K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation

—*Version 1*—

Grigore Roşu[*]
Department of Computer Science,
University of Illinois at Urbana-Champaign

Abstract

K is an algebraic framework for defining programming languages. It consists of a technique and of a specialized and highly optimized notation. The *K-technique*, which can be best explained in terms of rewriting modulo equations or in terms of rewriting logic, is based on a first-order representation of *continuations* with intensive use of *matching modulo* associativity, commutativity and identity. The *K-notation* consists of a series of high-level conventions that make the programming language definitions intuitive, easy to understand, to read and to teach, compact, modular and scalable. One important notational convention is based on *context transformers*, allowing one to automatically synthesize concrete rewrite rules from more abstract definitions once the concrete structure of the state is provided, by "completing" the contexts in which the rules should apply. The K framework is introduced by defining FUN, a concurrent higher-order programming language with parametric exceptions. A rewrite logic definition of a programming language can be executed on rewrite engines, thus providing an interpreter for the language for free, but also gives an initial model semantics, amenable to formal analysis such as model checking and inductive theorem proving. Rewrite logic definitions in K can lead to automatic, correct-by-construction generation of interpreters, compilers and analysis tools.

Note to readers: *The material presented in this report serves as a basis for programming language design and semantics classes and for several research projects. This report aims at giving a global snapshot of this rapidly evolving domain. Consequently, this work will be published on a version by version basis, each including and improving the previous ones, probably over a period of several years. This version already contains the desired structure of the final version, but not all sections are filled in yet. In this first version I put more emphasis on introducing the K framework and on how to use it. Here I focus less on related work, inductive verification and implementation; these will be approached in next versions of this work. My plan is to eventually transform this material into a book, so your suggestions and criticisms are welcome.*

[*] Supported by joint NSF grants CCF-0234524, CCF-0448501, and CNS-0509321.

Contents

1 Introduction

Appropriate frameworks for design and analysis of programming languages can not only help in understanding and teaching existing programming languages and paradigms, but can significantly facilitate our efforts and stimulate our desire to define and experiment with novel programming languages and programming language features, as well as programming paradigms or combinations of paradigms. But what makes a programming language definitional framework appropriate? We believe that an *ideal* such framework should satisfy at least some core requirements; the following are a few requirements that guided us in our quest for such a language definitional framework:

1. It should be *generic*, that is, not tied to any particular programming 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, 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.

2. It should be *semantics-based*, that is, based on formal definitions of languages rather than on ad-hoc implementations of interpreters/compilers or program analysis tools. Semantics is crucial to such an ideal definitional framework because, without a formal semantics of a language, the problems of program analysis and interpreter or compiler correctness are meaningless. One could use ad-hoc implementations to find errors in programs or compilers, but never to *prove* their correctness. Ideally, we would like to have *one* definition of a language to serve *all* purposes, including parsing, interpretation, compilation, as well as formal analysis and verification of programs, such as static analysis, theorem proving and model checking. Having to annotate or slightly change/augment a language definition for certain purposes is acceptable and unavoidable; for example, certain domain-specific analyses may need domain information which is not present in the language definition; or, to more effectively model-check a concurrent program, one may impose abstractions that cannot be inferred automatically from the formal semantics of the language. However, having to develop an entirely new language definition for each purpose should be unacceptable in an ideal definitional framework.

3. It should be *executable*. There is not much one can do about the correctness of a definition, except to accumulate confidence in it. In the case of a programming language, one can accumulate confidence by proving desired properties of the language (but how many, when to stop?), or by proving equivalence to other formal definitions of the same language if they exist (but what if these definitions all have the same error?), or most practical of all, by having the possibility to execute programs *directly* using the semantic definition of the language. In our experience, executing hundreds of programs exercising various features of a language helps to not only find and fix errors in that language definition, but also stimulates the desire to

experiment with new features. A *computational logic framework* with efficient executability and a spectrum of meta-tools can serve as a basis to define executable formal semantics of languages, and also as a basis to develop generic formal analysis techniques and tools.

4. It should be able to naturally support *concurrency*. Due to the strong trend in parallelizing computing architectures for increased performance, probably most future programming languages will be concurrent. To properly define and reason about concurrent languages, the semantics of the underlying definitional framework should be inherently concurrent, rather than artificially graft support for concurrency on an essentially sequential paradigm, for example, by defining or simulating a process/thread scheduler. A semantics for *true concurrency* would be preferred to one based on *interleavings*, because executions of concurrent programs on parallel architectures are *not* interleaved.

5. It should be *modular*, to facilitate reuse of language features. In this context, modularity of a programming language definitional framework means more than just allowing grouping language feature definitions in modules. What it means is the ability to add or remove language features without having to modify any definitions of other, unrelated features. For example, if one adds parametric exceptions to one's language, then one should just include the corresponding module and change no other definition of any other language feature. In a modular framework, one can therefore define languages by adding features in a "plug-and-play" manner. A typical structural operational semantics (SOS) [22] definition of a language lacks modularity because one needs to "update" all the SOS rules whenever the structure of the state, or configuration, changes (like in the case of adding support for exceptions).

6. It should allow one to define any desired level of *computation granularity*, to capture the various degrees of abstraction of computation encountered in different programming languages. For example, some languages provide references as first class value citizens (e.g., C and C++); in order to get the value v that a reference variable x points to, one wants to perform *two semantic steps* in such languages: first get the value l of x, which is a location, and then read the value v at location l. However, other languages (e.g., Java) prefer not to make references visible to programmers, but only to use them internally as an efficient means to refer to complex data-structures; in such languages, one thinks of grabbing the (value stored in a) data-structure in *one semantic step*, because its location is not visible. An ideal framework should allow one to flexibly define any of these computation granularities. Another example would be the definition of functions with multiple arguments: an ideal framework should *not* enforce one to eliminate multiple arguments by currying; from a programming language design perspective, the fact that the underlying framework supports only one-argument functions or abstractions is regarded as a limitation of the framework. Other examples are mentioned later in the paper. Having the possibility to define different computation granularities for different languages gives a language designer several important benefits: better understanding of the language by not having to bother with low level implementation-like or "encoding" details, more freedom in how to implement the language, more appropriate abstraction of the language for formal analysis of programs, such as theorem proving and model checking, etc. From a computation granularity perspective, an extreme worst case of a definitional framework would provide a fixed computation granularity for all programming languages, for example one based on encodings of language features in λ-calculus or on Turing machines.

There are additional desirable, yet more subjective and thus harder to quantify, requirements of an ideal language definitional framework, including: it should be simple, easy to understand and teach; it should have good data representation capabilities; it should scale well, to apply to arbitrarily large languages; it should allow proofs of theorems about programming languages that are easy to comprehend; etc. The six requirements above are nevertheless ambitious. Moreover, there are subtle tensions among them, making it hard, if not impossible, to find an ideal language definitional framework. 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 most, of these ideal features (we discuss several such frameworks and their limitations in Section 3). Others may argue that their favorite framework has some of the properties above, the "important ones", declaring the other properties either "not interesting" or "something else". For example, one may say that what is important in one's framework is to get a dynamic semantics of a language, but its (model-based) denotational semantics, proving properties about programs, model checking, etc., are "something else".

Following up recent work in rewriting logic semantics [19, 18], in this paper we argue that *rewriting logic* [17] can be a reasonable starting point towards the development of such an ideal framework. We call it a "starting point" because we believe that without appropriate language-specific front-ends (notations, conventions, etc.) and without appropriate definitional techniques, rewriting logic is simply too general, like the machine code of a processor or a Turing machine or a λ-calculus. In a nutshell, one can think of rewriting logic as a framework that gives complete semantics (that is, models that make the expected rewrite relation, or "deduction", complete) to the otherwise usual and standard *term rewriting modulo equations.* If one is specifically *not* interested in the model-theoretical semantic dimension of the proposed framework, but only in its operational (including proof theoretical, model checking and, in general, formal verification) aspects, then one can safely think of it as a framework to define programming languages as standard term rewrite systems modulo equations. The semantic counterpart is achieved at no additional cost (neither conceptual not notational), by just regarding rewrite systems as rewrite logic specifications.

A rewrite logic theory consists of a set of uninterpreted operations constrained equationally, together with a set of rewrite rules meant to define the concurrent evolution of the defined system. The distinction between equations and rewrite rules is only semantic. They are both executed as rewrite rules $l \to r$ by rewrite engines, following the simple, uniform and parallelizable *match-and-apply* principle of term rewriting: find a subterm matching l, say with a substitution θ, then replace it by $\theta(r)$. Therefore, if one is interested in just a dynamic semantics of a language, then, with few exceptions, one needs to make no distinction between equations and rewrite rules; the exceptions are some special equations, such as associativity and commutativity, enabling specialized algorithms in rewrite engines, which are used for non-computational purposes, namely for keeping structures, such as states, in convenient canonical forms.

Rewriting logic admits an *initial model semantics*, where equations form equivalence classes on terms and rewrite rules define transitions between such equivalence classes. Operationally, rewrite rules can be applied concurrently, thus making rewrite logic a very simple, generic and universal framework for concurrency; indeed, many other theoretical frameworks for concurrency, including π-calculus, process algebra, actors, etc., have been seamlessly defined in rewriting logic [16]. In our context of programming languages, a language definition is a rewrite logic theory in which (at least) the concurrent features of the language are defined using rewrite rules. A program together with its initial state are given as an uninterpreted term, whose denotation in the initial model is its corresponding transition system. Depending on the desired type of analysis, one can, using existing tool support, generate anywhere from one path in that transition system (e.g., when "executing" the program) to all paths (e.g., for model checking).

One must, nevertheless, treat the simplicity and generality of rewriting logic with caution; "general" and "universal" need not necessarily mean "better" or "easier to use", for the same reason that machine code is not better or easier to use than higher level programming languages that translate into it. In our context of defining programming languages in rewriting logic, the right questions to ask are whether rewriting logic provides a natural framework for this task or not, and whether we get any benefit from using it. In spite of its simplicity and generality, rewriting logic does *not* give us any immediate recipe for *how* to define languages as rewrite logic theories. Appropriate *definitional techniques* and *methodologies* are necessary in order to make rewriting logic an effective computational framework for programming language definition and formal analysis.

In this paper we propose K, a domain-specific language front-end to rewriting logic that allows for compact, modular, executable, expressive and easy to understand and change semantic definitions of programming languages, concurrent or not. K could be explained and presented orthogonally to rewriting logic, as a standalone language definitional framework (same as, e.g., SOS or reduction semantics), but we prefer to regard it as a language-specific front-end to rewrite logic to reflect from the very beginning the fact that it inherits all the good properties and techniques of rewriting logic, a well established formalism with many uses and powerful tool support.

As discussed in [19, 18, 4] and in Section 3 of this paper, other definitional frameworks, such as SOS [22], MSOS [21] and reduction semantics [8], can also be easily translated into rewriting logic. More precisely, for a particular language formalized using any of these formalisms, say F, one can devise a rewrite logic specification, say R_{F}, which is *precisely the intended original language definition* F*, not an artificial encoding of it*; in other words, there is a one-to-one correspondence between derivations using F and rewrite logic derivations using R_{F}, obviously modulo a different but minor and ultimately irrelevant syntactic notation. This way, R_{F} has all the properties, good or bad, of F. However, in order to achieve such a bijective correspondence and thus the faithful translation of F into rewriting logic, one typically has to significantly restrain the strength of rewriting logic, reducing it to a limited computational framework, just like F. For example, the faithful translation of SOS definitions requires one conditional rewrite rule per SOS rule, but the resulting rewrite logic definition can apply rewrites only at the top, just like SOS, thus enforcing always an interleaving semantics for concurrent languages, just like SOS. Therefore, the fact that these formalisms translate into rewriting logic does *not* necessarily mean that they inherit all the good properties of rewriting logic. However, K *extends* rewriting logic with features that are meaningful for programming language formal definitions but which can be translated automatically back into rewriting logic, so K *is* rewriting logic.

An Overview of K

To give the reader an early feel for how K works, we next define a very simple untyped language, that we call λ_{K}, including booleans and integers together with the usual operations on them, λ-expressions and λ-application, conditionals, references and assignments, and a halt statement. To emphasize the modularity aspect and the strength of *context transformers*, we then show how one can extend λ_{K} with threads. Appendix D shows a translation of the K definition of sequential λ_{K} below into Maude, while Appendix E a translation of its concurrent extension with threads.

Consider the following syntax of non-concurrent λ_{K}:

$$\begin{array}{rcl}
\mathit{Var} & ::= & \text{identifier} \\
\mathit{Bool} & ::= & \text{assumed defined, together with basic bool operations } \mathsf{not}_{\mathit{Bool}}\_ : \mathit{Bool} \to \mathit{Bool}, \text{etc.} \\
\mathit{Int} & ::= & \text{assumed, together with } \_ +_{\mathit{Int}} \_ : \mathit{Int} \times \mathit{Int} \to \mathit{Int},\ \_ <_{\mathit{Int}} \_ \ :\ \mathit{Int} \times \mathit{Int} \to \mathit{Bool}, \text{etc.} \\
\mathit{Exp} & ::= & \mathit{Var} \mid \mathit{Bool} \mid \mathit{Int} \mid \mathsf{not}\ \mathit{Exp} \mid \mathit{Exp} + \mathit{Exp} \mid \mathit{Exp} < \mathit{Exp} \mid ... \\
 & & \mid \lambda\, \mathit{VarList}^{[,]}.\mathit{Exp} \mid \mathit{Exp}\ \mathit{ExpList}^{[,]} \\
 & & \mid \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Exp}\ \mathsf{else}\ \mathit{Exp} \\
 & & \mid \mathsf{ref}\ \mathit{Exp} \mid\ *\ \mathit{Exp} \mid \mathit{Exp} := \mathit{Exp} \\
 & & \mid \mathsf{halt}\ \mathit{Exp}
\end{array}$$

Boolean and integer expressions come with standard operations, which are indexed for clarity; note that we use the infix notation for some of these operations. Expressions of λ_{K} extend variables, booleans, integers, as well as all the "builtin" operations. The λ-abstraction and λ-application are standard, except that we assume by default that λ-abstractions take any number of arguments; here, $\mathit{VarList}^{[,]}$ and $\mathit{ExpList}^{[,]}$ stay for comma separated lists of variables and expressions, respectively. We deliberately decided *not* to follow the standard approach in which λ-abstractions are defined with only one argument and then multiple arguments are eliminated via currying, because that would change the *granularity level* of the language definition (see requirement number 6 above); additionally, most language implementations treat multiple arguments of functions together, as a block. In our view, from a language definitional and design perspective, a framework imposing such changes of granularity in a language definition just for the purpose of "reducing every language feature to a basic set of well-chosen constructs" is rather limited and falls into the same category with a framework translating any language construct into a sequence of Turing machine operations. We will later see that rewriting logic, and implicitly K, allow us to tune the granularity of computation also via *equational abstraction*: if certain rules are not intended to generate computation steps in the semantics of a language, then we make them equations; the more equations versus rules in a language definition, the fewer computational steps (and the larger the equivalence classes of terms/expressions/programs in the initial model of that language's definition).

For simplicity, we here assume a call-by-value evaluation strategy in λ_{K}; the other evaluation strategies for languages defined in K present no difficulty and are taught on a regular basis to undergraduate students [24]. The conditional expression expects its first argument to evaluate to a boolean and then, depending on its truth value, evaluates to either its second or its third expression argument; note that for the conditional we used the "mix-fix" syntactic notation, rather then a prefix one. The expression $\mathsf{ref}\ E$ evaluates E to some value, stores it at some location or *reference*, and then returns that reference as a result of its evaluation; therefore, in λ_{K} we (semantically) assume that references are first-class values in the language even though the programmer cannot use them in programs explicitly, since the syntax of λ_{K} does not define references (it actually does not define values, either). The expression $*\ R$, called *dereferencing* of R, evaluates R to a reference and then returns the value stored at that reference. The expression $R := E$, called *assignment*, first evaluates R to an (existing) reference r and E to a value v, and then writes v at r; the value v is also the result of the evaluation of the assignment. Finally, the expression $\mathsf{halt}\ E$ evaluates E to some value v and then aborts the computation and returns v as a result.

We next define the usual let bindings and sequential composition as *syntactic sugar* over λ_{K}, that is, $\mathsf{let}\ X = E\ \mathsf{in}\ E'$ is $(\lambda X.E')E$, $\mathsf{let}\ F(X) = E\ \mathsf{in}\ E'$ is $(\lambda F.E')(\lambda X.E)$, $\mathsf{let}\ F(X,Y) = E\ \mathsf{in}\ E'$ is $(\lambda F.E')(\lambda X,Y.E)$, etc., and $E;E'$ is $(\lambda D.E')E$, where D is a fresh "dummy" variable. If these constructs were intended to be part of the language, then these definitions are clearly *not* suitable from a computational granularity point of view, because they change the intended granularity of these language constructs. In Section 5, we show how statements like these and many others can be defined *directly* (without translating them to other constructs), in the context of the more complex FUN language. For now, we can regard them as "notational conventions" for the more complex, equivalent expressions. With these conventions, the following is a λ_{K} expression that calculates factorial of n (here, n is regarded as a "macro"; also, note that this code cannot be used as part of a function to return the result of the factorial, because halt terminates the program — one could do it if one defines parametric exceptions as we do later in the paper, and then throw an exception instead of halt):

$$\begin{array}{l}
\mathsf{let}\ r = \mathsf{ref}\ n \\
\mathsf{in}\ \mathsf{let}\ g(m,h) =\ \mathsf{if}\ m > 1\ \mathsf{then}\ (r := (*r) * m;\ h(m-1,h))\ \mathsf{else}\ \mathsf{halt}\ (*r) \\
\quad \mathsf{in}\ g(n-1,g)
\end{array}$$

While λ_{K} is clearly a very simple toy language, we believe that it contains some canonical features that any ideal language definitional framework should be able to support naturally. Indeed, if a framework cannot define λ-expressions naturally than that framework is inappropriate to define most functional languages and most likely many other non-trivial languages. The conditional statement has been chosen for two reasons: first, most languages have conditional statements as a means to allow various control flows in programs, and, second, conditionals have an interesting evaluation strategy, namely strict in the first argument and lazy in the second and third; therefore, a naive use of a purely "lazy" or a purely "strict" definitional framework may lead to inconvenient encodings of conditionals. References are either directly or indirectly part of several mainstream languages, so they must present no difficulty in any language definitional framework. Finally, halt is one of the simplest control-intensive language constructs; if a language definitional framework cannot define halt naturally, then it most likely cannot define any control-intensive statements, including break and continue of loops, exceptions, and call/cc.

On the other hand, if a language definitional framework can define the λ_{K} language above easily, then it most likely can define many other programming language constructs of interest. Our purpose for introducing λ_{K} here is not to highlight specific features that languages could or should have, but instead to highlight that a language definitional framework should be flexible enough to support a rapid, yet formal, investigation of language features, lowering the boundary between new ideas and executable systems for trying these ideas. An important language feature that we have deliberately *not* added yet to λ_{K} is concurrency; that is because many existing frameworks were designed for sequential languages and one may therefore argue that a comparison of those with K on a concurrent language would not be fair. However, we will add threads to λ_{K} shortly and show how they can be given a semantics in K in a straightforward manner. Concurrent λ_{K} shows most of the subtleties of our framework.

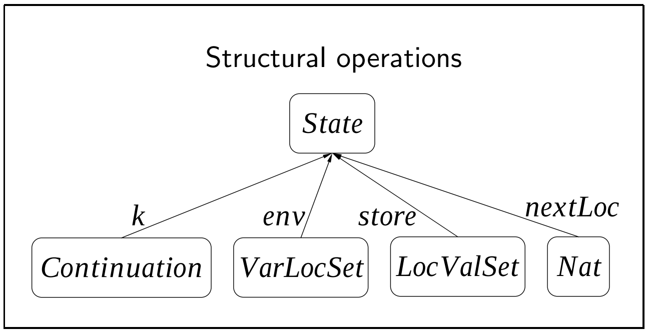
The language definitional framework K consists of two important components:

- The *K-notation*, which can be regarded as a programming language specific *front-end* to rewriting logic, allows users to focus fully on the actual semantics of language features, rather than on distracting details or artifacts of rewriting logic, such as complete sort and operation declarations even though these can be trivially inferred from context, or adding conceptually unrelated state context just for the purpose of well formedness of the rewrite logic theory defining the feature under consideration, even though some partially well formed theory could be completed in a unique way entirely automatically. Besides clarity of definitions of programming language concepts, an additional driving force underlying the design of the K-notation was compactness of definitions, which is closely related to modularity of programming language definitions: any unnecessary piece of information (including sort and operation declarations) that we mention in a language feature definition may work against us when using that feature in a different context.

- The *K-technique* is a technique to define programming languages algebraically, based on a first-order representation of *continuations*. Recall that a continuation encodes the *remaining computation* [23, 26]. Specifically, a continuation is a structure that comes with an implicit or explicit evaluation mechanism such that, when passed a value (thought of as the value of some local computation), it "knows" how to finish the entire computation. Since they provide convenient, explicit representations of computation flows as ordinary data, continuations are useful especially in the context of defining control-intensive statements, such as exceptions. Note, though, that continuations typically encode only the computation steps, not the store; therefore, one still needs to refer to external data during their evaluation. Traditionally, continuations are encoded as higher-order functions, which is very convenient in the presence of higher-order languages, because one uses the *implicit* evaluation mechanism of these languages to evaluate continuations as well. Our K-technique is, however, *not* based on encodings of continuations as higher-order functions. That is partly because our underlying algebraic infrastructure is *not* higher-order (though, as we are showing here, one can define higher-order functions); we can more easily give a different encoding of continuations, namely one based on lists of tasks, together with an *explicit* stack-like evaluation mechanism.

We introduce the K-notation and the K-technique together as part of our framework K, because we think that they fit very well together. However, one can nevertheless use them independently; for example, one can use the K-notation in algebraic specification or rewrite-based applications not necessarily related to programming languages, and one can employ the K-technique to define programming languages in any reduction-based framework, ignoring entirely the proposed notation. An ideal name for our framework would have been "C", since our technique is based on *continuations* and our trickiest notational convention is that for *context transformers*, but this letter is already associated to a programming language. To avoid confusion we use the letter "K" instead.

Figure 1 shows the complete definition of λ_{K} in K, including both its syntax and its semantics. The imported modules *BOOL*, *INT* and *K-BASIC* contain basic definitions of booleans, of integer numbers and of infrastructure operations that tend to be used by most language definitions. We will discuss these later, together with other technical details underlying the K framework; we here only mention that these modules can be either defined in the same style as other language features, that is, as rewrite logic theories using the K framework, or defined or even implemented as builtins. Here we only present the major characteristics and intuitions underlying the K definitional framework. It is fair to say upfront that we have not implemented either a parser of K or an automatic translator of K into rewriting logic or other formalisms yet; currently, we are using it as an alternative to SOS or reduction semantics to define languages, which can be then straightforwardly translated into rewrite logic definitions (note that SOS or reduction semantics language definitions *cannot* be "straightforwardly" translated into rewriting logic; these need to follow generic and systematic transformation steps resulting in rather constrained and unnatural rewrite logic theories – see import *BOOL*, *INT*, *K-BASIC*



$eval : Exp \to Val$

$result : State \to Val$

$$\frac{eval(E)}{result(k(E)\ env(\cdot)\ store(\cdot)\ nextLoc(0))} \qquad (1)$$

$$\frac{result\langle k(V)\rangle}{V} \qquad (2)$$

$Var,\ Bool,\ Int < Exp$

$Bool,\ Int < Val$

$$k(\underset{V}{\underline{X}}\rangle\ env\langle (X,L)\rangle\ store\langle (L,V)\rangle \qquad (3)$$

not_ $: Exp \to Exp\ [!,$ not$_{Bool} : Bool \to Bool]$

$\_+\_ : Exp \times Exp \to Exp\ [!,\ \_+_{Int}\_ : Int \times Int \to Int]$

$\_<\_ : Exp \times Exp \to Exp\ [!,\ \_<_{Int}\_ : Int \times Int \to Bool]$

$\lambda\_.\_ : VarList \times Exp \to Exp$

$\_\,\_ : Exp \times ExpList \to Exp\ [![app]]$

$closure : VarList \times Exp \times VarLocSet \to Val$

$$k(\underset{closure(Xl,E,Env)}{\underline{\lambda Xl.E}}\rangle\ env(Env) \qquad (4)$$

$$k(\underset{Vl \curvearrowright bind(Xl) \curvearrowright E \curvearrowright Env'}{\underline{(closure(Xl,E,Env),Vl) \curvearrowright app}}\rangle\ env(\underset{Env}{\underline{Env'}}) \qquad (5)$$

if_then_else_ $: Exp \times Exp \times Exp \to Exp\ [!(1)[if]]\}$

$$\frac{bool(true) \curvearrowright if(E_1,E_2)}{E_1} \qquad (6)$$

$$\frac{bool(false) \curvearrowright if(E_1,E_2)}{E_2} \qquad (7)$$

$Loc < Val$

ref $\_ : Exp \to Exp\ [!]$

$\star\ \_ : Exp \to Exp\ [!]$

$\_ := \_ : Exp \times Exp \to Exp\ [!]$

$$k(\underset{loc(L)}{\underline{V \curvearrowright ref}}\rangle\ nextLoc(\underset{next(L)}{\underline{L}})\ store\langle \underset{(L,V)}{\underline{\cdot}}\rangle \qquad (8)$$

$$k(\underset{V}{\underline{loc(L) \curvearrowright \star}}\rangle\ store\langle (L,V)\rangle \qquad (9)$$

$$k(\underset{V}{\underline{(loc(L),V) \curvearrowright :=}}\rangle\ store\langle (L,\underset{V}{\underline{\_}})\rangle \qquad (10)$$

halt $\_ : Exp \to Exp\ [!]\ \}$

$$k(V \curvearrowright \underset{\cdot}{\underline{halt \curvearrowright \_}}) \qquad (11)$$

Figure 1: Definition of Syntax and Semantics of λ_{K} in K

Section 3). However, K was designed to be implementable *as is*; therefore, even though we do not have machine support for it yet, we think of the subsequent definitions, in particular the one of λ_{K} in Figure 1, as machine executable.

The modules *BOOL* and *INT* come with sorts *Bool*, *Int* and *Nat*, where *Nat* is a subsort of *Int*. The module *K-BASIC* comes with sorts *Exp*, *Val* and *Loc*, for expressions, values and locations, respectively; no subsort relations are assumed among these sorts, in particular the sort *Val* is *not* a subsort of *Exp* – indeed, values are not expressions in general (e.g., a closure is a value but not an expression). Locations also come with a constructor operation $\mathit{location} : \mathit{Nat} \to \mathit{Loc}$ and an operation $\mathit{next} : \mathit{Loc} \to \mathit{Loc}$, which gives the next available location; for simplicity, in this paper we assume that $\mathit{next}(\mathit{location}(N)) = \mathit{location}(N+1)$, but one is free to define a garbage collector returning the next unoccupied location. One additional and very important sort that the module *K-BASIC* comes with is *Continuation*, which is a supersort of both *Exp* and *Val* (in fact, it is a supersort of lists of expressions and of lists of values), which is structured as a list (of "tasks"). K is a modular framework, that is, language features can be added to or dropped from a language without having to change any other unrelated features. However, for simplicity, we here refrain from defining modules and their corresponding module composition operations.

A K language definition contains module imports, declarations of *operations* (which can be structural, language constructs, or just ordinary), and declarations of *contextual rules*. For each sort *Sort*, in K we implicitly assume corresponding sorts *SortList* and *SortSet* for lists and sets of sort *Sort*, respectively. By default, we use the (same) infix associative comma operation $\_\,,\_$ as a constructor for lists of any sort, and the infix associative and commutative operation $\_\,\_$ as a constructor for sets of any sort. One important exception from the comma notation is the *continuation construct*, written $\_ \curvearrowright \_$, which sequentializes evaluation tasks: e.g., the list $(E, E') \curvearrowright + \curvearrowright \mathit{write}(X)$ should read "first evaluate the expressions E and E', then sum their values, then write the result to X"; in module *K-BASIC*, the sort *Continuation* is declared as an alias for the sort *ContinuationItemList*, where *ContinuationItem* is declared as a supersort of *ExpList* and *ValList*, and is typically extended with new constructs in a language definition (such as the "+" above). By convention, tuple sorts *Sort1Sort2...Sortn* and tuple operations $(\_, \_, ..., \_) : \mathit{Sort1} \times \mathit{Sort2} \times \cdots \mathit{Sortn} \to \mathit{Sort1Sort2...Sortn}$ are assumed tacitly, without declaring them explicitly, whenever needed. Also, some procedure will be assumed for *sort inference* for variables; this will relieve us from declaring obvious sorts and thus keep the formal programming language definitions more interesting and compact. By convention, variables start with a capital letter. If the sort inference of a variable is ambiguous, or simply for clarity reasons, one can manually sort, or annotate, any subterm, using the notation $t : \mathit{Sort}$; in particular, t can be a variable. This manual sorting can be used by the sort inference procedure to both eliminate ambiguities and improve efficiency.

Structural operations, which we prefer to define graphically as a tree whose nodes are sorts and whose edges are operation declarations, are those that give the structure of the state. When more edges go into the same node, the target sort is automatically assumed a multiset sort, each of the incoming operations generating an element of that set. For example, the sort *State* in Figure 1 is a set sort, while *k* (continuation), *env* (environment), *store*, and *nextLoc* (next location) can generate elements of *State*; these elements are also called *state attributes*. All the corresponding set operation declarations and equations are added automatically. Structural operations will be used in the process of context transformation of rules that will be explained shortly. The distinguished characteristic of K is its *contextual rule*. A contextual rule consists of a term C, called the *context*, in which some subterms t_1, ..., t_n, which are underlined, are substituted by other terms t'_1, ...,

t'_n, written underneath the lines. Contextual rules can be automatically translated into equations or rewrite rules. Our motivation for contextual rules came from the observation that in many rewriting logic definitions of programming languages, large terms tend to appear in the left hand sides of rewrite rules *only* to create the context in which a small change should take place; the right hand sides of such rewrite rules consist mostly of repeating the left ones, thus making them hard to read and error prone.

When writing K definitions, for clarity we prefer to write the syntax on the left and the corresponding semantics on the right. There will typically be one or at most two semantic rules per syntactic language construct. At this moment we do not make a distinction between declarations of operations intended to define the syntax of a language and the declarations of auxiliary operations that are necessary only to define the semantics; for the time being we assume that the distinction is clear, but in the forthcoming implementation of K we will have designated operator annotations for this purpose. Tacitly, we tend to use sans serif font for the former and italic font for the latter.

In the definition of λ_{K} in Figure 1, the auxiliary operations *eval* and *result* are used to create the appropriate state context and to extract the resulting value when the computation is finished, respectively. Their definitions consist of trivial contextual rules: *eval*(E) creates the initial state with E in the continuation, empty environment and store, and first free location 0, and then wraps it with the operator *result*. The rules corresponding to various features of the language will eventually evaluate E to a value, which will reside as the only item in the continuation at the end of the computation. Then the rule corresponding to *result* will collapse everything to that value.

Note the use of the angle brackets "$\langle$" and "$\rangle$" in the rule for *result*. These can be used wherever a list or a set is expected, to signify that there may be more elements to the left or to the right of the enclosed term, respectively: "$\langle t\rangle$" reads "the term t appears somewhere in the list or set", "$(t\rangle$" reads "t appears as a prefix of the list or set", and "$\langle t)$" reads "t appears as a suffix of the list or set"; they are all equivalent in the case of sets, in which case we prefer the former. Intuitively, "$\langle$" can be read "and so on to the left" and "$\rangle$" can be read "and so on to the right". Therefore, the contextual rule of *result* says that if the term to reduce will ever consist of a *result* operator wrapping a state that contains a continuation that contains a value V, then replace the entire term by V. The sort *Val* of V can be automatically inferred, because the result sort of *result* is *Val*.

We next say via subsort declarations that variables, booleans and integers are all expressions, and that booleans and integers are also values. K adds automatically an operation $valuesort : Valuesort \to Val$ for each sort *Valuesort* that is declared a subsort of *Val*. In our case, operations $bool : Bool \to Val$ and $int : Int \to Val$ are added now, and another operation $loc : Loc \to Val$ will be added later, when the subsort declaration $Loc < Val$ appears. Also, for those sorts *Valuesort* that are subsorts of both *Val* and *Exp*, a contextual rule of the form

$$k(\underset{valuesort(Ev)}{\underline{\qquad Ev \qquad}}\rangle$$

is automatically considered in K, saying that whenever an expression *Ev* of sort *Valuesort* (inferred from the fact that *Ev* appears as an argument of *valuesort*) appears at the top of the continuation, that is, if it is the next task to evaluate, then simply replace it by the corresponding value, *valuesort*(Ev). Rule (3) gives the semantics of variable lookup: if variable X is the next task in the continuation (note the "$\rangle$" angle bracket matching the rest of the continuation), and if the pair (X, L) appears in the environment (here note both angle brackets), and if the pair (L, V) appears in the store, then just replace X by V at the top of the continuation; note also that the right sorts for variables can be automatically inferred.

Let us next briefly discuss the operator attributes. The operations not, $\_ + \_$, as well as the other operators extending the builtin operations, have two attributes, an exclamation mark "!" and the operations they extend. The exclamation mark, called *strictness attribute*, says that the operation is *strict* in its arguments in the order they appear, that is, its arguments are evaluated first (from left to right) and then the actual operation is applied. An exclamation mark attribute can take a list of numbers as optional argument and one optional attribute, like in the case of the conditional. These decorations of the strictness attribute can be fully and automatically translated into corresponding rules. We next discuss these.

If the exclamation mark attribute has a list of numbers argument, then it means that the operation is declared strict in those arguments in the given order. Note that the missing argument of a strictness attribute is just a syntactic sugar convenience; indeed, an operator with n arguments which is declared just "strict" using a plain "!" attribute, is entirely equivalent to one whose attribute is "$!(1\ 2 \ldots n)$". When evaluating an operation declared using a strictness attribute, the arguments in which the operation was declared strict (i.e., the argument list of "!") are first scheduled for evaluation; the remaining arguments need to be "frozen" until the other arguments are evaluated. If the exclamation mark attribute has an attribute, say *att*, then *att* will be used as a continuation item constructor to "wrap" the frozen arguments. Similarly, the lack of an attribute associated to a strictness attribute corresponds to a default attribute, which by convention has the same name as the corresponding operation; to avoid confusion with underscore variables, we drop all the underscores from the original language construct name when calculating the default name of the attribute of "!". For example, an operation declared "$\_ + \_ \ : \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}$ [!]" is automatically desugared into "$\_ + \_ \ : \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}$ [!(1 2)[+]]".

Let us next discuss how K generates corresponding rules from strictness attributes. Suppose that we declare a strict operation, say *op*, whose strictness attribute has an attribute *att*. Then K automatically adds

(a) an auxiliary operation *att* to the signature, whose arguments (number, order and sorts) are precisely the arguments of *op* in which *op* is *not* strict; the result sort of this auxiliary operation is *ContinuationItem*;

(b) a rule initiating the evaluation of the strict arguments in the specified order, followed by the other arguments "frozen", i.e., wrapped, by the corresponding auxiliary operation *att*.

For example, in the case of λ_{K}'s conditional whose strictness attribute is "!(1)[*if*]", an operation "$\mathit{if} \ : \mathit{Exp} \times \mathit{Exp} \to \mathit{ContinuationItem}$ is automatically added to the signature, together with a rule

$$\frac{\mathsf{if}\ E\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2}{E \curvearrowright \mathit{if}(E_1, E_2)}$$

If the original operation is strict in all its arguments, like not, $\_ + \_$ and $\_\_$, then the auxiliary operation added has no arguments (it is a constant). For example, in the case of λ_{K}'s application whose strictness attribute is "![*app*]", an operation "$\mathit{app} \ : \to \mathit{ContinuationItem}$ is added to the signature, together with a rule

$$\frac{E\ \mathit{El}}{(E, \mathit{El}) \curvearrowright \mathit{app}}$$

The "builtin" operations added as attributes to some language constructs, such as $\_{+_{\mathit{Int}}}\_ : \mathit{Int} \times \mathit{Int} \to \mathit{Int}$ as an attribute of $\_ + \_ \ : \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}$, each corresponds to precisely one K-rule that "calls" the builtin on the right arguments. This convention will be explained in detail later in the paper; we here only show the two implicit rules corresponding to the attributes of the addition operator in Figure 1:

$$\frac{E_1 + E_2}{(E_1, E_2) \curvearrowright +}$$

$$\frac{(\mathit{int}(I_1), \mathit{int}(I_2)) \curvearrowright +}{\mathit{int}(I_1 +_{\mathit{Int}} I_2)}$$

Notice that these rules can apply anywhere they match, not only at the top of the continuation. In fact, the rules with exclamation mark attributes can be regarded as some sort of "precompilation rules" that transform the program into a tree (because continuations can be embedded) that is more suitable for the other semantic rules. If, in a particular language definition, code is not generated dynamically, then these pre-compilation rules can be applied all "statically", that is, before the semantic rules on the right apply; one can formally prove that, in such a case, these "pre-compilation" rules need not be applied anymore during the evaluation of the program. These rules associated to exclamation mark attributes should *not* be regarded as transition rules that "evolve" a program, but instead, as a means to put the program into a more convenient but *computationally equivalent* form; indeed, when translated into rewriting logic, the precompilation contextual rules become all equations, so the original program and the precompiled one are in the same equivalence class, that is, they are *the same* program within the initial model semantics that rewriting logic comes with.

To get a better feel for how the rules corresponding to strictness attributes change the syntax of a program, let us consider the λ-expression corresponding to the definition body of the g function in our definition of factorial in λ_{K} above, namely

$$\mathsf{if}\ m > 1\ \mathsf{then}\ (r := (*r) * m;\ h(m-1, h))\ \mathsf{else}\ \mathsf{halt}\ (*r).$$

The strictness rules associated automatically to the exclamation mark attributes transform this term in a few steps (that can also be applied in parallel on a parallel rewrite engine) into

$$(m, 1) \curvearrowright > \curvearrowright \mathit{if}((\lambda d\ .\ (r, (r \curvearrowright *, m) \curvearrowright *) \curvearrowright :=)((h, (m, 1) \curvearrowright -, h) \curvearrowright \mathit{app}),\ r \curvearrowright * \curvearrowright \mathit{halt}).$$

We have also applied the notational convention for sequential composition and assumed that the continuation constructor $\_ \curvearrowright \_$ binds the tightest. Therefore, these "precompilation" rules (that can be generated automatically from the exclamation mark attributes of operators) do nothing but iteratively transform the expression to evaluate in a postfix form, taking into account the strictness information of language constructs. Since we'll refer to this expression several times in Figure 2, we introduce the "macro" $\%G$ to refer to it.

Let us discuss the remaining contextual rules in Figure 1. Rule (4) says that a λ-abstraction evaluates to a closure. The current environment is needed, so it is mentioned as part of the rule. Note that a closure is a value but *not* an expression. Rule (5) gives the semantics of application (defined as a strict operation): it expects to be passed a closure and a value V; it then binds the value V to the parameter X in the closure in the environment *Env* of the closure, which becomes current, and then initiates the evaluation of E, the body of the closure; the original environment *Env'* is recovered after the resulting value is produced. The semantics of *bind*(Xl) and environment recovery are straightforward and assumed part of the module *K-BASIC*, since they are used in most languages that we defined; they will be formally defined and discussed in detail in Section 4.

Rules (6) and (7) give the expected semantics of the conditional statement. Recall that the conditional is declared strict in its first argument, so an auxiliary operation *if* is generated together with the rule above initiating the evaluation of the condition. Rules (8), (9) and (10) give the semantics of references, one per corresponding language construct. Note first that *Loc* is defined as a subsort of *Val*, so an operation $loc : Loc \to Val$ is implicitly assumed. Note also that all the language constructs for references are strict, so we only need to define how to combine the values produced by their arguments. Rule (8) applies when a value is passed to the auxiliary operator *ref* at the top of the continuation; when that happens, a corresponding location value is produced, the current next location is incremented, and the value is stored at the newly produced location in the store. Note the use of angle brackets to match the unit "·" of the store multiset. By convention, in K we use the dot "·" as the unit of all set and list structures. In this case, matching the unit somewhere in the store always succeeds, and replacing it by a pair corresponds to adding that pair to the store. Rule (9) for dereferencing location L grabs the value V at location L in the store, and rule (10) for assignment replaces the exiting value at location L in the store by V; underscore variables can be used any time when the corresponding matched term is not needed for any other purpose (except the actual matching) in the rule - in other words, variables that appear only once in a rule can well be underscores. Finally, rule (11) for halt simply dissolves the rest of the continuation, keeping only the value on which the program halted. If the evaluation of the argument expression E of halt(E) encounters another halt, then the nested halt will obviously terminate the execution of the program (the outer halt that initiated the evaluation of E will be dissolved as a consequence of applying the rule (11) for the nested halt).

Figure 2 shows the "execution"[1] of λ_{K} (as defined in K) on the factorial program above, where n is 3. Each line shows a term. The underlined subterms of each term are those that change by the application of contextual rules. The rules applied are mentioned on the right column, above the arrow ⇒: !* means a series of application of implicit rules associated to ! attributes; (4), for example, means an application of the contextual rule (4); $[K]$ stays for basic rules that come with the module *K-BASIC*. If one collapses all the applications of non-numbered contextual rules (which is precisely what happens semantically anyway, because all these contextual rules are translated into equations in the associated rewrite logic theory), then one can see exactly all the intended computation steps of this execution (making abstraction, of course, of their syntactic representation). Despite its computational feel, the "execution" in Figure 2 is actually a formal derivation, or *proof*, that the factorial of 3 indeed reduces to 6 within the formal semantics of our programming language. It is, however, entirely executable; in fact, the derivation in Figure 2 was produced by tracing Maude's reduction (with the command "`set trace on .`") of the factorial program within the Maude-ified version of the K semantics in Figure 1; this Maude semantics is shown in Appendix D.

Let us next add *concurrency* to λ_{K}; we do it by means of dynamic threads. To keep the language simple, we refrain from adding any built-in synchronization mechanism here. We will add synchronization via acquire and release of locks later in the paper, in the context of defining the more complex language FUN; for now, one can partially simulate synchronization by programming techniques (using wait-loops and shared variables). Our point here is to emphasize that adding threads to λ_{K} is a straightforward task, requiring *no change* of the already defined language features. The additional work needed is shown in Figure 3 and it is, in our view, conceptually unavoidable unless one makes stronger assumptions about the possible extensions of the language *apriori*; for example, if one assumes that any language can be potentially extended with threads, then one can

[1] We warmly thank Traian Florin Şerbănuţa for help with typing in this sample execution.

$$\mathit{eval}(\underline{(\lambda r\ .\ (\lambda g\ .\ g(3-1,g))(\lambda m,h\ .\ \textsf{if}\ m>1\ \textsf{then}\ (\lambda d.h(m-1,h)\ (r:=(*r)*m))\ \textsf{else halt}\ (*r)))\ (\textsf{ref}\ 3)})\quad\overset{!*}{\Rightarrow}$$

$$\underline{\mathit{eval}(((\lambda r\ .\ ((\lambda g\ .\ (g,(3,1)\curvearrowright -,g)\curvearrowright \mathit{app}),\lambda m,h\ .\%G)\curvearrowright \mathit{app}),3\curvearrowright \mathit{ref})\curvearrowright \mathit{app})}\quad\overset{(1)}{\Rightarrow}$$

$$\mathit{result}(k(\underline{((\lambda r\ .\ ((\lambda g\ .\ (g,(3,1)\curvearrowright -,g)\curvearrowright \mathit{app}),\lambda m,h\ .\ \%G)\curvearrowright \mathit{app}),3\curvearrowright \mathit{ref})}\curvearrowright \mathit{app})\ \ \mathit{store}(.)\ \ \mathit{nextLoc}(0)\ \ \mathit{env}(.))\quad\overset{[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{(\lambda r\ .\ ((\lambda g\ .\ (g,(3,1)\curvearrowright -,g)\curvearrowright \mathit{app}),\lambda m,h\ .\ \%G)\curvearrowright \mathit{app})}\curvearrowright (3\curvearrowright \mathit{ref},\cdot)\curvearrowright \mathit{app})\ \ \mathit{store}(.)\ \ \mathit{nextLoc}(0)\ \ \mathit{env}(.))\quad\overset{(4)}{\Rightarrow}$$

$$\mathit{result}(k(\underline{\mathit{closure}(r,((\lambda g\ .\ (g,(3,1)\curvearrowright -,g)\curvearrowright \mathit{app}),\lambda m,h\ .\ \%G)\curvearrowright \mathit{app},\cdot)\curvearrowright (3\curvearrowright \mathit{ref},\cdot)}\curvearrowright \mathit{app})\ \mathit{store}(.)\ \mathit{nextLoc}(0)\ \mathit{env}(.))\quad\overset{[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{3\curvearrowright \mathit{ref}}\curvearrowright (\cdot,\mathit{closure}(r,((\lambda g\ .\ (g,(3,1)\curvearrowright -,g)\curvearrowright \mathit{app}),\lambda m,h\ .\ \%G)\curvearrowright \mathit{app},\cdot))\curvearrowright \mathit{app})\ \mathit{store}(\underline{.})\ \mathit{nextLoc}(\underline{0})\ \mathit{env}(.))\quad\overset{(8),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{\mathit{loc}(0)\curvearrowright (\cdot,\mathit{closure}(r,((\lambda g.(g,(3,1)\curvearrowright -,g)\curvearrowright \mathit{app}),\lambda m,h.\%G)\curvearrowright \mathit{app},\cdot))}\curvearrowright \mathit{app})\ \mathit{store}((0,3))\ \mathit{nextLoc}(1)\ \mathit{env}(.))\quad\overset{[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{(\mathit{closure}(r,((\lambda g.(g,(3,1)\curvearrowright -,g)\curvearrowright \mathit{app}),\lambda m,h.\%G)\curvearrowright \mathit{app},\cdot),\mathit{loc}(0))\curvearrowright \mathit{app}})\ \mathit{store}((0,3)\ \underline{\cdot})\ \mathit{nextLoc}(\underline{1})\ \mathit{env}(\underline{.}))\quad\overset{(5),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{(\lambda g.(g,((3,1)\curvearrowright -),g)\curvearrowright \mathit{app})\curvearrowright (\lambda m,h.\%G,.)}\curvearrowright \mathit{app}\curvearrowright (.))\ \mathit{store}((0,3)(1,\mathit{loc}(0)))\ \mathit{nextLoc}(2)\ \mathit{env}(\underline{(r,1)}))\quad\overset{(4),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{(\mathit{closure}(g,(g,((3,1)\curvearrowright -),g)\curvearrowright \mathit{app},(r,1)),\%C1)\curvearrowright \mathit{app}}\curvearrowright (.))\ \mathit{store}((0,3)(1,\mathit{loc}(0))\ \underline{\cdot})\ \mathit{nextLoc}(\underline{2})\ \mathit{env}(\underline{(r,1)}))\quad\overset{(5),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{g\curvearrowright (((3,1)\curvearrowright -),g,.)}\curvearrowright \mathit{app}\curvearrowright ((r,1))\curvearrowright (.))\ \mathit{store}((0,3)(1,\mathit{loc}(0))(2,\%C1))\ \mathit{nextLoc}(3)\ \mathit{env}((g,2)(r,1)))\quad\overset{(3),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{(\%C1,2,\%C1)\curvearrowright \mathit{app}}\curvearrowright ((r,1))\curvearrowright (.))\ \mathit{store}((0,3)(1,\mathit{loc}(0))(2,\%C1)\ \underline{\cdot})\ \mathit{nextLoc}(\underline{3})\ \mathit{env}(\underline{(g,2)(r,1)}))\quad\overset{(5),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{m\curvearrowright (1,.)\curvearrowright >}\curvearrowright \%\mathit{IF}\curvearrowright ((g,2)(r,1))\curvearrowright ((r,1))\curvearrowright (.))\ \%S1)\quad\overset{(3),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{\mathit{bool}(\mathit{true})\curvearrowright \%\mathit{IF}}\curvearrowright \%e1)\ \%S1)\quad\overset{(6),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{(\lambda d.(h,((m,1)\curvearrowright -),h)\curvearrowright \mathit{app})\curvearrowright ((r,((r\curvearrowright *),m)\curvearrowright *)\curvearrowright :=,.)}\curvearrowright \mathit{app}\curvearrowright \%e1)\ \%S1)\quad\overset{(4,3),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{\mathit{loc}(0)\curvearrowright *\curvearrowright (m,.)}\curvearrowright *\curvearrowright (.,\mathit{loc}(0))\curvearrowright :=\curvearrowright (.,\%C2)\curvearrowright \mathit{app}\curvearrowright \%e1)\ \%S1)\quad\overset{(9),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{m\curvearrowright (.,3)\curvearrowright *\curvearrowright (.,\mathit{loc}(0))}\curvearrowright :=\curvearrowright (.,\%C2)\curvearrowright \mathit{app}\curvearrowright \%e1)\ \%S1)\quad\overset{(3),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{(\mathit{loc}(0),6)\curvearrowright :=\curvearrowright (.,\%C2)}\curvearrowright \mathit{app}\curvearrowright \%e1)\ \mathit{store}(\underline{(0,3)}\ \%s1\ )\ \mathit{nextLoc}(5)\ \mathit{env}((h,4)(m,3)(r,1)))\quad\overset{(10),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{(\%C2,6)\curvearrowright \mathit{app}}\curvearrowright \%e1)\ \mathit{store}((0,6)\ \%s1\ \ \underline{\cdot})\ \mathit{nextLoc}(\underline{5})\ \mathit{env}(\underline{(h,4)(m,3)(r,1)}))\quad\overset{(5),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{h\curvearrowright (((m,1)\curvearrowright -),h,.)}\curvearrowright \mathit{app}\curvearrowright \%e2)\ \mathit{store}((0,6)\ \%s1\ (5,6))\ \mathit{nextLoc}(6)\ \mathit{env}((d,5)(h,4)(m,3)(r,1)))\quad\overset{(3),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{(\%C1,1,\%C1)\curvearrowright \mathit{app}}\curvearrowright \%e2)\ \mathit{store}((0,6)\ \%s1\ (5,6)\ \underline{\cdot})\ \mathit{nextLoc}(\underline{6})\ \mathit{env}(\underline{(d,5)(h,4)(m,3)(r,1)}))\quad\overset{(5),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{m\curvearrowright (1,.)\curvearrowright >}\curvearrowright \%\mathit{IF}\curvearrowright \%e3)\ \%S2)\quad\overset{(3),[K]}{\Rightarrow}$$

$$\mathit{result}(k(\underline{\mathit{bool}(\mathit{false})\curvearrowright \%\mathit{IF}}\curvearrowright \%e3)\ \%S2)\quad\overset{(7)}{\Rightarrow}$$

$$\mathit{result}(k(\underline{r}\curvearrowright *\curvearrowright \mathit{halt}\curvearrowright \%e3)\ \%S2)\quad\overset{(3)}{\Rightarrow}$$

$$\mathit{result}(k(\underline{\mathit{loc}(0)\curvearrowright *}\curvearrowright \mathit{halt}\curvearrowright \%e3)\ \%S2)\quad\overset{(9)}{\Rightarrow}$$

$$\mathit{result}(k(6\curvearrowright \underline{\mathit{halt}\curvearrowright \%e3})\ \%S2)\quad\overset{(11)}{\Rightarrow}$$

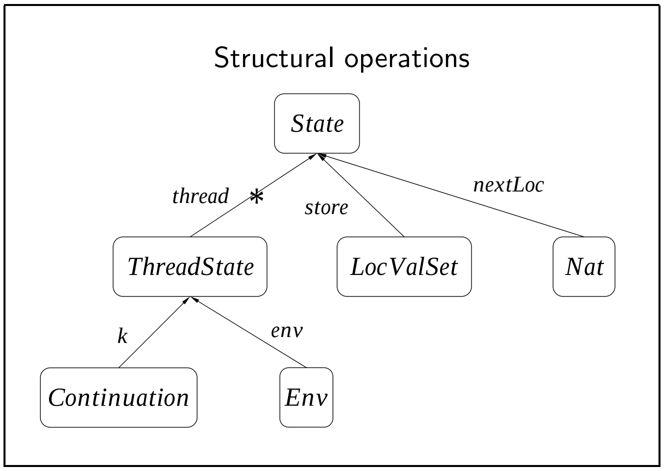
$$\underline{\mathit{result}(k(6)\ \%S2)}\quad\overset{(2)}{\Rightarrow}$$

6

$\%\mathit{IF}$ stands for $\mathit{if}((\lambda d.(r,(r\curvearrowright *,m)\curvearrowright *)\curvearrowright :=)((h,(m,1)\curvearrowright -,h)\curvearrowright \mathit{app}),\ r\curvearrowright *\curvearrowright \mathit{halt})$
$\%G$ stands for $(m,1)\curvearrowright >\curvearrowright \%\mathit{IF}$
$\%C1$ stands for $\mathit{closure}(m,h,\%G,(r,1))$
$\%S1$ stands for $\mathit{store}((0,3)\ \%s1\ )\ \mathit{nextLoc}(5)\ \mathit{env}((h,4)(m,3)(r,1)))$
$\%e1$ stands for $((g,2)(r,1))\curvearrowright ((r,1))\curvearrowright (.)$
$\%C2$ stands for $\mathit{closure}(d,(h,((m,1)\curvearrowright -),h)\curvearrowright \mathit{app},(h,4)(m,3)(r,1))$
$\%s1$ stands for $(1,\mathit{loc}(0))(2,\%C1)(3,2)(4,\%C1)$
$\%e2$ stands for $((h,4)(m,3)(r,1))\curvearrowright \%e1$
$\%e3$ stands for $((d,5)(h,4)(m,3)(r,1))\curvearrowright \%e2$
$\%S2$ stands for $\mathit{store}((0,6)\ \%s1\ (5,6)(6,1)(7,\%C1))\ \mathit{nextLoc}(8)\ \mathit{env}((h,7)(m,6)(r,1))$

Figure 2: Sample run of the factorial program in the executable semantics of λ_{K} in K.

The definition of λ_{K} in Figure 1 needs to be changed as follows:
1) replace structural operators with the ones in the picture to the right
2) replace definition of eval as below
3) add two more rules, as below:
a) one for creation of threads; and
b) one for termination of threads
No other changes needed.



$$\frac{eval(E)}{result(thread(k(E)\ env(\cdot))\ store(\cdot)\ nextLoc(0))} \qquad (1)$$

$$\left.\begin{array}{l} \mathsf{spawn}\ \_ \ : Exp \rightarrow Exp \\ die : \cdot \rightarrow ContinuationItem \end{array}\right\} \left\{\begin{array}{r} thread(k(\underset{0}{\underline{\mathsf{spawn}\ E}}\rangle\ env(Env))\ \dfrac{\cdot}{thread(k(E \curvearrowright die)\ env(Env))} \qquad (12) \\ \\ \dfrac{thread\langle k(V : Val \curvearrowright die)\rangle}{\cdot} \qquad (13) \end{array}\right.$$

Figure 3: Adding threads to λ_{K}

define for non-concurrent λ_{K} the state as in Figure 3 from the very beginning (rather than as in Figure 1). However, this would not be a general solution, because one cannot know in advance how a language will be extended; for example, how can one know that λ_{K} is not going to be extended with concurrency using actors [1] instead of threads? (In fact, this is in our opinion quite an appealing extension.) A major factor in the design of K, whose importance increasingly became clear to us empirically by experience with several language definition case studies, was the observation that whenever one extends an existing language by adding new features, or whenever one builds a language by combining features already defined in some library, one cannot and does not want to avoid analyzing and deciding on "the structural big picture" of the language, namely how the state items needed by the various desired language features are structured within the state of the language (recall that our notion of "state" is broad and generic).

We want the threads in λ_{K} to be dynamically created and terminated. Like in any multi-threaded language, threads in λ_{K} may share memory locations. More precisely, a newly created thread shares the same environment as its parent thread. Therefore, accesses (reads or writes) to variables shared by different threads may lead to non-deterministic behaviors of programs (in particular to data-races). All these suggest to the language designer the state structure depicted in Figure 3 for multi-threaded λ_{K}. Its state therefore contains, besides a store and a next available location like in the non-concurrent version, an arbitrary number of threads. The star ($\star$) on the arrow labeled *thread* in the graph in Figure 3 means that corresponding subterms wrapped by the state constructor operator *thread* can appear multiple times in the state — in this case, reflecting the fact that the number of threads created during the execution of a program varies. The star annotation happens to be irrelevant in the definition of multi-threaded λ_{K}, but in other language definitions it may play a critical role in the disambiguation of the context transformation process discussed next. Each subterm in the state soup wrapped by a *thread* construct (i.e., each thread) contains a continuation $k(...)$ and an environment $env(...)$.

The next step is to create the initial state of a program. Unlike in non-concurrent λ_{K} where the initial state contained the continuation and the environment at the same top level as the store and the next available location, in multi-threaded λ_{K} the initial state should contain one thread at the same top level as the store and the next available location, but the continuation associated to the program to evaluate as well as its corresponding (empty) environment should be contained as part of that thread, rather than at the top level (so we have a "nested soup"). This way, several threads with their corresponding continuations and environments can be unambiguously created, terminated and accessed dynamically. The semantics of the operation *eval* needs to be changed now to appropriately initiate the evaluation of its argument expression taking into consideration the new initial state.

An intriguing observation at this stage is that many of the contextual rules in Figure 1 do not parse anymore because of the new state structure, and so does the sample execution in Figure 2. Therefore, it looks as if the addition of threads to λ_{K} broke the modularity of the other language feature definitions. That would be, indeed, the case in the context of most language definitional frameworks, including SOS as well as plain rewrite logic definitions of languages. However, in K *we need to change no existing definitions of other language features*, even though they were defined before the state structure of multithreaded λ_{K} was known. That is thanks to one of the most important features of K, called *context transformer* and explained next.

In K, the structural operators play a crucial role in the definition of a language. Many of the contextual rules in a K definition of a language are *incomplete* unless full knowledge of the state structure is available. That incompleteness allows language designers to *only mention the relevant part* of the state when defining the semantics of each feature. An advantage of rewrite logic definitions compared to SOS definitions is that in a rewrite rule one needs only mention the local structure of a subterm on which a transformation is applied, while in traditional SOS definitions one needs to mention the entire state, or configuration, even though one wants to change only a very small portion of it. Thanks also to matching modulo associativity, commutativity and identity which bring additional modularity to language definitions, rewriting logic allows for elegant and compact definitions even of complex features of languages [19, 18]. However, rewriting logic imposes well-formedness of its rewrite rules w.r.t. signatures, so definitions of features cannot be easily transported from one language to another when the structure of the state changes. K's context transformers address precisely this limitation of rewriting logic. At this moment we solve all the context transformers statically because we assume that the structure of the state does not change. However, one could also imagine, at least in principle, more complex situations in which the structure of the state would change dynamically; in such a case one could informally think of context transformers as of *matching modulo structure*.

Context transformers will be explained in detail later in the paper; we here only discuss their application on multi-threaded λ_{K}. Intuitively, once all the structural operators are known, such as in the pictures in Figures 1 and 3, the contextual rules need not mention the complete "path" to each operation that appears in a rule, because that path can be inferred *automatically* and *unambiguously* from the declared structure of the state. For example, consider rule (2) in Figure 1 defined for non-concurrent λ_{K}, but when the structure of the language changes as in Figure 3. This rule obviously does not parse anymore, because *result* expects an argument of sort *State* which, because of the structural operators in Figure 3, can only be constructed with state constructors *thread(...)*, *store(...)*, and *nextLoc(...)*; therefore, there is no $k(...)$ construct at the top level of the state anymore, so $\mathit{result}\langle k(V)\rangle$ naturally does not parse. However, the crucial observation is that there is a *unique* way to have $k(V)$ make sense in a term below the operation *result*: when $k(V)$ is part of a thread state, wrapped by the operator *thread*. Therefore, by taking the structural operators into account, one can automatically and unambiguously transform $\mathit{result}\langle k(V)\rangle$ into $\mathit{result}\langle \mathit{thread}\langle k(V)\rangle\rangle$, the latter making now full sense in the new language definition. Similarly, with the new state structure in Figure 3, one can automatically transform rule (3) in Figure 1 into

$$\mathit{thread}\langle k(\underset{V}{\underline{X}}\rangle\ \ \mathit{env}\langle (X, L)\rangle\rangle\ \ \mathit{store}\langle (L, V)\rangle.$$

Since a thread has only two soup attributes each with multiplicity one, namely a continuation and an environment, the angle brackets of *thread* could have been just plain parentheses. There are guidelines that need to be followed and conventions that need to be adopted in order for the context transformation process of rules to be unambiguous; these are discussed in more depth later in the paper. As a rule of thumb, terms that appear in a contextual rule are grouped in a "depth-first" style. For example, in the rule (3) for variable lookup above, we grouped the continuation and the environment together and placed them inside one thread; one could have also put each of the two in one separate thread, but that would have violated the "depth first" style. However, if two continuations were involved in some contextual rule (that is indeed possible in some language definitions), then that rule would be transformed into one where each continuation appears in its separate thread; that's because the structural operations picture in Figure 3 specifically allows threads, not continuations, to appear multiple times in the state (the star "$\star$" on the arrow *thread*).

All contextual rules defining the sequential features of λ_{K} in Figure 1 can be automatically transformed in a similar manner when transported into the definition of multithreaded λ_{K}. Let us next discuss the concurrent features of multithreaded λ_{K}. Spawning a new thread to calculate an expression E can be done from any other thread. The contextual rule (12) shows the semantics of $\mathsf{spawn}(E)$: a new thread is added to the state soup that contains the expression to evaluate E in its continuation and shares the same environment as the creating thread; $\mathsf{spawn}(E)$ evaluates to 0 (by convention; like in FUN defined later in the paper, we could have also added a special "unit" value and evaluate all constructs used for their side effects to that particular value) with no delay, and a *die* continuation item is placed after the expression E in the created thread to state that that thread should be terminated (and therefore removed from the state) once E evaluates to a value (this is done by rule (13)). In a more complex language, such as FUN, the resources of a thread (e.g., the locks it holds) need to be released as well when the thread is terminated.

An interesting observation here, which has nothing to do with K as a definitional framework but only with our particular design of multi-threaded λ_{K}, is that any thread can halt the program. Indeed, the rule for halt can be applied for any thread and, if applied on a created thread, the terminating continuation item *die* is eliminated and therefore rule (2) for returning the result can apply, thus returning the halted value as a result of the entire multi-threaded program. Note that the only way that rule (2) can apply on the continuation of a spawned thread is that that thread halts the computation; otherwise, the continuation item *die* would follow the value in the continuation, so rule (2) would be inapplicable. Since rules can be applied non-deterministically (and they would indeed be applied non-deterministically on a parallel rewrite engine), there is no guarantee that all the threads will be properly terminated before the final result is produced, even if no thread halts the computation. For example, the original thread may spawn a thread as the last task in its computation; this spawning will produce a 0 in the continuation of the original thread and, as a side effect, will create the new thread; now rule (2) can apply on the continuation of the original thread without waiting for the created thread to terminate.

The K framework will be discussed more fully in Sections 4 and 5, where we also define FUN, a more complex programming language with higher-order functions with return, let and letrec, lists, sequential composition, assignments, loops with break and continue, input/output, parametric exceptions, call/cc and concurrency via threads and synchronization. In this introduction we only wanted to emphasize that once learned, K is simple, natural, and leads to compact and easy to understand and read language definitions. An interesting feature of K that we have not discussed so far but which is also worthwhile mentioning, is the distinction that it makes among contextual rules. These can be:

- *Structural*, or *non-computational*. These are rules whose only role is to allow modifications of the structure of the state, so that other contextual rules that have computational meaning can apply. For example, the rules generated automatically by K for the strictness attributes ("!") as well as rule (1) are all structural, their role being to prepare the expression for evaluation by prioritizing the subexpressions that need to be evaluated first. Also, all the set and list implicit associativity and commutativity equations are structural, having no computational meaning.

- *Computational*. These are rules that each captures precisely one intended execution step of the program. All numbered rules in the definition of λ_{K} except (1) are computational.

- *Non-deterministic*. These are rules that can potentially change the deterministic behavior of a program. If a language is sequential and deterministic, then there is no need to declare any contextual rule to be non-deterministic, because the rules are expected to be Church-Rosser on the terms of interest (well-formed programs). However, if a language admits concurrency, then some rules may lead to non-deterministic executions; for example, reads and writes of memory locations, or acquires of locks.

Non-deterministic rules are always a (typically quite small) subset of the computational rules. The structural and computational declarations of contextual rules are modular, in the sense that one does not need knowledge about the entire language in order to declare a rule structural or computational. However, the non-deterministic declaration of rules is trickier and needs global information about the program. For example, one cannot know when one defines a variable lookup rule (3) whether the defined language is concurrent or not; if it is, then that rule needs to be declared non-deterministic. Also, it is impossible to identify the non-deterministic rules automatically in general (this problem is harder than testing confluence).

K can be all resolved *statically*, so a definition of a language in K *is* a rewrite logic specification. One subtle aspect of the translation of K into rewriting logic is to decide which contextual rules translate into equations and which into rewrite rules (see Section 2 for the semantic distinction between the two). In general, if one is interested in obtaining just one interpreter or a dynamic semantics for the language, then one can translate all the computational contextual rules into rewrite rules and the non-computational ones into equations. If formal analysis of programs is of interest, then one may want to reduce the number of rewrite rules to a minimum; this can be done by transforming only the non-deterministic contextual rules into rewrite rules. Since rewriting logic is relatively efficiently executable (rewrite engines such as ASF+SDF and Maude are quite fast) relatively efficient interpreters are obtained *for free* from such formal language definitions. However, K specifications follow a certain definitional methodology which identifies them as particular rewrite logic specifications. Therefore, one should be able to execute these K definitions more efficiently using some specialized rewrite engines or compilers than using general purpose rewrite engines such as ASF+SDF or Maude. Some preliminary experiments have been made showing that that is indeed the case; Section 9 gives more details.

Also, formal analysis tools for rewrite logic specifications, such as those of Maude [5] (e.g., model checkers), translate into corresponding tools for languages defined using the presented technique. Also, since rewriting logic is a computational logical framework with both initial model semantics and formal analysis techniques for initial models, the presented framework can also serve as a foundation for program verification. The technique presented here, or previous versions of it, has been used to define language features including type inference, object-orientation and subtyping, concurrency, etc., as well as large fragments of real programming languages, such as the λ-calculus, System F, BETA, HASKELL, JAVA, LLVM, LISP, PYTHON, PICT, RUBY, and SMALLTALK [24]. As discussed in [7] and Section 8, the model checker obtained for free from the formal definition of Java compares favorably with two state of the art JAVA model checkers.

2 Rewriting Logic

The language definitional framework proposed in this paper can be presented and understood in isolation, like SOS or reduction semantics, and implemented following different programming paradigms. However, as mentioned previously, we prefer to present it via an automatic translation to term rewriting modulo equations or to rewriting logic, for two reasons:

1. to make it clear upfront that we inherit all the good properties of these well understood and generic computational frameworks, and

2. to use at no additional effort the remarkable rewrite-based tool support developed over the last few decades.

We here informally recall some basic notions of rewriting logic and of its important sublogic called equational logic, together with operational intuitions of term rewriting modulo equations.

2.1 Equational Logics

Equational logic is perhaps the simplest logic having the full expressivity of computability [2]. One can think of it as a logic of "term replacement": terms can be replaced by equal terms in any context. An *equational specification* is a pair (Σ, E), where Σ is a set of "uninterpreted" operations, also called its "syntax", and E is a set of *equations* of the form $(\forall X)\ t = t'$ constraining the syntax, where X is some set of variables and t, t' are well-formed terms over variables in X and operations in Σ. Equational logics can be *many-sorted* [10] (operations in Σ have arguments of specific sorts), or even *order-sorted* [11], i.e., sorts come with a partial order on them; we use order-sorted specifications in this paper. Also, equations can be *conditional*, where the condition is a (typically finite) set of pairs $u = u'$ over the same variables X. We write conditional equations (of finite condition) using the notation $(\forall X)\ t = t' \ \Leftarrow\ u_1 = u'_1 \wedge \cdots \wedge u_n = u'_n$.

Models of an equational specification (Σ, E) interpret sorts into sets of values and operations in Σ into corresponding functions satisfying all the equations in E, where a model satisfies an equation if and only if the two terms evaluate to the same value for any assignment of their variables to values in the model. Models are also called Σ-*algebras* and it is customary to regard them as "realizations", or even "implementations", of the equational specifications they satisfy. Equational deduction is complete and consists of five natural deduction rules, namely *reflexivity*, *symmetry*, *transitivity*, *congruence* and *substitution*. We write $E \vdash_\Sigma e$ if the equation e can be derived with these rules from (Σ, E). Among the variety of models of an equational specification, there is one which, up-to-an-isomorphism, captures precisely the intended meaning of the specification: its *initial model*. Because of the "all-in-one and one-in-all" flavor and properties of initial algebras, as well as because any computable domain can be shown isomorphic to a (restricted) initial model over a finite equational specification [2], *initial algebra semantics* [12] was introduced in 1977 as a theoretically self-contained approach to (non-concurrent) programming language semantics.

2.2 Term Rewriting

Term rewriting is a related approach in which equations are *oriented* left-to-right, written $(\forall X)\ l \to r$ and called *rewrite rules*, and can be applied to a term t at any position where l matches as follows: find some subterm t' of t, that is, $t = c[t']$ for some *context* c, which is an instance of the left-hand-side term (lhs) of some rewrite rule $(\forall X)\ l \to r$, that is, $t' = \theta(l)$ for some variable assignment θ, and replace t' by $\theta(r)$ in t. This way, the term t can be continuously transformed, or rewritten. A pair (Σ, R), where R is a set of such oriented *rewrite rules*, is called a *rewrite system*. The corresponding term rewriting relation is written $\rightarrow_R$ and its inverse is written $\leftarrow_R$. If no rule in R can rewrite a Σ-term, than that term is said to be in *normal form* w.r.t. R.

Term rewriting can be used as an operational mechanism to perform equational deduction. Specifically, $E \vdash_\Sigma (\forall X)\ t = t'$ if and only if $t\ (\rightarrow_{R_E} \cup \leftarrow_{R_E})^*\ t'$, where R_E is the set of rewrite rules obtained by orienting all the equations in E from left to right. Therefore, term rewriting is as powerful as equational deduction if used in both directions. However, in practice term rewriting is used as a heuristic for equational deduction. A very common case is to attempt the task $E \vdash_\Sigma (\forall X)\ t = t'$ by showing $t\ \ \rightarrow^*_{R_E} \cdot \leftarrow^*_{R_E}\ \ t'$, i.e., by reducing both t and t' to some common term using (Σ, R_E). In some cases, for example when (Σ, R_E) is confluent and terminates, this becomes a semi-decision procedure for equational deduction.

There are many software systems that either specifically implement term rewriting efficiently, known also as *rewrite engines*, or that support term rewriting as part of a more complex functionality. Any of these systems can be used as an underlying platform for execution and analysis of programming languages defined using the technique and the formalism proposed in this paper. Without attempting to be exhaustive, we here only mention (alphabetically) some engines that we are more familiar with, noting that many functional languages and theorem provers provide support for term rewriting as well: ASF/SDF [27], CafeOBJ [6], Elan [3], Maude [5], OBJ [13], Stratego [28]. Some of these can achieve remarkable speeds on today's machines, in the order of millions or tens of millions of rewrite steps per second. Many engines store terms as directed acyclic graphs (DAGs), so applications of rewrite rules consist in many cases of just permuting pointers, which can indeed be implemented quite efficiently. In our language definitions, we often place or remove environments or continuations onto other structures; even though these may look like heavy operations, in fact these do nothing but save or remove pointers to subterms when these definitions are executed.

Because of the forward chaining executability of term rewriting and also because of these efficient rewrite engines, equational specifications are often called *executable*. As programming languages tend to be increasingly more abstract due to the higher speeds of processors, and as specification languages tend to be provided with faster execution engines, the gap between executable specifications and implementations, in case there has ever been any, is becoming visibly narrower. For example, we encourage the curious reader to *specify* the factorial operation equationally as follows (s is the Peano "successor")

$$0! = 1$$
$$s(N)! = s(N) * N!$$

in a fast rewrite engine like Maude, versus *implementing* it in programming languages like ML or Scheme. In our experiments, the factorial of 50,000, a number of 213,237 digits, was calculated in 18.620 seconds by the executable equational engine Maude and in 19.280 and 16.770 seconds by the programming languages ML and Scheme, respectively. In case one thinks that the efficiency of builtin libraries should not be a factor in measuring the efficiency of a system, one can define permutations equationally instead, for example as follows

Sorts and Subsorts

$\quad$ *Nat* < *Permutation* < *Permutations*

Operations

$\quad$ _ ,_ : *Permutation Permutation* → *Permutation [assoc]*

$\_ ; \_$: *Permutations Permutations* → *Permutations [assoc]*
perm : *Nat* → *Permutations*
insert : *Nat Permutations* → *Permutations*
map-cons : *Nat Permutations* → *Permutations*

Equations

$perm(1) = 1$
$perm(s(N)) = insert(s(N),\ perm(N))$
$insert(N,\ (P{:}Permutation\ ;\ Ps{:}Permutations)) = insert(N,P)\ ;\ insert(N,Ps)$
insert(N, (M,P')) = *(N,M,P') ; map-cons(M, insert(N,P'))*
insert(N, M) = *(N,M) ; (M,N)*
map-cons(M, (P ; Ps)) = *map-cons(M, P) ; map-cons(M, Ps)*
map-cons(M, P) = *(M,P)*

The above is an *order-sorted* equational specification; for readability, we used the *mixfix* notation for operation declarations (underscores are argument placeholders) which is supported by many rewrite engines. Also, we declared the list constructor operations with the attribute *[assoc]*. Semantically this is equivalent to giving the equation of associativity, but rewrite engines typically use this information to enable specialized algorithms for *rewriting* and *matching modulo associativity*; we will discuss matching modulo attributes like associativity, commutativity and identity in more depth shortly. Like in the previous example, we assumed some builtin natural numbers coming with a successor operation.

In our experiments with permutations, the executable equational specifications outperformed the implementations. 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 experiments have been performed on a 2.5GHz Linux machine with 3.5GB of memory, and we[2] used Maude 2.0, PolyML and PLT Scheme (specifically mzscheme), all providing libraries for large numbers. These simplistic experiments should by no means be considered conclusive; our measurements favoring executable specifications may be due to fortunate uses of data-structures in the Maude implementation, or even to our lack of usage of Scheme and ML at their maximum efficiency. While more extensive comparisons and analyses would be interesting and instructive, this is *not* our goal here; nor to unreasonably claim that executable specifications will ever outperform implementations. All we are trying to say is that the pragmatic, semantics-reluctant language designer, can safely regard the subsequent semantic definitions of language features as implementations, in spite of their conciseness and mathematical flavor.

NEXT-VERSION: discuss the examples with white/black balls and with bubble sorting

2.3 Rewriting Logic

While equational logic and its execution via term rewriting provide as powerful computational properties as one can get in a sequential setting, these were *not* designed to specify or reason about *concurrent systems*. The initial algebra model of an equational specification collapses all the computationally equivalent terms, but it does not say anything about *evolution* of terms under concurrent transitions. There are at least two broad directions of research on the subject of specifying non-deterministic and/or concurrent systems [29]. One builds upon the Platonist belief that

[2] Warmest thanks to Mark Hills who helped with these experiments.

models *are* deterministic, but, by making use of *underspecification*, one never knows precisely in which model one is[3]. While underspecification is a very powerful approach in semantics, in our programming language definitional framework it suffers from a crucial impediment: it is *not executable enough* to allow one to execute actual concurrent programs. Nevertheless, we will make intensive use of underspecification by *not* specifying implementation details of programming languages (such as how environments, stores, lists or stacks are implemented). Another direction of thought in concurrency is to allow *non-determinism in models*, typically by means of non-deterministic transitions. Specifications become executable now, because all what one needs to do is to randomly pick some transition when more are possible. This is similar to what thread/process schedulers do in concurrent computer systems.

To properly define and analyze (concurrent) programming languages formally, we need a framework which provides *natural* support for concurrency. In other words, we would like a framework in which we can state what programming language features are meant to do *without* artificial encodings of these due to artifacts of the underlying definitional framework. For example, we would consider it "unnatural" to define, or simulate, a particular "thread or process scheduler" in order to define a concurrent language, just because the underlying definitional framework is inherently sequential. Since both underspecification and non-deterministic transitions seem important for capturing the meaning of programming language features, we would like an underlying framework that supports both.

Rewriting logic [17] is a logic for concurrency, which should not be confused with term rewriting. A *rewrite specification*, or *theory*, is a triple (Σ, E, R), where (Σ, E) is an equational specification and R is a set of *rewrite rules*. Rewriting logic therefore extends equational logic with rewrite rules, allowing one to derive both equations and rewrites (or transitions). Deduction remains the same for equations, but the symmetry rule is dropped for rewrite rules. Models of rewrite theories are (Σ, E)-algebras enriched with transitions satisfying all the rewrite rules in R. In our context, the equational part of a rewrite theory is allowed to "underspecify" features as far as the specification remains executable. Interestingly, rewrite theories also have initial models, consisting of term models factored by the equational derivability relation and enriched with appropriate transitions between the equational equivalence classes. They also follow the slogan "no junk, no confusion", but extend it also w.r.t. reachability of terms via transitions.

Rewriting logic is a framework for *true concurrency*. The reader interested in details is referred to [17]. We here only discuss this by means of examples. Suppose that (Σ, E, R) is the following rewrite theory:

Σ:

sort $State$
operation $\emptyset\ : \to State$
operation $\_\ \_\ :\ State \times State \to State$
operation $0\ : \to State$
operation $1\ : \to State$

E:

equation $(\forall S : State)\ \emptyset\ S = S$

[3] There is some resemblance here to the idea of parallel worlds promoted by Giordano Bruno 400+ years ago in his book "The Infinity, the Universe and Its Worlds". He was sentenced to death by fire by the Inquisition for his "heretic" thinking.

equation $(\forall S_1, S_2 : \mathit{State})\ S_1\ S_2 = S_2\ S_1$
equation $(\forall S_1, S_2, S_3 : \mathit{State})\ (S_1\ S_2)\ S_3 = S_1\ (S_2\ S_3)$

R:

rule $r_1\ :\ 0 \Rightarrow 1$
rule $r_2\ :\ 1 \Rightarrow 0$

The two equations state the associativity and commutativity of the binary "_ _" operator, thus making it a multi-set operator, and the two rules flip the two constants 0 and 1. If one starts with an initial term as a multi-set containing 0 and 1 constants, then the rules r_1 and r_2 can apply *concurrently*. For example, if the initial term is the multi-set 0 1 0 1 0 then three instances of r_1 and two of r_2 can apply in parallel and transform the multi-set into 1 0 1 0 1. Note, however, that there is no requirement on the number of rewrites applied concurrently; for example, one can apply only the instances of r_1, or only one instance of r_1 and one of r_2, etc. Consequently, on a multi-set of 0 and 1 constants, the rewrite theory above manifests all possible concurrent behaviors, not only those following an interleaving semantics. And indeed, if one "executes" this specification on a machine with an arbitrarily large number of processors, then one can observe any of these concurrent behaviors. Parallel execution of rewrite logic specifications is an important aspect of our language definitions; we will rediscuss it.

One can regard a rewrite logic specification as a compact means to encode transition systems, namely one that has the capability to generate for any given term a transition system manifesting all its "concurrent" behaviors. The states of that transition system are the terms to which the original term can evolve by iterative applications of rewrite rules; the equations are used to keep the states in canonical forms (in this case as AC multi-sets, but in general as any terms which are not reducible by applying equations from left to right – modulo particular axioms, such as ACI) and the rules are used to generate transitions (in this case the rules are not parametric, but in general they can have variables, each rule corresponding to a recursively enumerable set of ground transitions).

Given a rewrite logic specification and an initial term, the corresponding transition system may or may not need to be generated explicitly. For example, if one is interested in one execution of the specification on that term, then one only needs to generate one path in the transition system. If one is interested in testing whether a particular term can be reached (reachability analysis), then one can only generate the transition system by need, for example following a breadth-first strategy. However, if one is interested in checking some complex property against all possible executions (model-checking) then one may need to generate the entire transition system. Interestingly, one can regard a concurrent programming language also as a means to encode transition systems, namely one taking a program and, depending upon the intended purpose, generate one path, part of, or the entire transition system comprising all behaviors of that program. Thus, that programming languages can be given a rewriting logic semantics should come as no surprise. What may seem surprising in the sequel is the simplicity of such language definitions when one uses the K framework.

All rewrite engines, by their nature, generate one (finite or infinite) path in the transition system of a term when requested to reduce that term. Therefore, we can use any of these rewrite engines to execute our K specifications, thus converting them into interpreters for the programming languages defined in K. The Maude system also supports breadth-first exploration of the state space of the transition system of a term, as well as linear temporal logic (LTL) model checking. Using these features one can, for example, show that in the example above it is possible to start with the state of zeros 0 0 0 0 0 and reach a state of just ones; also, using the model checker one can show that it is *not* the case that whenever one reaches the state of zeros then one will eventually reach a state of ones. Indeed, there are infinite executions in which one can reach the state of zeros and then never reach the state of ones. Appendix A shows how these formal analyses can be performed in Maude. As one may expect, this capability of rewrite logic systems to explore the state space of the transition system associated to a term will allow us to obtain corresponding analysis tools for the programming languages that we will define as rewrite logic theories in K.

If rewrite rules can apply concurrently and in as many instances as the term to rewrite permits, then how can one attain *synchronous* applications of rules? How can one simulate situations in which one wants each application of a rule to be an atomic action, happening only one at a time? As usual, this can be achieved by introducing "synchronization objects" and making sure that each rule intended to synchronize grabs the synchronization object. In the example above, we can, e.g., introduce a new constant $\$: \to State$ and replace the two rewrite rules by

$$\begin{array}{ll} \mathsf{rule} & r_1^{\$} \; : \; 0, \$ \Rightarrow 1, \$ \\ \mathsf{rule} & r_2^{\$} \; : \; 1, \$ \Rightarrow 0, \$ \end{array}$$

and make sure that the multi-set to rewrite contains precisely one constant $. Rewrite rules can apply in parallel on a term only if that term can be matched a *multi-context* with one hole per rule application, the subterm corresponding to each hole further matching the left-hand-side (lhs) of the corresponding rule. In particular, that means that rule instances whose lhs's overlap *cannot* be applied in parallel. Since the rules above overlap on the "synchronization object" $, they can never be applied concurrently. Note that the equations are being applied "silently" in the background, to permute the constants in a way that rules can apply. Indeed, the role of equations is to generate *equivalence classes* on which rewrite rules can apply and thus transit to other equivalence classes, etc. Rewrite engines provide heuristics to choose a "good representative" of each equivalence class, typically by applying equations as rewrite rules until a normal form is obtained, potentially modulo associativity and/or commutativity.

Therefore, the underlying execution engine has the possibility to *non-deterministically* pick some rule application and thus disable the applications of other rules that happen to overlap it. In practice, rewrite logic theories contain both synchronous and asynchronous rules. In particular, our language definitions will contain asynchronous rules for thread local computations and synchronous rules for thread interactions; for example, reading/writing shared variables is achieved with rules that synchronize on the *store* (mapping *locations* to *values*). Thus, if one executes a concurrent program on a multi-processor rewrite engine in its programming language rewrite logic definition, one can obtain any of the possible (intended or unintended) concurrent behaviors of that program.

Together with concurrency and synchronization, the problem of deadlocking is almost unavoidable. Let us next show how deadlocking can appear in a rewrite logic context by means of a classical example, the dining philosophers. We can encode all the philosophers and the forks as elements in a set called "state", define equations to keep the state in a canonical form and to capture actions that need not split the state space, such as releasing forks, and define rules to capture those actions that split the state space, in our case the operation of acquiring a fork. Let us next define one possible rewrite logic theory specifying the dining philosophers problem and show by reachability analysis that it can, indeed, lead to a deadlock. We start by defining the state as a (multi-)set:

$$\begin{array}{ll} \mathsf{sort}\ State & \\ \mathsf{operation} & \emptyset \; : \; \to State \end{array}$$

operation $\_\ \_ \ :\ \mathit{State} \times \mathit{State} \to \mathit{State}$

equation $(\forall S : \mathit{State})\ \emptyset\ S = S$

equation $(\forall S_1, S_2 : \mathit{State})\ S_1\ S_2 = S_2\ S_1$

equation $(\forall S_1, S_2, S_3 : \mathit{State})\ (S_1\ S_2)\ S_3 = S_1\ (S_2\ S_3)$

When using rewrite logic systems, e.g., Maude, one would replace the three standard equations above by operation attributes. Let us next add the two important constructors for states, philosophers and forks. We identify philosophers and forks by natural numbers, assuming builtin numbers and integers. Note that a philosopher can keep some part of the state, namely a set of forks; we will make sure, by corresponding equations and rules, that philosophers will hold at most two forks:

operation $\mathit{ph}\ :\ \mathit{Nat} \times \mathit{State} \to \mathit{State}$

operation $\$\ :\ \mathit{Nat} \to \mathit{State}$

The following (uninteresting) operations and equations declare an initial state, in this case of 10 philosophers; one can change n to any other number:

operation $n\ :\ \to \mathit{Nat}$

equation $n = 9$

operation $\mathit{init}\ :\ \to\ \mathit{State}$

operation $\mathit{init}\ :\ \mathit{Nat} \to\ \mathit{State}$

equation $\mathit{init} = \mathit{init}(n)$

equation $\mathit{init}(-1) = \emptyset$

equation $(\forall N : \mathit{Nat})\ \mathit{init}(N) = \mathit{ph}(N, \emptyset)\ \$(N)\ \mathit{init}(N-1)$

We are now ready to give the three *rules* defining the actions that philosophers can perform, namely grabbing one of their neighbor forks:

rule $(\forall N : \mathit{Nat},\ \mathit{Fs} : \mathit{State})\ \mathit{ph}(N, \mathit{Fs})\ \$(N) \Rightarrow \mathit{ph}(N, \mathit{Fs}\ \$(N))$

rule $(\forall N : \mathit{Nat},\ \mathit{Fs} : \mathit{State})\ \mathit{ph}(s(N), \mathit{Fs})\ \$(N) \Rightarrow \mathit{ph}(s(N), \mathit{Fs}\ \$(N))$

rule $(\forall N : \mathit{Nat},\ \mathit{Fs} : \mathit{State})\ \mathit{ph}(0, \mathit{Fs})\ \$(N) \Rightarrow \mathit{ph}(0, \mathit{Fs}\ \$(n))$ if $N = n$

Assuming that the action of eating as well as that of releasing *both* forks when finished eating are local actions without involving any "competition" among the philosophers (these would happen anyway, regardless of the external environment), we can capture both these actions with just one equation:

equation $(\forall N, X, Y : \mathit{Nat})\ \mathit{ph}(N, \$(X)\ \$(Y)) = \mathit{ph}(N, \emptyset)\ \$(X)\ \$(Y)$

One can now use the rewrite logic theory above to generate a transition system comprising all the behaviors that can result from its initial state, the term *init*. That transition system will have equational equivalence classes as states and instances of the three rules above as transitions. It is easy to see that that transition system indeed encodes all the behaviors of the dining philosophers' problem, and also that it has a finite number of states. Using generic formal analysis tools for rewrite logic specifications, such as reachability analysis, one can show that there are precisely two scenarios in which the concurrent system above deadlocks. All one needs to show is that one can reach a state in which no rule can be applied anymore. Appendix B shows how such an analysis can be performed in Maude. For 10 philosophers, for example, Maude takes 2.7 seconds on a 2.5GHz/3.5GB to explore the entire state space of 15,127 states and find the two deadlock solutions. For 14 philosophers it takes Maude about 400 seconds to explore all the 710,647 states (Maude crashed when tried on 15 philosophers).

As already mentioned in the introduction, many theoretical frameworks for concurrency have been translated into rewriting logic [16], such as π-calculus, process algebra, actors, etc., suggesting that rewriting logic may be regarded as a general, universal logic for concurrency. However, one must treat this generality with caution; "general" and "universal" need not necessarily mean "better" or "easier to use", for the same reason for which machine code is not better or easier to use than higher level programming languages that translate into it. In our context of defining (concurrent) programming languages in rewriting logic, the question is whether rewriting logic provides a natural framework for this task or not, and whether we get any benefit from using it. This paper aims at giving a positive answer to this question, by introducing a definitional technique and syntactic sugar domain-specific notations for equations and rewrite rules that will make our definitions more readable and more modular.

Since both equations and rewrite rules in rewriting logic are executed as rewrite rules in the corresponding term rewriting system, an immediate benefit from defining programming languages in rewriting logic is that any of the existing rewrite engines gives an *interpreter for free*. Since some of these engines are quite fast, the obtained language interpreters can also be quite fast. Since rewriting logic is a *computational logical framework*, "execution" of programs becomes logical deduction. That means that one can formally analyze programs or their executions directly within the semantic definition of their programming language. In particular, executions can be regarded as proofs, so one can log and check them, thus obtaining a framework for *certifiable execution* of programs. Moreover, generic analysis tools for rewrite logic specifications can translate into analysis tools for the defined programming languages. For example, Maude provides a BFS reachability analyzer and an LTL model checker for rewrite logic specifications (or better say for their initial models); these translate immediately into corresponding BFS reachability analysis and LTL model checking tools for the defined languages, also *for free*. Additionally, we believe that these language specifications can be used *as are* to synthesize very efficient, correct by construction, interpreters and even compilers for the defined languages. Semantically, as stated in [18], *rewriting logic semantics* unifies algebraic denotational semantics and structural operational semantics: from a denotational perspective, the initial model of a language specification can be regarded as the *canonical* model of the language, comprising all computationally equivalent programs as well as all their concurrent behaviors[4]; from an operational perspective, the initial model is executable via rewriting at no additional effort.

[4] Note, however, that unlike other favorite (non-concurrent) models in denotational semantics, the initial models do *not* collapse all nonterminating programs into one element ($\perp$), but only those that can be shown equal using the equational axioms of the language. It is important to note that *induction* is a valid proof principle in the initial models, so our semantics is also amenable to theorem proving.

3 Related Work

There is much related work on defining programming languages in various computational logical frameworks, including term rewriting. In this version of the report we do not discuss these in depth, but in a future version each relevant related work will comprehensively be compared to K in a designated subsection.

The first extensive study on defining a programming language equationally, with an initial algebra semantics, seems to be [9]; there, OBJ [13] was used to execute the language specifications via term rewriting. Interesting work in not only defining languages by term rewriting but also in compiling those has been investigated under the ASF+SDF project [27]. Stratego [28] is a program transformation framework based also on term rewriting. What makes our work different from other language definitional works based on rewriting is precisely the use of a first-order representation of continuations and of ACI matching, which turn out to have a crucial effect on the compactness and simplicity of definitions.

The *rewriting logic semantics* project [19, 18] aims at providing a framework unifying algebraic denotational semantics and structural operational semantics (SOS) (as well as other operational variants, such as reduction semantics and evaluation contexts). We have shown in [19, 18] how one can use the Maude language to define toy languages, following also a continuation-based semantics. However, those toy languages missed some of the interesting features of FUN, such as callcc, and their definition was specific to Maude. In this paper we define a *domain-specific*, Maude-independent specification language for programming languages, called K, which eases enormously the task of defining a language; it has an intuitive notation and can be understood independently from rewriting logic.

There is some similarity between our approach and monads [15, 20]. The monad approach gains modularity by using monad transformers to lift program constructs from one level of specification to a richer one. In our case, modularity is achieved by the use of ACI-matching and context transformers, which allow selecting from the state "soup" only those attributes of interest. In fact, the complete enumeration of the state attributes is done only once, when defining the "eval" command.

3.1 Structural Operational Semantics (SOS)

NEXT-VERSION: both small- and big-step

3.2 Modular Structural Operational Semantics (MSOS)

NEXT-VERSION: describe MSOS here; refer to Section 6.3 for the translation of MSOS into K.

3.3 Reduction Semantics and Evaluation Contexts

NEXT-VERSION: the continuation can be regarded as an instantiated, reversed evaluation context: first element in the continuation is the expression placed in the hole, while the rest of the continuation is the "evaluation context". Therefore, we have a framework here where continuations and evaluation contexts are the same thing!

3.4 Rewriting Logic Semantics

NEXT-VERSION: not modular enough; context transformers bring additional modularity to K*.*

3.5 Abstract State Machines

NEXT-VERSION: establish the operational relationship between ASMs and rewriting logic; operationally, both can be regarded as a program which is applied iteratively (a loop at the top): in ASMs the program is written in a simple programming language which looks rather conventional, while in rewriting the program is "match-and-apply any rule".

3.6 Logic Programming Semantics

NEXT-VERSION: explain this approach and compare it with equational and rewriting logic. If one makes abstraction of the operational aspects, both the logic programming approach and the equational logic one are semantically Horn clauses. However, if one uses K *within equational logic then one needs no conditional equations.*

3.7 Monads

NEXT-VERSION: Due to our use of continuations, there is some remote relationship between K *and monads. However, monads are denotational, while* K *is essentially operational, though it also comes with a complete model theory; understand and explain the relationship between these two worlds. Also, monads fail to be fully modular, at least in the strong sense in which* K *is.*

3.8 SECD Machine

NEXT-VERSION:

4 The K-Notation

> *A good notation sets the mind free to think about really important things.*
>
> Alfred North Whitehead

In this section we introduce the K-notation, suitable to define rewrite systems in which the term(s) to rewrite involve operations that are associative and/or commutative and/or have identities. It consists of a series of notational conventions for matching modulo axioms, for unnecessary variables, for sort inference, and most importantly, for *context transformers*. Combined with the definitional technique presented in Section 5, the K-notation will allow us to develop compact, modular and intuitive definitions for programming language features.

4.1 Matching Modulo Associativity, Commutativity, Identity

In previous examples we have mentioned that equations like associativity, commutativity and identity can and should be declared as operation attributes. While semantically equivalent to their corresponding equations, the operation attributes not only allow a more compact specification of common properties, but also tell the underlying rewrite system to enable specialized matching algorithms. *Matching*, or the process of finding a substitution that makes a term with variables equal to a term without variables, is a core operation of any rewrite engine.

Matching modulo any equational theory is undecidable, because it can be ultimately reduced to arbitrary (ground) equational semantic entailment. However, matching modulo certain equational axioms, such as *associativity* (A), *commutativity* (C), and *identity* (I), also known as *ACI-matching*, is decidable. In spite of its intractability [14], ACI-matching tends to be relatively efficient in practical situations. Consequently, many rewrite engines support it in its full generality. Some rewrite engines only support AI-matching; our K-notation and our language definitions can be modified to only require AI-matching, though they would not be as compact and easy to read. ACI-matching leads to compact and elegant, yet efficient specifications. In what follows we discuss some uses of ACI-matching and our notational conventions.

Since different languages have different ways to state that certain binary operations are associative and/or commutative and/or have identities, to keep the discussion simple and generic we assume that all the ACI operations are written using the *mixfix*[5] concatenation notation "$\_\ \_$" and have identity "$\cdot$", while all but one of the AI operations use the comma notation "$\_,\_$" and have identity written also "$\cdot$". The exception to the comma notation for AI operations is the *continuation*, which, just for reading convenience, uses the notation $\_ \curvearrowright \_$; our encoding and use of continuations will be discussed in Section 5.3. In particular implementations of our subsequent specifications, either to avoid notational confusion or because the underlying system does not allow operator name overloading, one may want to use different names for the different ACI or AI operations. ACI operations correspond to multi-sets, while the AI operations correspond to lists. Therefore, for any sort *Sort*, we take the liberty to tacitly add supersorts "*SortSet*" and "*SortList*" of *Sort*, constant operations "$\cdot\ :\ \to \mathit{SortSet}$" and "$\cdot\ :\ \to \mathit{SortList}$", and ACI operation "$\_\ \_\ :\ \mathit{SortSet} \times \mathit{SortSet} \to \mathit{SortSet}$" and AI operation "$\_,\_\ :\ \mathit{SortList} \times \mathit{SortList} \to \mathit{SortList}$" of identities "$\cdot$", respectively. We also assume supersorts "*SortNeSet*" and "*SortNeList*" for non-empty sets and lists of elements of sort

[5] The mixfix notation is supported by most languages in the OBJ family; underscores "$\_$" stand for arguments.

Sort whenever needed. Appendix C shows how these operations can be defined in Maude; similar definitions are possible in other algebraic specification or rewrite engines.

In our language definitions, ACI operations will be used to define state data-structures as "soups" of state attributes. For example, the state of a programming language is typically a "soup" containing a store, locks which are busy, an input buffer, an output buffer, etc., as well as a set of threads. Soups can be nested; for example, a thread may itself contain a soup of thread attributes, such as an environment, a set of locks that it holds, several stacks (for functions, exceptions, loops, etc.); an environment is further a soup of pairs (variable,location), etc. Lists will be used to specify structures where the order of the attributes matters, such as buffers (for input/output), parameters of functions, etc.

As an example of how ACI matching works, let us consider defining equationally an operation $\mathit{update} \ : \mathit{Environment} \times \mathit{Var} \times \mathit{Loc} \to \mathit{Environment}$, where *Environment* is (an alias for) the set sort *VarLocSet* associated to a pairing sort *VarLoc* with one constructor pairing operation $(\_,\_) \ : \ \mathit{Var} \times \mathit{Loc} \to \mathit{VarLoc}$. The meaning of $\mathit{update}(\mathit{Env}, X, L)$ is that the resulting environment is the same as *Env* except in the location of X, say L', which should be replaced by L. With ACI-matching, this can be defined with just one equation,

$$\begin{array}{l}(\forall X : \mathit{Var};\ L, L' : \mathit{Loc};\ \mathit{Env} : \mathit{Environment}) \\ \qquad\qquad \mathit{update}((X, L')\ \mathit{Env},\ X,\ L) = (X, L)\ \mathit{Env}.\end{array}$$

The ACI-matching algorithm "knows" that the first argument of *update* has an ACI constructor, so it will be able to match the lhs of this equation even though the pair (X, L') does *not* appear on the first position in the environment. Note that the *update* operator defined above works only when the variable X was already in the environment. In our language definitions we prefer to work with a slightly different *update* operator, one which will add the pair (X, L) to the environment in case X was not already assigned a location in the environment.

4.2 Sort Inference

The equation in the previous section defining *update* contains three parts: variable declarations, a lhs and a rhs. Surprisingly, the variable declarations take almost half the size of the entire equation. In fact, it was often the case in our experiments with implementing the K-technique in Maude that variable declarations take a significant amount of space, sometimes more than half the space of the entire language specification. However, in most cases *the sorts of variables can be automatically inferred from the context*, that is, from the terms involved in the equation or rule. To simplify this process, we make the assumption that all variable names start with a capital letter (this Prolog-like convention is tacitly followed by many rewrite programmers).

Consider, for example, the two terms of the equation of *update* in the previous section, namely $\mathit{update}((X, L')\ \mathit{Env},\ X,\ L)$ and $(X, L)\ \mathit{Env}$. Since the signature of *update* is $\mathit{Environment} \times \mathit{Var} \times \mathit{Loc} \to \mathit{Environment}$ and since X and L appear as its second and third arguments, one can immediately infer that their sorts are *Var* and *Loc*, respectively. Moreover, since the first argument of *update* and/or the rhs have the sort *Environment*, and since environments are constructed using the multiset operation $\_\,\_ : \mathit{Environment} \times \mathit{Environment} \to \mathit{Environment}$, one can infer that the sort of *Env* is *Environment*.

Because of subsorting, a variable occurring on a position in a term may have multiple sorts. For example, the variable *Env* above can have both the sort *Environment* (which, as already mentioned, is nothing but *VarLocSet*) and the sort *VarLoc*, as well as the sort *VarLocNeSet*; however, note that there is only one largest sort among the three, namely *Environment*. If an occurrence of a variable can have multiple sorts, we assume by default, or by convention, that that variable occurrence has *the largest* sort among those sorts that it can have. If there is no such largest sort then we assume variable sorting to be "ambiguous"; if variable sorting is ambiguous for this reason, then most likely the order-sorted signature has problems which would lead to ambiguous parsing anyway, so these problems would probably be flagged by the parser.

This "largest sort" convention corresponds to the intuition that we assume the "least" information about each variable occurrence; indeed, the more concrete the (sub)sort of a variable, the "more information" one has about that variable. If the same variable appears on multiple positions then we infer for that variable the "most concrete" sort that it can have among them. Technically, this is the intersection of all the inferred largest sorts on the different positions where the variable appears; if there is no such intersection sort then, again, we declare the sorting process of that variable ambiguous.

If the variable sort-inference process is ambiguous, or if one is not sure, or if one really wants a different sort than the inferred one, or even simply for clarity, one is given the possibility to sort terms, in particular variables, "on-the-fly": we append the sort to the term or variable using ":", e.g., $t : \mathit{Sort}$ or $Z : \mathit{Sort}$. For example, from the term $\mathit{update}(\mathit{Env}, X, L)$ one can only infer that the sort of *Env* is *Environment*, the most general possible under the circumstances. If for some reason (though we do not see any reason in this particular case) one wants to refer to a "special" environment of just one pair, then one can write $\mathit{update}(\mathit{Env}\colon \mathit{VarLoc}, X, L)$.

Variable sort inference in the context of order sorted signatures is an interesting and challenging problem, that we only partly address here. Also, syntactic criteria for non-ambiguous sort inference seem to be quite useful and non-trivial. The complexity of sort inference is yet another interesting and apparently non-trivial problem. We hope that all these problems will be addressed in the near future, and that an efficient variable sorting algorithm will be implemented as part of a system supporting the K framework. For the time being we use sort inference just as a convenient way to avoid writing long and heavy equations and rules. For now, the reader interested in defining languages using the K framework is expected to perform the sort inference process "in mind" to deduce the sorts of all the variables, and then to declare them appropriately in a rewrite system. The name of variables can also serve as a hint to the user. Indeed, there is not much doubt that *Env* has sort *Environment*, or that L has sort *Loc*. One may even imagine a sorting convention in which the root of a variable name (i.e., after dropping prime or digit suffixes) is a prefix of a sort name or its alias; we prefer to postpone this issue until an implementation of K takes shape.

4.3 Underscore Variables

With the sort inference conventions, the equation defining the operation *update* can therefore be written as

$$\mathit{update}((X, L')\ \mathit{Env},\ X,\ L) = (X, L)\ \mathit{Env}.$$

Note that the location L' that occurs in the lhs is not needed; it is only used for "structural" purposes, i.e., it is there only to say that the variable X is allocated at some location, but we do not care what that location is (because we change it anyway). Since this will be a common phenomenon in our language definitions, we take the liberty to replace unnecessary letter variables by underscores, like in Prolog. Therefore, the equation above can be written

$$\mathit{update}((X, \_)\ \mathit{Env},\ X,\ L) = (X, L)\ \mathit{Env}.$$

Let us consider a more complex and yet very common situation, namely (here only a fragment of) the definition of variable lookup in a multi-threaded language. In our language definitions, many state attribute "soups" will be wrapped with specific operators to keep them distinct from other soups. For example, environments will be wrapped with an operation such as $\mathit{env} : \mathit{Environment} \to \mathit{ThreadAttribute}$ before they are placed in their threads' state attribute soup; the threads' state is further wrapped with an operation $\mathit{thread} : \mathit{ThreadAttributeSet} \to \mathit{StateAttribute}$. Thus, with the underscore conventions discussed in this section, if we want to find the location of a variable X in the environment of a thread, then we match the thread against the term

$$\mathit{thread}(\mathit{env}((X, L)\ \_)\ \_)$$

and find the desired location L (assume that X is already instantiated to some concrete program variable); in this "pattern" term, the first underscore variable matches the rest of the environment, while the second matches the rest of the thread attributes (such as the resources that thread holds, its various stacks, its continuation, etc.).

4.4 Tuples

Like we need to pair variables and locations to create environments, we will often need to tuple two or more terms in order to "save" current information for later processing. This is similar to "record" data-types in conventional programming languages. Most functional languages provide tuples as core data-types. However, algebraic specification and rewriting logic do not have builtin tuples, so they need to be defined; their definition is, as expected, straightforward. To save space and increase the readability of our language definitions, we avoid defining all these tupling operations. Like the sorts of variables, their arities can also be inferred from the context. Concretely, if the term $(X_1 : \mathit{Sort1},\ X_2 : \mathit{Sort2},\ \ldots,\ X_n : \mathit{Sortn})$ appears in some context (the variable sorts may be inferred from the remaining context using the techniques above), then we implicitly add to the signature the sort *Sort1Sort2...Sortn* and the operation

$$(\_,\_,\ldots,\_) : \mathit{Sort1} \times \mathit{Sort2} \times \cdots \times \mathit{Sortn} \to \mathit{Sort1Sort2...Sortn}.$$

To keep the number of tupling operations small and to also avoid parsing conflicts, if two or more tupling terms appear in a specification with the same number of arguments and the sorts of the arguments of one are larger than all the others, then only one tupling operation is added, the one for the larger sorts. In other words, if tuple terms $(X_1 : \mathit{Sort1},\ X_2 : \mathit{Sort2},\ \ldots,\ X_n : \mathit{Sortn})$ and[6] $(X'_1 : \mathit{Sort1'},\ X'_2 : \mathit{Sort2'},\ \ldots,\ X'_n : \mathit{Sortn'})$ are encountered and $\mathit{Sort1'} < \mathit{Sort1}$, $\mathit{Sort2'} < \mathit{Sort2}$, ..., $\mathit{Sortn'} < \mathit{Sortn}$, then only one operation is implicitly added to the signature, namely

$$(\_,\_,\ldots,\_) : \mathit{Sort1} \times \mathit{Sort2} \times \cdots \times \mathit{Sortn} \to \mathit{Sort1Sort2...Sortn}.$$

4.5 Contextual Notation for Equations and Rules

All the subsequent equations and rules will apply on just one (large) term, encoding the state of the program; note that our state is rather generous, containing everything needed to execute the entire program, including the program itself. Specifically, most of the equations and rules will apply on

[6] Recall that in order-sorted algebraic specification, "<" means "less than or equal to"; indeed, if $s < s'$ for sorts s and s', models in which the carriers of s and s' are equal are also accepted.

subterms selected via matching, but only if the structure of the state permits it. In other words, most of our equations/rules will be of the form

$$C[t_1]\cdots[t_n] = C[t'_1]\cdots[t'_n] \qquad \text{or} \qquad C[t_1]\cdots[t_n] \Rightarrow C[t'_1]\cdots[t'_n]$$

where C is some context term with n "holes" and t_1, ..., t_n are subterms that need to be replaced by t'_1, ..., t'_n in that context. Let us temporarily make no distinction between equations and rules; we will return to this important distinction shortly. The context C needs not match the entire state, but nevertheless sometimes it can be quite large. To simplify notation and ease reading, we prefer to use the notation

$$C[\underset{t'_1}{\underline{t_1}}]\cdots[\underset{t'_n}{\underline{t_n}}]$$

and call them all, generically, *contextual rules.* This notation follows a natural intuition: first write the state context in which the transformation is intended to take place, then underline what needs to change, then write the changes under the lines. Our contextual notation above proves to be particularly useful when combined with the "_" convention for variables: if "_" appears in a context C and it is not underlined, then it means that we do not care what is there (but we keep it unchanged).

As an example, let us consider the following common rule for variable lookup in the context of a concurrent language (see also the second example in Section 4.3), already simplified using all the previously discussed notational conventions:

$$\begin{array}{l} \mathit{thread}(k(X \curvearrowright K)\ \mathit{env}((X,L)\ \mathit{Env})\ \mathit{TS})\ \mathit{store}((L,V)\ \mathit{Store}) \quad \Rightarrow \\ \mathit{thread}(k(V \curvearrowright K)\ \mathit{env}((X,L)\ \mathit{Env})\ \mathit{TS})\ \mathit{store}((L,V)\ \mathit{Store}). \end{array}$$

The only difference between the two terms involved in the rule above is that the variable X at the top of the continuation in the left term is replaced by V. The sorts of X, L, V and *Env* can be readily inferred as discussed in Section 4.2. Also, the sort of *Store* and the sort of *TS* can be inferred similarly to the sort of *Env*; the sort of K, *Continuation*, can also be inferred similarly, noting that $\_ \curvearrowright \_$ is the list constructor for continuations (see Section 5.3). With the additional conventions in this section, the rule above can be transformed into the following contextual rule, which avoids repeating state fragments and inventing names for variables that do not change (but need to be there just to keep the above a well-formed rewrite rule):

$$\mathit{thread}(k(\underset{V}{\underline{X}} \curvearrowright \_)\ \mathit{env}((X,L)\ \_)\ \_)\ \mathit{store}((L,V)\ \_).$$

Our continuation-based language definitional technique, the K-technique proposed in the next section, makes the control/conditional context of a definition explicit as data context. A natural consequence of this definitional style is that the need for conditional rules almost disappears. In the very rare case that a condition is needed, then we use the expected notation:

$$C[\underset{t'_1}{\underline{t_1}}]\cdots[\underset{t'_n}{\underline{t_n}}] \ \Leftarrow\ b$$

In our case studies on defining programming languages using K, we only needed very few conditional contextual rules. No conditional contextual rules were needed for most languages, some of them relatively complex. This should not appear to be surprising, though: [25] shows how a continuation-passing-style transformation applied automatically to a confluent conditional rewrite system[7] can systematically eliminate all the conditional rules by replacing them with computationally equivalent unconditional ones. The conditions that we needed were all trivial *boolean* conditions, so we do not bother here to introduce notation for various types of conditions of equations or rules in rewriting logic, such as equations, memberships, rules, shortcuts, matching in conditions, etc.

4.6 Structural, Computational and Non-Deterministic Contextual Rules

Let us now return to the important distinction between equations and rules in rewriting logic, and discuss how we accommodate it in our notation. A careful analysis of several languages that we defined as rewrite logic theories led us to the conclusion that, in fact, one wants to distinguish among *three* kinds of contextual rules, not only between two (as rewriting logic would suggest by its distinction between equations and rules):

- *Structural*, or *non-computational.* These are rules whose only role is to allow modifications of the structure of the state, so that other contextual rules that have computational meaning can apply. For example, the contextual rules that transform a program into a continuation term, or the rules for creating the initial state, are all structural, their role being to "prepare" the expression for evaluation. These structural rules have absolutely no computational meaning. Intuitively, if one thinks of a programming language definition as a means to generate "computations" (i.e., sequences of execution steps) from programs, making complete abstraction of state or syntax representation, then the structural rules are intended to generate *no execution steps* in computations. Even though these rules may be executed by rewriting as well by most rewrite engines, their corresponding rewrite steps are not visible as part of the computation, in a similar way that equations in executions of rewrite logic specifications are not visible as transition steps in models. We can think of structural contextual rules as rules "modulo" which computations take place. We will use the following notation for these rules

  $$C[\underset{t'_1}{\underset{\cdots\cdots}{t_1}}]\cdots[\underset{t'_n}{\underset{\cdots\cdots}{t_n}}],$$

  whose lines underlying subterms that change are dotted, signifying that these rules are "lighter" than the other rules whose lines are full. As expected, when K is translated into rewriting logic, structural contextual rules always translate into equations. Also, all the set and list implicit associativity and commutativity equations are structural, having no computational meaning either.

- *Computational.* These are all the other contextual rules; we use the notation

  $$C[\underset{t'_1}{\underline{t_1}}]\cdots[\underset{t'_n}{\underline{t_n}}]$$

  for them, where the lines underlying subterms that change are full. Each of these rules captures precisely one intended execution step of the program. When we translate K into

[7] Definitions of non-concurrent non-deterministic languages are expected to be indeed confluent.

rewriting logic, depending on the desired abstraction and the purpose of the translation, these computational rules can be translated either into equations or into rewrite rules. If the language is sequential and deterministic and if one's interest is in capturing an algebraic denotational semantics of it (an initial algebra semantics where the denotation of each term is the value that it evaluates to), then all these computational rules can be translated into equations. If one's interest is in precisely capturing a dynamic semantics of the language that reflects all the computation steps that take place during any execution, then one should translate all the computational contextual rules into rewrite rules. While this appears to semantically be the most appropriate choice, it may not always be practical, because the resulting executions may be too detailed. For example, if for certain common analysis purposes, such as model checking of concurrent programs, one is deliberately not interested in steps that do not change the concurrent behavior of a program, such as initializations of function arguments, or break/continue of loops, or throwing exceptions, etc., then one can translate their corresponding contextual rules into equations. Recall that the initial model of a rewrite theory collapses all states that are provably equal by equational reasoning, so a significant state-space reduction can result as a consequence of such *equational abstractions*; experiments confirm this fact.

- *Non-deterministic.* Non-deterministic rules are always a (typically quite small) subset of the computational rules. To distinguish them from the other computational rules, we box them:

$$\boxed{C[\underset{t_1'}{\underline{t_1}}]\cdots[\underset{t_n'}{\underline{t_n}}]}$$

These are rules that can potentially change the deterministic behavior of a program. If a language is sequential and deterministic, then there is no need to declare any of the contextual rules non-deterministic, because the rules are expected to be Church-Rosser on the terms of interest[8] (well-formed programs) if the programming language is correctly defined. For example, a simple calculator language that only evaluates side-effect-free expressions is deterministic, even though several rules may possibly apply concurrently and non-deterministically during evaluation tasks; what is important is that the final result of an evaluation does not depend on the order in which or the places where these rules apply. However, if a language admits concurrency, then some rules may lead to non-deterministic behaviors; for example, reads and writes of memory locations, or acquire and release of locks can lead to different behaviors of concurrent programs. Technically, a rule is non-deterministic when there are terms of interest (well-formed programs or fragments of programs) on which two rule instances (possibly of the same rule) can be applied at the same time, at least one of them being an instance of our rule, and depending upon which of the two rules is chosen first one gets different behaviors of the program. If an instance of a rule can be swapped with any other rule instance at any moment, then the former rule is obviously not non-deterministic. Intuitively, one can think of the non-deterministic rules as those rules that show the *relevant*

[8] The contextual rules defining a programming language are *not* Church-Rosser in the strict sense of the term in general, even though the language is sequential. Indeed, one can imagine a term containing two continuation structures, both with the next task to write the same variable; depending upon which rule is picked first, one can obtain totally different normal forms. However, such a term will never appear during the evaluation of a well-formed sequential program (though one would need to prove that), so we say that such terms are *not* of interest.

execution steps in the execution of a concurrent program, namely those where the behavior of the program can possibly change if another choice is being made. We call *deterministic* all the rules which are *not* non-deterministic (these include also the structural rules).

The structural and computational declarations of contextual rules are *modular*, in the sense that one does not need knowledge about the entire language in order to declare a rule structural or computational. However, the non-deterministic declaration of rules is trickier and needs global information about the program. For example, one cannot know when one defines a variable lookup rule whether the defined language is concurrent or not; if it is, then that rule needs to be declared non-deterministic. Also, it is impossible to identify the non-deterministic rules automatically in general, because this problem is harder than testing confluence, a notorious undecidable problem.

The non-deterministic rule declarations should be regarded as *annotations*, rather than as a core part, of the semantics of a language, telling the designer that those rules, and only those, *may* non-deterministically change the behavior of the program. This information can be very precious for certain analysis purposes, such as model checking. Indeed, since the other computational rules are implicitly declared to not affect the overall behavior of the concurrent program, a model checker for the defined language can regard their transitive application as *atomic*, thus reducing significantly the state space that needs to be analyzed: only the instances of the non-deterministic rules can lead to increases in the number of states of the system that need to be analyzed. In terms of model checking, the deterministic rules can be regarded as *atomic abstractions* (also related to *partial-order reduction*) and their instances can therefore be grouped and applied together in one "big" atomic step, without worrying about missing any behavior of the concurrent program. The intuition underlying the distinction between non-deterministic and deterministic contextual rules closely resembles the distinction between rules and equations in rewriting logic; indeed, when we translate K language definitions into rewriting logic for program analysis purposes, we translate the non-deterministic ones into rewrite rules and the deterministic ones into equations.

4.7 Matching Prefixes, Suffixes and Fragments

We here introduce another important piece of notation, which will help us further compact language definitions by eliminating the need to mention unnecessary underscore variables. Recall the "pattern" term $\mathit{thread}(\mathit{env}((X, L)\ \_)\ \_)$, useful for variable lookup, that we discussed in Section 4.3. The two underscores in this pattern term make the pattern look heavier and harder to read than needed. Conceptually they bring nothing new: they are there just for formal reasons, to write terms using a mathematically rigorous notation. What one really wants to say here is that one is interested in the pair (X, L) that appears *somewhere* in the environment that appears *somewhere* in the thread. In any domain, good notation is meant to have a low, hopefully close to zero representational distance to the concepts that it refers to. In our particular domain of programming languages, we believe, admittedly subjectively, that the notation $\mathit{thread}\langle\mathit{env}\langle(X, L)\rangle\rangle$ for the same pattern term is better than the one using the underscores. We next formalize our notation.

By convention, whenever "$\_ \circ \_$" is an ACI or AI operator constructing a term that appears in some context, say $C\Box$, we write

- $C(T\rangle$ (i.e., left parenthesis right angle) as syntactic sugar for $C(T \circ \_)$,
- $C\langle T)$ (i.e., left angle right parenthesis) as syntactic sugar for $C(\_ \circ T)$,

- $C\langle T\rangle$ (i.e., left and right angles) as syntactic sugar for $C(\_ \circ T \circ \_)$.

If "$\_ \circ \_$" is an ACI operator then the three notations above have the same effect, namely that of matching T (which can be itself a set of several attributes) inside the soup in the hole of context C; for simplicity, in this case we just use the third notation, $C\langle T\rangle$. The intuition for this notation comes from the fact that the left and the right angles can be regarded as some hybrid between corresponding "directions" and parentheses. For example, if "$\_ \circ \_$" is AI (not C) then $(T\rangle$ can be thought of as a list starting with T (the left parenthesis) and potentially continuing to the right (the right angle); in other words, it says that T is the *prefix* of the list in the hole of C. Similarly, $\langle T)$ says that T is a *suffix* of and $\langle T\rangle$ says that T is a contiguous *fragment* of the list in the hole of the context C. If "$\_ \circ \_$" is also commutative, i.e., an ACI operator, then the notions of prefix, suffix and fragment are equivalent, all saying that T is a subset of the set at the hole of C.

This notational convention will be particularly useful in combination with other conventions part of the K notation. For example, the input and output of the programming language defined in the sequel will be modeled as comma separated lists of integers, using an AI binary operation "$\_ , \_$" of identity "$\cdot$"; then in order to read (consume) the next two integers I_1, I_2 from the input buffer, or to output (produce) integers I_1, I_2 to the output buffer, all one needs to do (as part of a larger context that we do not mention here) is:

$$\mathit{in}(\underset{\cdot}{\underline{I_1, I_2}}\rangle \qquad \text{and, respectively,} \qquad \mathit{out}\langle\underset{I_1, I_2}{\underline{\quad\cdot\quad}})$$

The first matches the first two integers in the buffer and removes them (the "$\cdot$" underneath the line; recall that "$\cdot$" is by convention the unit, or the identity of all ACI or AI operators), while the second matches the end of the buffer (the "$\cdot$" above the line) and appends the two integers there. Note that the later works because of the matching modulo identity: $\mathit{out}\langle\cdot)$ is a shorthand for $\mathit{out}(\_ , \cdot)$, where the underscore matches the entire list; replacing "$\cdot$" by the list I_1, I_2 is nothing but appending the two integers to the end of the list wrapped by *out* (thanks to the AI attributes of the comma operator).

As another interesting example, this time using an ACI operator, consider changing the location of a variable X in the environment of a thread to another location, say L (in order for such a definition to make sense, one should make sure that there is enough context in order for L to be properly instantiated); this could be necessary in the definition of a language allowing declarations of local variables, when a variable with the same name, X, is declared locally and thus "shadows" a previously declared variable with the same name. This can be done as follows (part of a larger context):

$$\mathit{thread}\langle \mathit{env}\langle (X, \underset{L}{\underline{\_}})\rangle\rangle.$$

We take the liberty to use the "angle" notation above also in situations where the corresponding soups are *not* necessarily "wrapped" by specific attributes, but only when, by using all the sort and tuple operation inference conventions above, such a use does not lead to any confusion. For example, as part of the semantics of most programming languages, we need to evaluate lists of expressions sequentially. To achieve this elegantly, we will just place the list of expressions on the top of the continuation and "wait" for it to be transformed into a list of corresponding values. As part of this process, we use an auxiliary operator pairing a list of expressions and a list of values, and then evaluate the expressions one by one, moving their corresponding values (as they are generated) at the end of the list of values. The following is the contextual rule that picks the next expression and places it on the top of the continuation for evaluation:

$$k(\underset{E}{\underline{\;\cdot\;}} \curvearrowright ((\underset{\cdot}{\underline{E\!:\!\mathit{Exp}\rangle}}, \_ : \mathit{ValList}))\rangle$$

The "·" on the top of the continuation is generated by the matching modulo identity; by replacing it with the expression E, we are saying that we place E at the top of the continuation. From the sorting information present in the second continuation item, it becomes clear that that item corresponds to the pairing operation of *ExpList* and *ValList*; since its first argument is undoubtedly a list of expressions, we use the angle-bracket convention to identify the first expression in the list, E, and then remove it by replacing it with the "·". All the contextual rules for this common task of evaluating lists of expressions are part of the core of K and will be discussed in detail in Section 5. Note that the rule above is structural, because it carries no computational meaning.

4.8 Structural Operations

We next introduce the most subtle of our notational conventions, which plays a crucial role in increasing the modularity of programming language definitions. This notational convention is based on the observation that, in programming language definitions, it is always the case that the state of the program, a term in our representation, does not change its significant structure during the execution of the program. For example, the store will always stay at the same level in the state structure, typically at the top level, and the environments will be declared as part of each thread's soup and will stay there during the execution. If certain state infrastructure is known to stay unchanged during the evaluation of any program, and if one is interested in certain attributes that can be unambiguously located in that state infrastructure, then we are going to take the liberty to only mention those attributes as part of the context assuming that the remaining part of the context can be generated automatically (statically).

To make this intuition clearer, let us first consider an artificial example. Suppose that attributes a_1 and a_2 wrap ACI soups and that their concrete positions in the state structure can be inferred unambiguously (we will see shortly how). Suppose also that one is interested in changing subterms t_1 (to t'_1) and t_2 (to t'_2) that are part of the soups wrapped by attributes a_1 and a_2. Then one would like to write this contextual rule simply as:

$$a_1\langle\underset{t'_1}{\underline{t_1}}\rangle \quad a_2\langle\underset{t'_2}{\underline{t_2}}\rangle$$

With the current notational conventions, this rule would parse only if a_1 and a_2 lay together within the same soup. However, due to various reasons, the two attributes may be located at different levels in the state. For example, a_1 and a_2 may be part of the soups of other attributes, say b_1 and b_2, respectively, in which case one is expected to write the rule above as

$$b_1\langle a_1\langle\underset{t'_1}{\underline{t_1}}\rangle\rangle \quad b_2\langle a_2\langle\underset{t'_2}{\underline{t_2}}\rangle\rangle$$

The second rule above is not only unnecessarily more complicated and harder to read than the first one (since the places of a_1 and a_2, i.e., inside the soups wrapped by b_1 and b_2, are known, then why mention them again?), but also suffers from being *more rigid* than the first. Indeed, if for any reason one decides to change the structure of the state and move the attributes a_1 and a_2 to other places in the state (still keeping their unique identity), then one would also need to transform the second rule accordingly; note, however, that one would not need to change the first rule.

Such changes of state structure are quite common in language definitions. For example, one may consider different state structures for sequential languages than for concurrent ones. Indeed, in a concurrent language one may want the various threads to each wrap their own state (environment, locks, control structures, etc.) for clarity and to avoid confusion among attributes with the same name associated with different threads, while in a sequential language that is not necessary because there is only one thread, so the content of that thread, including its environment and control structures, can be moved to the top level of the state. This way, rules defined for the concurrent language involving both thread attributes (e.g., environments) and other attributes (e.g., the store), are not well-formed anymore *as are*: these need to be modified to account for the new state structure, thus breaking the encapsulation principle of the definitions of language features in particular and the modularity of programming language definitions in general.

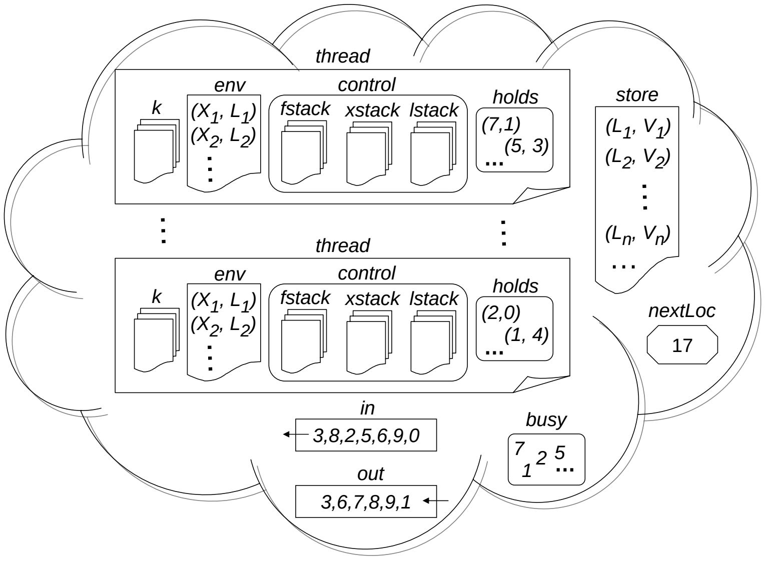


Figure 4: FUN state infrastructure.

The state structure which is not intended to change can be, fortunately, specified very easily by just "marking" certain operations as *structural*. Consider, for example, a state instance like the one in Figure 4 of the programming language FUN that we will define in the next sections. As can be seen, the state is a nested soup of ingredients, or attributes: the store wraps a set of pairs (location,value) and is located at the top level in the state together with input/output buffers, a set of busy locks, a next free location, as well as one attribute per thread (added dynamically); each thread contains a soup of its attributes (that's why we use the term "nested soup"), such as continuation, an environment, a set of locks it holds, as well as its control information; the control information is itself a soup whose ingredients are the various stacks needed to encode the control context of the program (for functions, loops, exceptions, etc.). Most of the operations that are constructors for states are structural, i.e., give the state a structure which does not change as the programs are executed. In our example language, the following top level state constructor operations can be declared as structural:

Structural operations
$thread : ThreadState \to StateAttribute$ [$\star$]
$busy : IntSet \to StateAttribute$
$in : IntList \to StateAttribute$
$out : IntList \to StateAttribute$
$store : LocValSet \to StateAttribute$
$nextLoc : Nat \to StateAttribute$

The star "$\star$" attribute of *thread* means that that attribute can have multiple occurrences in the state, all at the same level; this notation will be discussed shortly. Here we assumed that other basic sorts and operations are already declared, such as those for locations, values, integers, naturals, etc. Also, following our previous notational conventions, we assume that sets and lists of such basic sorts exist whenever we need them. The state of the programming language is nothing but a set of state attributes, i.e., of terms of sort *StateAttribute*. Indeed, we take the liberty to use the sort *State* as a more intuitive "alias" for *StateAttributeSet*. At this moment, we define and use sort aliasing informally; introducing a formal notation for sort aliasing is not difficult, but we prefer to postpone it until we implement K. Further, constructor operators for thread states can, and should also be declared as structural. We declare a sort *ThreadAttribute* and, as before, take the liberty to alias the sort *ThreadAttributeSet* by the more intuitive sort name *ThreadState*:

Structural operations
$k : Continuation \to ThreadAttribute$
$env : Environment \to ThreadAttribute$
$control : Control \to ThreadAttribute$
$holds : IntIntSet \to ThreadAttribute$

In the above operation declarations, we used the sort *Continuation* as an alias for *ContinuationItemList*, *Environment* as an alias for the sort *VarLocSet* and *Control* as an alias for *ControlAttributeSet*; we may also use *Ctrl* as an alias sort for *Control*.

The control of a thread is also a soup of various control attributes, which should be declared as structural operations as well:

Structural operations
$fstack : FunctionStack \to ControlAttribute$
$xstack : ExceptionStack \to ControlAttribute$
$lstack : LoopStack \to ControlAttribute$

The sorts *FunctionStack*, *ExceptionStack* and *LoopStack* alias various list sorts which we do not need here but will explain in the next section.

The structural operations above form a tree whose edges are the structural operations and whose nodes are the largest sorts that are supersorts of the result sort of each structural operation, like in Figure 5. For example, the target node of the edge *control* is *ThreadState*, which aliases the sort *ThreadAttributeSet*; that is because the target sort of the operation *control* was declared to be

ThreadAttribute and its largest supersort is *ThreadAttributeSet*. Note that, in general, the structural operations may yield a multi-graph whose multi-edges correspond to the structural operations; that is because operations may have more than one argument and because one may have cycles in operation declarations. However, for simplicity, in this paper we assume that structural operations are defined in such a way that their corresponding multi-graph is a tree; we believe, however, that our notational conventions can be extended to general multi-graphs (but we are currently not motivated to do it).

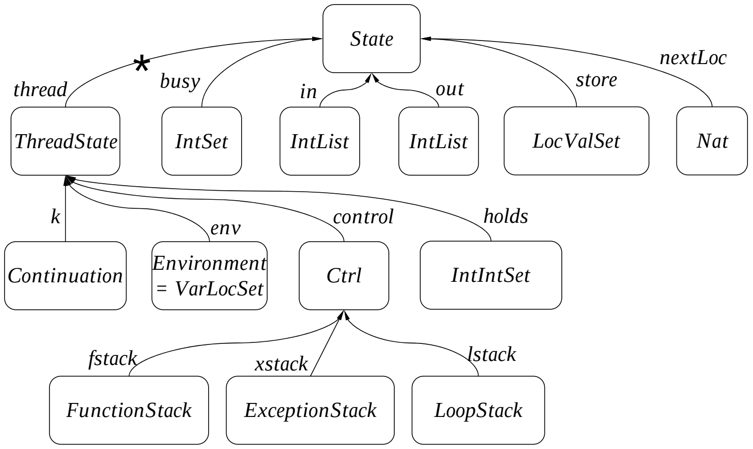


Figure 5: FUN state structural operations.

4.9 Context Transformers

We are now ready to introduce our notational convention for *context transformers*. Assuming that the various structural operations have unique names, or in other words that their corresponding tree has unique labels on its edges, then one can use the structural tree to *automatically* transform partial, possibly non-well-formed contexts into concrete ones that are well-formed within the current structure of the state. These context transformers are applied statically, before the rewrite system is executed. Suppose, for example, that in order to define a particular language feature, such as printing a value, one is interested only in the continuation (k) of a thread (the continuation tells what is the next task to perform, in this case printing of some value) and the output buffer (*out*). In spite of the fact that these two attributes are located at different levels in the state nested soup, thanks to the context transformer notational convention one is allowed to write contextual rules as if they are at the same level, that is, of the form:

$$k(...)\ out(...)$$

Here we do not care about the particular terms that are wrapped by the attributes (the "..."), nor whether they are replaced by some other terms or not. Complete details will be given in the next section. We are only interested in the structure of the state context in which the attributes mentioned in the partial contextual rule make sense. For example, by examining the tree associated to the state structural operations in Figure 5, one can easily see that the only way to complete the partial context above is to mention the entire path to the attribute k, that is:

$$\mathit{thread}\langle k(...)\rangle\ \mathit{out}(...)$$

To properly transform the context, one may need to add artificial variables corresponding to non-interesting branches at each node in the tree. By convention, we only use the special underscore "_" variables for this task; combined with the left and right angle convention to eliminate unnecessary underscore variables, we get the concrete contextual rule above.

Let us now consider a more complex example, corresponding to retrieving the value associated to some program variable. In order to find the value corresponding to a variable that appears as the next task in the continuation k, one needs to retrieve the location associated to that variable from the environment *env* and then the value associated with that location in the *store*. Ignoring again the non-structural details, thanks to the context transformer convention one can simply write this contextual rule as follows:

$$k(...)\ \mathit{env}(...)\ \mathit{store}(...)$$

By examining the unique way in which these attributes make sense together, this partial contextual rule can be automatically transformed into:

$$\mathit{thread}\langle k(...)\ \mathit{env}(...)\rangle\ \mathit{store}(...)$$

A subtle aspect of the context transformation above is that both k and *env* attributes are part of each thread. Since there can be many threads running concurrently in a program, how can one know that one did not mean the k attribute of one thread and the *env* attribute of another thread in the partial contextual rule above? The simple answer to this concern is the following: the context transformers are applied statically, based on just the structural information provided by the syntactic definition of the structural operators. The fact that there can be several threads at the same time part of the top-level state soup is a purely semantic issue, consequence of the other contextual rules defining the language under consideration. There can be language definitions where the state may contain more than one store (for example a distributed language) during the evaluation of the program. However, once the store and the operations on the path to it in the state were declared as structural and the tree associated to the structural operations is unambiguous, there is only one way to complete/transform any partial context referring to the store.

Then what if, for some reason which may not make sense for our particular language but could make sense for other languages, one really means the k of one thread and the *env* or another thread? In such a case one is supposed to give the context transformation procedure more information, namely to mention explicitly the two threads:

$$\mathit{thread}\langle k(...)\rangle\ \mathit{thread}\langle \mathit{env}(...)\rangle\ \mathit{store}(...)$$

Another interesting situation which can appear in some language definitions is when one wants to refer to two attributes with the *same name* in a contextual rule. In the language defined in this paper thread communication is via shared memory and locks, but in some languages one may want to synchronize threads using a wait/notify mechanism and one may want to define this by matching both the notifying thread and the waiting one, in which case one would end with two attributes with the same name in some contextual rules (*thread* in this case). Since attribute names are distinct in the structural operations' tree, an attribute operation, say a, can appear multiple times in a contextual rule only if some attribute on the path from the root of the tree to a appears multiple times in its corresponding attribute set; this attribute can be either a or some other attribute at a higher level.

From practical considerations and our experience with using the K notation, we believe that a language designer typically knows in advance which attributes are allowed to appear multiple times in their soup. For example, if one designs a multi-threaded language, one knows that the attribute *thread* can appear multiple times in the soup of threads, because threads are intended to be created and terminated dynamically. However, one also knows that environments, for example, are not intended to appear multiple times within one thread. In principle, a careful analysis of a complete language definition would allow one to deduce which attributes should be allowed to appear multiple times and which should not; for example, one can see that there are rules which add one more thread to the state, while there are no rules that add an environment to an existing thread state. However, we prefer *not* to infer this important structural information automatically from the way the contextual rules are written for two reasons:

- first, because it is meaningful and useful for a language designer to be aware and explicitly state which attributes are allowed to multiply; and

- second, because an automation of this process seems non-trivial to implement.

Since attributes that multiply are by far less frequent that attributes which do not multiply, we introduce a notation in K for those which multiply: we put a star "$\star$" on the arrow labeled with those attributes in the state attributes' tree; see, for example, the star on the *thread* attribute in Figure 5. With this convention, it is now easy to disambiguate contextual rules. In particular, the partial contextual rule

$$k(...)\ k(...)$$

would be completed to

$$thread\langle k(...)\rangle\ thread\langle k(...)\rangle$$

rather than to

$$thread\langle k(...)\ k(...)\rangle.$$

If some contextual rules cannot be transformed with this convention then, at this moment, we say that the specification is ambiguous or faulty.

It is fair to mention it here again that the K notation proposed in this section and intensively used in the next is *not* yet supported by any mechanical system, tool or prototype. We are only using it by hand to define languages compactly and modularly on paper. However, it is a straightforward exercise to transform a definition of a language using the K notation into a rewrite logic specification that can be executed on existing systems. For example, it took a graduate student (Traian Florin Şerbănuţă, who is also currently implementing a parser and a mechanical translation of K into rewriting logic) little time to hand translate into Maude the K-definition of FUN discussed next (Appendixes F and G show the Maude definitions of both sequential and concurrent FUN).

In our current programming language definitions we regard context transformers as well as the translation of all the other notational conventions as a *pre-compilation* step of our language definitions, which can only be performed once the structural operators are completely defined. The modularity of our language definitions comes from the fact that the same definition of a particular language feature pre-compiles differently under different state structures. As an example, in several of our language definitions we were faced with the controversial issue of whether the continuation of a thread should really be at the same level with the control and the environment, letting the control structure maintain only the execution stacks, or it should instead be part of the control of that thread. Fortunately, with the K notation it essentially does not matter whether k is defined as part of the control of threads or at the same level with it. If one wants the former, all one needs to do is to replace the operation declaration

$$k : \mathit{Continuation} \to \mathit{ThreadAttribute}$$

with the declaration

$$k : \mathit{Continuation} \to \mathit{ControlAttribute}$$

Nothing else needs to be changed (but the entire language definition needs to be pre-compiled again). As with the other notational conventions, one will become more familiar with the context transformers as one defines concrete languages. The next section shows a middle-complexity case study.

5 The K-Technique: Defining the FUN Language

We exemplify our K-technique by defining a simple, yet non-trivial programming language called FUN, which contains higher-order functions with return, let and letrec, lists, sequential composition, assignments, loops with break and continue, input/output, parametric exceptions, callcc and concurrency via threads and synchronization. We assume the reader familiar with these programming language features. In this section we define all features except callcc and concurrency, as part of a sublanguage that we call "sequential FUN"; we complete the definition of FUN in the next section, where we define callcc and concurrency as language extensions, to reflect the modularity of our definitional framework. Our main goal is to focus on FUN's semantics, so we just pick some familiar syntax and assume an existing parser. Depending upon the rewriting engine one uses to formalize it, one can use the definition below *as is* to execute and to analyze (BFS reachability and/or LTL model checking) FUN programs. The complete K-definition of sequential FUN is shown in Figure 6, and a Maude-ified version of it in Appendix F. Figure 7 shows how the language can be extended with callcc and concurrency, and Appendix G a complete Maude-ified definition.

5.1 Syntax

FUN is an expression language. Expressions may evaluate to values of different types. We define the syntax of FUN as an order-sorted algebraic signature using the mixfix notation. It is known that the mixfix notation is equivalent to the more common (but slightly harder to relate to term rewriting) BNF notation, by interpreting each sort into a non-terminal and each operation as a production. If underscores are not explicitly mentioned in an operation's name then that operation is assumed in prefix notation. We assume already existing definitions of integers and identifiers, coming with sorts *Int* and *Var*, respectively. One can either use builtins or define these. We therefore introduce one sort *Exp* with $Int < Exp$ and $Var < Exp$, and define the following operations:

$\mathsf{true},\ \mathsf{false},\ \mathsf{skip}\ :\ \to Exp$

$\_+\_,\ \_-\_,\ \_*\_,\ \_/\_,\ \ldots,\ \_\leq\_,\ \_\geq\_,\ \ldots, \_\wedge\_,\ \_\vee\_,\ \ldots\ :\ Exp \times Exp \to Exp$

$\mathsf{if\_then\_}\ :\ Exp \times Exp \to Exp$

$\mathsf{if\_then\_else\_}\ :\ Exp \times Exp \times Exp \to Exp$

$\mathsf{fun\_} \to \_\ :\ VarList \times Exp \to Exp$

$\_\,(\_\,)\ :\ Exp \times ExpList \to Exp$

$\mathsf{return}\ :\ Exp \to Exp$

$\mathsf{let},\ \mathsf{letrec}\ :\ VarList \times ExpList \times Exp \to Exp$

$\_;\_\ :\ Exp \times Exp \to Exp$

$[\_\,]\ :\ ExpList \to Exp$

$\mathsf{car},\ \mathsf{cdr},\ \mathsf{null?} : Exp \to Exp$

$\mathsf{cons}\ :\ Exp \times Exp \to Exp$

$\_ := \_\ :\ Var \times Exp \to Exp$

$\mathsf{read}()\ :\ \to Exp$

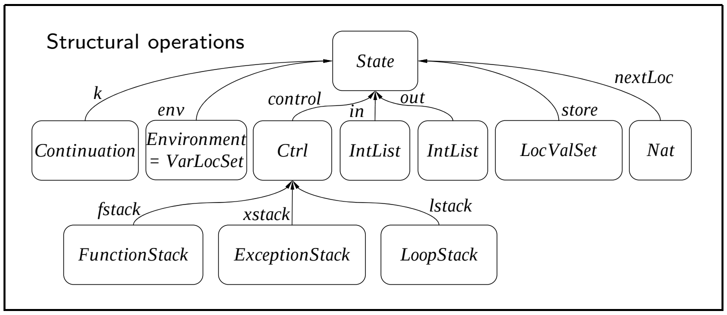
$\mathsf{print}\ :\ Exp \to Exp$

$\mathsf{try\_\ catch}(\_\,)\,\_\ : Exp \times Var \times Exp \to Exp$

$\mathsf{throw}\ :\ Exp \to Exp$

$\mathsf{while}(\_\,)\ \_\ :\ Exp \times Exp \to Exp$

import $BOOL,\ INT,\ REAL,\ K\text{-}BASIC$



$$\left.\begin{array}{l} eval : Exp \times IntList \to IntList \\ result : State \to IntList \end{array}\right\} \cdots \left\{\begin{array}{r} \frac{eval(E,\ Il)}{result(k(E)\ env(\cdot)\ control(fstack(\cdot)\ xstack(\cdot)\ lstack(\cdot))\ in(Il)\ out(\cdot)\ store(\cdot)\ nextLoc(0))} \\ \\ \frac{result\langle k(\_ :\ Val)\ out(Il)\rangle}{Il} \end{array}\right.$$

$$\left.\begin{array}{l} Var,\ Bool,\ Int,\ Real < Exp \\ Bool,\ Int,\ Real < Val \end{array}\right\} \cdots\cdots\cdots \left\{\ k(\frac{X}{V}\rangle\ env\langle(X, L)\rangle\ store\langle(L, V)\rangle\right.$$

$\underline{\mathsf{not}\_}\ :\ Exp \to Exp\ \ [!,\ \mathsf{not}_{Bool} :\ Bool \to Bool]$

$\underline{\_ + \_} : Exp \times Exp \to Exp\ [!,\ \_ +_{Int} \_ : Int \times Int \to Int,\ \_ +_{Real} \_ : Real \times Real \to Real]$

$\underline{\_ \leq \_} :\ Exp \times Exp \to Exp\ [!,\ \_ \leq_{Int} \_ :\ Int \times Int \to Bool,\ \_ \leq_{Real} \_ :\ Real \times Real \to Bool]$

$\underline{\mathsf{skip}} : \to Exp\ [unit : \to Val]$

$$\left.\begin{array}{l} \underline{\mathsf{if\_then\_}}\ :\ Exp \times Exp \to Exp \\ \underline{\mathsf{if\_then\_else\_}}\ :\ Exp \times Exp \times Exp \to Exp\ [!(1)[if]] \end{array}\right\} \cdots \left\{ \frac{\mathsf{if}\ B\ \mathsf{then}\ E}{\mathsf{if}\ B\ \mathsf{then}\ E\ \mathsf{else\ skip}} \right. \bigg| \frac{bool(true) \curvearrowright if(E_1, E_2)}{E_1} \bigg| \frac{bool(false) \curvearrowright if(E_1, E_2)}{E_2}$$

$$\left.\begin{array}{l} \underline{\mathsf{fun}\_ \to \_} : VarList \times Exp \to Exp \\ \underline{\_(\_)}\ : \\ \quad Exp \times ExpList \to Exp\ [![app]] \\ \underline{\mathsf{return}}\ :\ Exp \to Exp\ [!] \\ closure : \\ \quad VarList \times Exp \times VarLocSet \to Val \\ popFstack : \to ContinuationItem \end{array}\right\} \cdots \left\{\begin{array}{l} k(\frac{\mathsf{fun}\ Xl \to E}{closure(Xl, E, Env)}\rangle\ env(Env) \\ (k(\frac{(closure(Xl, E, Env), Vl) \curvearrowright app \curvearrowright K}{Vl \curvearrowright bind(Xl) \curvearrowright E \curvearrowright popFstack})\ fstack(\frac{\cdot}{(K, Env', C)}\rangle\ C : Ctrl)\ env(\frac{Env'}{Env}) \\ k(\_ : Val \curvearrowright \frac{popFstack}{K})\ fstack(\frac{(K, Env, \_)}{\cdot}\rangle\ env(\frac{\_}{Env}) \\ (k(\_ : Val \curvearrowright \frac{return \curvearrowright \_}{K})\ fstack(\frac{(K, Env, C)}{\cdot}\rangle\ \frac{\_ : Ctrl}{C})\ env(\frac{\_}{Env}) \end{array}\right.$$

$$\left.\begin{array}{l} \underline{\mathsf{let}},\ \underline{\mathsf{letrec}} : \\ \quad VarList \times ExpList \times Exp \to Exp \end{array}\right\} \cdots \left\{ k(\frac{\mathsf{let}(Xl, El, E)}{El \curvearrowright bind(Xl) \curvearrowright E \curvearrowright Env}\rangle env(Env) \right. \bigg| \ k(\frac{\mathsf{letrec}(Xl, El, E)}{bind(Xl) \curvearrowright El \curvearrowright write(Xl) \curvearrowright E \curvearrowright Env}\rangle env(Env)$$

$$\left.\begin{array}{l} \underline{\_ ; \_}\ :\ Exp \times Exp \to Exp\ [!] \\ \underline{\_ := \_}\ :\ Var \times Exp \to Exp \end{array}\right\} \cdots\cdots\cdots \left\{ (\frac{V_1\ :\ Val}{\cdot}, V_2\ :\ Val) \curvearrowright \frac{;}{\cdot} \right. \bigg| \frac{X := E}{E \curvearrowright write(X) \curvearrowright unit}$$

$$\left.\begin{array}{l} \underline{[\_]}\ :\ ExpList \to Exp\ [!,\ [\_] : ValList \to Val] \\ \underline{\mathsf{car}},\ \underline{\mathsf{cdr}},\ \underline{\mathsf{null?}} :\ Exp \to Exp\ [!] \\ \underline{\mathsf{cons}}\ :\ Exp \times Exp \to Exp\ [!] \end{array}\right\} \cdots\cdots\cdots \left\{\begin{array}{l} \frac{[V : Val, \_] \curvearrowright car}{V} \bigg| \frac{[\_ : Val, Vl] \curvearrowright cdr}{Vl} \\ \frac{[\cdot] \curvearrowright null?}{bool(true)} \bigg| \frac{[\_ : Val, \_] \curvearrowright null?}{bool(false)} \bigg| \frac{(V, [Vl]) \curvearrowright cons}{[V, Vl]} \end{array}\right.$$

$$\left.\begin{array}{l} \underline{\mathsf{read()}}\ :\ \to Exp \\ \underline{\mathsf{print}}\ :\ Exp \to Exp\ [!] \end{array}\right\} \cdots\cdots\cdots \left\{ k(\frac{\mathsf{read()}}{int(I)}\rangle\ in(\frac{I}{\cdot}\rangle \right. \bigg| \ k(\frac{int(I) \curvearrowright print}{unit}\rangle\ out\langle\frac{\cdot}{I})$$

$$\left.\begin{array}{l} \underline{\mathsf{try}\_\ \mathsf{catch}(\_)\ \_}\ : \\ \quad Exp \times Var \times Exp \to Exp \\ \underline{\mathsf{throw}}\ :\ Exp \to Exp\ [!] \end{array}\right\} \cdots \left\{\begin{array}{l} (k(\frac{\mathsf{try}\ E'\ \mathsf{catch}(X)\ E \curvearrowright K}{E' \curvearrowright popXstack})\ xstack(\frac{\cdot}{(X, E, Env, K, C)}\rangle\ C : Ctrl)\ env(Env) \\ k(\_ : Val \curvearrowright \frac{popXstack}{K})\ xstack(\frac{(\_, \_, \_, K, \_)}{\cdot}\rangle \\ (k(\_ : Val \curvearrowright \frac{throw \curvearrowright \_}{bind(X) \curvearrowright E \curvearrowright Env \curvearrowright K})\ xstack(\frac{(X, E, Env, K, C)}{\cdot}\rangle\ \frac{\_ : Ctrl}{C})\ env(\frac{\_}{Env}) \end{array}\right.$$

$$\left.\begin{array}{l} \underline{\mathsf{while}(\_)\ \_} : Exp \times Exp \to Exp \\ \underline{\mathsf{for}(\_ ; \_ ; \_)\ \_}\ : \\ \quad Exp \times Exp \times Exp \times Exp \to Exp \\ \underline{\mathsf{break}} :\ \to Exp \\ \underline{\mathsf{continue}}\ :\ \to Exp \\ \circlearrowleft : \to ContinuationItem \end{array}\right\} \cdots \left\{\begin{array}{l} \frac{\mathsf{while}(B)\ E}{\mathsf{for(skip};\ B;\ \mathsf{skip)}\ E} \\ (k(\frac{\mathsf{for}(S;\ B;\ J)\ E \curvearrowright K}{S;\ B \curvearrowright \circlearrowleft})\ lstack(\frac{\cdot}{(B, E, J, Env, K, C)}\rangle\ C : Ctrl)\ env(Env) \\ k(\frac{bool(false) \curvearrowright \circlearrowleft}{unit \curvearrowright K})\ lstack(\frac{(\_, \_, \_, \_, K, \_)}{\cdot}\rangle \ \bigg|\ k(\frac{bool(true)}{E;\ J;\ B} \curvearrowright \circlearrowleft)\ lstack((B, E, J, \_, \_, \_)\rangle \\ (k(\frac{\mathsf{break} \curvearrowright \_}{unit \curvearrowright K})\ lstack(\frac{(\_, \_, \_, Env, K, C)}{\cdot}\rangle\ \frac{\_ : Ctrl}{C})\ env(\frac{\_}{Env}) \\ (k(\frac{\mathsf{continue} \curvearrowright \_}{J;\ B \curvearrowright \circlearrowleft})\ lstack((B, \_, J, Env, \_, C)\rangle\ \frac{\_ : Ctrl}{C})\ env(\frac{\_}{Env}) \end{array}\right.$$

Figure 6: K definition of sequential FUN

$\mathsf{for}(\_\,;\_\,;\_\,)\ \_\ :\ \mathit{Exp} \times \mathit{Exp} \times \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}$

$\mathsf{break},\ \mathsf{continue}\ :\ \to \mathit{Exp}$

In Section 6 we extend the syntax above with callcc and with threads and synchronization.

5.2 State Infrastructure

The picture at the top of Figure 6 shows the structural operators of sequential FUN. The structure of the state of the language will change when we add threads, as shown in Figure 7; Figure 4 shows an instance of the state of concurrent FUN. For all our language definitions in K, the state of the language is a soup of *attributes*, each potentially comprising other soups of attributes. The particular sort names that one chooses for the various soup ingredients are not technically important, though they can help users read and understand language definitions more easily. If the modularity of the language definition is one's concern, then one can define the various soup ingredients "on-the-fly", as the language features using them are introduced to the language. Modularity of our definitional framework will be discussed in the next section, together with a detailed comparison to Modular SOS [21]. Let us next discuss the various state attributes of sequential FUN, whose programs contain only one, implicit execution thread; all these attributes will be reused in the definition of concurrent FUN, with the same meaning.

One of the distinguished state attributes that appear in all our definitions in K is the *continuation*, which is wrapped using the attribute name *k* and which encodes the remaining computation tasks; our encoding and use of continuations is discussed in Section 5.3. The *env* attribute wraps a set of pairs (variable, location), and the *store* attribute wraps a set of pairs (location,value). Locations can be any structures, e.g., wrapped natural numbers such as *location*(7), providing a next location operation: if L is some location, then *next*(L) is the "next" location, where the ordering is left uninterpreted. We assume the existence of environment and store *update* operations; for example, for stores we assume that $\mathit{Store}[L \leftarrow V]$ updates the value at location L in *Store* to V; if L is not in the store yet, then it adds the pair (L, V) to *Store*. The state attribute *nextLoc* wraps the next *available* location. Note that FUN implementations/compilers can interpret locations and the *next* operation many different ways, in particular as needed by an underlying garbage collector.

The attributes *in* and *out* contain the input and the output buffers (lists of integers) needed to define the read() and print statements. Because of the various control-intensive statements of FUN, the *control* attribute contains itself several other attributes which are stacks that will store, into tuples, information needed to define the different control-intensive statements: *fstack* for functions with return, *xstack* for exceptions, and *lstack* for loops with break and continue.

5.3 Continuations

There are many different ways to formally represent or define continuations; these are ultimately all equivalent and capture the same idea: a continuation encodes the *remaining of the computation* [23, 26]. More precisely, a continuation, say K, is a structure which, when passed a value, say V, which can be thought of as the value of some local computation, "knows" how to finish the entire computation. One can use various notations for continuation invocation, such as "$K(V)$", or "$K\ V$", or even "$V \rightsquigarrow K$" or "$V \curvearrowright K$", or any other meaningful or desired notation, all symbolizing the same intuition: the value V is passed to the continuation K. The first two notations are more common and are preferred in frameworks where continuations are encoded as function structures.

In this paper we prefer one of the other notations, namely $V \curvearrowright K$, to distinguish continuation applications, that we use for purely semantic purposes, from actual function invocations or term structures that appear in some languages that we intend to define using the proposed framework (function invocations are needed in definitions of functional, imperative and object-oriented languages, while term structures are needed in logic programming languages). To keep our syntax simple and non-ambiguous, we assume that the operation $\_ \curvearrowright \_$ is reserved for the K framework.

Regardless of the notation and of the explicit or implicit evaluation mechanism that one chooses for continuations, if K is a continuation corresponding to some intermediate stage/context during the evaluation of a program in which a subexpression E is being evaluated, if V is the value obtained after evaluating E, then $V \curvearrowright K$ will eventually evaluate to the final result of the program. To achieve this non-trivial task, continuations provide explicit representations of computation flows as ordinary data: they encode and manipulate the control-context of a program as data-context. Continuations typically encode only the remaining computational steps, not the store, or the environments, or other resources needed during the execution of the program; therefore, one still needs to refer to external data during the evaluation of a continuation invocation. Continuations are useful especially in the context of defining control-intensive statements, such as halt or exit, exceptions, break or continue of loops, callcc, etc., because one can relatively easily change the "future" in a continuation structure: the remaining computation steps are explicit in the continuation structure, rather than implicit in the program control flow, so one can change them as desired.

Traditionally, continuations are encoded as higher-order functions. This particular encoding of continuations as functions is very convenient in the presence of higher-order functional languages, because one uses the *implicit* evaluation mechanism of these languages to evaluate continuations as well: passing a value V to a continuation K becomes a function invocation, namely $K(V)$. Our K-technique is, however, *not* based on encodings of continuations as higher-order functions. Our underlying algebraic infrastructure is *not* higher-order – though, as our language definitions in this paper show, defining higher-order functions is not problematic. In our algebraic first-order framework, a continuation is a list, or stack, of computational tasks. The rules introduced as part of the language semantics will all together provide an *explicit* mechanism to evaluate applications of continuations. In other words, our continuations are inert by themselves; they need the rest of the semantics to become alive.

To distinguish continuations from other lists in the definition of a language, we do *not* use the common comma "_ , _" AI constructor notation for continuations; we use $\_ \curvearrowright \_$ instead, same like the operator for passing a value to a continuation. For example, a continuation $1 \curvearrowright +$ applied to a value V, i.e., the structure $V \curvearrowright 1 \curvearrowright +$, adds[9] 1 to V. Also, a continuation $1 \curvearrowright + \curvearrowright \mathit{halt} \curvearrowright K$ adds 1 to a passed value and then halts the computation with that sum, discarding the remaining computations encoded by K (via appropriate rules). K's module *K-BASIC*, which is included in all language definitions, provides a sort *Continuation*, as an alias for the sort *ContinuationItemList*. The elements/terms/tasks that we can store in a continuation are called *continuation items* and will be defined as having sort *ContinuationItem*. For example, the symbols + and *halt* that appear in the continuation $1 \curvearrowright + \curvearrowright \mathit{halt} \curvearrowright K$ are continuation items and are, as operations, *distinct* from the language constructors "_ + _" and "halt_"; they are generated automatically from the latter, because of their strictness attributes.

Since continuations in our framework can be constructed arbitrarily using the AI operator

[9] It may be more appropriate to say "is meant to add" instead of "adds", because our continuation structures need a state context in order to "return to life" and evaluate.

$\_ \curvearrowright \_$ and various continuation items, one should therefore be aware of the fact that there may be "garbage" terms of sort *Continuation*, that is, ones which do not correspond to any meaningful continuation; for example, the term $+ \curvearrowright + \curvearrowright \mathit{halt} \curvearrowright + \curvearrowright 7$ corresponds to no meaningful computation. However, such meaningless terms will never be reached in a correct language definition executing on a well-formed program. That is because, in such a case, the "continuation wrapper" $k(...)$ will always contain a well-formed intermediate computation; this can be shown by induction on the length of the computation, showing first that each contextual rule preserves the well-formedness of the continuation structure. In other words, one can therefore show that whenever a term $k(V \curvearrowright K)$ appears in a state structure reached during the "execution" of a program, the term K of sort *Continuation* corresponds to a well-formed continuation that contains the remaining computation tasks. Such proofs correspond to "type preservation and progress" proofs, so they are nevertheless important in general (we here focus on the design aspect of languages, delegating the proving aspects to Section 8.2).

The use of our continuations will somehow resemble stack machines: tasks at the top of the continuations are decomposed in subtasks, then their results will be composed back into a desired result for the original task. The goal of the definition of each language construct is to eventually put some value back on the top of the continuation. As a consequence of this divide-and-conquer approach, during the evaluation of an expression the continuation structure wrapped by $k(...)$ "evolves" from the original expression to its resulting value, at each moment during this process containing a snapshot of the remaining execution tasks. To achieve this divide-and-conquer functionality elegantly, we prefer to place *lists* of (sub)expressions on top of continuations, rather than plain expressions, and then "wait" for them to be evaluated into lists of corresponding values. Then a continuation of the form $(E_1 + E_2) \curvearrowright \mathit{write}(X) \curvearrowright K$, whose meaning is "the result of the sum $E_1 + E_2$ is written at X and then continue with K", evolves (the "divide" part) into $(E_1, E_2) \curvearrowright + \curvearrowright \mathit{write}(X) \curvearrowright K$; the latter is expected to evolve to $(\mathit{int}(I_1), \mathit{int}(I_2)) \curvearrowright + \curvearrowright \mathit{write}(X) \curvearrowright K$, which evolves (the "conquer" part) into $\mathit{int}(I_1 +_{\mathit{Int}} I_2) \curvearrowright \mathit{write}(X) \curvearrowright K$, and so on. The technicalities of this extension are discussed in Section 5.4.

5.4 Helping Operators

Same operations are used frequently in K definitions of languages. The operations below appear in the definitions of almost all the languages that we considered, so we group them together into a module called *K-BASIC* and imported in all language definitions. These are also used in the definitions of λ_{K} and FUN in this paper, among other languages that we defined in K. We give their definitions here; they are instructive and necessary to understand the rest of FUN's definition.

Expressions and values appear in the definitions of all the languages that we defined so far. A list of expressions at the top of a continuation is regarded as the task of evaluating all the expressions, propagating their side effects sequentially, and then producing a list of corresponding values on the top of the continuation; the continuation will "know" how to continue the rest of the computation once those values are available. Specifically, a term $k(\mathit{El} \curvearrowright K)$ in the state soup is expected to reduce eventually to a term $k(\mathit{Vl} \curvearrowright K)$, where *Vl* is a list of values corresponding to *El*, respectively, calculated by evaluating the expressions in *El* sequentially, propagating appropriately their side effects. Technically, to allow lists of expressions and of values to be added as continuation items, one needs to declare these as appropriate subsorts[10]: *ExpList* < *ContinuationItem* and *ValList* <

[10] The parsers of some term rewriting systems may complain when collapsing the sort connected components of

ContinuationItem. In practice, we actually assume that any sort is a subsort of *ContinuationItem*, so continuations can be regarded as ordered containers containing elements of any sort.

The rewrite logic semantics of a language consists of providing contextual rules that transform a task E on top of a continuation K into a term V on top of K, where E is *one* expression and V is *one* value. The following rules decompose the task of evaluating a list of expressions:

$$k\langle \underset{E \curvearrowright (El,\ \cdot : \mathit{ValList})}{\underline{E : \mathit{Exp}, \mathit{El} : \mathit{ExpNeList}}}\rangle$$

$$\underset{\cdot}{\underline{V : \mathit{Val}}} \curvearrowright (\_\,, \langle \underset{V}{\underline{\cdot}}\,))$$

$$k\langle \underset{E}{\underline{\ \cdot\ }} \curvearrowright ((\underset{\cdot}{\underline{E}}\rangle, \_ : \mathit{ValList})\rangle$$

$$k\langle \underset{\mathit{Vl}}{\underline{(\cdot : \mathit{ExpList}, \mathit{Vl} : \mathit{ValList})}}\rangle$$

The first rule above decomposes the task of evaluating a list of two or more expressions, in two tasks: first evaluate the first expression and then the rest; the remaining expressions are placed in a tuple together with an empty list of values, where the values of already evaluated expressions will be collected one by one. The second rule is applied after the first expression evaluates to a value, say V; that value is added at the end of the list of values in the tuple. One can prove that, for any well-formed program, this rule will always apply only on top of a continuation and only in situations in which V resulted from the evaluation of an expression put on top of the continuation using the first rule above. Since a language will be defined formally in a logical settings admitting initial model semantics, one can prove, potentially using induction, many properties about the definition of a language. While this is an interesting subject by itself, we will not approach it in this paper. Here we will only focus on the definition and the use of the K framework, assuming that all the language specifications are correct "by definition". The third rule above picks the next expression to evaluate from the list and places it in top of the continuation for evaluation. Finally, the last rule keeps only the list of values when all the expressions were evaluated. Note that, by the convention on implicit tuple operation declarations in Section 4.4, only one tuple operation is declared for the rules above, namely $(\_\,,\_) : \mathit{ExpList} \times \mathit{ValList} \to \mathit{ExpListValList}$.

All the contextual rules above are structural (they all use dotted lines). Indeed, none of these rules has any computational meaning; they only move expressions and values to their places in order for the other rules to apply.

The operation $\mathit{bind} : \mathit{VarList} \to \mathit{ContinuationItem}$ generating the continuation item task to bind variables provided in an argument list is crucial in any language definition. There are two uses of this operator:

1. If it is preceded by a list of values in the continuation (generated, e.g., by a variable binding such as a let or a function call) then the variables will be bound to new locations containing those values;

expressions and values; if that is the case, then one can employ the usual technique of wrapping intended subsorts into corresponding operations, in this case by defining operations $\mathit{exp} : \mathit{ExpList} \to \mathit{ContinuationItem}$ and $\mathit{val} : \mathit{ValList} \to \mathit{ContinuationItem}$. However, such a decision would complicate some of the subsequent definitions.

2. Otherwise, it will just bind the variables to new "dangling" locations (needed, e.g., by letrec).

When the lists of variables and values are voided by the first or the second contextual rule below, we simply eliminate the redundant continuation items with the last two structural rules below:

$$k((\underset{\cdot}{\underline{V}}\rangle \curvearrowright bind(\underset{\cdot}{\underline{X}}\rangle\,\rangle \;\; env(\frac{Env}{Env[X \leftarrow L]}) \;\; store(\frac{Store}{Store[L \leftarrow V]}) \;\; nextLoc(\frac{L}{next(L)})$$

$$k(bind(\underset{\cdot}{\underline{X}}\,\rangle\,\rangle \;\; env(\frac{Env}{Env[X \leftarrow L]}) \;\; nextLoc(\frac{L}{s(L)})$$

$$k(\underset{\cdot}{\underline{\cdot : ValList}}\rangle$$

$$\underset{\cdot}{\underline{bind(\cdot)}}$$

In case the program to execute is well-formed, which can be checked with an auxiliary type-checker that can also be defined using the K-framework (see Section 8.1), one can show that the lists of values and of variables to bind have the same length whenever the first rule above matches. The third rule above replaces an empty list of values that appears as a continuation item by the identity on continuations, or in other words, it simply removes it. The fourth rule removes empty lists of bindings. The operations $\_[\_ \leftarrow \_] : Environment \times Var \times Loc \to Environment$ and $\_[\_ \leftarrow \_] : Store \times Loc \times Val \to Store$ are standard update operations that are easy to define and we do not do it here. They update the pair in the first argument whose left element is the second argument, in case such a pair exists, to one whose right element is the third argument. If such a pair does not exist, then they add one formed with their second and third arguments. The operation $next : Loc \to Loc$ gives the next location.

Of particular interest in the rules above is to notice how the various conventions of the K notation work. For example, one can infer that the sort of V is *Val* because V appears in the term $Store[L \leftarrow V]$ and the sort of the last argument of the store update operation is *Val*. That further implies that the first continuation item in the first rule is a *ValList*. Also, note that the state attributes appearing in the rules above may live at different levels in the state structure, which may differ from language definition to language definition. For example, in the case of sequential FUN, these attributes live all at the top-level of the state, but in the case of concurrent FUN, $k(...)$ and $env(...)$ will move from the top-level to the thread state level, by applying the grouping convention. The context transformer conventions will guarantee that the contextual rules above are instantiated accordingly to the state structure of the language under consideration.

Languages with side effects, including FUN, need to write values at *existing* locations. To achieve that, we introduce a new continuation item operation, $write : VarList \to ContinuationItem$, together with its expected definition (like for empty binding, the second rule below is structural):

$$k((\underset{\cdot}{\underline{V}}\,\rangle \curvearrowright write(\underset{\cdot}{\underline{X}}\,\rangle\,\rangle \;\; env\langle(X, L)\rangle \;\; store(\frac{Store}{Store[L \leftarrow V]})$$

$$\underset{\cdot}{\underline{write(\cdot)}}$$

The first rule above is obviously computational, so we used full lines to underline subterms that replace. In the case of concurrent languages, including the extension of FUN in the next section, this rule may in fact be non-deterministic. Indeed, multiple threads may read or write the same locations; different orderings of the applications of these rules when they all match may lead to different behaviors. To properly allow all these behaviors as part of the corresponding rewrite logic initial model semantics, one needs to make sure that the semantics of those language constructs by which threads "compete" on resources are defined with rewrite rules, *not* with equations; non-deterministic rules always correspond to rewrite rules. As already mentioned, if one is interested in just obtaining an interpreter for a language from its K definition, then the distinction between equations and rules is insignificant. But one should be aware of the fact that by defining an interpreter one does *not* define a language, because a language definition should comprise *all* possible behaviors.

In most languages, some language constructs change execution environments. For example, a function call changes the environment (to its closure's) and then binds its parameters in the new environment in order to evaluate is body. Once the result value is computed, the caller's environment needs to be restored. Other language constructs (let, letrec, etc.) also need to restore environments. A uniform way to provide such an environment restoring mechanism is to "freeze" the environment to be restored at the appropriate place in the continuation structure, and then restore it whenever a value is "passed" to it. Thanks to term rewrite engines' capability to maximize sub-term sharing by storing terms as DAGs, this freezing operation can actually be executed quite efficiently: just places a pointer in a list.[11] Therefore, we assume that *Environment* is a subsort of *ContinuationItem* (alternatively, one can introduce a corresponding "wrapper", e.g., *restore* : *Environment* → *ContinuationItem* producing a new continuation item/task) and define environment restoration as a structural rule:

$$k(\_ : Val \curvearrowright \underset{\cdot}{\underline{Env}}\rangle \;\; env(\underset{Env}{\underline{\;\_\;}})$$

5.5 Defining FUN's Features

We are now ready to define formally the sequential FUN language in K. Except features with identical semantics, in what follows we define and discuss *all* the language features. The next subsection will show how the language can be extended with callcc and concurrency via threads and synchronization. Figure 6 contains the complete definition of sequential FUN, including both syntax and semantics. The syntax is on the left and the semantics on the right of the figure. To distinguish the syntactic constructs from other operations defined for semantic reasons, we underline the former; also, if a language construct has alphanumeric characters in its name, then we use a sans serif font. We believe that most of the definition is self-explanatory.

5.5.1 Global Operations: Eval and Result

Unlike SOS languages definitions where the semantics of each language construct is given mentioning a complete (state) configuration, in K there are very few *global* contextual rules, i.e., ones that need the entire state structure. We have encountered only two types of global rules, ones for initiating

[11] A trickier consequence of this, which works in our favor, is that the frozen environment will *not* be allowed to be garbage-collected by the rewrite engine in case there is no other reference to it.

the computation and the others for collecting and reporting the result of the computation. We believe that there is little or no chance to remove such rules entirely. Moreover, in our experience, such rules are rather meaningful and useful, because they give one a global understanding of how the various fragments of state fit together in one's language. We found that an early understanding of how to initiate the computation and how to retrieve the final results is so important, that we prefer to start every language definition with the semantic definitions of the global operations.

In sequential FUN we need to define an operation "$\mathit{eval} : \mathit{Exp} \times \mathit{IntList} \to \mathit{IntList}$" which takes an expression E and an input list of integers Il and reduces to an output list of integers, the result of evaluating E under the input Il. This operation creates an initial state containing the expression in the continuation and the list of integers in the input buffer, all the other structures being empty, nil or 0, depending on their type; this initial state is wrapped by an auxiliary operator "$\mathit{result} : \mathit{State} \to \mathit{IntList}$", whose role is to collect the result of the computation when ready:

$$\frac{\mathit{eval}(E, \mathit{Il})}{\mathit{result}(k(E)\ \mathit{env}(\cdot)\ \mathit{control}(\mathit{fstack}(\cdot)\ \mathit{xstack}(\cdot)\ \mathit{lstack}(\cdot))\ \mathit{in}(\mathit{Il})\ \mathit{out}(\cdot)\ \mathit{store}(\cdot)\ \mathit{nextLoc}(0))}$$

The *eval* and *result* operations above divide-and-conquer the task of evaluating an expression.

The contextual rules for the various language features in the sequel will all modify the state structure wrapped by *result*. The result to be output is available when there is no computation to be done in the continuation. Since the evaluation of an expression at the top of the continuation structure wrapped by "$k(...)$" eventually generates a value on top of the same remaining continuation (empty in our case) if it terminates, we can define the semantics of *result* as the following structural rule:

$$\frac{\mathit{result}\langle k(\_ : \mathit{Val})\ \mathit{out}(\mathit{Il})\rangle}{\mathit{Il}}$$

In words, the collected list of integers in the output buffer is returned as the result of the evaluation whenever there is nothing but a value in the continuation; that value left in the continuation is simply discarded, as well as all the state structures.

5.5.2 Syntactic Subcategories: Variables, Bool, Int, Real

Each syntactic subcategory of *Exp* needs to be given a semantics. We declared via subsorting that variables, booleans, integers and reals are all expressions. We split the syntactic subcategories in two groups:

1. Syntactic subcategories that come with no meaning from their existing domain, or whose existing meaning is not intended to be inherited in the programming language. Variables form such a syntactic subcategory. Variables are plain identifiers, which may come with their semantic definition as strings of characters, including concatenation, etc., but which is neither interesting nor meaningful for our purpose of using identifiers as variables. We want variables to have the meaning of holders of values in our language, not of strings of characters. A variable X evaluates to the value V contained in the store at the appropriate location, which is found in the environment:

$$k(\underset{V}{\underline{X}}\rangle\ \mathit{env}\langle(X, L)\rangle\ \mathit{store}\langle(L, V)\rangle$$

This contextual rule is computational (so the use of a full line instead of a dotted one) and, like the rule for variable writing in Section 5.4, this rule may lead to non-deterministic behaviors of programs if the language is concurrent. Consequently, we will declare this rule "non-deterministic" in Section 6 when we add threads to FUN.

2. Syntactic subcategories whose meaning is intended to be cast in the programming language using them, such as booleans and integers. Elements of these sorts become values in the language semantics, so they are defined as subsorts of *both Exp* and *Val*. Therefore, both sorts *Exp* and *Val* have a special meaning and treatment in K. K adds automatically an operation *valuesort* : *Valuesort* → *Val* for each sort *Valuesort* that is declared a subsort of *Val*. In our case, operations *bool* : *Bool* → *Val*, *int* : *Int* → *Val* and *real* : *Real* → *Val* are added. For those sorts *Valuesort* that are subsorts of both *Val* and *Exp*, a contextual structural rule of the form

$$k(\underset{\mathit{valuesort}(\mathit{Ev})}{\underline{\quad \mathit{Ev} \quad}}\rangle$$

is automatically considered in K, saying that whenever an expression *Ev* of sort *Valuesort* appears at the top of the continuation, that is, if it is the next task to evaluate, then simply replace it by the corresponding value, *valuesort*(*Ev*). Therefore, elements of sort *Valuesort* that appear in expressions are instantaneously, i.e., at no computational expense, regarded as values borrowed from their domain. In our case, the following three contextual rules are automatically added by K:

$$k(\underset{\mathit{bool}(B)}{\underline{\quad B \quad}}\rangle$$

$$k(\underset{\mathit{int}(I)}{\underline{\quad I \quad}}\rangle$$

$$k(\underset{\mathit{real}(R)}{\underline{\quad R \quad}}\rangle$$

5.5.3 Operator Attributes

We next discuss the operator attributes that decorate some of the language construct declarations in our K definitions. We here discuss only the programming language specific attributes and explain how they can be automatically translated into contextual rules; standard algebraic specification or rewriting logic operator attributes can also be used, such as associativity, commutativity, etc., with precisely the same meaning as in algebraic specification and rewriting logic.

Attributes of operators are listed in square brackets, following the declaration of the operation. Note, for example, that many operations have an exclamation mark attribute; some operations also have other operations as attributes, such as the arithmetic operations of addition, comparison, etc. The exclamation mark, called *strictness attribute*, says that the operation is *strict* in its arguments in the order its arguments appear, that is, its arguments are evaluated from left to right and then the actual operation is "applied"; we will shortly explain what we mean by "applied" in this context. An exclamation mark attribute can take a list of numbers as optional argument and an operation as optional attribute; e.g., the conditional has the strictness attribute "!(1)[*if*]", the application has the strictness attribute "![*app*]", etc. As shown below, these decorations of the strictness attribute can be fully and automatically translated into corresponding contextual rules.

If the exclamation mark attribute has a list of numbers as argument, then it means that the operation is declared strict in those arguments in the given order. Therefore, a missing argument of a strictness attribute is just a syntactic sugar convenience; indeed, an operator with n arguments which is declared just "strict" using a plain "!" attribute, is entirely equivalent to one whose strictness attribute is "$!(1\ 2 \ldots n)$". When evaluating an operation declared using a strictness attribute, the arguments in which the operation was declared strict (i.e., the explicit or implicit argument list of "!") are first scheduled for evaluation; the remaining arguments need to be "frozen" until the other arguments are evaluated. If the exclamation mark attribute has an attribute, say *att*, then *att* will be used as a continuation item constructor to "wrap" the frozen arguments. Similarly, the lack of an attribute associated to a strictness attribute corresponds to a default attribute, which by convention has the same name as the corresponding operation; to avoid confusion with underscore variables, we drop all the underscores from the original language construct name when calculating the default name of the attribute of "!". For example, an operation declared "$\_ + \_ \ : \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}$ [!]" is automatically desugared into "$\_ + \_ \ : \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}$ [!(1 2)[+]]".

Let us next discuss how K generates corresponding rules from strictness attributes. Suppose that we declare a strict operation, say *op*, whose strictness attribute's attribute is *att*. Then K automatically adds

(a) an auxiliary operation *att* to the signature, whose arguments (number, order and sorts) are precisely the arguments of *op* in which *op* is *not* strict; by default, the result sort of this auxiliary operation is *ContinuationItem*;

(b) a rule initiating the evaluation of the strict arguments in the specified order, followed by the other arguments "frozen", i.e., wrapped, by the corresponding auxiliary operation *att*.

For example, in the case of FUN's conditional whose strictness attribute is "$!(1)[\mathit{if}]$", an operation "$\mathit{if}\ : \mathit{Exp} \times \mathit{Exp} \to \mathit{ContinuationItem}$ is automatically added to the signature, together with a rule

$$\frac{\mathsf{if}\ B\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2}{B \curvearrowright \mathit{if}(E_1, E_2)}$$

If the original operation is strict in all its arguments, like not, $\_ + \_$ and $\_\_$, then the auxiliary operation added has no arguments (it is a constant). For example, in the case of λ_{K}'s application whose strictness attribute is "$![\mathit{app}]$", an operation "$\mathit{app}\ : \to \mathit{ContinuationItem}$ is added to the signature, together with a rule

$$\frac{E\ \mathit{El}}{(E, \mathit{El}) \curvearrowright \mathit{app}}$$

There is an additional category of attributes associated to operations that are already declared strict, consisting of other operations, such as $\_{+_{\mathit{Int}}}\_ : \mathit{Int} \times \mathit{Int} \to \mathit{Int}$ and $\_{+_{\mathit{Real}}}\_ : \mathit{Real} \times \mathit{Real} \to \mathit{Real}$ declared as attributes of $\_ + \_ \ : \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}$. We call these attributes *builtin (operation) attributes*. Each of these builtin operation attributes corresponds to precisely one K-rule that "calls" the corresponding builtin operation on the right arguments. For example, the implicit rules corresponding to the three attributes of the addition operator in Figure 6 are:

$$\frac{E_1 + E_2}{(E_1, E_2) \curvearrowright +}$$

$$\frac{(\mathit{int}(I_1), \mathit{int}(I_2)) \curvearrowright +}{\mathit{int}(I_1 +_{\mathit{Int}} I_2)}$$

$$\frac{(\mathit{real}(R_1), \mathit{real}(R_2)) \curvearrowright +}{\mathit{real}(R_1 +_{\mathit{real}} R_2)}$$

The three rules above can be generated entirely automatically, by just investigating the attributes of the addition operator. To generate the second rule, for example, one first notes that the arguments of the builtin operation $\_ +_{\mathit{Int}} \_$ are both *Int*, so one can generate the list of values $(\mathit{int}(I_1), \mathit{int}(I_2))$ that is expected to be "passed" to the continuation item "+", and then one notes that the result of $\_ +_{\mathit{Int}} \_$ is also *Int*, so one can generate the right resulting value $\mathit{int}(I_1 +_{\mathit{Int}} I_2)$. Similarly, the three rules corresponding to the attributes of "$\_ \leq \_ : \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}$ [!, $\_ \leq_{\mathit{Int}} \_ : \mathit{Int} \times \mathit{Int} \to \mathit{Bool}$, $\_ \leq_{\mathit{Real}} \_ : \mathit{Real} \times \mathit{Real} \to \mathit{Bool}$]" are

$$\frac{E_1 \leq E_2}{(E_1, E_2) \curvearrowright \leq}$$

$$\frac{(\mathit{int}(I_1), \mathit{int}(I_2)) \curvearrowright \leq}{\mathit{bool}(I_1 \leq_{\mathit{Int}} I_2)}$$

$$\frac{(\mathit{real}(R_1), \mathit{real}(R_2)) \curvearrowright \leq}{\mathit{bool}(R_1 \leq_{\mathit{real}} R_2)}$$

Also, the two rules corresponding to "$\mathsf{not}\_ : \mathit{Exp} \to \mathit{Exp}$ [!, $\mathsf{not}_{\mathit{Bool}} : \mathit{Bool} \to \mathit{Bool}$]" are

$$\frac{\mathsf{not}\ E}{E \curvearrowright \mathit{not}}$$

$$\frac{\mathit{bool}(B) \curvearrowright \mathit{not}}{\mathit{bool}(\mathsf{not}_{\mathit{Bool}}(B))}$$

A builtin operation attribute can also take as argument a list of values (as opposed to a fixed number of arguments, like the addition and comparison attributes above); if that is the case, then all the list of values is passed to the builtin operation. For example, the two contextual rules generated automatically from the attributes of the list constructor operator "$[\_] : \mathit{ExpList} \to \mathit{Exp}$ [!, $[\_] : \mathit{ValList} \to \mathit{Val}$]" are the following:

$$\frac{[\mathit{El}]}{\mathit{El} \curvearrowright []}$$

$$\frac{\mathit{Vl} : \mathit{ValList} \curvearrowright []}{[\mathit{Vl}]}$$

As the example above shows, "builtin" operation attributes need not necessarily refer only to operations whose semantics have already been defined as part of some libraries. K simply desugars blindly all the operation attributes: it first adds to the signature all the operations declared as builtin operation attributes (we here assume no errors in case of multiple declarations of the same operator in the signature; the duplicate declarations are ignored), and then generates the corresponding contextual rules.

There is one more case to discuss here, namely the one where the operation declared a builtin operation attribute has no arguments, such as the operation "$\mathsf{skip} : \to \mathit{Exp}\ [\mathit{unit} : \to \mathit{Val}]$". Since constant constructs are strict by default, we refrain from declaring them a strictness attribute and also from adding the redundant continuation item. In other words, we add only one contextual rule corresponding to the skip operation,

$$\frac{\mathsf{skip}}{\mathit{unit}}$$

instead of the two below that one would add following blindly the general desugaring procedure (in addition to defining the redundant operation "$\mathit{skip} : \to \mathit{ContinuationItem}$"):

$$\frac{\mathsf{skip}}{\mathit{skip}}$$

$$\frac{\mathit{skip}}{\mathit{unit}}$$

Notice that these rules associated to strictness attributes and builtin operations can apply anywhere they match, not only at the top of the continuation. In fact, the rules with exclamation mark attributes can be regarded as some sort of "precompilation rules" that transform the program into a tree (because continuations can be embedded) that is more suitable for the other semantic rules. If, in a particular language definition, code is not generated dynamically, then these precompilation rules can be applied all "statically", that is, before the semantic rules on the right apply; one can formally prove that, in such a case, these "pre-compilation" rules need not be applied anymore during the evaluation of the program. These rules associated to exclamation mark attributes should *not* be regarded as transition rules that "evolve" a program, but instead, as a means to put the program into a more convenient but *computationally equivalent* form; indeed, when translated into rewriting logic, the precompilation contextual rules become all equations, so the original program and the precompiled one are in the same equivalence class, that is, they are *the same* program within the initial model semantics that rewriting logic comes with.

5.5.4 The Conditional

We first translate any if_then_ into an if_then_else_, using the structural contextual rule

$$\frac{\mathsf{if}\ B\ \mathsf{then}\ E}{\mathsf{if}\ B\ \mathsf{then}\ E\ \mathsf{else\ skip}}$$

Since the rule above is structural, it does not count as a computation step, exactly as desired.

Thanks to the rule generated automatically from the strictness attribute of the conditional constructor, we only need to define the following two (computational) rules:

$$\frac{bool(true) \curvearrowright if(E_1, E_2)}{E_1}$$

$$\frac{bool(false) \curvearrowright if(E_1, E_2)}{E_2}$$

Note that the expressions E_1 and E_2 can be pre-processed into their corresponding continuation structures; this corresponds to "compilation-on-the-fly" and can be beneficial when executed on parallel rewrite engines[12]. It is interesting to note that E_1 or E_2 needs not be in normal form when selected (by one of the two bottom rules above) to be placed on top of the continuation. In other words, some pre-processing of E_1 and/or E_2 may take place during the evaluation of the condition, while the remaining part of pre-processing would be performed after the expression is selected.

5.5.5 Functions

To define the semantics of functions and function applications, closure values need to be added; we introduce a new value constructor $closure : VarList \times Exp \times Environment \to Val$ for this purpose. Then the semantics of function declaration is

$$k(\underset{closure(Xl, E, Env)}{\underline{\mathsf{fun}\ Xl \to E}}\rangle\ env(Env)$$

As for the conditional, some pre-processing on-the-fly of the body of the function within the closure can (safely) take place. Since we want to be able to exit a function abruptly using `return`, we need to freeze the control state and the environment when a function is called. We do it via a tuple operator $(\_, \_, \_)$ that stacks useful information in *fstack*, as well as a continuation item $popFstack : \to ContinuationItem$ telling when the function call terminates; this covers the normal termination of a function, that is, the one in which its body evaluates as a normal expression, without a return:

$$(k(\underset{Vl \curvearrowright bind(Xl) \curvearrowright E \curvearrowright popFstack}{\underline{(closure(Xl, E, Env), Vl) \curvearrowright app \curvearrowright K}})\ fstack(\underset{(K, Env', C)}{\underline{\quad\cdot\quad}}\rangle\ C : Ctrl)\ env(\underset{Env}{\underline{Env'}})$$

The control structures remain unchanged when a function returns normally, so there is no need to recover them in the rule below. However, the environment does need to change, because the function body is evaluated in an extended environment, namely one that provides bindings for the function parameters:

$$k(\_ : Val \curvearrowright \underset{K}{\underline{popFstack}})\ fstack(\underset{\cdot}{\underline{(K, Env, \_)}}\rangle\ env(\underset{Env}{\underline{\ \_\ }})$$

As a general rule, if one is not sure whether certain structures in the state change or not during the evaluation of a particular expression or language construct, then it is safer to recover those structures anyway; then, if one realizes informally or proves formally that some structures do not

[12] There is no problem here with propagation of unexpected side effects or non-termination of rewriting, because values can only be introduced at the top of the continuation attribute, $k(...)$.

change, then one can refine the definition of the corresponding feature appropriately – one can regard this later step as an "optimization" of the language definition.

The semantics of return proceeds as expected:

$$(k(\_ : \mathit{Val} \curvearrowright \underset{K}{\underline{\mathit{return} \curvearrowright \_}})\ \mathit{fstack}(\underset{\cdot}{\underline{(K, \mathit{Env}, C)}}\rangle\ \underset{C}{\underline{\_ : \mathit{Ctrl}}})\ \mathit{env}(\underset{\mathit{Env}}{\underline{\ \_\ }})$$

5.5.6 Let and Letrec

The semantics of let and letrec are straightforward and self-explanatory. All we need to do is to put the two constructs in a computationally equivalent continuation form, using for each of them precisely one structural rule:

$$k(\underset{\mathit{El} \curvearrowright \mathit{bind}(\mathit{Xl}) \curvearrowright E \curvearrowright \mathit{Env}}{\underline{\mathsf{let}(\mathit{Xl}, \mathit{El}, E)}}\rangle\ \mathit{env}(\mathit{Env})$$

$$k(\underset{\mathit{bind}(\mathit{Xl}) \curvearrowright \mathit{El} \curvearrowright \mathit{write}(\mathit{Xl}) \curvearrowright E \curvearrowright \mathit{Env}}{\underline{\mathsf{letrec}(\mathit{Xl}, \mathit{El}, E)}}\rangle\ \mathit{env}(\mathit{Env})$$

5.5.7 Sequential Composition and Assignment

Sequential composition can in principle be expressed using let, but, for clarity and simplicity, we prefer to define it independently. Since we defined the sequential composition constructor strict in Figure 6, all we need to do now is to define a rule saying how to "combine" the two values obtained after evaluating the two subexpressions:

$$(\underset{\cdot}{\underline{V_1 : \mathit{Val},}} V_2 : \mathit{Val}) \curvearrowright \underset{\cdot}{\underline{;}}$$

Therefore, our semantics of sequential composition here is that the first subexpression is evaluated only for its side-effects, and the result value of the sequential composition is the result value of its second subexpression: indeed, we only keep V_2, the value of the second subexpression, discarding both the first value and the continuation item corresponding to sequential composition. One should not get tricked here and say that our semantics does not deal well with abrupt change of control, wrongly thinking that sequential composition always expects its subexpressions to evaluate to values, so its subexpressions cannot, for example, throw exceptions.

As we'll see shortly, abrupt changes of control consist of discarding the remaining computation. Since the sequential composition was declared strict, its two subexpressions were scheduled first for evaluation, sequentially in a list, while the corresponding sequential composition continuation item was placed next in the continuation structure to "remind" us what to do *just in case* both subexpressions evaluate normally. In case the first expression throws an exception, then the second expression will never even be considered for evaluation, being discarded together with everything else following the control-changing statement. The above may seem straightforward, but it is a crucial aspect of our definitional style, that is the basis of the modularity of our language definitions. Unlike in a conventional SOS semantics, we need to change *nothing* in the definition of the other language constructs when we add exceptions or any other control-intensive language construct.

Assignment is used for its side effect only, and it evaluates to *unit*:

$$\frac{X := E}{E \curvearrowright write(X) \curvearrowright unit}$$

5.5.8 Lists

One can have different ways to define lists. We here take a simple approach, which may not work in all language definitions, that lists are regarded just as any other values in the language, in particular being stored at *one* location in the store; alternatively, one could consider lists as pairs value/location, and then store them element by element in the store. Thanks to the strictness attributes (“!”) of all the list language constructs and to the builtin operation (“[_] : *ValList* → *Val*”) attribute of the language construct “[_] : *ExpList* → *Exp*”, we need to only define the semantics of the list operators on expression lists of the form “[*Vl*]”; we only give the semantics of car and cons, the other operations' semantics being similar:

$$\frac{[V : Val, \_] \curvearrowright car}{V}$$

$$\frac{(V, [Vl]) \curvearrowright cons}{[V, Vl]}$$

5.5.9 Input/Output: Read and Print

When read() is the next task to evaluate in the continuation, it grabs and returns the next integer in the input buffer; therefore, read() has a side effect on the state (it modifies the input buffer):

$$k\langle \underset{int(I)}{\underline{\mathsf{read}()}} \rangle \;\; in\langle \underset{\cdot}{\underline{I}} \rangle$$

Print also has a side effect, this time on the output buffer:

$$k\langle \underset{unit}{\underline{int(I) \curvearrowright print}} \rangle \;\; out\langle \underset{I}{\underline{\cdot}} \rangle$$

The output buffer will be eventually returned as the result of the evaluation of the original program. Since both read and print have side effects, their rules can lead to non-deterministic behaviors of the program if the language was concurrent. Consequently, we are going to declare the two rules above “non-deterministic” when we extend the language with threads.

For simplicity, we only allow input/output operations with integers; if one wants to allow such operations with other values as well, then one would need to change the input/output buffers in the state (wrapped by the attributes *in* and *out*) to take other values as well, and then correspondingly change the definitions of read and print.

Note that the input and output are *not* interactive: the input is expected to be provided with the original program, while the output will be returned all at the end of the computation. That is because K, like algebraic specification and rewriting logic, is a pure “formalism”, so its reduction process cannot be stopped for I/O operations. While this is acceptable for defining the semantics of a language, it is clearly inconvenient if one's goal is to use it to execute arbitrary programs. We do not discuss this issue in depth here, but just mention that one can adopt at least two solutions:

1. slightly modify the semantics to use a rewrite engine feature such as Maude's loop mode [5] which was designed specifically for dealing with I/O and user interaction; or
2. provide functionality in the rewrite engine to link certain lists to I/O buffers.

5.5.10 Parametric Exceptions

Parametric exceptions can be defined with just three contextual rules (we do not count the trivial one generated automatically from the strictness attribute of throw):

1. One structural rule for initiating the evaluation of the try_catch(_)_ statement. Operation "$\mathit{popXstack} : \to \mathit{ContinuationItem}$" is needed to mark the execution of the main body of the exception code; returning information is also stacked with a tupling operation:

$$(k(\underset{E' \curvearrowright \mathit{popXstack}}{\underline{\mathsf{try}\ E'\ \mathsf{catch}(X)\ E \curvearrowright K}})\ \mathit{xstack}(\underset{(X, E, \mathit{Env}, K, C)}{\underline{\qquad\cdot\qquad}}\rangle\ C : \mathit{Ctrl})\ \mathit{env}(\mathit{Env})$$

2. One structural contextual rule for normal, non-exceptional evaluation. If E' evaluates normally, i.e., if its evaluation does not encounter any exception throwing, then the execution is recovered with the original continuation; note that one only needs to recover the continuation, the other control ingredients being the same:

$$k(\_ : \mathit{Val} \curvearrowright \underset{K}{\underline{\mathit{popXstack}}})\ \mathit{xstack}(\underset{\cdot}{\underline{(\_, \_, \_, K, \_)}}\rangle$$

3. One for performing the actual change of context in case of an exception throwing. The expression which is thrown is first evaluated (note that there is no problem if that expression throws itself an exception) and then its value is bound to the parameter and the execution context recovered:

$$(k(\_ : \mathit{Val} \curvearrowright \underset{\mathit{bind}(X) \curvearrowright E \curvearrowright \mathit{Env} \curvearrowright K}{\underline{\mathit{throw} \curvearrowright \_}})\ \mathit{xstack}(\underset{\cdot}{\underline{(X, E, \mathit{Env}, K, C)}}\rangle\ \underset{C}{\underline{\_ : \mathit{Ctrl}}})\ \mathit{env}(\underset{\mathit{Env}}{\underline{\ \_\ }})$$

5.5.11 Loops with Break and Continue

Because of break and continue, which have a similar behavior to (non-parametric) exceptions, an additional stack for loops is necessary in the control soup of the state. Also, a helping special continuation marker "$\circlearrowleft : \to \mathit{ContinuationItem}$" is defined. Since continue has a more general semantics for "for" loops than for "while" loops (the "step", J below, is executed), we first translate "while" into "for" using a structural rule:

$$\underset{\mathsf{for}(\mathsf{skip};\ B;\ \mathsf{skip})\ E}{\underline{\mathsf{while}(B)\ E}}$$

Whenever a loop is encountered, an appropriate execution context is created, by stacking information needed to recover in case of a "break" or "continue":

$$(k(\underset{S;\ B \curvearrowright \circlearrowleft}{\underline{\mathsf{for}(S;\ B;\ J)\ E}} \curvearrowright K)\ \mathit{lstack}(\underset{(B, E, J, \mathit{Env}, K, C)}{\underline{\qquad\cdot\qquad}}\rangle\ C : \mathit{Ctrl})\ \mathit{env}(\mathit{Env})$$

There are two possibilities for the evaluation of B (false or true):

$$k(\underset{\mathit{unit} \curvearrowright K}{\underline{\mathit{bool}(\mathit{false}) \curvearrowright \circlearrowleft}})\ \mathit{lstack}(\underset{\cdot}{\underline{(\_, \_, \_, \_, K, \_)}}\rangle$$

$$k(\underset{E; J; B}{\underline{\mathit{bool}(\mathit{true})}} \curvearrowright \circlearrowleft)\ \mathit{lstack}((B, E, J, \_, \_, \_)\rangle$$

With the infrastructure above, the semantics of break and continue are immediate:

$$(k(\underset{\mathit{unit} \curvearrowright K}{\underline{\mathsf{break} \curvearrowright \_}})\ \mathit{lstack}(\underset{\cdot}{\underline{(\_, \_, \_, \mathit{Env}, K, C)\rangle}}\ \underset{C}{\underline{\_}} : \mathit{Ctrl})\ \mathit{env}(\underset{\mathit{Env}}{\underline{\ \_\ }})$$

$$(k(\underset{J; B \curvearrowright \circlearrowleft}{\underline{\mathsf{continue} \curvearrowright \_}})\ \mathit{lstack}((B, \_, J, \mathit{Env}, \_, C)\rangle\ \underset{C}{\underline{\_}} : \mathit{Ctrl})\ \mathit{env}(\underset{\mathit{Env}}{\underline{\ \_\ }})$$

6 On Modular Language Design

In this section we show how the K framework addresses the important aspect of *modularity* in programming language definition and design. By "modular" in this context it is meant significantly more than just grouping together all the rules corresponding to a particular language feature in a module, which is ultimately what most existing language definitional frameworks do anyway, formally or informally. What a framework supporting modular language design means here is one having

> *the capability to add or remove programming language features without a need to modify any of the other, unrelated language features.*

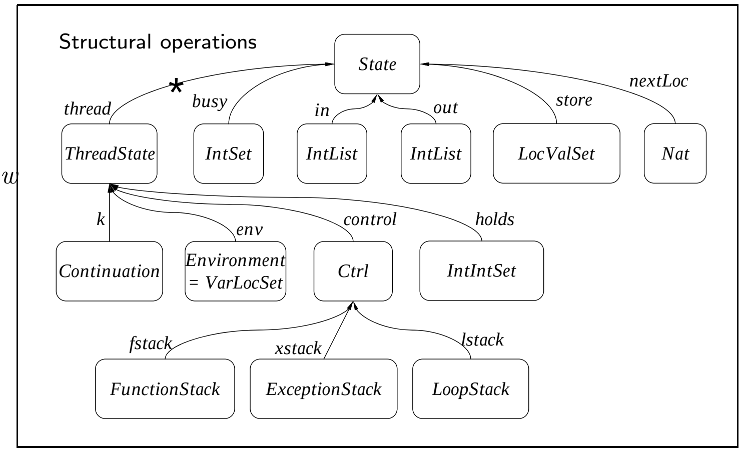
Most existing definitional frameworks either cannot support properly some important language features at all, or when they do, they do not do it in a modular manner. A notorious example is the big-step SOS, also called "natural" semantics and discussed in Section 3.1, which cannot be used at all to define non-trivial concurrent programming languages, and which is hardly suitable for defining features that change the control abruptly, such as exceptions, halt, break, continue, etc. For example, to define an innocent "halt(E)" statement, one would need to add for each existing language construct as many new rules as subexpressions that can halt the program that language construct has. From a modular language design perspective, it is hard to imagine anything worse than that in a language definitional framework.

In this section we discuss the modularity of K both empirically and theoretically. Empirically, we show how one can add to the sequential FUN language defined in the previous section two important and nontrivial features, namely a callcc construct and concurrency via threads and synchronization with locks. Figure 7 shows the new operations and rules that need to be added and the changes that need to be applied to the definition of sequential FUN to accommodate the new language features; note that no change needs to be done to support callcc, and that the changes needed to support concurrency are global, not specific to other language features. On the theoretical side, we show how Mosses' Modular SOS (MSOS) [21], designed specifically to address the lack of modularity of standard SOS, can be translated into K. This is not surprising, because K *is* rewriting logic and MSOS has already been shown to map into rewriting logic [4], but it is instructive because it shows the connections and relationships between two modular frameworks.

6.1 Adding Call/cc to FUN

Call with the current continuation, or call/cc, is a very powerful control-intensive language construct. Scheme is one of the most known languages providing an explicit call/cc construct, though other languages also provide it, or at least some limited version of it. Briefly, call/cc takes an argument, expected to evaluate to a function, and passes it the current computation context. If the function that was the argument of call/cc passes its argument any value, then the computation is reset back to the point the call/cc took place. This can be regarded as a jump back in time, recovering also the original computation context. Because of the variety of control-intensive FUN language constructs, call/cc in FUN needs to freeze more than the current continuation. It must actually freeze the entire control context. To understand why this is needed, imagine that the call/cc appears in the body of some function, in some nested loops and try/catch blocks; once a value is passed back to the context in which the call/cc took place, one would, obviously, want to still be able to returning from the function using return, to breaking/continuing loops, as well

The definition of sequential FUN in
Figure 6 needs to change as follows:
1) add callcc *and its two rules below*
2) replace structural operators with
the ones in the picture to the right
3) replace eval and result definitions as below
4) declare non-deterministic the rules
for write, variable lookup, read and print
5) add six more rules for threads:
a) one rule for creation of threads
and one for termination of threads;
b) two rules for acquiring a lock
c) two rules for releasing a lock
No other changes needed.



$$
\mathit{newThread} : \mathit{Exp} \times \mathit{Environment} \to \mathit{StateAttribute}\} \;\ldots\; \begin{cases}
\dfrac{\mathit{eval}(E, \mathit{Il})}{\mathit{result}(\mathit{newThread}(E, \cdot)\ \mathit{busy}(\cdot)\ \mathit{in}(\mathit{Il})\ \mathit{out}(\cdot)\ \mathit{store}(\cdot)\ \mathit{nextLoc}(0))} \\[2ex]
\dfrac{\mathit{result}(\mathit{busy}(\_)\ \mathit{in}(\_)\ \mathit{out}(\mathit{Il})\ \mathit{store}(\_)\ \mathit{nextLoc}(\_))}{\mathit{Il}} \\[2ex]
\dfrac{\mathit{newThread}(E, \mathit{Env})}{\mathit{thread}(k(E)\ \mathit{env}(\mathit{Env})\ \mathit{control}(\mathit{fstack}(\cdot)\ \mathit{xstack}(\cdot)\ \mathit{lstack}(\cdot))\ \mathit{holds}(\cdot))}
\end{cases}
$$

$$
\left.\begin{array}{l}
\underline{\mathsf{callcc}}\ :\ \mathit{Exp} \to \mathit{Exp}\ [!] \\
\mathit{cc} : \mathit{Continuation} \times \mathit{Ctrl} \times \mathit{Environment} \to \mathit{Val}
\end{array}\right\} \ldots\ldots\ldots \begin{cases}
(k(\underset{(V, \mathit{cc}(K, C, \mathit{Env})) \curvearrowright \mathit{app}}{\underline{V : \mathit{Val} \curvearrowright \mathit{callcc}}} \curvearrowright K)\ C{:}\mathit{Ctrl})\ \mathit{env}(\mathit{Env}) \\
(k(\underset{V \curvearrowright K}{\underline{((\mathit{cc}(K, C, \mathit{Env}), V) \curvearrowright \mathit{app} \curvearrowright \_)}}\ \underset{C}{\underline{\_{:}\mathit{Ctrl}}})\ \mathit{env}(\underset{\mathit{Env}}{\underline{\ \_\ }})
\end{cases}
$$

$$
\left.\begin{array}{l}
\underline{\mathsf{spawn}}\ :\ \mathit{Exp} \to \mathit{Exp} \\
\underline{\mathsf{acquire}} :\ \mathit{Exp} \to \mathit{Exp}\ [!] \\
\underline{\mathsf{release}}\ :\ \mathit{Exp} \to \mathit{Exp}\ [!]
\end{array}\right\} \ldots\ldots \begin{cases}
\mathit{thread}\langle k(\underset{\mathit{unit}}{\underline{\mathsf{spawn}(E)}}\rangle\ \mathit{env}(\mathit{Env})\rangle\ \underset{\mathit{newThread}(E, \mathit{Env})}{\underline{\cdot}} \\
\underset{\cdot}{\underline{\mathit{thread}\langle k(\_ : \mathit{Val})\ \mathit{holds}(\mathit{LCs})\rangle}}\ \mathit{busy}(\underset{\mathit{Is} - \mathit{LCs}}{\underline{\mathit{Is}}}) \;\Big|\; k(\underset{\mathit{unit}}{\underline{\mathit{int}(I) \curvearrowright \mathit{acquire}}}\rangle\ \mathit{holds}\langle(I, \underset{s(N)}{\underline{N}})\rangle \\
\boxed{k(\underset{\mathit{unit}}{\underline{\mathit{int}(I) \curvearrowright \mathit{acquire}}}\rangle\ \mathit{holds}\langle\underset{(I, 0)}{\underline{\cdot}}\rangle\ \mathit{busy}(\underset{\mathit{Is}, I}{\underline{\mathit{Is}}}) \Leftarrow \mathit{not}(I\ \mathit{in}\ \mathit{Is})} \\
k(\underset{\mathit{unit}}{\underline{\mathit{int}(I) \curvearrowright \mathit{release}}}\rangle\ \mathit{holds}\langle(I, \underset{N}{\underline{s(N)}})\rangle \;\Big|\; k(\underset{\mathit{unit}}{\underline{\mathit{int}(I) \curvearrowright \mathit{release}}}\rangle\ \mathit{holds}\langle\underset{\cdot}{\underline{(I, 0)}}\rangle\ \mathit{busy}\langle\underset{\cdot}{\underline{I}}\rangle
\end{cases}
$$

Figure 7: Adding call/cc and threads to FUN

as to throwing exceptions, exactly like in the original context. Therefore, in our language, call/cc appears to better abbreviate "call with current context".

Even though call/cc is conceptually more powerful than our other control-intensive language constructs (their translation would be technically involved, though), it is actually very easy to define in our framework. A special "current context" value is needed, so we define an operation $\mathit{cc} : \mathit{Continuation} \times \mathit{Ctrl} \times \mathit{Environment} \to \mathit{Val}$, as well as a corresponding continuation item, $\mathit{callcc} : \to \mathit{ContinuationItem}$. Note that callcc is declared strict in its argument. When a value (expected to be a function closure, namely the evaluated argument expression of callcc) is passed to it at the top of the continuation, a special "control context" value is created and passed to its argument function:

$$(k(\underset{(V, \mathit{cc}(K, C, \mathit{Env})) \curvearrowright \mathit{app}}{\underline{V : \mathit{Val} \curvearrowright \mathit{callcc}}} \curvearrowright K)\ C{:}\mathit{Ctrl})\ \mathit{env}(\mathit{Env})$$

If the special control context value is ever passed a value, then the original execution context is restored and the value is passed to it:

$$(k(\underset{V \curvearrowright K}{\underline{((cc(K,C,Env),V) \curvearrowright app \curvearrowright \_}})\ \underset{C}{\underline{\_ : Ctrl}})\ env(\underset{Env}{\underline{\ \_\ }})$$

6.2 Adding Concurrency to FUN

To add threads to a language, the structure of the state typically needs to change. That is because threads contain and repeat part of the state, including a continuation, an environment and a control structure. The picture in Figure 7 shows the new structure of the state of FUN. There are several new attributes and some old ones have been restructured:

- An attribute *thread* has been added, which is allowed to multiply as part of the state (because of the star "$\star$" on its edge to *State*);

- The continuation, environment and control previously at the top level in the state, were moved within the *thread* attribute; indeed, each thread needs enough functionality to carry over its own computation; the store will be shared among all threads, so it remains at the top level in the state;

- A new attribute, *busy*, has been added at the top level in the state, which maintains the set of (integer values of) locks that are being held by the different threads; this is needed for thread synchronization: a thread is not allowed to acquire a lock which is busy;

- A new attribute, *holds*, has been added to the thread soup, which contains all the locks held by the corresponding thread. A tricky aspect of thread synchronization is that a thread can acquire the same lock multiple times; however, it then needs to release it the same number of times. For that reason, the holds attribute maintains a set of pairs (lock,counter).

Together with a change in the structure of the state, the semantics of all the operations that create states need to change accordingly. In our case, the semantics of *eval* needs to change, to incorporate the new desired structure of state. To ease thread creation later, we prefer to add an auxiliary *newThread* operation[13] which takes an expression and an environment and creates a new thread:

$$\frac{newThread(E, Env)}{thread(k(E)\ env(Env)\ control(fstack(\cdot)\ xstack(\cdot)\ lstack(\cdot))\ holds(\cdot))}$$

Then we use this operation to create the initial execution thread that contains the expression to evaluate and an empty environment:

$$\frac{eval(E, Il)}{result(newThread(E, \cdot)\ busy(\cdot)\ in(Il)\ out(\cdot)\ store(\cdot)\ nextLoc(0))}$$

Threads can not only be created dynamically, but can also be terminated dynamically. In sequential FUN, the execution of the program was terminated when the (unique) continuation could not be advanced anymore, that is, when it had only one value in it; the semantics of *result*

[13] If one does not want to add any other auxiliary global operations besides *eval* and *result*, then one can think of *newThread*(E, *Env*) as a "macro".

was defined to detect this terminating situation and then to return the output buffer. However, in the context of multiple threads, since a terminating thread needs to release its resources anyway, it is easier to simply discard the entire thread structure of a thread which terminates. This way, the *result* operation can be defined as follows:

$$\frac{\mathit{result}(\mathit{busy}(\_)\ \mathit{in}(\_)\ \mathit{out}(\mathit{Il})\ \mathit{store}(\_)\ \mathit{nextLoc}(\_))}{\mathit{Il}}$$

All it needs to do is to ensure, via matching, that there is no thread structure in the state, and then to return the output buffer.

In concurrent FUN, at any given moment during the execution of a program there can be several threads running concurrently. An immediate consequence of this non-trivial language design extension is that an underlying execution engine may *non-deterministically* pick some rule instance and thus disable the other rule instances that happen to overlap it, thus leading to different behaviors of the multi-threaded program when different rule instances are chosen. That means that several of our previously defined contextual rules need to become *non-deterministic*, as explained in Section 4.6. Among the rules that have been already defined for sequential FUN, the obvious ones that can change the deterministic behavior of a program, thus having to be now declared non-deterministic, are those defining accesses (reads or writes) to variables. The rules for I/O accesses need to also be declared non-deterministic, because the input and the output buffers are also shared by all the threads. Consequently, the following four rules, previously declared as just computational, need to become non-deterministic when we add threads to FUN, so we box them:

$$\boxed{k(\underset{V}{\underline{X}}\rangle\ \mathit{env}\langle(X,L)\rangle\ \mathit{store}\langle(L,V)\rangle}$$

$$\boxed{k((\underset{\cdot}{\underline{V}}\rangle \curvearrowright \mathit{write}(\underset{\cdot}{\underline{X}}\rangle\rangle\ \mathit{env}\langle(X,L)\rangle\ \mathit{store}(\underset{\mathit{Store}[L \leftarrow V]}{\underline{\quad\mathit{Store}\quad}})}$$

$$\boxed{k(\underset{\mathit{int}(I)}{\underline{\mathsf{read}()}}\rangle\ \mathit{in}(\underset{\cdot}{\underline{I}}\rangle}$$

$$\boxed{k(\underset{\mathit{unit}}{\underline{\mathit{int}(I) \curvearrowright \mathit{print}}}\rangle\ \mathit{out}\langle\underset{I}{\underline{\cdot}})}$$

Notice that all the changes above, that incurred because of the addition of threads to the language, are at the *global* level. The computational, or operational, or executional semantics of no individual language feature that was previously defined needs to change. The fact that we had to annotate some of the existing rules as non-deterministic and hence box them is *not* a break in modularity in what regards the computational semantics of the corresponding language features. The role of these annotations is to give additional, sometimes very useful, information about how much one can abstract computation away as atomic; recall from Section 4.6 that, for program analysis purposes, transitive applications of deterministic rules are considered atomic, thus significantly reducing the state-space that needs to be analyzed.

We believe that, regardless of the definitional framework used, some global changes need to be made when one extends a language non-trivially, as we did. That is because, on the one hand, the structure of the state, or configuration, or whatever data-structure a framework uses to store "execution" information about a program, may need to change in order to properly store the additional information needed by the new features that one adds to one's language. On the other hand, some language features may interact with each other in very subtle, indirect ways. For example, reads and writes of variables, whose computational semantics remain the same when one extends a language with concurrency, may lead to non-deterministic behaviors in the extended language; this can be entirely ignored if one is only interested in a dynamic semantics of the language, but it may be regarded as very useful and otherwise hard to infer information if one is interested in other aspects of a language definition, such as formal analysis. The challenge is to devise frameworks in which the definitions of language features do *not* strictly depend on the structure of the state or configuration. Unlike in classic SOS where a structural change of the configuration requires to change *all* the SOS rules, in K we need to change *no* contextual rule, providing that one does not change the names of state attributes when restructuring the state.

We next define thread creation and termination, as well as thread synchronization based on a simple locking mechanism. Creating/spawning a thread results in simply adding a new thread attribute to the state "soup"; the newly created thread is passed an expression for evaluation and it inherits the parent's environment (this way the two threads share data):

$$\mathit{thread}\langle k(\underset{\mathit{unit}}{\underline{\mathsf{spawn}(E)}}\rangle\ \mathit{env}(\mathit{Env})\rangle\ \underset{\mathit{newThread}(E,\mathit{Env})}{\underline{\qquad\qquad\cdot\qquad\qquad}}$$

When a thread (including the original one) finishes evaluating its expression, it is eliminated from the state releasing all its resources (only locks in our case):

$$\underset{\cdot}{\underline{\mathit{thread}\langle k(\_ : \mathit{Val})\ \mathit{holds}(\mathit{LCs})\rangle}}\ \mathit{busy}(\underset{\mathit{Is} - \mathit{LCs}}{\underline{\qquad \mathit{Is} \qquad}})$$

Above we assumed an easy to define operation "$\_ - \_$" that takes a set of integers *Is* (all the busy locks) and a set *LCs* of pairs (lock,counter) and returns the set of locks that are in *Is* but not in any pair in *LCs*.

We adopt a simple locking mechanism for thread synchronization, where locks are plain integers. Acquiring a lock requires two cases to analyze, depending upon whether the thread already has the lock or not:

- If the thread already has the lock then "it acquires it once more", i.e., it increments the counter associated to that lock:

$$k(\underset{\mathit{unit}}{\underline{\mathit{int}(I) \curvearrowright \mathit{acquire}}}\rangle\ \mathit{holds}\langle(I, \underset{s(N)}{\underline{\quad N \quad}})\rangle$$

- If the thread does not have the lock and if the lock is available (note that the rule below is conditional) then the thread grabs it and initiates its counter to 0:

$$\boxed{k(\underset{\mathit{unit}}{\underline{\mathit{int}(I) \curvearrowright \mathit{acquire}}}\rangle\ \mathit{holds}\langle\underset{(I,0)}{\underline{\quad\cdot\quad}}\rangle\ \mathit{busy}(\underset{\mathit{Is},I}{\underline{\ \mathit{Is}\ }}) \Leftarrow \mathit{not}(I\ \mathit{in}\ \mathit{Is})}$$

The condition of the rule above uses an easy to define set membership operation. Also, note that the conditional (computational) rule above is marked as "non-deterministic". Indeed, the acquisition of a lock by a thread disables the acquisition of that same lock by other threads "blocking" them, so this action may lead to non-deterministic execution behaviors.

The semantics of release is dual to that of acquire:

$$k(\underset{\textit{unit}}{\underline{\textit{int}(I)) \curvearrowright \textit{release}}}\rangle \;\; \textit{holds}\langle(I, \underset{N}{\underline{s(N)}})\rangle$$

$$k(\underset{\textit{unit}}{\underline{\textit{int}(I) \curvearrowright \textit{release}}}\rangle \;\; \textit{holds}\langle\underset{\cdot}{\underline{(I,0)}}\rangle \;\; \textit{busy}\langle\underset{\cdot}{\underline{I}}\rangle$$

It is interesting to note that none of the release rules above is non-deterministic. Indeed, the release of a lock cannot lead to non-deterministic behaviors of multi-threaded programs; in other words, if at any moment there are two rules that can be applied, one of them being a release rule, the same behaviors are manifested regardless of which one is chosen.

6.3 Translating MSOS to K

NEXT-VERSION:

7 On Language Semantics

NEXT-VERSION:

7.1 Initial Algebra Semantics Revisited

NEXT-VERSION:

7.2 Initial Model Semantics in Rewriting Logic

NEXT-VERSION:

8 On Formal Analysis

NEXT-VERSION:

8.1 Type Checking and Type Inference

NEXT-VERSION:

8.2 Type Preservation and Progress

NEXT-VERSION:

8.3 Concurrency Analysis

NEXT-VERSION:

8.4 Model Checking

NEXT-VERSION:

9 On Implementation

NEXT-VERSION:

10 Conclusion

An algebraic framework to define programming languages was presented, based on a first-order representation of continuations and on ACI-matching. The technique was introduced by defining the language FUN. The language definitions following the discussed methodology are executable by term rewriting, so one can get interpreters for free for the defined languages. Moreover, these definitions can be used to generate, also for free, certain formal analysis tools, such as model checkers, for the defined programming languages. As future work, we would like to: (1) implement the domain-specific language used in this paper; and (2) compile these definitions into efficient code running on parallel architectures.

References

[1] G. Agha. *Actors.* MIT Press, 1986.

[2] J. Bergstra and J. V. Tucker. Equational specs complete term rewriting systems, and computable and semicomputable algebras. *J. of ACM*, 42(6):1194–1230, 1995.

[3] P. Borovanský, H. Cîrstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. ELAN. User manual – `http://www.loria.fr`.

[4] C. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Mapping Modular SOS to Rewriting Logic. In M. Leuschel, editor, *12th International Workshop, LOPSTR 2002, Madrid, Spain*, volume 2664 of *Lecture Notes in Computer Science*, pages 262–277, 2002.

[5] M. Clavel, F. J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic, Mar. 1999. Maude System documentation at `http://maude.csl.sri.com/papers`.

[6] R. Diaconescu and K. Futatsugi. *CafeOBJ Report.* World Scientific, 1998. AMAST Series in Computing, volume 6.

[7] A. Farzan, F. Cheng, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. in Proc. CAV'04, Springer LNCS, 2004.

[8] M. Felleisen and R. Hieb. A Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 103(2):235–271, 1992.

[9] J. Goguen and G. Malcolm. *Alg. Semantics of Imperative Programs.* MIT, 1996.

[10] J. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *SIGPLAN Notices*, 16(7):24–37, July 1981.

[11] J. Goguen and J. Meseguer. Order-sorted algebra I: Eq. ded. for multiple inheritance, overloading, exceptions and partial ops. *J. of TCS*, 105(2):217–273, 1992.

[12] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Initial algebra semantics and continuous algebras. *J. of ACM*, 24(1):68–95, January 1977.

[13] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Soft. Eng. with OBJ: alg. spec. in action.* Kluwer, 2000.

[14] D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In *CADE'86*, pages 489–495, 1986.

[15] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL'95*, pages 333–343. ACM Press, 1995.

[16] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

[17] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *J. of TCS*, 96(1):73–155, 1992.

[18] J. Meseguer and G. Roşu. Rewriting logic semantics: From language specifications to formal analysis tools. In *IJCAR'04*, pages 1–44. Springer LNAI 3097, 2004.

[19] J. Meseguer and G. Roşu. The rewriting logic semantics project. Technical Report UIUCDCS-R-2005-2639, Deptartment of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[20] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989.

[21] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60–61:195–228, 2004.

[22] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, July-December 2004 2004.

[23] J. C. Reynolds. The Discoveries of Continuations. *LISP and Symbolic Computation*, 6(3–4):233–247, 1993.

[24] G. Roşu. Programming language classes. Department of Computer Science, University of Illinois at Urbana-Champaign, `http://fsl.cs.uiuc.edu/~grosu/classes/`.

[25] T. F. Şerbănuţă and G. Roşu. 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.

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

[27] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling lang. defs.: ASF+SDF compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[28] E. Visser. Program transf. with Stratego/XT: Rules, strategies, tools, and systems. In *Domain-Specific Program Generation*, pages 216–238, 2003.

[29] M. Walicki and S. Meldal. Algebraic approaches to nondeterminism: An overview. *ACM Computing Surveys*, 29:30–81, 1997.

A A Simple Rewrite Logic Theory in Maude

We here show how the simple rewrite logic theory discussed in Section 2.3 can be formalized in the Maude system. As one can see, there is an almost zero representational distance in how rewrite theories are specified. Note the use of the *operation attributes* `assoc` and `comm` instead of the equations of associativity and commutativity. These attributes are used both by the Maude's parser (to release the user from providing parentheses for associative operators) and by its rewriting engine: rewrites will be performed *modulo* operation attributes, enabling specialized algorithms:

```
mod 01 is
  sort S .
  op empty : -> S .
  op _ _ : S S -> S [assoc comm id: empty] .
  op 0 : -> S .
  op 1 : -> S .
  rl [r1] : 0 => 1 .
  rl [r2] : 1 => 0 .
endm
```

Once a rewrite theory is defined, one can perform different types of analyses. The most obvious one would be to execute it on different terms using the command `rewrite`, for example:

```
rewrite 0 1 0 1 0 =>* 1 1 1 1 1 .
```

The theory above, however, specifies a non-terminating concurrent system. Indeed, the command above will never terminate. Therefore, executing the specification above is not interesting. Maude also provides a `search` command, which enables a breadth-first search (BFS) state exploration algorithm on the transition system associated to the term to search from:

```
search 0 1 0 1 0 =>* 1 1 1 1 1 .
```

This `search` command enables the BFS engine to search a path of length zero or more (the "`=>*`") from the term `0 1 0 1 0` to the term `1 1 1 1 1`. In general, the target term can contain variables and the `search` command can take an argument saying how many solutions to search for. In our case there is only one solution, found after exploring five additional states. This is Maude's output:

```
Solution 1 (state 5)
states: 6  rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
empty substitution
```

If one wants to see the sequence of states from `0 1 0 1 0` to `1 1 1 1 1`, that is, the path to state 5 as the above output says, then one can type the command:

```
show path 5 .
```

ad get the result:

```
states: 6  rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
state 0, S: 0 0 0 1 1
===[ rl 0 => 1 [label r1] . ]===>
state 1, S: 0 0 1 1 1
===[ rl 0 => 1 [label r1] . ]===>
state 3, S: 0 1 1 1 1
===[ rl 0 => 1 [label r1] . ]===>
state 5, S: 1 1 1 1 1
```

Finally, if one wants to see all the 6 states explored together with all the transitions in-between, then one can type the command:

```
show search graph .
```

and get the result:

```
state 0, S: 0 0 0 1 1
arc 0 ===> state 1 (rl 0 => 1 [label r1] .)
arc 1 ===> state 2 (rl 1 => 0 [label r2] .)

state 1, S: 0 0 1 1 1
arc 0 ===> state 0 (rl 1 => 0 [label r2] .)
arc 1 ===> state 3 (rl 0 => 1 [label r1] .)

state 2, S: 0 0 0 0 1
arc 0 ===> state 0 (rl 0 => 1 [label r1] .)
arc 1 ===> state 4 (rl 1 => 0 [label r2] .)

state 3, S: 0 1 1 1 1
arc 0 ===> state 1 (rl 1 => 0 [label r2] .)
arc 1 ===> state 5 (rl 0 => 1 [label r1] .)

state 4, S: 0 0 0 0 0
arc 0 ===> state 2 (rl 0 => 1 [label r1] .)

state 5, S: 1 1 1 1 1
arc 0 ===> state 3 (rl 1 => 0 [label r2] .)
```

As mentioned, the `search` command allows target terms that may contain variables; this way, the target terms are regarded as *patterns*. The BFS algorithm will explore the state space of the corresponding transition system and match each state term against the pattern, reporting all the successful matches by returning the corresponding substitution. For example, one can type the command:

```
search 01 : 0 1 0 0 1 =>* 1 1 1 X:S .
```

where `X:S` declares a variable `X` of sort `S` on-the-fly (it could have also been declared in the rewrite theory using the syntax “`var X : S`”). All three solutions will be reported by Maude:

```
Solution 1 (state 1)
states: 2  rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
X:S --> 0 0

Solution 2 (state 3)
states: 4  rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
X:S --> 0 1

Solution 3 (state 5)
states: 6  rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
X:S --> 1 1

No more solutions.
states: 6  rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
```

The above command investigates the entire state space. Sometimes the state space can be infinite or very large, and one is not interested in all the solutions of the search anyway. If that is the case, then one can specify a fixed number of solutions one wants the search command to find; once it succeeds in finding those, the state exploration is terminated. For example, the following `search` command explores the state space only until it finds two solutions:

```
search[2] 01 : 0 1 0 0 1 =>* 1 1 1 X:S .
```

The result shows that only 4 of the total 6 states have been discovered:

```
Solution 1 (state 1)
states: 2  rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
X:S --> 0 0

Solution 2 (state 3)
states: 4  rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
X:S --> 0 1
```

Note that, even though the rewrite theory above admits models in which the two rewrite rules can apply concurrently, for efficiency reasons the search algorithm (as well as the model-checking one below) explores the state space following an interleaving semantics, i.e., it tries the rules one at a time. This does not imply any unsoundness in reachability (`search`) analysis because the same state space is explored anyway.

Let us next formalize in Maude the model-checking example on multi-sets of 0 and 1 discussed in Section 2.3, namely to show that the rewrite theory above does *not* satisfy the property "it is always the case that from a state of zeros one can eventually reach a state of ones". This property can be expressed as a linear temporal logic (LTL) formula, namely `[](zeros -> <> ones)`. To use the model checking capability of Maude, one needs to first import the model checking package (which is only partly builtin; it allows users to add/remove LTL simplification rules) and to define the atomic predicates. The model checking module comes with a sort `State` for the states of the transition system to analyze, with a sort `Prop` for the atomic propositions on those states, and with a satisfaction operation `_|=_ : State Prop -> Bool` (defined in the module `SATISFACTION`). One is free to define as many atomic propositions as one wants, but one should make sure that one also states when these are true in a given state. For our example, one can do all these as follows:

```
in MODEL_CHECKER

mod 01-PREDS is
  protecting 01 .
  including MODEL-CHECKER .
  subsort S < State .
  var S : S .
  op zeros : -> Prop .
  eq (1 S) |= zeros  = false .
  eq (0 S) |= zeros  = S |= zeros .
  eq empty |= zeros = true .
  op ones : -> Prop .
  eq (0 S) |= ones  = false .
  eq (1 S) |= ones  = S |= ones .
  eq empty |= ones = true .
endm
```

One can now launch the model checking by invoking the operation `modelCheck` (defined in the `MODEL_CHECKER` package) on an initial term (whose state space is requested to be model checked) and an LTL formula to be checked against. Below are three invocations of the model checker:

```
reduce modelCheck(0,[](zeros -> <> ones)) .
reduce modelCheck(0 1,[](zeros -> <> ones)) .
reduce modelCheck(0 1 0 1 0,[](zeros -> <> ones)) .
```

Interestingly, the first does not violate the property: from the state containing just one 0 there is no way to avoid the state containing just one 1. However, the property is violated, and one possible counterexample (not necessarily the smallest one) given, when checked against any multi-set of size larger than:

```
reduce in 01-PREDS : modelCheck(0, [](zeros -> <> ones)) .
rewrites: 18 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in 01-PREDS : modelCheck(0 1, [](zeros -> <> ones)) .
rewrites: 24 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult: counterexample({0 1,'r2}, {0 0,'r1} {0 1,'r2})
==========================================
reduce in 01-PREDS : modelCheck(0 1 0 0 1, [](zeros -> <> ones)) .
rewrites: 42 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult: counterexample({0 0 0 1 1,'r2} {0 0 0 0 1,'r2} {0 0 0
    0 0,'r1} {0 0 0 0 1,'r1} {0 0 0 1 1,'r1}, {0 0 1 1 1,'r1} {0 1 1 1 1,'r2})
```

This reminds us that, in general, model-checking can find errors in non-deterministic systems but it does *not* prove the correctness of those systems: the fact that a system is model-checked and shown correct on some inputs does not necessarily imply that the system is correct on all its inputs.

B Dining Philosophers in Maude

In this appendix we show how the dining philosophers problem discussed in Section 2.3 can be specified and analyzed in Maude. The following module does nothing but formalizes in Maude the rewrite logic theory in Section 2.3, replacing the equations of associativity, commutativity and identity with the corresponding attributes `assoc`, `comm`, and `id`:

```
mod DINING-PHILOSOPHERS is protecting INT .
--- state infrastructure
  sort State .
  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .

  op ph : Nat State -> State .
  op $ : Nat -> State .

--- number of philosophers + 1
  op n : -> Nat .
  eq n = 9 .

--- creating the initial state
```

```
    op init : -> State .
    eq init = init(n) .
    op init : Int -> State .
    eq init(-1) = empty .
    eq init(N) = ph(N, empty) $(N) init(N - 1) .

    vars N X Y : Nat .  var Fs : State .

  --- acquiring locks: philosophers need to compete on forks
    rl ph(N, Fs) $(N) => ph(N, Fs $(N)) .
    rl ph(s(N), Fs) $(N) => ph(s(N), Fs $(N)) .
   crl ph(0, Fs) $(N) => ph(0, Fs $(n)) if N = n .

  --- no competition involved in releasing locks
    eq ph(N, $(X) $(Y)) = ph(N, empty) $(X) $(Y) .
  endm
```

To find all the states in which no rules can be applied anymore, or in other words to find all the states which are *normal forms*, one needs to use the special symbol "`=>!`" in the `search` command:

```
  search init =>! S:State .
```

The above will return the two deadlock solutions for 10 philosophers in about 2.7 seconds on a 2.5GHz/3.5GB Linux machine, while for 14 philosophers (change "`n = 9`" to "`n = 13`" in the module above) in about 400 seconds. In our experiments, Maude crashed when tried on 15 philosophers on the machine above.

C Defining Lists and Sets in Maude

```
fmod SORT is
  sort Sort .
  ops a b c : -> Sort .
endfm

fmod SORT-LIST is protecting SORT .
  sorts SortList SortNeList .
  subsorts Sort < SortNeList < SortList .
  op nil : -> SortList .
  op _,_ : SortList SortList -> SortList [assoc id: nil] .
  op _,_ : SortNeList SortNeList -> SortNeList [ditto] .
endfm

parse  a,b,b,c,a,b,c .
parse  a .
parse  nil .
parse  nil,a,nil .
reduce nil,a,nil .


fmod SORT-SET is protecting SORT .
  sorts SortSet SortNeSet .
  subsorts Sort < SortNeSet < SortSet .
  op empty : -> SortSet .
  op _ _ : SortSet SortSet -> SortSet [assoc id: empty] .
  op _ _ : SortNeSet SortNeSet -> SortNeSet [ditto] .
endfm

parse  a b b c a b c .
parse  a .
parse  empty .
parse  empty a empty .
reduce empty a empty .
```

D Definition of Sequential λ_{K} in Maude

In this appendix we show an instance of the K-technique presented in the paper, in the context of the Maude language. The Maude code below is a translation of the K-definition of sequential λ_{K} in Figure 1. Appendix E shows a similar instance, but for the concurrent extension of λ_{K} in Figure 3. Appendixes F and G show the more complex Maude instances of the FUN language. All the Maude language definitions in Appendixes D, E, F and G, have been hand-crafted by Traian Florin Şerbănuţă (to whom I warmly thank for his continuous help and interest in this project) in a few hours, by following a mechanical K-to-Maude translation procedure that he is also currently implementing.

At this moment, in our Maude-ified K language definitions we collapse the sorts *Exp* and *Continuation*; we assume that programs are syntactically well-formed, in particular that one does not explicitly use K continuations in programs (some external parser or static checker can be used for this purpose as well). One should not find it surprising that expressions and continuations are collapsed in our semantics; in our framework, continuations are nothing but computationally equivalent variants of expressions that are structurally more suitable for defining the semantics of other operators.

```
fmod SYNTAX is
  including INT .
  including QID .

--- Var
  sort Var .
  subsort Qid < Var .
  ops a b c d e f g h i j k l m n o p q r t u v x y z : -> Var [format (g o)] .

  sort Exp .
  subsort Var < Exp .
  op #_ : Int -> Exp [prec 30 format (b r o)] .
  ops _+_ _-_ : Exp Exp -> Exp [prec 33 gather (E e) format (d b o d)] .
  ops _*_ _/_ : Exp Exp -> Exp [prec 32 gather (E e) format (d b o d)] .

  ops true false : -> Exp [format (r o)] .
  ops _<=_ _>=_ _==_ : Exp Exp -> Exp [prec 37 format (d b o d)] .
  op _and_ : Exp Exp -> Exp [prec 55 format (d b o d)] .
  op _or_ : Exp Exp -> Exp [prec 57 format (d b o d)] .
  op not_ : Exp -> Exp [prec 53 format (b o d)] .

  sort VarList ExpList .
  op _,_ : VarList VarList -> VarList [assoc id: .] .
  op . : -> VarList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
  subsort Var < VarList Exp < ExpList .

  op \_._ : VarList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 60] .

  op if_then_else_ : Exp Exp Exp -> Exp
                     [prec 70 format (b o bn+i o+ bn-i o+ --)] .
  op ref_ : Exp -> Exp [prec 31 format (b o d)].
  op ^_ : Exp -> Exp [format (b o d)] .
  op _:=_ : Exp Exp -> Exp [format (d b o d)] .
  op halt_ : Exp -> Exp [format (b o d)] .

  op let_=_in_ : Exp Exp Exp -> Exp
     [format (b o b o bni++ o --) prec 70] .

  op _;_ : Exp Exp -> Exp [gather (e E) prec 80 format (d b noi d)] .
endfm

fmod LOC is
  including NAT .
  sort Loc .
  op loc : Nat -> Loc .
  op next : Loc -> Loc .
  var N : Nat .
  eq next(loc(N)) = loc(s(N)) .
endfm

fmod ENV is
  including SYNTAX .
  including LOC .
  sort Env .
  op ‘(_‘,_‘) : Var Loc -> Env .
  op __ : Env Env -> Env [assoc comm id: empty ] .
  op empty : -> Env .


  var Env : Env . var X : Var . var L L’ : Loc .
  --- retrieves a location from the environment
  op _[_] : Env Var -> Loc .
  eq (Env (X,L))[X] = L .

  --- updates a variable’s location in the environment
  op _[_<-_] : Env Var Loc -> Env .
  eq (Env (X,L))[X <- L’] = Env (X,L’) .
  eq Env[X <- L] = Env (X,L) [owise] .
endfm

fmod VAL is
  including INT .
  sort Val .
  op int : Int -> Val .
  op bool : Bool -> Val .
endfm

fmod STORE is
```

```
  including VAL .
  including LOC .
  sort Store .
  op ‘(_‘,_‘) : Loc Val -> Store .
  op __ : Store Store -> Store [assoc comm id: empty ] .
  op empty : -> Store .

  var S : Store . var L : Loc . var V V’ : Val .

  --- retrieves a value from the state
  op _[_] : Store Loc -> Val .
  eq (S (L,V))[L] = V .

  --- updates a variable in the state
  op _[_<-_] : Store Loc Val -> Store .
  eq (S (L,V))[L <- V’] = S (L,V’) .
  eq S[L <- V] = S (L,V) [owise] .
endfm


mod VAL-LIST is
  including VAL .
  sort ValList .
  subsort Val < ValList .
  op _,_ : ValList ValList -> ValList [assoc id: .] .
  op . : -> ValList .
endm

mod K is
  including SYNTAX .
  including VAL-LIST .
  sort Kitem K .
  subsort Kitem < K .
  op exp : Exp -> K .

  sort KList .
  subsort K < KList .
  op _,_ : KList KList -> KList [assoc id: .] .
  op . : -> KList .

  op kList : KList -> Kitem .
  op valList : ValList -> Kitem .
  op _->_ : K K -> K [assoc id: nothing] .
  op nothing : -> K .
endm

mod K-STATE is
  including K .
  including STORE .
  including ENV .
  sort State .
  op __ : State State -> State [assoc comm id: .] .
  op . : -> State .

  op k : K -> State .
  op store : Store -> State .
  op env : Env -> State .
  op nextLoc : Loc -> State .
endm

mod K-BASIC is
  including K-STATE .
  op kv : KList ValList -> Kitem .
  var Ke : K . var Kel : KList . var K : K .
  var V : Val . var Vl : ValList .

  eq k(kList(Ke,Kel) -> K) = k(Ke -> kv(Kel,.) -> K) .
  eq valList(V) -> kv(Kel,Vl) = kv(Kel,Vl,V) .
  eq k(kv(Ke,Kel,Vl) -> K) = k(Ke -> kv(Kel,Vl) -> K) .
  eq k(kv(.,Vl) -> K) = k(valList(Vl) -> K) .
endm

mod SYNTAX-SUGAR-RULES is
  including K .
  var X : Var . var E E’ : Exp . var Xl : VarList .

  op dummy : -> Var .
  eq exp(let X = E in E’) = exp((\ X . E’) E) .
  eq exp(let X(Xl)= E in E’) = exp((\ X . E’)(\ Xl . E)) .
  eq exp(E ; E’) = exp((\ dummy . E’) E) .
endm

mod AEXP-RULES is
  including K-STATE .
  var I I1 I2 : Int . var X : Var . var K : K . var Store : Store .
  var A1 A2 : Exp . var Env : Env .
  eq exp(# I)  = valList(int(I)) .
  rl k(exp(X) -> K) env(Env) store(Store)
   => k(valList(Store[Env[X]]) -> K) env(Env) store(Store) .
  ops + - * / : -> Kitem .
  eq exp(A1 + A2) = kList(exp(A1),exp(A2)) -> + .
  rl valList(int(I1),int(I2)) -> + => valList(int(I1 + I2)) .

  eq exp(A1 - A2) = kList(exp(A1),exp(A2)) -> - .
  rl valList(int(I1),int(I2)) -> - => valList(int(I1 - I2)) .

  eq exp(A1 * A2) = kList(exp(A1),exp(A2)) -> * .
  rl valList(int(I1),int(I2)) -> * => valList(int(I1 * I2)) .

  eq exp(A1 / A2) = kList(exp(A1),exp(A2)) -> / .
  crl valList(int(I1),int(I2)) -> / => valList(int(I1 quo I2))
   if I2 =/= 0 .
endm

mod BEXP-RULES is
  including K-STATE .
  var X : Var . var Store : Store . var B B1 B2 : Exp .
  var A1 A2 : Exp . var I1 I2 : Int . var T T1 T2 : Bool .
  eq exp(true)  = valList(bool(true)) .
  eq exp(false)  = valList(bool(false)) .
  ops <= >= == and or not : -> Kitem .
  eq exp(A1 <= A2) = kList(exp(A1),exp(A2)) -> <= .
  rl valList(int(I1),int(I2)) -> <= => valList(bool(I1 <= I2)) .

  eq exp(A1 >= A2) = kList(exp(A1),exp(A2)) -> >= .
  rl valList(int(I1),int(I2)) -> >= => valList(bool(I1 >= I2)) .

  eq exp(A1 == A2) = kList(exp(A1),exp(A2)) -> == .
  rl valList(int(I1),int(I2)) -> == => valList(bool(I1 == I2)) .

  eq exp(B1 and B2) = kList(exp(B1),exp(B2)) -> and .
  rl valList(bool(T1),bool(T2)) -> and => valList(bool(T1 and T2)) .

  eq exp(B1 or B2) = kList(exp(B1),exp(B2)) -> or .
  rl valList(bool(T1),bool(T2)) -> or => valList(bool(T1 or T2)) .

  eq exp(not B) = exp(B) -> not .
  rl valList(bool(T)) -> not => valList(bool(not T)) .
endm

mod FUNCTION-RULES is
  including K-STATE .
  op closure : VarList K Env -> Val .
  var Xl : VarList . var E : Exp . var El : ExpList . var X : Var .
  var Env Env’ : Env . var V : Val . var Vl : ValList . var K K’ : K .
  var Store : Store . var L : Loc .
  eq k(exp(\ Xl . E) -> K) env(Env)
   = k(valList(closure(Xl,exp(E),Env)) -> K) env(Env) .
  op bind : VarList -> Kitem .
  eq k(valList(Vl,V) -> bind(Xl,X) -> K)
     env(Env) store(Store) nextLoc(L)
  = k(valList(Vl) -> bind(Xl) -> K)
     env(Env[X <- L]) store(Store (L,V)) nextLoc(next(L)) .
  eq k(valList(.) -> bind(.) -> K) = k(K) .
  op app : -> Kitem .
  eq exp(E El) = kList(exp(E),expList(El)) -> app .
  op expList : ExpList -> KList .
  eq expList(E,El) = exp(E),expList(El) .
  eq expList(.) = . .
  rl k(valList(closure(Xl,K,Env),Vl) -> app -> K’) env(Env’)
  => k(valList(Vl) -> bind(Xl) -> K -> restore(Env’) -> K’) env(Env) .
  op restore : Env -> Kitem .
  eq k(restore(Env) -> K) env(Env’) = k(K) env(Env) .
endm

mod IMPERATIVE-RULES is
  including K-STATE .
  op loc : Loc -> Val .
  ops ref ^ := halt : -> Kitem .
  op if : K K -> Kitem .
  var E E1 E2 E’ : Exp . var K K’ K1  K2 : K . var L : Loc .
  var V V’ : Val . var Store : Store .
  eq exp(if E then E1 else E2) = exp(E) -> if(exp(E1),exp(E2)) .
  rl k(valList(bool(true)) -> if(K1,K2) -> K) => k(K1 -> K) .
  rl k(valList(bool(false)) -> if(K1,K2) -> K) => k(K2 -> K) .
  eq exp(ref E) = exp(E) -> ref .
  rl k(valList(V) -> ref -> K) nextLoc(L) store(Store)
  => k(valList(loc(L)) -> K) nextLoc(next(L)) store(Store (L,V)) .
  eq exp(^ E) = exp(E) -> ^ .
  rl k(valList(loc(L)) -> ^ -> K) store(Store (L,V))
  => k(valList(V) -> K) store(Store (L,V)) .

  eq exp(E := E') = kList(exp(E),exp(E')) -> := .
  rl k(valList(loc(L),V) -> := -> K) store(Store (L,V'))
  => k(valList(V) -> K) store(Store (L,V)) .

  eq exp(halt E) = exp(E) -> halt .
  rl k(valList(V) -> halt -> K) => k(valList(V)) .
endm

mod K-SEMANTICS is
  including K-BASIC .
  including SYNTAX-SUGAR-RULES .
  including AEXP-RULES .
  including BEXP-RULES .
  including FUNCTION-RULES .
  including IMPERATIVE-RULES .

  op <_> : Exp -> [Val] .
  op result : State -> [Val] .
  var P : Exp . var V : Val . var Store : Store . var Env : Env .
  var L : Loc .
  eq < P >
   = result(k(exp(P)) env(empty) store(empty) nextLoc(loc(0))) .
  eq result(k(valList(V)) env(Env) store(Store) nextLoc(L))
   = V .
endm

rew < let r = ref # 100 in
let g m,h = if m >= # 2
 then r := ^ r * m ;
  h m - # 1,h
 else halt ^ r
in g # 100 - # 1,g > .
***> should compute 100!
quit
```

E Definition of Concurrent λ_{K} in Maude

In this appendix we show the Maude-ified version of the concurrent λ_{K} language, whose K-definition is shown in Figure 3. Note that there is much repetition between the subsequent definition and the one of sequential λ_{K} in Appendix D, but yet, we cannot just import sequential λ_{K} and define the additional features on top of it, as we did in K. That is because rewriting logic semantics is not as modular as K is in what regards changes of the state structure. Once we have an implementation of K together with translations into Maude and/or other rewrite engines, definitions like those in Appendix D and in this appendix, as well as those in Appendixes F and G, can all be generated automatically from the corresponding modular K definitions.

```
fmod SYNTAX is
  including INT .
  including QID .

--- Var
  sort Var .
  subsort Qid < Var .
  ops a b c d e f g h i j k l m n o p q r t u v x y z : -> Var [format (g o)] .

  sort Exp .
  subsort Var < Exp .
  op #_ : Int -> Exp [prec 30 format (b r o)] .
  ops _+_ _-_ : Exp Exp -> Exp [prec 33 gather (E e) format (d b o d)] .
  ops _*_ _/_ : Exp Exp -> Exp [prec 32 gather (E e) format (d b o d)] .

  ops true false : -> Exp [format (r o)] .
  ops _<=_ _>=_ _==_ : Exp Exp -> Exp [prec 37 format (d b o d)] .
  op _and_ : Exp Exp -> Exp [prec 55 format (d b o d)] .
  op _or_ : Exp Exp -> Exp [prec 57 format (d b o d)] .
  op not_ : Exp -> Exp [prec 53 format (b o d)] .

  sort VarList ExpList .
  op _,_ : VarList VarList -> VarList [assoc id: .] .
  op . : -> VarList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
  subsort Var < VarList Exp < ExpList .

  op \_._ : VarList Exp -> Exp .
  op __ : Exp ExpList -> Exp [prec 60] .

  op if_then_else_ : Exp Exp Exp -> Exp
                     [prec 70 format (b o bn+i o+ bn-i o+ --)] .
  op ref_ : Exp -> Exp [prec 31 format (b o d)].
  op ^_ : Exp -> Exp [format (b o d)] .
  op _:=_ : Exp Exp -> Exp [format (d b o d)] .
  op halt_ : Exp -> Exp [format (b o d)] .

  op let_=_in_ : Exp Exp Exp -> Exp
     [format (b o b o bni++ o --) prec 70] .

  op _;_ : Exp Exp -> Exp [gather (e E) prec 80 format (d b noi d)] .

  op spawn_ : Exp -> Exp .
endfm

fmod LOC is
  including NAT .
  sort Loc .
  op loc : Nat -> Loc .
  op next : Loc -> Loc .
  var N : Nat .
  eq next(loc(N)) = loc(s(N)) .
endfm

fmod ENV is
  including SYNTAX .
  including LOC .
  sort Env .
  op ‘(_‘,_‘) : Var Loc -> Env .
  op __ : Env Env -> Env [assoc comm id: empty ] .
  op empty : -> Env .


  var Env : Env . var X : Var . var L L’ : Loc .
  --- retrieves a location from the environment
  op _[_] : Env Var -> Loc .
  eq (Env (X,L))[X] = L .

  --- updates a variable’s location in the environment
  op _[_<-_] : Env Var Loc -> Env .
  eq (Env (X,L))[X <- L’] = Env (X,L’) .
  eq Env[X <- L] = Env (X,L) [owise] .
endfm

fmod VAL is
  including INT .
  sort Val .
  op int : Int -> Val .
  op bool : Bool -> Val .
endfm

fmod STORE is
  including VAL .
  including LOC .
  sort Store .
  op ‘(_‘,_‘) : Loc Val -> Store .
  op __ : Store Store -> Store [assoc comm id: empty ] .
  op empty : -> Store .

  var S : Store . var L : Loc . var V V’ : Val .

  --- retrieves a value from the state
  op _[_] : Store Loc -> Val .
  eq (S (L,V))[L] = V .

  --- updates a variable in the state
  op _[_<-_] : Store Loc Val -> Store .
  eq (S (L,V))[L <- V’] = S (L,V’) .
  eq S[L <- V] = S (L,V) [owise] .
endfm


mod VAL-LIST is
  including VAL .
  sort ValList .
  subsort Val < ValList .
  op _,_ : ValList ValList -> ValList [assoc id: .] .
  op . : -> ValList .
endm

mod K is
  including SYNTAX .
  including VAL-LIST .
  sort Kitem K .
  subsort Kitem < K .
  op exp : Exp -> K .

  sort KList .
  subsort K < KList .
  op _,_ : KList KList -> KList [assoc id: .] .
  op . : -> KList .

  op kList : KList -> Kitem .
  op valList : ValList -> Kitem .
  op _->_ : K K -> K [assoc id: nothing] .
  op nothing : -> K .
endm

mod K-THREAD-STATE is
  including K .
  including ENV .
  sort ThreadState .
  op __ : ThreadState ThreadState -> ThreadState [assoc comm id: .] .
  op . : -> ThreadState .
  op k : K -> ThreadState .
  op env : Env -> ThreadState .
endm

mod K-STATE is
  including STORE .
  including K-THREAD-STATE .
  sort State .
  op __ : State State -> State [assoc comm id: .] .
  op . : -> State .

  op thread : ThreadState -> State .
  op store : Store -> State .
  op nextLoc : Loc -> State .
endm

mod K-BASIC is
  including K-STATE .
  op kv : KList ValList -> Kitem .
  var Ke : K . var Kel : KList . var K : K .
  var V : Val . var Vl : ValList .

  eq k(kList(Ke,Kel) -> K) = k(Ke -> kv(Kel,.) -> K) .
  eq valList(V) -> kv(Kel,Vl) = kv(Kel,Vl,V) .
  eq k(kv(Ke,Kel,Vl) -> K) = k(Ke -> kv(Kel,Vl) -> K) .
  eq k(kv(.,Vl) -> K) = k(valList(Vl) -> K) .
endm

mod SYNTAX-SUGAR-RULES is
  including K .
  var X : Var . var E E’ : Exp . var Xl : VarList .

  op dummy : -> Var .
  eq exp(let X = E in E’) = exp((\ X . E’) E) .
  eq exp(let X(Xl)= E in E’) = exp((\ X . E’)(\ Xl . E)) .
  eq exp(E ; E’) = exp((\ dummy . E’) E) .
```

```
endm

mod AEXP-RULES is
  including K-STATE .
  var I I1 I2 : Int . var X : Var . var K : K . var Store : Store .
  var A1 A2 : Exp . var Env : Env .
  eq exp(# I)  = valList(int(I)) .
  rl thread(k(exp(X) -> K) env(Env)) store(Store)
   => thread(k(valList(Store[Env[X]]) -> K) env(Env)) store(Store) .
  ops + - * / : -> Kitem .
  eq exp(A1 + A2) = kList(exp(A1),exp(A2)) -> + .
  rl valList(int(I1),int(I2)) -> + => valList(int(I1 + I2)) .

  eq exp(A1 - A2) = kList(exp(A1),exp(A2)) -> - .
  rl valList(int(I1),int(I2)) -> - => valList(int(I1 - I2)) .

  eq exp(A1 * A2) = kList(exp(A1),exp(A2)) -> * .
  rl valList(int(I1),int(I2)) -> * => valList(int(I1 * I2)) .

  eq exp(A1 / A2) = kList(exp(A1),exp(A2)) -> / .
  crl valList(int(I1),int(I2)) -> / => valList(int(I1 quo I2))
   if I2 =/= 0 .
endm

mod BEXP-RULES is
  including K-STATE .
  var B B1 B2 : Exp .
  var A1 A2 : Exp . var I1 I2 : Int . var T T1 T2 : Bool .
  eq exp(true)  = valList(bool(true)) .
  eq exp(false)  = valList(bool(false)) .
  ops <= >= == and or not : -> Kitem .
  eq exp(A1 <= A2) = kList(exp(A1),exp(A2)) -> <= .
  rl valList(int(I1),int(I2)) -> <= => valList(bool(I1 <= I2)) .

  eq exp(A1 >= A2) = kList(exp(A1),exp(A2)) -> >= .
  rl valList(int(I1),int(I2)) -> >= => valList(bool(I1 >= I2)) .

  eq exp(A1 == A2) = kList(exp(A1),exp(A2)) -> == .
  rl valList(int(I1),int(I2)) -> == => valList(bool(I1 == I2)) .

  eq exp(B1 and B2) = kList(exp(B1),exp(B2)) -> and .
  rl valList(bool(T1),bool(T2)) -> and => valList(bool(T1 and T2)) .

  eq exp(B1 or B2) = kList(exp(B1),exp(B2)) -> or .
  rl valList(bool(T1),bool(T2)) -> or => valList(bool(T1 or T2)) .

  eq exp(not B) = exp(B) -> not .
  rl valList(bool(T)) -> not => valList(bool(not T)) .
endm

mod FUNCTION-RULES is
  including K-STATE .
  op closure : VarList K Env -> Val .
  var Xl : VarList . var E : Exp . var El : ExpList . var X : Var .
  var Env Env' : Env . var V : Val . var Vl : ValList . var K K' : K .
  var Store : Store . var L : Loc .
  eq k(exp(\ Xl . E) -> K) env(Env)
   = k(valList(closure(Xl,exp(E),Env)) -> K) env(Env) .
  op bind : VarList -> Kitem .
  eq thread(k(valList(Vl,V) -> bind(Xl,X) -> K) env(Env))
     store(Store) nextLoc(L)
  = thread(k(valList(Vl) -> bind(Xl) -> K) env(Env[X <- L]))
    store(Store (L,V)) nextLoc(next(L)) .
  eq k(valList(.) -> bind(.) -> K) = k(K) .
  op app : -> Kitem .
  eq exp(E El) = kList(exp(E),expList(El)) -> app .
  op expList : ExpList -> KList .
  eq expList(E,El) = exp(E),expList(El) .
  eq expList(.) = . .
  rl k(valList(closure(Xl,K,Env),Vl) -> app -> K’) env(Env’)
  => k(valList(Vl) -> bind(Xl) -> K -> restore(Env’) -> K’) env(Env) .
  op restore : Env -> Kitem .
  eq k(restore(Env) -> K) env(Env’) = k(K) env(Env) .
endm

mod IMPERATIVE-RULES is
  including K-STATE .
  op loc : Loc -> Val .
  ops ref ^ := halt : -> Kitem .
  op if : K K -> Kitem .
  var E E1 E2 E’ : Exp . var K K’ K1  K2 : K . var L : Loc .
  var V V’ : Val . var Store : Store . var TS : ThreadState .
  eq exp(if E then E1 else E2) = exp(E) -> if(exp(E1),exp(E2)) .
  rl k(valList(bool(true)) -> if(K1,K2) -> K) => k(K1 -> K) .
  rl k(valList(bool(false)) -> if(K1,K2) -> K) => k(K2 -> K) .
  eq exp(ref E) = exp(E) -> ref .
  rl thread(k(valList(V) -> ref -> K) TS)
     nextLoc(L) store(Store)
  => thread(k(valList(loc(L)) -> K) TS)
     nextLoc(next(L)) store(Store (L,V)) .

  eq exp(^ E) = exp(E) -> ^ .
  rl thread(k(valList(loc(L)) -> ^ -> K) TS) store(Store (L,V))
  => thread(k(valList(V) -> K) TS) store(Store (L,V)) .

  eq exp(E := E’) = kList(exp(E),exp(E’)) -> := .
  rl thread(k(valList(loc(L),V) -> := -> K) TS) store(Store (L,V’))
  => thread(k(valList(V) -> K) TS) store(Store (L,V)) .

  eq exp(halt E) = exp(E) -> halt .
  rl k(valList(V) -> halt -> K) => k(valList(V)) .
endm

mod SPAWN-RULES is
  including K-STATE .
  op die : -> Kitem .
  var E : Exp . var K : K . var Env : Env . var V : Val .
  var TS : ThreadState .
  rl thread(k(exp(spawn E) -> K) env(Env))
  => thread(k(valList(int(0)) -> K) env(Env))
     thread(k(exp(E) -> die) env(Env)) .
  rl thread(k(valList(V) -> die) TS) => . .
```

```
endm

mod K-SEMANTICS is
  including K-BASIC .
  including SYNTAX-SUGAR-RULES .
  including AEXP-RULES .
  including BEXP-RULES .
  including FUNCTION-RULES .
  including IMPERATIVE-RULES .
  including SPAWN-RULES .

  op <_> : Exp -> [Val] .
  op result : State -> [Val] .
  var P : Exp . var V : Val . var Store : Store . var Env : Env .
  var L : Loc .
  eq < P >
   = result(thread(k(exp(P)) env(empty)) store(empty) nextLoc(loc(0))) .
  rl result(thread(k(valList(V)) env(Env)) store(Store) nextLoc(L))
  => V .
endm

search < let r = ref # 4 in
let g m,h = if m >= # 2
 then spawn(r := ^ r * m) ;
  h m - # 1,h
 else halt ^ r
in g # 4 - # 1,g > =>! V:Val .
***> erroneous program for computing 4!
quit
```

F Definition of Sequential FUN in Maude

The Maude code below is a mechanical, hand-crafted translation of the K-definition of sequential FUN in Figure 6. Appendix G shows a similar instance, but for the full FUN language in Figure 7.

```
fmod SYNTAX is
  including INT .
  including QID .

--- Var
  sort Var .
  subsort Qid < Var .
  ops a b c d e f g h i j k l m n o p q r t u v x y z : -> Var [format (g o)] .

  sort Exp .
  subsort Var < Exp .
  op #_ : Int -> Exp [prec 30 format (b r o)] .
  ops _+_ _-_ : Exp Exp -> Exp [prec 33 gather (E e) format (d b o d)] .
  ops _*_ _/_ _%_ : Exp Exp -> Exp
                 [prec 32 gather (E e) format (d b o d)] .

  ops true false : -> Exp [format (r o)] .
  ops _<=_ _>=_ _==_ : Exp Exp -> Exp [prec 37 format (d b o d)] .
  op _and_ : Exp Exp -> Exp [prec 55 format (d b o d)] .
  op _or_ : Exp Exp -> Exp [prec 57 format (d b o d)] .
  op not_ : Exp -> Exp [prec 53 format (b o d)] .

  op skip : -> Exp [format (b o)] .

  op if_then_ : Exp Exp -> Exp
             [prec 70 format (b o bn+i o+ --)] .
  op if_then_else_ : Exp Exp Exp -> Exp
                     [prec 70 format (b o bn+i o+ bn-i o+ --)] .

  sort VarList ExpList .
  op _,_ : VarList VarList -> VarList [assoc id: .] .
  op . : -> VarList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
  subsort Var < VarList Exp < ExpList .

  op fun_->_ : VarList Exp -> Exp [format (b o b no++i n--)] .
  op _`(_`) : Exp ExpList -> Exp [prec 60 format(d b o b o)] .
  op return : Exp -> Exp [format (b o)] .

  ops let letrec : VarList ExpList Exp -> Exp [format (b o)] .

  op _;_ : Exp Exp -> Exp [gather (e E) prec 80 format (d b noi d)] .
  op _:=_ : Var Exp -> Exp [format (d b o d)] .

  op `[_`] : ExpList -> Exp [format (b o b o)] .
  ops car cdr null? : Exp -> Exp [format (b o)] .
  op cons : Exp Exp -> Exp [format (b o)] .

  op read`(`) : -> Exp [format (b d d o)] .
  op print : Exp -> Exp [format (b o)] .

  op try_catch`(_`)_ : Exp Var Exp -> Exp [format (b o b d o b o d)] .
  op throw : Exp -> Exp [format (b o)] .

  op while`(_`)_ : Exp Exp -> Exp [format (b d o b no++i --)] .
  op for`(_;_;_`)_ : Exp Exp Exp Exp -> Exp
                   [format (b d o b o b o b no++i --)] .
  op break : -> Exp [format (b o)] .
  op continue : -> Exp [format (b o)] .
endfm

fmod LOC is
  including NAT .
  sort Loc .
  op loc : Nat -> Loc .
  op next : Loc -> Loc .
  var N : Nat .
  eq next(loc(N)) = loc(s(N)) .
endfm

fmod ENV is
  including SYNTAX .
  including LOC .
  sort Env .
  op `(_`,_`) : Var Loc -> Env .
  op __ : Env Env -> Env [assoc comm id: empty ] .
  op empty : -> Env .


  var Env : Env . var X : Var . var L L’ : Loc .
  --- retrieves a location from the environment
  op _[_] : Env Var -> Loc .
  eq (Env (X,L))[X] = L .

  --- updates a variable’s location in the environment
  op _[_<-_] : Env Var Loc -> Env .
  eq (Env (X,L))[X <- L’] = Env (X,L’) .
  eq Env[X <- L] = Env (X,L) [owise] .
endfm

fmod VAL is
  including INT .
  sort Val .
  op int : Int -> Val .
  op bool : Bool -> Val .
  op unit : -> Val .
endfm

fmod STORE is
  including VAL .
  including LOC .
  sort Store .
  op ‘(_‘,_‘) : Loc Val -> Store .
  op __ : Store Store -> Store [assoc comm id: empty ] .
  op empty : -> Store .

  var S : Store . var L : Loc . var V V’ : Val .

  --- retrieves a value from the state
  op _[_] : Store Loc -> Val .
  eq (S (L,V))[L] = V .

  --- updates a variable in the state
  op _[_<-_] : Store Loc Val -> Store .
  eq (S (L,V))[L <- V’] = S (L,V’) .
  eq S[L <- V] = S (L,V) [owise] .
endfm

mod VAL-LIST is
  including VAL .
  sort ValList .
  subsort Val < ValList .
  op _,_ : ValList ValList -> ValList [assoc id: .] .
  op . : -> ValList .
endm

mod INT-LIST is
  including INT .
  sort IntList .
  subsort Int < IntList .
  op _,_ : IntList IntList -> IntList [assoc id: .] .
  op . : -> IntList .
```

```
endm

mod K is
  including SYNTAX .
  including VAL-LIST .
  sort Kitem K .
  subsort Kitem < K .
  op exp : Exp -> K [memo] .

  sort KList .
  subsort K < KList .
  op _,_ : KList KList -> KList [assoc id: .] .
  op . : -> KList .

  op kList : KList -> Kitem .
  op valList : ValList -> Kitem .
  op _->_ : K K -> K [assoc id: nothing] .
  op nothing : -> K .
endm

mod STACKS is
  sorts FunctionStack ExceptionStack LoopStack .
  op _->_ : FunctionStack FunctionStack -> FunctionStack [assoc id: .] .
  op . : -> FunctionStack .
  op _->_ : ExceptionStack ExceptionStack -> ExceptionStack [assoc id:
.] .
  op . : -> ExceptionStack .
  op _->_ : LoopStack LoopStack -> LoopStack [assoc id: .] .
  op . : -> LoopStack .
endm

mod CONTROL is
  including STACKS .
  sort Ctrl .
  op __ : Ctrl Ctrl -> Ctrl [assoc comm id: .] .
  op . : -> Ctrl .
  op fstack : FunctionStack -> Ctrl .
  op xstack : ExceptionStack -> Ctrl .
  op lstack : LoopStack -> Ctrl .
endm

mod K-STATE is
  including CONTROL .
  including ENV .
  including INT-LIST .
  including K .
  including STORE .
  sort State .
  op __ : State State -> State [assoc comm id: .] .
  op . : -> State .

  op k : K -> State .
  op env : Env -> State .
  op control : Ctrl -> State .
  ops in out : IntList -> State .
  op store : Store -> State .
  op nextLoc : Loc -> State .
endm

mod K-BASIC is
  including K-STATE .
  op kv : KList ValList -> Kitem .
  var Ke : K . var Kel : KList . var K : K .
  var V V' : Val . var Vl : ValList .
  var X : Var . var Xl : VarList . var Env Env' : Env . var L : Loc .
  var E : Exp . var El : ExpList . var Store : Store .

  eq k(kList(Ke,Kel) -> K) = k(Ke -> kv(Kel,.) -> K) .
  eq valList(V) -> kv(Kel,Vl) = kv(Kel,Vl,V) .
  eq k(kv(Ke,Kel,Vl) -> K) = k(Ke -> kv(Kel,Vl) -> K) .
  eq k(kv(.,Vl) -> K) = k(valList(Vl) -> K) .

  op bind : VarList -> Kitem .
  eq k(valList(Vl,V) -> bind(Xl,X) -> K)
     env(Env) store(Store) nextLoc(L)
   = k(valList(Vl) -> bind(Xl) -> K)
     env(Env[X <- L]) store(Store (L,V)) nextLoc(next(L)) .
  eq k(valList(.) -> bind(.) -> K) = k(K) .
  eq k(bind(Xl,X) -> K)
     env(Env) nextLoc(L)
   = k(bind(Xl) -> K)
     env(Env[X <- L]) nextLoc(next(L)) .
  eq k(bind(.) -> K) = k(K) .

  op write : VarList -> Kitem .
  eq k(valList(Vl,V) -> write(Xl,X) -> K)
     env(Env (X,L)) store(Store)
   = k(valList(Vl) -> write(Xl) -> K)
     env(Env (X,L)) store(Store[L <- V]) .
  eq k(valList(.) -> write(.) -> K) = k(K) .

  op expList : ExpList -> KList .
  eq expList(E,El) = exp(E),expList(El) .
  eq expList(.) = . .

  op restore : Env -> Kitem .
  eq k(valList(V) -> restore(Env) -> K) env(Env')
   = k(valList(V) -> K) env(Env) .
endm

mod AEXP-RULES is
  including K-STATE .
  var I I1 I2 : Int . var X : Var . var K : K . var Store : Store .
  var A1 A2 : Exp . var Env : Env .
  eq exp(# I)  = valList(int(I)) .
  rl k(exp(X) -> K) env(Env) store(Store)
   => k(valList(Store[Env[X]]) -> K) env(Env) store(Store) .
  ops + - * / % : -> Kitem .
  eq exp(A1 + A2) = kList(exp(A1),exp(A2)) -> + .
  rl valList(int(I1),int(I2)) -> + => valList(int(I1 + I2)) .

  eq exp(A1 - A2) = kList(exp(A1),exp(A2)) -> - .
  rl valList(int(I1),int(I2)) -> - => valList(int(I1 - I2)) .

  eq exp(A1 * A2) = kList(exp(A1),exp(A2)) -> * .
  rl valList(int(I1),int(I2)) -> * => valList(int(I1 * I2)) .

  eq exp(A1 / A2) = kList(exp(A1),exp(A2)) -> / .
  crl valList(int(I1),int(I2)) -> / => valList(int(I1 quo I2))
   if I2 =/= 0 .

  eq exp(A1 % A2) = kList(exp(A1),exp(A2)) -> % .
  crl valList(int(I1),int(I2)) -> % => valList(int(I1 rem I2))
   if I2 =/= 0 .
endm

mod BEXP-RULES is
  including K-STATE .
  var X : Var . var Store : Store . var B B1 B2 : Exp .
  var A1 A2 : Exp . var I1 I2 : Int . var T T1 T2 : Bool .
  eq exp(true)  = valList(bool(true)) .
  eq exp(false)  = valList(bool(false)) .
  ops <= >= == and or not : -> Kitem .
  eq exp(A1 <= A2) = kList(exp(A1),exp(A2)) -> <= .
  rl valList(int(I1),int(I2)) -> <= => valList(bool(I1 <= I2)) .

  eq exp(A1 >= A2) = kList(exp(A1),exp(A2)) -> >= .
  rl valList(int(I1),int(I2)) -> >= => valList(bool(I1 >= I2)) .

  eq exp(A1 == A2) = kList(exp(A1),exp(A2)) -> == .
  rl valList(int(I1),int(I2)) -> == => valList(bool(I1 == I2)) .

  eq exp(B1 and B2) = kList(exp(B1),exp(B2)) -> and .
  rl valList(bool(T1),bool(T2)) -> and => valList(bool(T1 and T2)) .

  eq exp(B1 or B2) = kList(exp(B1),exp(B2)) -> or .
  rl valList(bool(T1),bool(T2)) -> or => valList(bool(T1 or T2)) .

  eq exp(not B) = exp(B) -> not .
  rl valList(bool(T)) -> not => valList(bool(not T)) .
endm

mod IF-RULES is
  including K-STATE .
  op if : K K -> Kitem .
  var E E1 E2 : Exp . var K K1 K2 : K .
  eq exp(if E then E1) = exp(if E then E1 else skip) .
  eq exp(if E then E1 else E2) = exp(E) -> if(exp(E1),exp(E2)) .
  rl k(valList(bool(true)) -> if(K1,K2) -> K) => k(K1 -> K) .
  rl k(valList(bool(false)) -> if(K1,K2) -> K) => k(K2 -> K) .
endm

mod FUNCTION-RULES is
  including K-BASIC .
  op closure : VarList K Env -> Val .
  var Xl : VarList . var E : Exp . var El : ExpList . var X : Var .
  var Env Env’ : Env . var V : Val . var Vl : ValList . var K K’ : K .
  var Store : Store . var L : Loc . var FS : FunctionStack .
  var C C’ : Ctrl .
  eq k(exp(fun Xl -> E) -> K) env(Env)
   = k(valList(closure(Xl,exp(E),Env)) -> K) env(Env) .
  op app : -> Kitem .
  eq exp(E(El)) = kList(exp(E),expList(El)) -> app .

  op ‘(_,_,_‘) : K Env Ctrl -> FunctionStack .
  ops popFstack return : -> Kitem .

  rl k(valList(closure(Xl,K,Env),Vl) -> app -> K’)
     control(fstack(FS) C) env(Env’)
  => k(valList(Vl) -> bind(Xl) -> K -> popFstack)
     control(fstack((K’,Env’,C) -> FS) C) env(Env) .
  eq k(valList(V) -> popFstack)
     control(fstack((K,Env,C’) -> FS) C) env(Env’)
   = k(valList(V) -> K) control(fstack(FS) C) env(Env) .

  eq exp(return(E)) = exp(E) -> return .
  eq k(valList(V) -> return -> K’)
     control(fstack((K,Env,C) -> FS) C’) env(Env’)
   = k(valList(V) -> K) control(fstack(FS) C) env(Env) .
endm

mod LET-RULES is
  including K-BASIC .
  var Env Env’ : Env . var K : K . var Xl : VarList . var El : ExpList .
  var E : Exp .
  rl k(exp(let(Xl,El,E)) -> K) env(Env)
  => k(kList(expList(El)) -> bind(Xl) -> exp(E) -> restore(Env) -> K)
     env(Env) .

  rl k(exp(letrec(Xl,El,E)) -> K) env(Env)
  => k(bind(Xl) -> kList(expList(El)) -> write(Xl) -> exp(E)
         -> restore(Env) -> K) env(Env) .
endm

mod BASIC-STMT-RULES is
  including K-BASIC .
  eq exp(skip) = valList(unit) .
  op ; : -> Kitem .
  var E E1 E2 : Exp . var V1 V2 : Val . var X : Var .
  eq exp(E1 ; E2) = kList(exp(E1),exp(E2)) -> ; .
  rl valList(V1,V2) -> ; => valList(V2) .
  eq exp(X := E) = exp(E) -> write(X) -> valList(unit) .
endm

mod LIST-RULES is
  including K-BASIC .
  ops ‘[‘] car cdr null? cons : -> Kitem .
  var El : ExpList . var Vl : ValList . var E E1 E2 : Exp .
  var V : Val .
  eq exp([El]) = kList(expList(El)) -> [] .
  op ‘[_‘] : ValList -> Val .
  eq valList(Vl) -> [] = valList([Vl]) .
  eq exp(car(E)) = exp(E) -> car .
  rl valList([V,Vl]) -> car => valList(V) .
  eq exp(cdr(E)) = exp(E) -> cdr .
  rl valList([V,Vl]) -> cdr => valList([Vl]) .
  eq exp(null?(E)) = exp(E) -> null? .
  rl valList([V,Vl]) -> null? => valList(bool(false)) .
  rl valList([.]) -> null? => valList(bool(true)) .
  eq exp(cons(E1,E2)) = kList(exp(E1),exp(E2)) -> cons .
  rl valList(V,[Vl]) -> cons => valList([V,Vl]) .
endm

mod IO-RULES is
  including K-STATE .
  var E : Exp . var I : Int . var Il : IntList . var K : K .
  rl k(exp(read()) -> K) in(I,Il) => k(valList(int(I)) -> K) in(Il) .
  op print : -> Kitem .
  eq exp(print(E)) = exp(E) -> print .
  rl k(valList(int(I)) -> print -> K) out(Il)
  => k(valList(unit) -> K) out(Il,I) .
endm

mod EXCEPTION-RULES is
  including K-BASIC .
  op ‘(_,_,_,_,_‘) : Var K Env K Ctrl -> ExceptionStack .
  ops popXstack throw : -> Kitem .

  var E E’ : Exp . var K K’ Ke : K . var X : Var . var C C’ : Ctrl .
  var Env Env’ : Env . var XS : ExceptionStack . var V : Val .
  eq k(exp(try E’ catch(X) E) -> K) control(xstack(XS) C) env(Env)
   = k(exp(E’) -> popXstack)
    control(xstack((X,exp(E),Env,K,C) -> XS) C) env(Env) .
  rl k(valList(V) -> popXstack)
    control(xstack((X,Ke,Env,K,C) -> XS) C’) env(Env’)
  => k(valList(V) -> K) control(xstack(XS) C’) env(Env) .

  eq exp(throw(E)) = exp(E) -> throw .
  rl k(valList(V) -> throw -> K’)
    control(xstack((X,Ke,Env,K,C) -> XS) C’) env(Env’)
  => k(valList(V) -> bind(X) -> Ke -> restore(Env) -> K)
    control(xstack(XS) C) env(Env) .
endm

mod LOOP-RULES is
  including K-STATE .

  var S B J E : Exp . var K K’ : K . var Env Env’ : Env .
  var LS : LoopStack . var C C’ : Ctrl .


  eq exp(while(B) E) = exp(for(skip ; B ; skip) E) .
  op loop : -> Kitem .
  op ‘(_,_,_,_,_,_‘) : Exp Exp Exp Env K Ctrl -> LoopStack .

  eq k(exp(for(S ; B ; J) E) -> K) control(lstack(LS) C) env(Env)
   = k(exp(S ; B) -> loop)
    control(lstack((B,E,J,Env,K,C) -> LS) C) env(Env) .
  rl k(valList(bool(false)) -> loop)
    control(lstack((B,E,J,Env,K,C’) -> LS) C)
  => k(valList(unit) -> K) control(lstack(LS) C) .
  rl k(valList(bool(true)) -> loop)
     control(lstack((B,E,J,Env,K,C') -> LS) C)
  => k(exp(E ; J ; B) -> loop)
     control(lstack((B,E,J,Env,K,C') -> LS) C) .
  rl k(exp(break) -> K')
     control(lstack((B,E,J,Env,K,C) -> LS) C') env(Env')
  => k(valList(unit) -> K) control(lstack(LS) C) env(Env) .
  rl k(exp(continue) -> K')
     control(lstack((B,E,J,Env,K,C) -> LS) C') env(Env')
  => k(exp(J ; B) -> loop)
     control(lstack((B,E,J,Env,K,C) -> LS) C) env(Env) .
endm

mod K-SEMANTICS is
  including AEXP-RULES .
  including BEXP-RULES .
  including IF-RULES .
  including FUNCTION-RULES .
  including LET-RULES .
  including BASIC-STMT-RULES .
  including LIST-RULES .
  including IO-RULES .
  including EXCEPTION-RULES .
  including LOOP-RULES .

  op <_,_> : Exp IntList -> [IntList] .
  op result : State -> [IntList] .
  var P : Exp . var V : Val . var Store : Store . var Env : Env .
  var L : Loc . var Il : IntList . var Cfg : State .
  eq < P,Il >
   = result(
      k(exp(P))
      env(empty)
      control(fstack(.) xstack(.) lstack(.))
      in(Il) out(.)
      store(empty) nextLoc(loc(0))) .
  rl result(k(valList(V)) out(Il) Cfg)
  => Il .
endm
```

One can now use the rewrite engine of Maude to execute programs in FUN, thus getting an interpreter for the language directly from its formal definition:

```
rew < skip,. > .
***> should be .
rew < # 5,. > .
***> should be .
rew < print(# 5),. > .
***> should be 5
rew < print(read()),5 > .
***> should be 5
rew < print(# 2 + read()),3 > .
***> should be 5
rew < if read() >= # 3 then print(# 3) else print(read()), (2,5) > .
***> should be 5
```

```
rew < if read() >= # 3 then print(# 5) else print(read()), (4,2) > .
***> should be 5
rew < let(x,# 5,print(x)),. > .
***> should be 5
rew < (fun x -> print(x))(# 5),. > .
***> should be 5
rew < letrec(f,fun x -> (if x == # 0 then # 0 else (# 1 + (f(x /
# 2)))),
            print(f(# 31))),. > .
***> should be 5
rew < letrec('max,fun l ->
        ((if null?(cdr(l)) then return(car(l))) ;
         let(x,('max(cdr(l))),
         ((if (x <= car(l)) then return(car(l))) ;
         return(x)))),
        print('max([# 1,# 3,# 5,# 2,# 4,# 0,# -1,# -9]))),. > .
***> should be 5
rew < letrec('len,fun l -> (if null?(l) then # 0 else (# 1 +
('len(cdr(l))))),
            print('len(cons(# 3,[# 4,# 7,# 9,# 2])))),. > .
***> should be 5
rew < (try throw(read()) catch(x) print(x)),5 > .
***> should be 5
rew < (try let((x,y),(read(),read()),if y == # 0 then throw(x) else
print(x / y)) catch(x) print(x)),(15,3) > .
***> should be 5
rew < (try let((x,y),(read(),read()),if y == # 0 then throw(x) else
print(x / y)) catch(x) print(x)),(5,0) > .
***> should be 5
rew < let ((x,y),(read(),read()),
    ((while(not(y == # 0))
      ((x := (x % y)) ;
       (x := (x + y)) ;
       (y := (x - y)) ;
       (x := (x - y)) )) ;
     print(x))),(25,15) > .
***> should be 5

rew < let ((x,y),(read(),read()),
     ((for(skip ; not(y == # 0) ; ((x := (x % y)) ;
                                  (x := (x + y)) ;
                                  (y := (x - y)) ;
                                  (x := (x - y)) )) skip) ; print(x))),(35,25) > .
***> should be 5

rew < let (x,read(),for( skip ; true ; x := read()) ((if x <= # 0 then
continue) ; print(x) ; break)),(-2,0,-3,-7,5,-4,30,2) > .
***> should be 5
```

G Definition of Full FUN in Maude

In this appendix we show the Maude-ified version of the full FUN language, whose K-definition is shown in Figure 7.

```
fmod SYNTAX is
```

```
  including INT .
  including QID .

--- Var
  sort Var .
  subsort Qid < Var .
  ops a b c d e f g h i j k l m n o p q r t u v x y z : -> Var [format (g o)] .

  sort Exp .
  subsort Var < Exp .
  op #_ : Int -> Exp [prec 30 format (b r o)] .
  ops _+_ _-_ : Exp Exp -> Exp [prec 33 gather (E e) format (d b o d)] .
  ops _*_ _/_ _%_ : Exp Exp -> Exp
                 [prec 32 gather (E e) format (d b o d)] .

  ops true false : -> Exp [format (r o)] .
  ops _<=_ _>=_ _==_ : Exp Exp -> Exp [prec 37 format (d b o d)] .
  op _and_ : Exp Exp -> Exp [prec 55 format (d b o d)] .
  op _or_ : Exp Exp -> Exp [prec 57 format (d b o d)] .
  op not_ : Exp -> Exp [prec 53 format (b o d)] .

  op skip : -> Exp [format (b o)] .

  op if_then_ : Exp Exp -> Exp
             [prec 70 format (b o bn+i o+ --)] .
  op if_then_else_ : Exp Exp Exp -> Exp
                     [prec 70 format (b o bn+i o+ bn-i o+ --)] .

  sort VarList ExpList .
  op _,_ : VarList VarList -> VarList [assoc id: .] .
  op . : -> VarList .
  op _,_ : ExpList ExpList -> ExpList [ditto] .
  subsort Var < VarList Exp < ExpList .

  op fun_->_ : VarList Exp -> Exp [format (b o b no++i n--)] .
  op _`(_`) : Exp ExpList -> Exp [prec 60 format(d b o b o)] .
  op return : Exp -> Exp [format (b o)] .

  ops let letrec : VarList ExpList Exp -> Exp [format (b o)] .

  op _;_ : Exp Exp -> Exp [gather (e E) prec 80 format (d b noi d)] .
  op _:=_ : Var Exp -> Exp [format (d b o d)] .

  op `[_`] : ExpList -> Exp [format (b o b o)] .
  ops car cdr null? : Exp -> Exp [format (b o)] .
  op cons : Exp Exp -> Exp [format (b o)] .

  op read`(`) : -> Exp [format (b d d o)] .
  op print : Exp -> Exp [format (b o)] .

  op try_catch`(_`)_ : Exp Var Exp -> Exp [format (b o b d o b o d)] .
  op throw : Exp -> Exp [format (b o)] .

  op while`(_`)_ : Exp Exp -> Exp [format (b d o b no++i --)] .
  op for`(_;_;_`)_ : Exp Exp Exp Exp -> Exp
                   [format (b d o b o b o b no++i --)] .
  op break : -> Exp [format (b o)] .
  op continue : -> Exp [format (b o)] .

  op callcc : Exp -> Exp [format (b o)] .
  ops spawn acquire release : Exp -> Exp [format (b o)] .
endfm

fmod LOC is
  including NAT .
  sort Loc .
  op loc : Nat -> Loc .
  op next : Loc -> Loc .
  var N : Nat .
  eq next(loc(N)) = loc(s(N)) .
endfm

fmod ENV is
  including SYNTAX .
  including LOC .
  sort Env .
  op ‘(_‘,_‘) : Var Loc -> Env .
  op __ : Env Env -> Env [assoc comm id: empty ] .
  op empty : -> Env .


  var Env : Env . var X : Var . var L L’ : Loc .
  --- retrieves a location from the environment
  op _[_] : Env Var -> Loc .
  eq (Env (X,L))[X] = L .

  --- updates a variable’s location in the environment
  op _[_<-_] : Env Var Loc -> Env .
  eq (Env (X,L))[X <- L’] = Env (X,L’) .
  eq Env[X <- L] = Env (X,L) [owise] .
endfm

fmod VAL is
  including INT .
  sort Val .
  op int : Int -> Val .
  op bool : Bool -> Val .
  op unit : -> Val .
endfm

fmod STORE is
  including VAL .
  including LOC .
  sort Store .
  op ‘(_‘,_‘) : Loc Val -> Store .
  op __ : Store Store -> Store [assoc comm id: empty ] .
  op empty : -> Store .

  var S : Store . var L : Loc . var V V’ : Val .

  --- retrieves a value from the state
  op _[_] : Store Loc -> Val .
  eq (S (L,V))[L] = V .

  --- updates a variable in the state
  op _[_<-_] : Store Loc Val -> Store .
  eq (S (L,V))[L <- V'] = S (L,V') .
  eq S[L <- V] = S (L,V) [owise] .
endfm

mod VAL-LIST is
  including VAL .
  sort ValList .
  subsort Val < ValList .
  op _,_ : ValList ValList -> ValList [assoc id: .] .
  op . : -> ValList .
endm

mod INT-LIST is
  including INT .
  sort IntList .
  subsort Int < IntList .
  op _,_ : IntList IntList -> IntList [assoc id: .] .
  op . : -> IntList .
endm

mod INT-SET is
  including INT .
  sort IntSet .
  subsort Int < IntSet .
  op __ : IntSet IntSet -> IntSet [assoc comm id: empty] .
  op empty : -> IntSet .
endm

mod INTINT-SET is
  including INT .
  sort IntIntSet .
  op `(_,_`) : Int Int -> IntIntSet .
  op __ : IntIntSet IntIntSet -> IntIntSet [assoc comm id: empty] .
  op empty : -> IntIntSet .
endm

mod K is
  including SYNTAX .
  including VAL-LIST .
  sort Kitem K .
  subsort Kitem < K .
  op exp : Exp -> K [memo] .

  sort KList .
  subsort K < KList .
  op _,_ : KList KList -> KList [assoc id: .] .
  op . : -> KList .

  op kList : KList -> Kitem .
  op valList : ValList -> Kitem .
  op _->_ : K K -> K [assoc id: nothing] .
  op nothing : -> K .
```

```
endm

mod STACKS is
  sorts FunctionStack ExceptionStack LoopStack .
  op _->_ : FunctionStack FunctionStack -> FunctionStack [assoc id: .] .
  op . : -> FunctionStack .
  op _->_ : ExceptionStack ExceptionStack -> ExceptionStack [assoc id:
.] .
  op . : -> ExceptionStack .
  op _->_ : LoopStack LoopStack -> LoopStack [assoc id: .] .
  op . : -> LoopStack .
endm

mod CONTROL is
  including STACKS .
  sort Ctrl .
  op __ : Ctrl Ctrl -> Ctrl [assoc comm id: .] .
  op . : -> Ctrl .
  op fstack : FunctionStack -> Ctrl .
  op xstack : ExceptionStack -> Ctrl .
  op lstack : LoopStack -> Ctrl .
endm

mod K-THREAD-STATE is
  including CONTROL .
  including ENV .
  including K .
  including INTINT-SET .
  sort ThreadState .
  op __ : ThreadState ThreadState -> ThreadState [assoc comm id: .] .
  op . : -> ThreadState .
  op k : K -> ThreadState .
  op env : Env -> ThreadState .
  op control : Ctrl -> ThreadState .
  op holds : IntIntSet -> ThreadState .
endm

mod K-STATE is
  including INT-LIST .
  including INT-SET .
  including STORE .
  including K-THREAD-STATE .
  sort State .
  op __ : State State -> State [assoc comm id: .] .
  op . : -> State .

  ops in out : IntList -> State .
  op store : Store -> State .
  op nextLoc : Loc -> State .
  op thread : ThreadState -> State .
  op busy : IntSet -> State .

  var E : Exp . var Env : Env .

  op newThread : Exp Env -> State .
  eq newThread(E,Env) = thread(k(exp(E)) env(Env)
      control(fstack(.) xstack(.) lstack(.)) holds(empty)) .
endm

mod K-BASIC is
  including K-STATE .
  op kv : KList ValList -> Kitem .
  var Ke : K . var Kel : KList . var K : K .
  var V V' : Val . var Vl : ValList . var TS : ThreadState .
  var X : Var . var Xl : VarList . var Env Env' : Env . var L : Loc .
  var E : Exp . var El : ExpList . var Store : Store .

  eq k(kList(Ke,Kel) -> K) = k(Ke -> kv(Kel,.) -> K) .
  eq valList(V) -> kv(Kel,Vl) = kv(Kel,Vl,V) .
  eq k(kv(Ke,Kel,Vl) -> K) = k(Ke -> kv(Kel,Vl) -> K) .
  eq k(kv(.,Vl) -> K) = k(valList(Vl) -> K) .

  op bind : VarList -> Kitem .
  eq thread(k(valList(Vl,V) -> bind(Xl,X) -> K)
     env(Env) TS) store(Store) nextLoc(L)
   = thread(k(valList(Vl) -> bind(Xl) -> K)
     env(Env[X <- L]) TS) store(Store (L,V)) nextLoc(next(L)) .
  eq k(valList(.) -> bind(.) -> K) = k(K) .
  eq thread(k(bind(Xl,X) -> K)
     env(Env) TS) nextLoc(L)
   = thread(k(bind(Xl) -> K)
     env(Env[X <- L]) TS) nextLoc(next(L)) .
  eq k(bind(.) -> K) = k(K) .

  op write : VarList -> Kitem .
  eq thread(k(valList(Vl,V) -> write(Xl,X) -> K)
     env(Env (X,L)) TS) store(Store)
   = thread(k(valList(Vl) -> write(Xl) -> K)
     env(Env (X,L)) TS) store(Store[L <- V]) .
  eq k(valList(.) -> write(.) -> K) = k(K) .

  op expList : ExpList -> KList .
  eq expList(E,El) = exp(E),expList(El) .
  eq expList(.) = . .

  op restore : Env -> Kitem .
  eq k(valList(V) -> restore(Env) -> K) env(Env')
   = k(valList(V) -> K) env(Env) .
endm

mod AEXP-RULES is
  including K-STATE .
  var I I1 I2 : Int . var X : Var . var K : K . var Store : Store .
  var A1 A2 : Exp . var Env : Env . var TS : ThreadState .
  eq exp(# I)  = valList(int(I)) .
  rl thread(k(exp(X) -> K) env(Env) TS) store(Store)
  => thread(k(valList(Store[Env[X]]) -> K) env(Env) TS) store(Store) .
  ops + - * / % : -> Kitem .
  eq exp(A1 + A2) = kList(exp(A1),exp(A2)) -> + .
  rl valList(int(I1),int(I2)) -> + => valList(int(I1 + I2)) .

  eq exp(A1 - A2) = kList(exp(A1),exp(A2)) -> - .
  rl valList(int(I1),int(I2)) -> - => valList(int(_-_(I1,I2))) .

  eq exp(A1 * A2) = kList(exp(A1),exp(A2)) -> * .
  rl valList(int(I1),int(I2)) -> * => valList(int(I1 * I2)) .

  eq exp(A1 / A2) = kList(exp(A1),exp(A2)) -> / .
  crl valList(int(I1),int(I2)) -> / => valList(int(I1 quo I2))
   if I2 =/= 0 .

  eq exp(A1 % A2) = kList(exp(A1),exp(A2)) -> % .
  crl valList(int(I1),int(I2)) -> % => valList(int(I1 rem I2))
   if I2 =/= 0 .
endm

mod BEXP-RULES is
  including K-STATE .
  var X : Var . var Store : Store . var B B1 B2 : Exp .
  var A1 A2 : Exp . var I1 I2 : Int . var T T1 T2 : Bool .
  eq exp(true)  = valList(bool(true)) .
  eq exp(false)  = valList(bool(false)) .
  ops <= >= == and or not : -> Kitem .
  eq exp(A1 <= A2) = kList(exp(A1),exp(A2)) -> <= .
  rl valList(int(I1),int(I2)) -> <= => valList(bool(I1 <= I2)) .

  eq exp(A1 >= A2) = kList(exp(A1),exp(A2)) -> >= .
  rl valList(int(I1),int(I2)) -> >= => valList(bool(I1 >= I2)) .

  eq exp(A1 == A2) = kList(exp(A1),exp(A2)) -> == .
  rl valList(int(I1),int(I2)) -> == => valList(bool(I1 == I2)) .

  eq exp(B1 and B2) = kList(exp(B1),exp(B2)) -> and .
  rl valList(bool(T1),bool(T2)) -> and => valList(bool(T1 and T2)) .

  eq exp(B1 or B2) = kList(exp(B1),exp(B2)) -> or .
  rl valList(bool(T1),bool(T2)) -> or => valList(bool(T1 or T2)) .

  eq exp(not B) = exp(B) -> not .
  rl valList(bool(T)) -> not => valList(bool(not T)) .
endm

mod IF-RULES is
  including K-STATE .
  op if : K K -> Kitem .
  var E E1 E2 : Exp . var K K1 K2 : K .
  eq exp(if E then E1) = exp(if E then E1 else skip) .
  eq exp(if E then E1 else E2) = exp(E) -> if(exp(E1),exp(E2)) .
  rl k(valList(bool(true)) -> if(K1,K2) -> K) => k(K1 -> K) .
  rl k(valList(bool(false)) -> if(K1,K2) -> K) => k(K2 -> K) .
endm

mod FUNCTION-RULES is
  including K-BASIC .
  op closure : VarList K Env -> Val .
  var Xl : VarList . var E : Exp . var El : ExpList . var X : Var .
  var Env Env' : Env . var V : Val . var Vl : ValList . var K K' : K .
  var Store : Store . var L : Loc . var FS : FunctionStack .
  var C C’ : Ctrl .
  eq k(exp(fun Xl -> E) -> K) env(Env)
   = k(valList(closure(Xl,exp(E),Env)) -> K) env(Env) .
  op app : -> Kitem .
  eq exp(E(El)) = kList(exp(E),expList(El)) -> app .

  op ‘(_,_,_‘) : K Env Ctrl -> FunctionStack .
  ops popFstack return : -> Kitem .

  rl k(valList(closure(Xl,K,Env),Vl) -> app -> K’)
     control(fstack(FS) C) env(Env’)
  => k(valList(Vl) -> bind(Xl) -> K -> popFstack)
     control(fstack((K’,Env’,C) -> FS) C) env(Env) .
  eq k(valList(V) -> popFstack)
     control(fstack((K,Env,C’) -> FS) C) env(Env’)
   = k(valList(V) -> K) control(fstack(FS) C) env(Env) .

  eq exp(return(E)) = exp(E) -> return .
  eq k(valList(V) -> return -> K’)
     control(fstack((K,Env,C) -> FS) C’) env(Env’)
   = k(valList(V) -> K) control(fstack(FS) C) env(Env) .
endm

mod LET-RULES is
  including K-BASIC .
  var Env Env’ : Env . var K : K . var Xl : VarList . var El : ExpList .
  var E : Exp .
  rl k(exp(let(Xl,El,E)) -> K) env(Env)
  => k(kList(expList(El)) -> bind(Xl) -> exp(E) -> restore(Env) -> K)
     env(Env) .

  rl k(exp(letrec(Xl,El,E)) -> K) env(Env)
  => k(bind(Xl) -> kList(expList(El)) -> write(Xl) -> exp(E)
         -> restore(Env) -> K) env(Env) .
endm

mod BASIC-STMT-RULES is
  including K-BASIC .
  eq exp(skip) = valList(unit) .
  op ; : -> Kitem .
  var E E1 E2 : Exp . var V1 V2 : Val . var X : Var .
  eq exp(E1 ; E2) = kList(exp(E1),exp(E2)) -> ; .
  rl valList(V1,V2) -> ; => valList(V2) .
  eq exp(X := E) = exp(E) -> write(X) -> valList(unit) .
endm

mod LIST-RULES is
  including K-BASIC .
  ops ‘[‘] car cdr null? cons : -> Kitem .
  var El : ExpList . var Vl : ValList . var E E1 E2 : Exp .
  var V : Val .
  eq exp([El]) = kList(expList(El)) -> [] .
  op ‘[_‘] : ValList -> Val .
  eq valList(Vl) -> [] = valList([Vl]) .
  eq exp(car(E)) = exp(E) -> car .
  rl valList([V,Vl]) -> car => valList(V) .
  eq exp(cdr(E)) = exp(E) -> cdr .
  rl valList([V,Vl]) -> cdr => valList([Vl]) .
  eq exp(null?(E)) = exp(E) -> null? .
  rl valList([V,Vl]) -> null? => valList(bool(false)) .
  rl valList([.]) -> null? => valList(bool(true)) .
  eq exp(cons(E1,E2)) = kList(exp(E1),exp(E2)) -> cons .
  rl valList(V,[Vl]) -> cons => valList([V,Vl]) .
endm

mod IO-RULES is
  including K-STATE .
  var E : Exp . var I : Int . var Il : IntList . var K : K .
  var TS : ThreadState .
  rl thread(k(exp(read()) -> K) TS) in(I,Il)
  => thread(k(valList(int(I)) -> K) TS) in(Il) .
  op print : -> Kitem .
  eq exp(print(E)) = exp(E) -> print .
  rl thread(k(valList(int(I)) -> print -> K) TS) out(Il)
  => thread(k(valList(unit) -> K) TS) out(Il,I) .
endm

mod EXCEPTION-RULES is
  including K-BASIC .
  op ‘(_,_,_,_,_‘) : Var K Env K Ctrl -> ExceptionStack .
  ops popXstack throw : -> Kitem .

  var E E’ : Exp . var K K’ Ke : K . var X : Var . var C C’ : Ctrl .
  var Env Env’ : Env . var XS : ExceptionStack . var V : Val .
  eq k(exp(try E’ catch(X) E) -> K) control(xstack(XS) C) env(Env)
   = k(exp(E’) -> popXstack)
    control(xstack((X,exp(E),Env,K,C) -> XS) C) env(Env) .
  rl k(valList(V) -> popXstack)
    control(xstack((X,Ke,Env,K,C) -> XS) C’) env(Env’)
  => k(valList(V) -> K) control(xstack(XS) C’) env(Env) .

  eq exp(throw(E)) = exp(E) -> throw .
  rl k(valList(V) -> throw -> K’)
    control(xstack((X,Ke,Env,K,C) -> XS) C’) env(Env’)
  => k(valList(V) -> bind(X) -> Ke -> restore(Env) -> K)
    control(xstack(XS) C) env(Env) .
endm

mod LOOP-RULES is
  including K-STATE .

  var S B J E : Exp . var K K’ : K . var Env Env’ : Env .
  var LS : LoopStack . var C C’ : Ctrl .


  eq exp(while(B) E) = exp(for(skip ; B ; skip) E) .
  op loop : -> Kitem .
  op ‘(_,_,_,_,_,_‘) : Exp Exp Exp Env K Ctrl -> LoopStack .

  eq k(exp(for(S ; B ; J) E) -> K) control(lstack(LS) C) env(Env)
   = k(exp(S ; B) -> loop)
    control(lstack((B,E,J,Env,K,C) -> LS) C) env(Env) .
  rl k(valList(bool(false)) -> loop)
     control(lstack((B,E,J,Env,K,C') -> LS) C)
  => k(valList(unit) -> K) control(lstack(LS) C) .
  rl k(valList(bool(true)) -> loop)
     control(lstack((B,E,J,Env,K,C') -> LS) C)
  => k(exp(E ; J ; B) -> loop)
     control(lstack((B,E,J,Env,K,C') -> LS) C) .
  rl k(exp(break) -> K')
     control(lstack((B,E,J,Env,K,C) -> LS) C') env(Env')
  => k(valList(unit) -> K) control(lstack(LS) C) env(Env) .
  rl k(exp(continue) -> K')
     control(lstack((B,E,J,Env,K,C) -> LS) C') env(Env')
  => k(exp(J ; B) -> loop)
     control(lstack((B,E,J,Env,K,C) -> LS) C) env(Env) .
endm

mod CALLCC-RULES is
  including FUNCTION-RULES .
  op callcc : -> Kitem .
  op cc : K Ctrl Env -> Val .
  var E : Exp . var V : Val . var K K' : K . var Env Env' : Env .
  var C C' : Ctrl .
  eq exp(callcc(E)) = exp(E) -> callcc .
  rl k(valList(V) -> callcc -> K) control(C) env(Env)
  => k(valList(V,cc(K,C,Env)) -> app -> K) control(C) env(Env) .
  rl k(valList(cc(K,C,Env),V) -> app -> K') control(C') env(Env')
  => k(valList(V) -> K) control(C) env(Env) .
endm

mod THREAD-RULES is
  including K-STATE .
  var V : Val . var TS : ThreadState . var E : Exp . var Env : Env .
  var I : Int . var N : Nat . var Is : IntSet . var LCs : IntIntSet .
  var K : K .

  rl thread(k(exp(spawn(E)) -> K) env(Env) TS)
  => thread(k(valList(unit) -> K) env(Env) TS) newThread(E,Env) .
  rl thread(k(valList(V)) holds(LCs) TS) busy(Is)
  => busy(Is - LCs) .

  ops acquire release : -> Kitem .
  eq exp(acquire(E)) = exp(E) -> acquire .
  rl k(valList(int(I)) -> acquire -> K) holds((I,N) LCs)
  => k(valList(unit) -> K) holds((I,s(N)) LCs) .
  crl thread(k(valList(int(I)) -> acquire -> K) holds(LCs) TS) busy(Is)
   => thread(k(valList(unit) -> K) holds((I,0) LCs) TS) busy(Is I)
   if not(I in Is) .

  op _in_ : Int IntSet -> Bool .
  eq I in I Is = true .
  eq I in Is = false [owise] .

  eq exp(release(E)) = exp(E) -> release .
  rl k(valList(int(I)) -> release -> K) holds((I,s(N)) LCs)
  => k(valList(unit) -> K) holds((I,N) LCs) .
  rl thread(k(valList(int(I)) -> release -> K) holds((I,0) LCs) TS)
```

```
     busy(Is I)
  => thread(k(valList(unit) -> K) holds(LCs) TS) busy(Is) .

  op _-_ : IntSet IntIntSet -> IntSet .
  eq (I Is) - ((I,N) LCs) = Is - LCs .
  eq Is - LCs = Is [owise] .
endm

mod K-SEMANTICS is
  including AEXP-RULES .
  including BEXP-RULES .
  including IF-RULES .
  including FUNCTION-RULES .
  including LET-RULES .
  including BASIC-STMT-RULES .
  including LIST-RULES .
  including IO-RULES .
  including EXCEPTION-RULES .
  including LOOP-RULES .
  including CALLCC-RULES .
  including THREAD-RULES .

  op <_,_> : Exp IntList -> [IntList] .
  op result : State -> [IntList] .
  var P : Exp . var Store : Store . var Is : IntSet .
  var L : Loc . var Il Il' : IntList . var Cfg : State .
  var TS : ThreadState .
  eq < P,Il >
   = result(
      newThread(P,empty)
      in(Il) out(.)
      store(empty) nextLoc(loc(0)) busy(empty)) .
  rl result(busy(Is) in(Il') out(Il) store(Store) nextLoc(L))
  => Il .
endm
```

Like for the Maude definition of sequential FUN in Appendix F, one can now use the rewrite engine to execute programs in FUN, thus getting an interpreter for the language directly from its formal definition. All the examples in Appendix F can be executed, plus many others involving callcc and threads. Also, one can use Maude's generic formal analysis tools to obtain corresponding formal analysis tools for FUN. For example, the two search commands below invoke the reachability analysis capability of Maude:

```
rew <
let('+,(fun (x,y) -> print(x + y)),
(callcc((fun k -> (k('+))))(read(),read()))),
(2, 3) > .
***> sum using callcc - should print 5

search < let(x,# 0,(spawn(x := (x + # 2)) ; spawn(x := (x + # 3)) ;
print(x))),. >
=>! I:Int .
***> should give 4 solutions, one of them 5 : dataraces & print may not wait
search < let((x,'t1,'t2),(# 0,# 0,# 0),
    (spawn((x := (x + # 2)); ('t1 := # 1)) ;
     spawn((x := (x + # 3)) ; ('t2 := # 1)) ;
     (while(not(('t1 + 't2) == # 2)) skip) ;
     print(x)
    )),. >
=>! I:Int .
***> should give 3 solutions, one of them 5 : dataraces, print should wait
search < let((x,t),(# 0,# 0),
    (spawn(acquire(# 0) ; (x := (x + # 2)); (t := (t + # 1)) ;
release(# 0)) ;
     spawn(acquire(# 0) ; (x := (x + # 3)) ; (t := (t + # 1)) ;
release(# 0)) ;
     (while(not(t == # 2)) skip) ;
     print(x)
    )),. >
=>! I:Int .
***> should give 1 solution: 5
```

Many more examples of formal analysis of concurrent programs are shown in Section 8.