e)\ env(\cdot)\ store(\cdot)\ nextLoc(loc(0))]\!]$

$\langle\!| k(v) |\!\rangle_\top = v$

functions & application

$$
\begin{cases}
(\!|\underset{closure(pl,e,\rho)}{\underline{\mathsf{fun}\ pl\ \texttt{->}\ e}}|\!\rangle_k\ env(\rho) \\
closure(pl,e,\rho)\ el = \ddagger pl\ddagger el\ddagger\cdot\ddagger\cdot\ddagger \curvearrowright @closure(e, restore(\rho)) \\
(\!|\underset{strict(el) \curvearrowright k \curvearrowright bindTo(xl) \curvearrowright e \curvearrowright restore(\rho)}{\underline{\ddagger\cdot\ddagger\cdot\ddagger xl\ddagger el\ddagger \curvearrowright @closure(e,k)}}|\!\rangle_k\ env(\rho)
\end{cases}
$$

let

$$
\begin{cases}
\mathsf{let}\ pl = el\ \mathsf{in}\ e = \ddagger pl\ddagger el\ddagger\cdot\ddagger\cdot\ddagger \curvearrowright @let(e) \\
(\!|\underset{strict(el) \curvearrowright bindTo(xl) \curvearrowright e \curvearrowright restore(\rho)}{\underline{\ddagger\cdot\ddagger\cdot\ddagger xl\ddagger el\ddagger \curvearrowright @let(e)}}|\!\rangle_k\ env(\rho)
\end{cases}
$$

letrec

$$
\begin{cases}
\mathsf{letrec}\ pl = el\ \mathsf{in}\ e = \ddagger pl\ddagger el\ddagger\cdot\ddagger\cdot\ddagger \curvearrowright @letrec(e) \\
(\!|\underset{bind(xl) \curvearrowright strict(el) \curvearrowright writeTo(xl) \curvearrowright e \curvearrowright restore(\rho)}{\underline{\ddagger\cdot\ddagger\cdot\ddagger xl\ddagger el\ddagger \curvearrowright @letrec(e)}}|\!\rangle_k\ env(\rho)
\end{cases}
$$

call-by-value

$$
\begin{cases}
\ddagger(\!|\underset{\cdot}{\underline{x\{\mathsf{val}\}}}|\!\rangle_{???}\ddagger(\!|\underset{\cdot}{\underline{e}}|\!\rangle_{???}\ddagger\langle\!|\underset{x}{\underline{\cdot}}|\!)_{???}\ddagger\langle\!|\underset{e}{\underline{\cdot}}|\!)_{???}\ddagger \\
(\!|\underset{\sigma[\rho[x]]}{\underline{x}}|\!\rangle_k\ env(\rho)\ store(\sigma)\quad \text{when } \sigma[x] \text{ is not of the form } frozen(e,\rho') \text{ or } unfreeze(e,\rho')
\end{cases}
$$

call-by-name

$$
\begin{cases}
\ddagger(\!|\underset{\cdot}{\underline{x\{\mathsf{name}\}}}|\!\rangle_{???}\ddagger(\!|\underset{\cdot}{\underline{e}}|\!\rangle_{???}\ddagger\langle\!|\underset{x}{\underline{\cdot}}|\!)_{???}\ddagger\langle\!|\underset{mkFrozenVal(e)}{\underline{\cdot}}|\!)_{???}\ddagger \\
(\!|\underset{frozen(e,\rho)}{\underline{mkFrozenVal(e)}}|\!\rangle_k\ env(\rho) \\
(\!|\underset{e \curvearrowright restore(\rho)}{\underline{x}}|\!\rangle_k\ env(\underset{\rho'}{\underline{\rho}})\ store(\sigma)\quad \text{when } \sigma[x] \text{ is } frozen(e,\rho')
\end{cases}
$$

call-by-need

$$
\begin{cases}
\ddagger(\!|\underset{\cdot}{\underline{x\{\mathsf{need}\}}}|\!\rangle_{???}\ddagger(\!|\underset{\cdot}{\underline{e}}|\!\rangle_{???}\ddagger\langle\!|\underset{x}{\underline{\cdot}}|\!)_{???}\ddagger\langle\!|\underset{mkUnfreezeVal(e)}{\underline{\cdot}}|\!)_{???}\ddagger \\
(\!|\underset{unfreeze(e,\rho)}{\underline{mkUnfreezeVal(e)}}|\!\rangle_k\ env(\rho) \\
(\!|\underset{e \curvearrowright writeTo\,\rho[x]\ andKeep \curvearrowright restore(\rho)}{\underline{x}}|\!\rangle_k\ env(\underset{\rho'}{\underline{\rho}})\ store(\sigma)\quad \text{when } \sigma[x] \text{ is } unfreeze(e,\rho')
\end{cases}
$$

references

$$
\begin{cases}
(\!|\underset{l}{\underline{\mathsf{ref}\ v}}|\!\rangle_k\ store(\underset{\sigma[v/l]}{\underline{\sigma}})\ nextLoc(\underset{l +_{Int} 1}{\underline{l}}) \\
(\!|\underset{\rho[x]}{\underline{\&\ x}}|\!\rangle_k\ env(\rho) \\
(\!|\underset{\sigma[l]}{\underline{*\ l}}|\!\rangle_k\ store(\sigma) \\
(\!|\underset{unit}{\underline{l := v}}|\!\rangle_k\ store(\underset{\sigma[v/l]}{\underline{\sigma}})
\end{cases}
$$

$\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \to e_1,\quad \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \to e_2$

$\mathsf{car\ list}(v,vl) \to v,\quad \mathsf{cdr\ list}(v,vl) \to \mathsf{list}(vl),\quad \mathsf{null?\ list}() \to \mathsf{true},\quad \mathsf{null?\ list}(v,vl) \to \mathsf{false},\quad \mathsf{cons}\ v\ \mathsf{list}(vl) \to \mathsf{list}(v,vl)$

$unit; v \to v$

C.3 Polymorphic FUN with Type Declarations

$\boxed{\text{K-Annotated Syntax of Polymorphic FUN with Type Declarations}}$

Int	$::=$	$\ldots$ all integer numbers	
$Bool$	$::=$	$\mathsf{true} \mid \mathsf{false}$	
$Name$	$::=$	all identifiers; to be used as names of variables, functions, and types	
$Type$	$::=$	$int \mid bool \mid unit \mid\ Name \mid \mathsf{List}^{()}[Type] \to Type \mid \mathsf{ref}\ Type \mid \mathsf{list}\ Type$	
Exp	$::=$	$Int \mid Bool \mid Name$	
	$\mid$	$Exp + Exp$	$[strict]$
	$\mid$	$Exp - Exp$	$[strict]$
	$\mid$	$Exp * Exp$	$[strict]$
	$\mid$	$Exp\,/\,Exp$	$[strict]$
	$\mid$	$Exp \% Exp$	$[strict]$
	$\mid$	$-Exp$	$[strict]$
	$\mid$	$Exp < Exp$	$[strict]$
	$\mid$	$Exp <= Exp$	$[strict]$
	$\mid$	$Exp > Exp$	$[strict]$
	$\mid$	$Exp >= Exp$	$[strict]$
	$\mid$	$Exp == Exp$	$[strict]$
	$\mid$	$Exp\ != Exp$	$[strict]$
	$\mid$	$Exp\ \mathsf{and}\ Exp$	$[strict]$
	$\mid$	$Exp\ \mathsf{or}\ Exp$	$[strict]$
	$\mid$	$\mathsf{not}\ Exp$	$[strict]$
	$\mid$	$\mathsf{fun}\ Params\ \texttt{->}\ Exp$	$[\mathsf{fun}\ ps\ p\ \texttt{->}\ e = \mathsf{fun}\ ps\ \texttt{->}\ \mathsf{fun}\ p\ \texttt{->}\ e]$
	$\mid$	$Exp\ ExpList$	$[strict]$
	$\mid$	$\mathsf{let}\ Binding\ \mathsf{in}\ Exp$	
	$\mid$	$\mathsf{letrec}\ Binding\ \mathsf{in}\ Exp$	
	$\mid$	$\mathsf{if}\ Exp\ \mathsf{then}\ Exp\ \mathsf{else}\ Exp$	$[strict]$
	$\mid$	$\mathsf{ref}\ Exp$	$[strict]$
	$\mid$	$*Exp$	$[strict]$
	$\mid$	$Exp := Exp$	$[strict]$
	$\mid$	$\&Name$	$[strict]$
	$\mid$	$\mathsf{list}\ ExpList$	$[strict]$
	$\mid$	$\mathsf{list()}\ :\ \mathsf{list}(Type)$	
	$\mid$	$\mathsf{car}\ Exp$	$[strict]$
	$\mid$	$\mathsf{cdr}\ Exp$	$[strict]$
	$\mid$	$\mathsf{null?}\ Exp$	$[strict]$
	$\mid$	$\mathsf{cons}\ Exp\ Exp$	$[strict]$
	$\mid$	$Exp\ ;Exp$	$[strict]$
$ExpList$	$::=$	$\mathsf{List}^{()}_{,}[Exp]$	
$Params$	$::=$	$\mathsf{List}^{()}[Name] : \mathsf{List}^{()}[T] \mid Params, Params$	$[(xl : tl), (xl' : tl') = (xl, xl') : (tl, tl')]$
$ParamsSeq$	$::=$	$Params \mid ParamsSeq\ Params$	
$Binding$	$::=$	$Params = ExpList$	$[(ps\ p = e) = (ps = \mathsf{fun}\ p\ \texttt{->}\ e)]$
	$\mid$	$\mathsf{List}_{\_\mathsf{and}\_}[Binding]$	$[(p = el\ \mathsf{and}\ p' = el') = (p, p' = el, el')]$

$\boxed{\text{K Configuration and Semantics of Polymorphic FUN with Type Declarations}}$

$$\begin{array}{rcl} \mathit{KResult} & ::= & \mathit{Type} \\ \mathit{TEnv} & ::= & \mathit{Map}[\mathit{Name}, \mathit{Type}] \\ \mathit{ConfigItem} & ::= & k(K) \mid \mathit{tenv}(\mathit{TEnv}) \\ \mathit{Config} & ::= & \mathit{Type} \mid [\![K]\!] \mid [\![\mathsf{Set}[\mathit{ConfigItem}]]\!] \end{array}$$

$[\![e]\!] = [\![k(e)\ \mathit{tenv}(\cdot)]\!]$
$\langle\!| k(t) |\!\rangle_\top = t$

$$\text{lookup}\ \left\{ \begin{array}{l} (\!| \underline{x} \atop \rho[x] |\!\rangle_k\ \mathit{tenv}(\rho) \end{array} \right.$$

$$\text{fun \& app}\ \left\{ \begin{array}{l} (\!| \dfrac{\mathsf{fun}\ xl : tl\ \texttt{->}\ e}{(tl \to e) \curvearrowright \mathit{restore}(\rho)} |\!\rangle_k\ \mathit{env}(\dfrac{\rho}{\rho[tl/xl]}) \\ K ::= \cdots \mid \mathsf{List}[\mathit{Type}] \to K \qquad\qquad [\mathit{strict}(2)] \\ (tl \to t)\, tl' = \mathit{applySubst}(\mathit{getSubst}(tl, tl'), t) \end{array} \right.$$

$$\mathsf{let}\ \left\{ (\!| \dfrac{\mathsf{let}\ xl : tl = el\ \mathsf{in}\ e}{\mathit{strict}(el) \curvearrowright \mathit{checkEqualTo}\ tl\ \mathit{andKeep} \curvearrowright \mathit{bindTo}(xl) \curvearrowright e \curvearrowright \mathit{restore}(\rho)} |\!\rangle_k\ \mathit{env}(\rho) \right.$$

$$\mathsf{letrec}\ \left\{ (\!| \dfrac{\mathsf{letrec}\ xl : tl = el\ \mathsf{in}\ e}{\mathit{strict}(el) \curvearrowright \mathit{checkEqualTo}\ tl\ \mathit{andDiscard} \curvearrowright e \curvearrowright \mathit{restore}(\rho)} |\!\rangle_k\ \mathit{env}(\dfrac{\rho}{\rho[tl/xl]}) \right.$$

$$\text{ord ops}\ \left\{ \begin{array}{l} int + int \to int,\ int - int \to int,\ int\ *\ int \to int, \\ int\ /\ int \to int,\ int\ \%\ int \to int,\ \ -int\ \to int, \\ int < int \to bool,\ int <= int \to bool,\ int > int \to bool, \\ int >= int \to bool,\ int == int \to bool,\ int\ != int \to bool, \\ bool\ \mathsf{and}\ bool \to bool,\ bool\ \mathsf{or}\ bool \to bool,\ \mathsf{not}\ bool \to bool \end{array} \right.$$

$$\text{references}\ \left\{ \begin{array}{l} \&\ t \to \mathsf{ref}\ t \\ *\ \mathsf{ref}\ t \to t \\ \mathsf{ref}\ t := t \to unit \end{array} \right.$$

$$\text{conditional}\ \left\{ \mathsf{if\ true\ then}\ t\ \mathsf{else}\ t \to t \right.$$

$$\text{lists}\ \left\{ \begin{array}{l} \mathsf{list()} : (\mathsf{list}(t)) \to \mathsf{list}(t) \\ \mathsf{list}(t, t, tl) \to \mathsf{list}(t, tl) \\ \mathsf{car\ list}(t) \to t \\ \mathsf{cdr\ list}(t) \to \mathsf{list}(t) \\ \mathsf{null?\ list}(t) \to bool \\ \mathsf{cons}\ t\ \mathsf{list}(t) \to \mathsf{list}(t) \end{array} \right.$$

$$\text{seq comp}\ \left\{ unit; t \to t \right.$$

D Defining SKOOL in K

SKOOL is a simple object-oriented language. It is defined as an extension of SILF with classes and objects. SKOOL is single-inheritance: except for the topmost builtin class `object`, each user-definable class in SKOOL extends precisely one other existing class; if an `extends` clause is not specified for a class, then it is assumed that that class extends `object`. Classes contain fields and methods. Objects can be created as instances of classes and messages can be send to objects under the form of method calls. Fields of an object are not visible directly from the outside of the object, but only by means of method calls. A `self` construct can be used as an object; it refers to the current object and it is particularly useful to achieve recursion. Method invocations can be preceded by the keyword `super`, which redirects the call to the superclass; this is particularly useful when methods are overridden, to have access to methods in the superclass in case they were overridden by the current class. Object can be inspected for their type with an "instanceOf" language construct. SKOOL's methods have static scoping and dynamic dispatch. A SKOOL program consists of a set of classes. One class must be called `main` and must have a method called `main`. A program execution starts by creating an instance of the class `main` and then invoking the method `main` on that instance.

An untyped version of SKOOL is presented in Appendix D.1. Like in the case of untyped SILF, that means that variables and functions are are not declared any types. Nevertheless, the execution of programs gets stuck if certain incorrect operations are attempted, such as calling an inexistent method on an object, or adding an integer with a Boolean, etc. Appendix D.2 defines a dynamically typed variant of SKOOL, and Appendix D.3 discusses a static type checker for SKOOL taking into account subtype polymorphism. Here is a SKOOL program that can be executed by our semantics (both columns); the Maude output after executing this program is `2 2 3 11 3 3 11 111 111`:

```
class a {
  var i ; var j ;
  method a() {
    i = 1 ;
    j = i + 1 ;
    return j
  }
  method f() { return self . a() }
  method g() { return self . f() }
  method h() { return (i + j) }
}

class b extends a {
  var j ; var k ;
  method a() {
    return self . b()
  }
  method b() {
    call super a() ;
    j = 10 ;
    k = j + 1 ;
    return k
  }
  method g() { return super h() }
  method h() { return self . g() }
}
```

```
class c extends b {
  method a() { return super a() }
  method b() { return super b() }
  method c() {
    i = 100 ;
    j = i + 1 ;
    k = j + 1 ;
    return k
  }
  method g() { return (i + k * j) }
}

class main {
  method p(o) {
    write(o . f()) ;
    write(o . g()) ;
    write(o . h()) ;
    return 0
  }
  method main() {
    call self . p(new a()) ;
    call self . p(new b()) ;
    call self . p(new c()) ;
    return 0
  }
}
```

D.1 Untyped SKOOL

As mentioned above, we define SKOOL as an extension of SILF. We start by importing the K annotated syntax of untyped SILF *as is*, except for the syntax of function declaration and invocation constructs; we need to exclude the latter because we want functions to become methods and unrestricted function calls to become method invocation messages sent to individual objects. In some sense, a SKOOL program can be regarded as a set of SILF programs, one per class, which can "communicate" with each other by means of method invocation messages sent to objects. We then add the following SKOOL-specific syntax:

$\boxed{\text{K-Annotated Syntax of Untyped SKOOL}}$

(include all syntax from SILF, except constructs for function declaration and invocation)

$$\begin{array}{rcll}
\mathit{Exp} & ::= & \ldots & \\
 & \mid & \mathsf{new}\ \mathit{Name}(\mathsf{List}[\mathit{Exp}]) & [\mathit{strict}(2)] \\
 & \mid & \mathsf{self} & \\
 & \mid & \mathit{Exp}.\mathit{Name}(\mathsf{List}[\mathit{Exp}]) & [\mathit{strict}(1,3)] \\
 & \mid & \mathsf{super}\ \mathit{Name}(\mathsf{List}[\mathit{Exp}]) & [\mathit{strict}(2)] \\
 & \mid & \mathit{Exp}\ \mathsf{instanceOf}\ \mathit{Name} & [\mathit{strict}(1)] \\
\mathit{MethDecl} & ::= & \mathsf{method}\ \mathit{Name}(\mathsf{List}[\mathit{Name}])\ \mathit{Stmt} & \\
 & \mid & \mathit{MethDecl}\ \mathit{MethDecl} & [md_1\ md_2 = md_1 \curvearrowright md_2] \\
\mathit{ClsDecl} & ::= & \mathsf{class}\ \mathit{Name}\ \mathsf{extends}\ \mathit{Name}\ \{\mathit{VarDecl}\ ;\ \mathit{MethDecl}\} & \\
 & \mid & \mathsf{class}\ \mathit{Name}\ \mathsf{extends}\ \mathit{Name}\ \{\mathit{MethDecl}\} & [\mathsf{class}\, c\, \mathsf{extends}\, c'\ \{md\} = \mathsf{class}\, c\, \mathsf{extends}\, c'\ \{\cdot\ ;\ md\}] \\
 & \mid & \mathsf{class}\ \mathit{Name}\ \{\mathit{VarDecl}\ ;\ \mathit{MethDecl}\} & [\mathsf{class}\ c\ \{vd;\ md\} = \mathsf{class}\ c\ \mathsf{extends\ object}\ \{vd;\ md\}] \\
 & \mid & \mathsf{class}\ \mathit{Name}\ \{\mathit{MethDecl}\} & [\mathsf{class}\ c\ \{md\} = \mathsf{class}\ c\ \mathsf{extends\ object}\ \{md\}] \\
 & \mid & \mathit{ClsDecl}\ \mathit{ClsDecl} & [cd_1\ cd_2 = cd_1 \curvearrowright cd_2]
\end{array}$$

To define the K semantics of SKOOL, we also proceed by extending SILF's. However, in addition to the rules corresponding to the syntactic constructs for function declaration and invocation that have been replaced by method declaration and invocation, we also need to change a few other rules in SILF:

- The rules for variable (indexed or not) lookup and update need to change, because of the name resolution policy in object-oriented languages: if a name is not in the current environment (case in which the program would get stuck in SILF), then one should search for that name's location in the current class' layer in the current object; if not there, then one should move with the search up in the class hierarchy of that object, and so on until either the name is found or otherwise the topmost layer, object, is reached (in the latter case, the execution would get stuck).

- The rule for return also needs to change, because the structure of the stack needs to change. Indeed, the new stack will also push/pop the current class (in addition to the current environment and computation already considered in SILF).

Note that an object is a list of layers, each layer being an environment tagged with a class name. There is layer for each superclass of that object, as well as for its class; in other words, there is a layer for each class that the object can be regarded as an instance of. The semantics of object creation, as well as that of object and super method invocation, are all defined in terms of an auxiliary *invoke* computation item. When the first item in the computation, $invoke(m, vl)$ starts searching for the method m with the current class. As it traverses the class hierarchy, the current class is also changed. This is very important, because we want static scoping in our language; therefore, once the method m is found and invoked, any variable should be lookup first in the current environment, and then in the class hierarchy of the current object starting with the class where the method was found! Note that that is indeed what our new definition for variable lookup does (using the auxiliary and easy to define operation *getLoc*).

The new $c(vl)$ construct first creates an object instance of class c and then calls the constructor method of c with arguments vl; constructor methods have the same name as their corresponding classes and are typically intended to initialize the values of the fields. The object creation process allocates to values in the store; it only creates an object layered environment, mapping each field in each layer to a fresh location. Note that the needed computation when returning from the construct invocation is stored in the stacked computation in anticipation: the return value of the construct is discarded (constructs are called for their side effects), then the current object is placed in the computation, and then the current object and computation are restored. As in SILF, methods are expected to return explicitly (using return statements). Objects and super method invocations are easier to define than new. We prefer not to stack the current object because super does not change the current object, so it does not need to stack it.

$\boxed{\text{K Configuration and Semantics of Untyped SKOOL}}$

$$\begin{array}{rcll}
Val & ::= & ... \mid Object & \\
Object & ::= & \mathsf{List}[Name : Env] & \\
Stack & ::= & \mathsf{List}[Name \times Env \times K] & \\
ConfigItem & ::= & k(K) \mid stack(Stack) \mid env(Env) \mid in(\mathsf{List}[Int]) \mid out(\mathsf{List}[Int]) \mid store(Store) \mid nextLoc(Loc) & \\
 & \mid & obj(Object) \mid class(Name) \mid pgm(K) & \\
Config & ::= & \mathsf{List}[Int] \mid \llbracket K, \mathsf{List}[Int]\rrbracket \mid \llbracket \mathsf{Set}[ConfigItem]\rrbracket & \\
 & \mid & \llbracket K \rrbracket & [\llbracket k \rrbracket = \llbracket k, \cdot \rrbracket] \\
KLabel & ::= & ... \mid create \mid addOEnvLayer \mid invoke & \\
KConstant & ::= & ... \mid restore(Object) & [(\!| v \curvearrowright \underset{\cdot}{\underline{restore(o)}} |\!\rangle_k\ obj(\underset{o}{\underline{\_}})]
\end{array}$$

$\llbracket cls, il \rrbracket = \llbracket k(\mathsf{new\ main}())\ stack(\cdot)\ env(\cdot)\ obj(\cdot)\ class(\mathsf{main})\ pgm(cls)\ in(il)\ out(\cdot)\ store(\cdot)\ nextLoc(loc(0))\rrbracket$
$\langle\!| k(o)\ out(il) |\!\rangle_\top = il$

$(\!|\underset{o}{\underline{\mathsf{self}}}|\!\rangle_k\ obj(o)$
$\langle\!| c : \_ |\!\rangle_{???}\ \mathsf{instanceOf}\ c = \mathsf{true}$
$o\ \mathsf{instanceOf}\ c = \mathsf{false}$, otherwise (i.e., when o has no layer labeled c)

$k(\underset{create(c) \curvearrowright invoke(c, vl)}{\underline{\mathsf{new}\ c(vl) \curvearrowright k}})\ env(\underset{\cdot}{\underline{\rho}})\ obj(\underset{\cdot}{\underline{o}})\ class(\underset{c}{\underline{c'}})\ (\!|\underset{(c', \rho, discard \curvearrowright \mathsf{self} \curvearrowright restore(o) \curvearrowright k)}{\underline{\cdot}}|\!\rangle_{stack}$
$k(\underset{invoke(m, vl)}{\underline{(c:\eta,\omega).m(vl) \curvearrowright k}})\ env(\underset{\cdot}{\underline{\rho}})\ obj(\underset{(c:\eta,\omega)}{\underline{o}})\ class(\underset{c}{\underline{c'}})\ (\!|\underset{(c', \rho, restore(o) \curvearrowright k)}{\underline{\cdot}}|\!\rangle_{stack}$
$k(\underset{invoke(m,vl)}{\underline{\mathsf{super}\ m(vl) \curvearrowright k}})\ env(\underset{\cdot}{\underline{\rho}})\ class(\underset{c'}{\underline{c}})\ (\!|\underset{(c,\rho,k)}{\underline{\cdot}}|\!\rangle_{stack}\ pgm\langle\!|\mathsf{class}\ c\ \mathsf{extends}\ c'\ \{\_\ ;\_\}|\!\rangle_{???}$

$(\!|\underset{vd \curvearrowright addOEnvLayer \curvearrowright create(c')}{\underline{create(c)}}|\!\rangle_k\ pgm\langle\!|\mathsf{class}\ c\ \mathsf{extends}\ c'\ \{vd; md\}|\!\rangle_{???}$
$create(\mathsf{object}) = \cdot$
$k(\underset{bind\ xl\ to\ vl \curvearrowright s}{\underline{invoke(m, vl)}})\ class(c)\ pgm\langle\!|\mathsf{class}\ c\ \mathsf{extends}\ \_\ \{\_\ ;\langle\!|\mathsf{method}\ m(xl)\ s|\!\rangle_{???}\}|\!\rangle_{???}$
$(\!|invoke(m,vl)|\!\rangle_k\ class(\underset{c'}{\underline{c}})\ pgm\langle\!|\mathsf{class}\ c\ \mathsf{extends}\ c'\ \{\_\ ;\_\}|\!\rangle_{???}$, otherwise (i.e., when c has no method m)
$k(\underset{v \curvearrowright k}{\underline{\mathsf{return}(v) \curvearrowright \_}})\ class(\underset{c}{\underline{\_}})\ env(\underset{\rho}{\underline{\_}})\ (\!|\underset{\cdot}{\underline{(c,\rho,k)}}|\!\rangle_{stack}$

$(\!|\underset{\sigma[getLoc(x, (\_:\rho, c:\eta, \omega'))]}{\underline{x}}|\!\rangle_k\ env(\rho)\ class(c)\ obj(\omega, c:\eta,\omega')\ store(\sigma)$
$(\!|\underset{\sigma[getLoc(x, (\_:\rho, c:\eta, \omega')) +_{Loc} n]}{\underline{x[n]}}|\!\rangle_k\ env(\rho)\ class(c)\ obj(\omega, c:\eta,\omega')\ store(\sigma)$
$(\!|\underset{\cdot}{\underline{x = v}}|\!\rangle_k\ env(\rho)\ class(c)\ obj(\omega,c:\eta,\omega')\ store(\underset{\sigma[v/getLoc(x,(env:\rho, c:\eta,\omega'))]}{\underline{\sigma}})$
$(\!|\underset{\cdot}{\underline{x[n] = v}}|\!\rangle_k\ env(\rho)\ class(c)\ obj(\omega,c:\eta,\omega')\ store(\underset{\sigma[v/getLoc(x,(env:\rho, c:\eta,\omega')) +_{Loc} n]}{\underline{\sigma}})$

$getLoc(x, (\_ : \eta, \omega)) = \eta[x]$ when $\eta[x]$ is defined
$getLoc(x, (\_ : \eta, \omega)) = getLoc(x, \omega)$ when $\eta[x]$ is not defined

D.2 Type Checking SKOOL Dynamically

D.3 Type Checking SKOOL Statically

E K for the Maude User

In this section we discuss two approaches that one can use to incorporate K language definitions in Maude. The first approach makes plain use of Maude and should be easy to translate and use in the context of any other rewrite engine. Also, the first approach is slightly easier to understand than the second one. The second approach makes use of Maude's special reflective capabilities using its provided meta-level; at our knowledge Maude is the only rewrite engine providing such capabilities. In spite of the apparently heavier syntax notation that it involves and its slightly lower performance (when executed) in Maude, the second approach has a series of advantages over the first one, explained in Appendix **??**, which currently make it our preferred approach. Nevertheless, if the language that one wants to define does not include (rather complex) features that involve code generation or reflection, then the first approach is just as good. In fact, the only thing one looses when one uses the first approach for such complex language features is modularity: one needs to change its definition when moving the feature from one language to another, because the syntax to be used in the generated code of the language changes. This non-trivial sort of modularity can be achieved very elegantly in K, thanks to its uniform representation of computations and because of its deliberate decision to disobey the syntax. Both approaches are relatively straightforward and mechanical.

We discuss the two approaches separately, using for concreteness' sake the simple imperative language in Figure **??**. We have developed, for each of the approaches, a little "library", or "prelude", of K-related definitions that turned out to be useful in our experiments. These libraries include definitions for lists, sets, mappings, locations, environments and stores, etc. Before starting a new language definition, one should include the specific prelude file. Both our current prelude files, which may change in the future, are discussed in Appendix **??**; however, one needs not understand all the details of how they are defined in order to use them. All one needs to know is what one can use from them. For example, they both provide a sort `K`, together with sub-sorts `KResult` and `KProper`; the former corresponds to computations that are finished and therefore contain no further computational tasks (such as values, or types, or results of analysis, etc.), while the latter corresponds to proper (and well-formed!) computations that still contain computational tasks to be processed. In both approaches, proper computations can be further "heated", while results are used for cooling and for matching purposes in order to apply the actual, irreversible rewrite rules.

E.1 Using Plain Maude

Step 1 Load the provided prelude file for K definitions using plain Maude, called "`k-plain-prelude.maude`":

```
in k-plain-prelude
```

Step 2 Define the syntax of the desired programming language as an algebraic signature, making sure that you include the module `K-PLAIN`. Here is, for example, the syntax of our simple language:

```
fmod IMP-SYNTAX is including INT + BOOL + NAME + K-PLAIN .
  sorts #Int #Bool #Name .
  op #int : Int -> #Int .
  op #bool : Bool -> #Bool .
  op #name : Name -> #Name .
  subsorts #Int #Bool < K .
  subsort #Name < KProper .
  op _+_ : K K -> KProper [prec 33] .
  op _<=_ : K K -> KProper [prec 37] .
  op _and_ : K K -> KProper [prec 55] .
  op not_ : K -> KProper [prec 53] .
  op skip : -> KProper .
  op _:=_ : K K -> KProper .
  op _;_ : K K -> KProper [prec 100 gather(e E)] .
  op if___ : K K K -> KProper .
  op while__ : K K -> KProper .
  op halt_ : K -> KProper .
--- kept only one ‘‘_;_’’, because the two have the same computational meaning
endfm
```

One should be aware of several requirements and guidelines here:

- Most importantly, *one should use only one language syntactic category, K*. The sort `K` is already provided in "`k-plain-prelude.maude`", so one does not need to redeclare it. In other words, for our language in Figure **??**, we collapse `AExp`, `BExp`, `Stmt` and `Pgm` into only one sort `K`, which will be the only one visible from here on. As already mentioned, the sort `K` comes with an important sub-sort, `KProper`, which, intuitively, stays for those computations that need to be further processed to become results. It is important to declare all the result sorts of language constructs which are not already values as `KProper`. No computation constructs need to be declared as constructs for `KResult` when defining the language syntax. The sort collapsing process described above may certainly appear to be very inconvenient because one can now write programs which are not well formed, such as "$3 + \texttt{true}$". Nevertheless, at this moment one should think of the syntax of the language as a syntax for its *abstract syntax tree (AST)*. In other words, its syntax at this stage is nothing but a list of AST node labels. When defining real languages for research or prototyping purposes, one may want to implement an external parser, using the state of the art parsing technology, which takes as input a program following the desired user-friendly syntax and generates as output an AST following the simplified syntax that we define in Maude. As seen in Section **??**, one can also define type-checkers using the same K technique, in which case one can reject programs which are regarded as inappropriate (for typing or other safety reasons).
- A good practice is to *wrap the built-in sorts imported in the syntax of the language*, as well as those that one may use as an intrinsic part of the semantics later on. For example, we import both integers and booleans in our language. Since we collapsed the different language syntactic categories into just `K`, then, without an appropriate separation, the various imported sorts would also collapse, which may lead to many Maude warnings and errors. For example, one cannot even put together integers and booleans under a common sort in Maude, because it just happens that the two modules have operators sharing the same name and number of arguments. Maude rightfully complains, because one cannot always know from context which operator one refers to. Our convention is to use a wrapper name "`#sort`" for each sort that one wants "protected" from here on. We protect three such sorts in our language syntax: `Int`, `Bool`, and `Name`. Names are also protected in almost all our language definitions, because we are going to manipulate them as part of the semantics (save in environments, states, etc.). For clarity, though not strictly needed, one may define sub-sorts of `K` corresponding to the imported sorts, such as `#Int`, `#Bool`, etc.
- Since there is only one syntactic category, *one may have to either remove or redeclare some original language constructs*. For example, in our original syntax we had two semicolon constructs, *Stmt*; *Stmt* and *Stmt*; *AExp*, one for statements and the other for programs. Fortunately, these two constructs have the same semantics as computations: process the first computation and then the second. Therefore, we only need to add one operation "`_;_ : K K -> KProper`". If the two semicolon constructs had different semantics as computations, then one would have had to define them with different names as computation constructs.

Note that we also added precedence and gathering attributes to some of the language/computation constructs. Those attributes are used by Maude's internal parser and are given exclusively for simplifying the reading of programs by not having to write some obvious parentheses. The lower the precedence the tighter the binding of that operator. Gathering attributes can specify, among other things, left or right associativity of operators; fore example, we defined the sequential composition to be right associative (left associativity would have also worked).

Step 3 Test the syntax by asking Maude to parse several programs. For example, the program below calculates the sum of all the numbers between 1 and 1000:

```
parse (
        #name(n) := #int(1000) ;
        #name(s) := #int(0) ;
```

```
          #name(i) := #name(n) ;
          while (not(#name(i) <= #int(0))) (
            #name(s) := #name(s) + #name(i) ;
            #name(i) := #name(i) + #int(-1)
          ) ;
          #name(s)
  ) .
```

This is a good moment to write up many interesting programs that one would like to eventually run in the defined language, because at this moment one shoots two rabbits with one stone doing it: (1) comprehensively tests the syntax, and (2) prepares a benchmark to test the subsequent semantics. It is very convenient to properly "divide-and-conquer" the task of defining a language between defining its syntax and defining its semantics. Once the syntax is done, one can move on to the semantics without having to worry about syntactic details anymore. In our experience with designing languages using K, we found fewer aspects that were more annoying than having to go back and modify the syntax while testing the semantics, just because we were not able to parse certain desirable programs in order to execute them; each change of syntax will almost unavoidably trigger corresponding changes of semantics. It is therefore very important to get the syntax right once and for all at this stage.

Step 4 Define all the structural computation equations corresponding to the strictness attributes. These equations can be derived automatically, but we assume no particular translation or implementation of K in Maude here; we simply use Maude to write K language definitions and, obviously, Maude is not aware of K's particularities. The structural computation equations for our language are given below:

```
fmod IMP-K-STRICTNESS is including IMP-SYNTAX .
  vars k k1 k2 : K . vars pk pk1 pk2 : KProper .  vars r r1 r2 : KResult .

  ops (_+[]) ([]+_) : K -> K .
  eq pk1 + k2 = pk1 -> [] + k2 .
  eq r1 -> [] + k2 = r1 + k2 .
  eq k1 + pk2 = pk2 -> k1 + [] .
  eq r2 -> k1 + [] = k1 + r2 .

  ops (_<=[]) ([]<=_) : K -> K .
  eq pk1 <= k2 = pk1 -> [] <= k2 .
  eq r1 -> [] <= k2 = r1 <= k2 .
  eq r1 <= pk2 = pk2 -> r1 <= [] .
  eq r2 -> r1 <= [] = r1 <= r2 .

  op []and_ : K -> K .
  eq pk1 and k2 = pk1 -> [] and k2 .
  eq r1 -> [] and k2 = r1 and k2 .

  op not[] : -> K .
  eq not pk = pk -> not [] .
  eq r -> (not []) = not r .

  op _:=[] : K -> K .
  eq  k1 := pk2 = pk2 -> k1 := [] .
  eq r2 -> k1 := [] = k1 := r2 .

  op if[]__ : K K -> K .
  eq if pk k1 k2 = pk -> if [] k1 k2 .
  eq r -> if [] k1 k2 = if r k1 k2 .

  op halt[] : -> K .
  eq halt pk = pk -> halt [] .
  eq r -> halt [] = halt r .
endfm
```

Note that only proper computations are "heated", while only results are "cooled". As mentioned, the sort *KResult* is a built-in sub-sort of K, disjoint from *KProper*. The subsequent semantic definitions are going to tell precisely how results are generated and propagated. Also, note that for each strict argument of each language construct we declared a new operation, whose name was that of the original language construct with an apparent "hole" replacing the strict argument. As mentioned, there is no hole in K, but just an intuitive name convention: that's a result "placeholder" for the scheduled subcomputation. The reader is warned that defining the structural computation equations is perhaps the most boring and error prone part of a K language definition. Implementations of K should generate all these automatically from higher-level and compact strictness information.

Step 5 Define the configuration, if needed. With very few exceptions (one example being the truly concurrent semantics of CCS in Section **??**), a K language definition will include some conveniently chosen configuration which will include a set of configuration items. The prelude already defines the sorts `Config` and `ConfigItem`, as well as an operation `[[_]] : KSet{ConfigItem} -> Config`. The sort `KSet{ConfigItem}` is also defined in the prelude; in fact, a parametric module `KSET` is defined for multi-sets (defined using a binary associative and commutative operator), which provides a parametric sort `KSet{X}`, that in this case we instantiate for configuration items. One is free not to use our constructs for configurations if one prefers one's own way to organize them. Here is a configuration definition for our language:

```
fmod IMP-K-CONFIGURATION is including STATE + IMP-SYNTAX + CONFIG .
  op k : K -> ConfigItem .
  op state : State -> ConfigItem .
  subsort Val < Config KResult .
  op [[_]] : K -> Config .
  var p : K .  var v : Val .  var c : KSet{ConfigItem} .
  eq [[p]] = [[k(p) state(.State)]] .
  eq [[k(v) c]] = v .
endfm
```

One should make sure that the language syntax module is included, as well as other modules necessary for the configuration; in our case, we only need to include the modules `STATE` and `CONFIG`, both provided in the prelude. One may need to define new modules if they are not provided in the prelude. `STATE` is defined as an instance of a parametric module `KMAP`, more precisely as `KMAP{Name,Val}`; therefore, values are also automatically included. At this stage in the language definition values are generic, in the sense that we have no constructors for them yet; those will be added when we define the actual semantics of the language. Since values are intended to be results of initial configurations as well as results of computations, this is a good place to state that fact explicitly with the sub-sort declaration `Val < Config KResult`. To distinguish among the various empty lists or sets for which we generically used a central dot in the K notation, as well as to avoid Maude ambiguous parsing, we suffixed the various "dots" with their corresponding sorts in our prelude (e.g., `.State`, `.K`, etc.) The two equations initiate and terminate the computation process, respectively.

Step 6 Define the actual semantics. The very first step is to "initialize the semantics" by establishing the connection between "terminals" in the syntax, i.e., leaves in the abstract syntax tree, and computations. Recall that our convention was to wrap those terminals with corresponding operator names, such as `#int`, `#bool`, and `#name`. Our next convention is to use similar wrapper operator names, but dropping the `#` character, for terminals into the computational world; more precisely, we use wrapper operations like "`int : Int -> Val`", "`bool : Bool -> Val`", and "`name : Name -> KProper`". Recall that values are also result computations in `KResult`. The reason for which we declare the first two value wrappers as constructors for values as opposed to constructors for K results is because we want to allow them to be used as values, in particular to be stored in our state structure, for example (in this particular language only integers can be stored, but in general one can store any type of values in a language). The third wrapper is a construct for `KProper`, because names still need to be heated in order to be computed. Expected equations of the form "`#int(i)=int(i)`" establish the desired "bottom-AST" relationship between syntax and computation semantics. This step was not necessary in our "paper" K definitions because there we did not need to wrap AST-terminals at all on paper. The Maude module below shows the complete K semantics of our simple language:

```
mod IMP-K-SEMANTICS is including IMP-K-CONFIGURATION + IMP-K-STRICTNESS .
  vars k1 k2 rest : K . var x : Name . var v : Val . vars i i1 i2 : Int . var b : Bool . var sigma : State .

  op int : Int -> Val .  op bool : Bool -> Val .  op name : Name -> KProper .
  eq #int(i) = int(i) .  eq #bool(b) = bool(b) .  eq #name(x) = name(x) .

  rl k(name(x) -> rest) state(sigma) => k(sigma[x] -> rest) state(sigma) .
  rl int(i1) + int(i2) => int(i1 + i2) .
  rl int(i1) <= int(i2) => bool(i1 <= i2) .
  rl bool(true) and k2 => k2 .
  rl bool(false) and k2 => bool(false) .
  rl not bool(b) => bool(not b) .
  rl k(name(x) := v -> rest) state(sigma) => k(rest) state(sigma[x <- v]) .
  rl k1 ; k2 => k1 -> k2 .
  rl skip => .K .
  rl if bool(true) k1 k2 => k1 .
  rl if bool(false) k1 k2 => k2 .
  eq k(while k1 k2 -> rest) => k(if k1 (k2 -> while k1 k2) .K -> rest) .
  rl k(halt int(i) -> rest) => k(int(i)) .
endm
```

Note that rules use the rule "`rl left => right .`" Maude syntax, denoting the fact that they are, semantically speaking, irreversible rewrite rules and not equations. The type of the module is `mod` instead of `fmod`, saying that it may contain rewrite rules and thus is not a functional module anymore. Since this particular language happens to be deterministic, nothing would be lost if all the rules above were equations. However, since we also want to capture the intended computational granularity of each language construct as part of the semantics, in particular that `halt` stops the entire program in *one step*, we prefer to keep them rules.

Step 7 Test the semantics using several programs:

```
rewrite [[
        #name(n) := #int(1000) ;
        #name(s) := #int(0) ;
        #name(i) := #name(n) ;
        while (not(#name(i) <= #int(0))) (
          #name(s) := #name(s) + #name(i) ;
          #name(i) := #name(i) + #int(-1)
        ) ;
        #name(s)
]] .
```

The Maude output for the above reduction, on a 1GH/1GB tablet PC running Windows XP, is:

```
rewrites: 33048 in ... cpu (371ms real) (~ rewrites/second)
result Val: int(500500)
```

Fortunately, when writing the semantic rules above one needs not worry much about forgetting any particular rule. If the benchmarks of collected programs cover all the language constructs, which should always be the case, then one will easily notice the missing rule(s), because the normal-form computation returned by Maude will have the corresponding unreduced language construct at its top. For example, if one omits the first rule, then reducing the term above will give a normal form of the form "`[[k(name(x) -> ...) state(...)]]`", which almost suggests what was forgotten.

E.2 Using Maude's Meta-Level

Step 1 Load the provided K prelude file for definitions using Maude' meta-level capabilities:

```
in k-meta-prelude.maude
```

Step 2 Define the syntax of the desired programming language as an algebraic signature. Unlike in the previous approach, there is no need to include any K module. K will "swallow" the entire language syntax later at the meta-level. Therefore, we define the language syntax in isolation, as if there was no K:

```
fmod IMP-SYNTAX is including INT + BOOL + NAME .
  sorts #Int #Bool #Name AExp BExp Stmt Pgm .
  op #int : Int -> #Int .
  op #bool : Bool -> #Bool .
  op #name : Name -> #Name .
  subsorts #Name #Int < AExp .
  op _+_ : AExp AExp -> AExp [prec 33] .
  subsort #Bool < BExp .
  op _<=_ : AExp AExp -> BExp [prec 37] .
  op _and_ : BExp BExp -> BExp [prec 55] .
  op not_ : BExp -> BExp [prec 53] .
  op skip : -> Stmt .
  op _:=_ : #Name AExp -> Stmt .
  op _;_ : Stmt Stmt -> Stmt [prec 100 gather(e E)] .
  op if___ : BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .
  op halt_ : AExp -> Stmt .
  op _;_ : Stmt AExp -> Pgm [prec 110] .
endfm
```

As before, one should be aware of several requirements and guidelines here:

- There is no need to collapse the syntactic categories into only one like before, because that will be done automatically at the meta-level when we define the semantics. Even though Maude's parser can reject many ill-formed programs now using its internal parser, one should not expect Maude's parser to parse arbitrarily complex language syntax. Like in the plain Maude approach, if a language designer wants a syntax more complex than what Maude can parse, then she should implement an external parser for the desired syntax, using state of the art parsing technology, to translate it into one that Maude can parse.
- As before, one should also wrap the built-in sorts imported in the syntax of the language, as well as those that one may be used as an intrinsic part of the semantics later on. That will make it easier to recognize these important elements after the program will be lifted at the meta-level. We follow the same wrapper naming conventions like in the plain Maude approach.
- Once the language syntax will be lifted at the meta-level, the operator declarations in the syntax module will be forgotten and their names will be used as node labels in the meta-term (as plain quited identifiers). The sort information of all the language constructs will be discarded; consequently, it is important to rename constructs with the same name but with different sorts in case they have different computational semantics. In our case, we do not need to rename the semicolon constructs *Stmt*; *Stmt* and *Stmt*; *AExp*, because they have the same semantics as computations.

Step 3 Like in the previous approach using plain Maude, test the syntax by parsing several programs. This time, the result sort of well-formed programs will be `Pgm` instead of `KProper`.

Step 4 Define all the structural computation equations corresponding to the strictness attributes. As before, these equations can and should be derived automatically in implementations of K in Maude, but we assume no particular translation or implementation of K in Maude. Some discussion on our meta-term representation is necessary before we can show the structural computation equations for our simple language. The provided module `K-META` imports the module `K-PLAIN` and "hooks" the Maude internal representation for terms into computations. More precisely, computations can now be constructed with the usual task sequentialization operator `_->_` (corresponding to $\curvearrowright$), with *label constants* of sort `KLabel`, and with terms of the form $L(K_1, K_2, ..., K_n)$, where L is a label and K_1, K_2, ..., K_n are subcomputations. We prefer constants instead of labeled terms with an empty list of children to stay consistent with Maude's meta-term representation conventions. Labels include all the quoted identifiers and can be added more constructs if needed. One built-in label construct, `f : String -> Freezer`, is already provided, the one for "freezers" (`Freezer` is a sub-sort of `Label`) . A freezer label has therefore the form `f("any sequence of characters")`; the string will most likely include holes, written `[]`. To ease reading of meta-terms with holes, we renamed the Maude's square brackets enclosing meta-subterms to normal parentheses. Here are the structural computation equations for our language:

```
fmod IMP-K-STRICTNESS is including K-META .
  vars k k1 k2 : K .  vars r r1 r2 : KResult . vars pk pk1 pk2 : KProper .

 eq ’_+_(pk1,k2) = pk1 -> f(’‘[‘]+_)(k2) .
 eq r1 -> f(’‘[‘]+_)(k2) = ’_+_(r1,k2) .
 eq ’_+_(k1,pk2) = pk2 -> f(’_+‘[‘])(k1) .
 eq r2 -> f(’_+‘[‘])(k1) = ’_+_(k1,r2) .

 eq ’_<=_(pk1,k2) = pk1 -> f(’‘[‘]<=_)(k2) .
 eq r1 -> f(’‘[‘]<=_)(k2) = ’_<=_(r1,k2) .
 eq ’_<=_(r1,pk2) = pk2 -> f(’_<=‘[‘])(r1) .
 eq r2 -> f(’_<=‘[‘])(r1) = ’_<=_(r1,r2) .

 eq ’_and_(pk1,k2) = pk1 -> f(’‘[‘]and_)(k2) .
 eq r1 -> f(’‘[‘]and_)(k2) = ’_and_(r1,k2) .

 eq ’not_(pk) = pk -> f(’not‘[‘]) .
 eq r -> f(’not‘[‘]) = ’not_(r) .

 eq ’_:=_(k1,pk2) = pk2 -> f(’_:=‘[‘])(k1) .
 eq r2 -> f(’_:=‘[‘])(k1) = ’_:=_(k1,r2) .

 eq ’if___(pk,k1,k2) = pk -> f(’if‘[‘]__)(k1,k2) .
 eq r -> f(’if‘[‘]__)(k1,k2) = ’if___(r,k1,k2) .

 eq ’halt_(pk) = pk -> f(’halt‘[‘]) .
 eq r -> f(’halt‘[‘]) = ’halt_(r) .
endfm
```

The `KResult` and `KProper` computations have precisely the same meaning as in the plain Maude approach. We use the `Qid` labels for K nodes corresponding to the original language syntax, and frozen labels for auxiliary freezers. Note that the module above did not need to import the language syntax, because it only refers to quoted identifier and frozen string labels.

Step 5 Define the configuration, if needed:

```
fmod IMP-K-CONFIGURATION is including STATE + K-META + IMP-SYNTAX + CONFIG .
  op k : K -> ConfigItem .
  op state : State -> ConfigItem .
  subsort Val < Config KResult .
  op [[_]] : Pgm -> Config .
  var p : Pgm .  var v : Val .  var c : KSet{ConfigItem} .
  eq [[p]] = [[k(mkK(p)) state(.State)]] .
  eq [[k(v) c]] = v .
endfm
```

The only difference between the module above and its corresponding one in the plain Maude approach is that the evaluation operator takes a program now instead of a computation, but lifts it with the provided operator `mkK` into a `KProper` computation when placing it into the configuration.

Step 6 Define the actual semantics. Like before, we should first "initialize the semantics" by establishing the connection between "terminals" in the syntax, i.e., meta-subterms with `Qid` labels starting with `'#`, and computations. One easy way to achieve that is to use `downK`, also provided as part of the prelude, as shown in the module below, which completes the semantics of our language using the K-meta approach:

```
mod IMP-K-SEMANTICS is including IMP-K-CONFIGURATION + IMP-K-STRICTNESS .
  vars k k1 k2 rest : K . var x : Name . var v : Val . vars i i1 i2 : Int . var b : Bool . var sigma : State .

  op int : Int -> Val .  op bool : Bool -> Val .  op name : Name -> KProper .
  eq '#int(k) = int(downTerm(k,errInt)) .     op errInt : -> [Int] .
  eq '#bool(k) = bool(downTerm(k,errBool)) .  op errBool : -> [Bool] .
  eq '#name(k) = name(downTerm(k,errName)) .  op errName : -> [Name] .

  rl k(name(x) -> rest) state(sigma) => k(sigma[x] -> rest) state(sigma) .
  rl '_+_(int(i1),int(i2)) => int(i1 + i2) .
  rl '_<=_(int(i1),int(i2)) => bool(i1 <= i2) .
  rl '_and_(bool(true),k2) => k2 .
  rl '_and_(bool(false),k2) => bool(false) .
  rl 'not_(bool(b)) => bool(not b) .
  rl k('_:=_(name(x),v) -> rest) state(sigma) => k(rest) state(sigma[x <- v]) .
  rl '_;_(k1,k2) => k1 -> k2 .
  rl 'skip.Stmt => .K .
  rl 'if___(bool(true),k1,k2) => k1 .
  rl 'if___(bool(false),k1,k2) => k2 .
  eq (k('while__(k1,k2) -> rest)).ConfigItem = k('if___(k1, (k2 -> 'while__(k1,k2)), .K) -> rest) .
  rl (k('halt_(int(i)) -> rest)).ConfigItem => k(int(i)) .
endm
```

Step 7 Test the semantics using several programs.

```
mod IMP-K-RUN is
  including IMP-PROGRAMS + IMP-K-SEMANTICS .
endm

rew [[pSum]] .
```

E.3 Comparing the Plain and Meta Approaches

Note that the one using the meta-level has several advantages: - it works directly with the syntax of the PL, so one does not need to first collapse everything into computations - it is slightly more compact, at least in number of lines and of operators defined. - it is more modular, in that one can add new features like self-generation of code, etc., without worrying of the actual language syntax.

The only advantage the core-level one has is that its rules look more readable.

F The K Prelude

$\langle\!| v \curvearrowright \underset{\cdot}{\underline{restore(\rho)}} |\!\rangle_k \;\; env(\underset{\rho}{\underline{\_}})$

$strict(kl_1, k, kl_2) \rightleftharpoons k \curvearrowright strict(kl_1, \square, kl_2)$

$strict(rl) = results(rl)$

$\langle\!| \underset{\cdot}{\underline{results(vl) \curvearrowright bindTo(xl)}} |\!\rangle_k \;\; env(\underset{\rho[ll'/xl]}{\underline{\rho}}) \;\; store(\underset{\sigma[vl/ll']}{\underline{\sigma}}) \;\; nextLoc(\underset{l'}{\underline{l}})$ where l' is $l +_{Loc} |xl|$, and ll' is $l \ldots (l' +_{Loc} -1)$

$\langle\!| \underset{\cdot}{\underline{bindTo(xl)}} |\!\rangle_k \;\; env(\underset{\rho[ll'/xl]}{\underline{\rho}}) \;\; nextLoc(\underset{l'}{\underline{l}})$ where l' is $l +_{Loc} |xl|$, and ll' is $l \ldots (l' +_{Loc} -1)$

$\langle\!| \underset{\cdot}{\underline{results(vl) \curvearrowright writeTo(xl)}} |\!\rangle_k \;\; env(\rho) \;\; store(\underset{\sigma[vl/\rho[xl]]}{\underline{\sigma}})$

The following definitions of the auxiliary computation items *restore*, *strict*, *bindTo* and *writeTo* are so useful in language definitions, that, in our Maude implementation of K, they are part of the prelude file which is included in all K definitions:

$\langle\!| v \curvearrowright \underset{\cdot}{\underline{restore(\rho)}} |\!\rangle_k \;\; env(\underset{\rho}{\underline{\_}})$

$strict(kl_1, k, kl_2) \rightleftharpoons k \curvearrowright strict(kl_1, \square, kl_2)$

$strict(rl) = results(rl)$

$\langle\!| \underset{\cdot}{\underline{results(vl) \curvearrowright bindTo(xl)}} |\!\rangle_k \;\; env(\underset{\rho[ll'/xl]}{\underline{\rho}}) \;\; store(\underset{\sigma[vl/ll']}{\underline{\sigma}}) \;\; nextLoc(\underset{l'}{\underline{l}})$ where l' is $l +_{Loc} |xl|$, and ll' is $l \ldots (l' +_{Loc} -1)$

$\langle\!| \underset{\cdot}{\underline{bindTo(xl)}} |\!\rangle_k \;\; env(\underset{\rho[ll'/xl]}{\underline{\rho}}) \;\; nextLoc(\underset{l'}{\underline{l}})$ where l' is $l +_{Loc} |xl|$, and ll' is $l \ldots (l' +_{Loc} -1)$

$\langle\!| \underset{\cdot}{\underline{results(vl) \curvearrowright writeTo(xl)}} |\!\rangle_k \;\; env(\rho) \;\; store(\underset{\sigma[vl/\rho[xl]]}{\underline{\sigma}})$