The $\mathbb{K}$ Primer (version 2.5)

Traian Florin Șerbănuta$^{1,2,3}$ Andrei Arusoaei$^{1,3}$ David Lazar$^2$
Chucky Ellison$^2$ Dorel Lucanu$^{1,3}$ Grigore Roșu$^{1,2,3}$

$^1$ University Alexandru Ioan Cuza of Iasi
$^2$ University of Illinois at Urbana-Champaign

Abstract

This paper serves as a brief introduction to the $\mathbb{K}$ tool, a system for formally defining programming languages. It is shown how sequential or concurrent languages can be defined in $\mathbb{K}$ simply and modularly. These formal definitions automatically yield an interpreter for the language, as well as program analysis tools such as a state-space explorer.

Keywords: Theory and formal methods, programming language design

1 Introduction

Programming languages are the key link between computers and the software that runs on them. While programming languages usually have a formally defined syntax, this is not true of their semantics. Semantics is most often given in natural language, in the form of a reference manual or reference implementation, but rarely using mathematics. However, without a formal language semantics, it is impossible to rigorously reason about programs in that language. Moreover, a formal semantics of a language is a specification offering its users and implementers a solid basis for agreeing on the meaning of programs. Of course, providing a complete formal semantics for a programming language is notoriously difficult. This is partly because of the mathematics involved, and partly because of poor tool support.

To address this difficulty in writing language semantics, the $\mathbb{K}$ framework [16] was introduced as a semantic framework in which programming languages, calculi, as well as type systems or formal analysis tools can be defined. The aim of $\mathbb{K}$ in general is to demonstrate that a formal specification language for programming...
languages can be simultaneously simple, expressive, analyzable, and scalable. This paper serves as an introduction, tutorial, and reference for the $K$ tool [9] version 2.5, an implementation of the $K$ framework. We show how using the tool one can not only develop modular, executable definitions, but also easily experiment with language design by means of testing and exhaustive behavior exploration.

$K$ definitions are written in machine-readable ASCII, which the $K$ tool accepts as input. For execution and analysis purposes, the definitions are translated into Maude [4] rewrite theories. For visualization and documentation purposes, definitions are typeset into their $\LaTeX$ mathematical representation. Figure 1 gives the $K$ definition (both ASCII and mathematical representations) of a simple calculator language with input and output. This language will be described in great detail in Section 2.

Besides didactic and prototypical languages (such as lambda calculus, System F, and Agents), the $K$ tool has been used to formalize C [6], Scheme [11] and OCL [1,17]; additionally, definitions of Haskell and JavaScript are underway. With respect to analysis tools, the $K$ tool has been used to define type checkers and type inferencers [7], and is currently being used in the development of a new program verification tool using Hoare-like assertions based on matching logic [12,14], in a model checking tool [2] based on predicate abstraction, and in researching runtime verification techniques [5,15]. All of these definitions and analysis tools can be found on the $K$ tool website [8].

2 Writing the first $K$ definition

In this section we will guide the reader through the process of writing a simple language definition using the $K$ tool, using the definition of the EXP language presented in Figure 1. To keep the presentation succinct, we will write our definition so that it works with the default compilation, parsing, and execution options of the tool. Sections 4 and 5 will provide descriptions of the more advanced options.

2.1 Basic ingredients

When beginning to write $K$ definitions using the $K$ tool, it is recommended to test the definition as often as possible, thus catching problems early in the development process while they are easier to fix. Therefore, we will start by showing how to get a testable definition as early as possible in the development process.

2.1.1 Modules

The $K$ tool provides modules for grouping language features. A module is defined by the syntax:

```
module (NAME)
...
end module
```

4 All fragments of $K$ definitions presented in this paper were generated using the $K$ tool.
module EXP-SYNTAX
//@ Arithmetics Syntax
syntax Exp ::= #Int
| Exp * Exp [strict] //addition
| Exp * Exp [strict] //multiplication
| Exp / Exp [strict] //division
| Exp / Exp [strict] //modulo
| Exp ^ Exp [strict] //exponentiation
| Exp ? Exp : [transition]
| Exp ; [transition]
end module

//@ Concurrency features
syntax Exp ::= "spawn" Exp
| "rendezvous" Exp [transition]
end module

module EXP
imports EXP-SYNTAX
syntax KResult ::= #Int
configuration
⟨k color="green" multiplicity="*"⟩ $PGM:K ⟨/k⟩
⟨streams⟩
⟨in color="magenta" stream="stdin"⟩ .List ⟨/in⟩
⟨out color="Fuchsia" stream="stdout"⟩ .List ⟨/out⟩
⟨/streams⟩
end module

//@ Arithmetics Semantics
rule 11:#Int + 12:#Int => 11 +Int 12
rule 11:#Int * 12:#Int => 11 *Int 12
rule 11:#Int / 12:#Int => 11 /Int 12 when 12 /=Bool 0
rule 0 ? _ : E:Exp => E
rule 1:#Int ? E:Exp : _ => E when 1 /=Bool 0
rule _:#Int ; 12:#Int => 12
//@ Input / Output Semantics
rule (k) read => 1:#Int -⟨/k⟩
⟨in⟩ ListItem(1) => . -⟨/in⟩
rule (k) print 1:#Int => 1 -⟨/k⟩
⟨out⟩- . => ListItem(1) ⟨/out⟩
//@ Concurrency Semantics
rule (k) spawn E => 0 -⟨/k⟩
⟨. => (k) E ⟨/k⟩
rule (k) rendezvous 1 => 0 -⟨/k⟩
⟨(k) rendezvous 1 => 0 -⟨/k⟩
end module

module EXP-SYNTAX
Arithmetics Syntax
syntax Exp ::= #Int
| Exp + Exp [strict]
| Exp + Exp [strict]
| Exp / Exp [strict]
| Exp ? Exp : Exp [strict(1)]
| Exp ; Exp [strict]
Input / Output Syntax
syntax Exp ::= spawn Exp
| rendezvous Exp [strict]
Concurrency features
syntax Exp ::= read
| print Exp [strict]
Input / Output Semantics
syntax Exp ::= spawn Exp
| rendezvous Exp [strict]
Concurrency features
end module

Fig. 1. Κ definition of a calculator language with I/O (left: ASCII source; right: generated \LaTeX)
where ⟨NAME⟩ is a name identifying the module. In Figure 1, lines 1–20 define the EXP–SYNTAX module. It is customary to use only capital letters and hyphens for a module name.

A $\mathbb{K}$ definition is required to have at least one main module, but the $\mathbb{K}$ tool expects at least two by default: ⟨NAME⟩ and ⟨NAME⟩–SYNTAX. Separating the syntax from the rest of the definition minimizes parsing ambiguities in the tool. To import modules, one needs to add at least one imports ⟨NAME⟩ directive after the header of a module (e.g., line 23 in Figure 1). Multiple modules can be imported using the same imports directive by summing their names with the “+” symbol.

2.1.2 Compiling definitions
$\mathbb{K}$ definitions are usually stored into files with extension “.k”. Assume a file exp.k containing the following text:

```
module EXP–SYNTAX
end module

module EXP
imports EXP–SYNTAX
end module
```

To compile this definition (and, indirectly, check its validity), one can execute the following command:

```
$ kompile exp
Compiled version written in exp-compiled.maude.
```

kompile assumes the default extension for the file to be compiled. Moreover, the file ⟨name⟩.k to be compiled is assumed by default to contain a module ⟨NAME⟩, where ⟨NAME⟩ is the fully capitalized version of ⟨name⟩.

With the basic skeleton modules in place and the tool working, we can start to build up our definition, which we describe in the following sections.

2.1.3 Comments
The $\mathbb{K}$ tool allows C-like comments, introduced by “//” for single-line comments and “/* ... */” for multi-line comments. In addition, the $\mathbb{K}$ tool offers literate programming [10] capabilities via $\LaTeX$-specific comments (introduced by “//@” and “/*@”), which can be used to generate ready-to-publish definitions (see Section 4.3).

2.2 Language Syntax
Before we can give a formal semantics to a language, we first need to provide a formal syntax. Although the $\mathbb{K}$ parser can parse many non-trivial languages, it is not meant to be a substitute for real parsers. We often call the syntax defined in $\mathbb{K}$ “the syntax of the semantics”, to highlight the fact that its role is to serve as a convenient notation when writing the semantics, not as a means to define concrete syntax of arbitrarily complex programming languages. Programs written in these languages can be parsed using an external parser and transformed into the $\mathbb{K}$ AST (Abstract Syntax Tree) for execution and analysis purposes (see Section 5.1).
2.2.1 User-defined syntax
Syntax in K is defined using a variant of the familiar BNF notation, with terminals enclosed in quotes and nonterminals starting with capital letters (except for builtin nonterminals, which start with “#”). For example, the syntax declaration displayed on lines 4–9 in Figure 1 defines a syntactic category Exp, containing the builtin integers and three basic arithmetic operations on expressions. Productions can have attributes, which are used to help the parser with operator precedences and grouping (the prec and gather attributes—see Section 5.1), to instruct the PDF generator how to display the various constructs (the latex attribute—see Section 4.3), or to define the evaluation strategy of the corresponding constructs (the strict and seqstrict attributes—see Section 2.3.1).

2.2.2 Builtins
The K tool provides a number of builtin syntactic categories and semantic operations for them. To distinguish them from user-defined syntax, the names of these builtin categories are prefixed by the # symbol. The currently supported builtins are Booleans (#Bool), arbitrary precision integers (#Int), floats (#Float), strings (#String), and identifiers (#Id). Additionally, the K tool admits user-specified builtins (see Section 3.4).

2.2.3 Parsing programs
We can test a syntax definition by parsing programs written using that syntax. Suppose the file exp.k contains the modules EXP and EXP–SYNTAX, and that EXP imports EXP–SYNTAX (lines 22, 23, 73 from Figure 1). Suppose also there exists an EXP program 2avg.exp:

```k
print((read + read) / 2)
```

printing the average of two numbers read from the console.

Assuming that the definition was already compiled using kompile, the kast command can be used to test that the program parses and to see its corresponding K abstract syntax tree:

```
$ kast p1.exp
'print_(',,'_+(',read(.List{K}),'read(.List{K})),# 2(.List{K})))
```

For the above command to work, kast must be run from the directory containing the K definition file, say ⟨name⟩.k, and the compiled definition. The definition is assumed to contain a module called ⟨NAME⟩–SYNTAX. Moreover, all tokens not declared in the user syntax are assumed to be identifiers.

Note that the current parser allows the use of parentheses for grouping purposes, without the need to declare parentheses in the grammar. Also, note that the label associated to a production in the K AST (e.g., '_+') abstracts away the nonterminal information. This means that the tool does not distinguish between productions having the same “shape”; the user should be aware of that and avoid similar shapes for constructs which should have different semantics (see Section 3.5 for more details about the K AST).
2.3 Language Semantics

Specifying semantics within the K tool consists of three parts: providing evaluation strategies that conveniently (re)arrange computations, giving the structure of the configuration to hold program state, and writing K rules to describe transitions between configurations.

2.3.1 Evaluation strategies: Strictness

Evaluation strategies serve as a link between syntax and semantics, by specifying in what order the arguments of a language construct should be evaluated. For example, both arguments of an addition operator must be evaluated before computing their sum, whereas for the conditional operator “_?_:”, only the first argument should be evaluated. Although the order in which summands are evaluated might not matter, it matters crucially for a sequential composition operator (e.g., “_:”).

Since rules for specifying evaluation strategies are tedious to write in most definitional formalisms, the K tool allows the user to annotate the syntax declarations with strictness constraints specifying how the arguments of a construct are to be evaluated. Although strictness constraints have semantic meaning, it is more convenient to write them as annotations to the syntax, as they often refer to multiple nonterminal positions in the syntax declaration.

The K tool provides two strictness attributes: strict and seqstrict. Each optionally takes a list of space-separated numbers as arguments, denoting the positions on which the construct is strict (1 being the leftmost position). For example, the annotation strict(1) on line 8 in Figure 1 specifies that only the first argument of the conditional expression must be evaluated before giving semantics to the construct itself. If no argument is provided, then all of the positions are considered strict.

The only difference between strict and seqstrict is that the latter ensures the arguments are evaluated in the order given as an argument in the list, while the former allows non-determinism. In particular, if no argument is provided, seqstrict enforces the left-to-right order of evaluation for the arguments of the considered construct. For example, the strict annotation on line 5 in Figure 1 says that we want to evaluate all arguments of the “+” operator before giving its semantics, but does not constrain the evaluation strategy. In contrast, the seqstrict annotation on line 9 also specifies that both subexpressions must be evaluated, but that the left expression must be evaluated first, to guarantee the effects of sequential composition.

The K tool distinguishes a category of terms, KResult, which is used to determine which terms are values, or results. The syntax declaration on line 24 in Figure 1 specifies that integers are values in the EXP language.

More advanced details about strictness (including the more generalized notion of evaluation contexts) are presented in Section 3.2.4. How to use the K tool to explore the non-determinism associated with strictness is discussed in Section 3.3.

2.3.2 Computations

The sequencing of evaluation (see Section 3.2.4) is made possible in K by computation structures. Computation structures, called “computations” for short, extend the
abstract syntax of a language with a list structure using the separator \( \bowtie \) (read “followed by” or “and then”, and written \( \Rightarrow \) in ASCII). \( K \) provides a distinguished sort, \( K \), for computations. The extension of the user-defined syntax of the language into computations is done automatically for the constructs declared using the \texttt{syntax} keyword. The \texttt{KResult} sort described in the previous section is a subsort of \( K \).

The intuition for computation structures of the form \( t_1 \bowtie t_2 \bowtie \cdots \bowtie t_n \) is that the listed tasks are to be processed in order. The initial computation in an evaluation typically contains the original program as its sole task, but rules can then modify it into task sequences. Primarily this is done by the rules generated from strictness annotations, which sequence the evaluation of the arguments before the evaluation of the construct itself (see Section 3.2.4).

2.3.3 Configurations, Initial configuration

In \( K \), the state of a running program/system is represented by a \textit{configuration}. Configurations are structured as nested, labeled cells containing various computation-based data structures. Within the \( K \) tool, configuration cells are represented using an XML-like notation, with the label of the cell as the tag name and the contents between the opening and closing tags.

Lines 25–30 in Figure 1 describe both the initial configuration and the general structure of a configuration for EXP. It is introduced by the \texttt{configuration} keyword, and consists of a cell labeled \( k \) which is meant to hold the running program (denoted here by the variable \$PGM of type \( K \)), and a cell \texttt{streams} holding the \texttt{in} and \texttt{out} cells, which model an executing program’s input/output streams (abstracted as lists). The cells in the configuration declaration can contain the following XML attributes: \texttt{color}, used for display purposes (Section 4.3); \texttt{stream}, used for real interpreter-like I/O (Section 2.4); and \texttt{multiplicity}, used to specify how many copies of a cell are allowed (either 0 or 1 (?), 0 or more (+), or one or more (+)). Variables in the initial configuration, e.g., \$PGM:K on line 26 in Figure 1, are place-holders. They are initialized at the beginning of runtime (see Sections 2.4 and 5.2).

The only types of cell contents currently allowed by the \( K \) tool are computations and lists/bags/sets/maps of computations with their sorts/injection constructors being \texttt{List}, \texttt{Bag}, \texttt{Set}, and \texttt{Map}, respectively. The corresponding injections from the computation sort \( K \) into these sorts are \texttt{ListItem}, \texttt{BagItem}, \texttt{SetItem}, and \( \mapsto \) (written in ASCII as \( \mapsto \)) respectively. As the \( K \) sort encompasses all builtin datatypes and user-defined syntax, this allows for items like \texttt{ListItem(I:\#Int)}, \texttt{SetItem(5)}, and \( x \mapsto 5 \) to be written. Multiple items are space-separated, for example: \texttt{BagItem(4) BagItem(2)}. The unit of these sorts is denoted by \( . \) which, for disambiguation purposes, can be suffixed by the sort name, e.g., \( .List \) inside the \texttt{in} cell on line 28 in Figure 1.

The configuration declaration serves as the backbone for the process of \textit{configuration abstraction} (see Section 3.2.3) which allows users to only mention the relevant cells in each semantic rule, the rest of the configuration context being inferred automatically.
2.3.4 Rules

The semantic rules, or $\mathbb{K}$ rules, describe how a running configuration evolves by advancing the computation and potentially altering the state/environment. A $\mathbb{K}$ rule describes how a term or subterm in the configuration can change into another term, in a way similar to that of a rewrite rule: any term matching the left-hand side of a rule can be replaced by the right-hand side. In the $\mathbb{K}$ tool, semantic rules are introduced by the rule keyword.

In $\mathbb{K}$, rewriting is extended using an idea known as local rewriting. By pushing the rewrite action inside their contexts, $\mathbb{K}$ rules can omit parts of a term that would otherwise be duplicated on both sides of a rewrite rule. In the $\mathbb{K}$ tool this is done by allowing multiple occurrences of the rewrite symbol $\Rightarrow$ (written in ASCII as $\Rightarrow$), linking the parts of the matching contexts which are changed by the rule (the left side of $\Rightarrow$) and their corresponding replacements (the right side of $\Rightarrow$). In the \LaTeX notation, this replacement is displayed on the vertical axis, using a horizontal line instead of the rewrite symbol. For example, the print rule (lines 54–55 in Figure 1) performs two changes in the configuration: (1) the print expression is replaced by its integer argument; (2) a list item containing that integer replaces the empty list in the output cell (i.e., it is added to that list).

In $\mathbb{K}$, parts of the configuration can be omitted and inferred, so that as little of the configuration as possible needs to be given in a rule. This inference process, called configuration abstraction, relies on the fixed structure of the specified initial configuration. For example, this policy allows the print rule mentioned above to omit the streams cell, as it can be easily inferred. More details about configuration abstraction are discussed in Section 3.2.3.

As an additional notational shortcut, $\mathbb{K}$ allows omission of the contents of the cells at either end. In the $\mathbb{K}$ tool this is specified by attaching three dots (an ellipsis) at the left or the right end of a cell. In the \LaTeX notation, this is signified by the corresponding edge of the cell to look jagged or torn. With this convention, we now have the full semantics for the print rule: if print I (where I is an integer) is found at the beginning of the computation cell, then it is replaced by I and I is added at the end of the output list.

The above techniques make $\mathbb{K}$ rules simple and modular. By keeping rules compact and redundancy low, it is less likely a rule will need to be changed as the configuration is changed or new constructs are added to the language. For basic operations which do not require matching multiple parts of the configuration, a $\mathbb{K}$ rule might simply look like a rewrite rule, with just one rewrite symbol at the top of the rule. Furthermore, all rules can have side conditions which are introduced by the when keyword, and are expected to be boolean predicates.

Operations on builtin categories (such as those giving semantics for the basic arithmetic operators) are postfixed with the name of the category (excluding the # symbol): for example, $5 \mathbin{\texttt{+int}} 7$ applies the built-in integer addition on integers 5 and 7, while $5 \mathbin{\texttt{<=int}} 7$ produces the #Bool true constant. The full Maude specification of the provided builtins can be found in the pl-builtins.maude from the core/maude/lib directory of the distribution.

Variables are initial-cap letters or words followed by numbers or primes, as in Foo2 or X'. Variables can be sorted using a colon followed by the sort name. X:K is
Fig. 2. Three EXP programs and their interactive executions using krun

<table>
<thead>
<tr>
<th>$\text{cat } p0.exp$</th>
<th>$\text{cat } 2avg.exp$</th>
<th>$\text{cat } p4.exp$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 \cdot (4 + 6) / 2$</td>
<td>print((read + read) / 2)</td>
<td>print(100 / read)</td>
</tr>
<tr>
<td>$\langle k \rangle$</td>
<td>$\langle k \rangle$</td>
<td>(k) 100 / 0 $\Rightarrow$ print HOLE</td>
</tr>
<tr>
<td>$\langle /k \rangle$</td>
<td>(k)</td>
<td>(k)</td>
</tr>
<tr>
<td>$\langle \text{streams} \rangle$</td>
<td>(streams)</td>
<td>(streams)</td>
</tr>
<tr>
<td>$\langle /in \rangle$</td>
<td>(in)</td>
<td>(in)</td>
</tr>
<tr>
<td>$\langle \text{out} \rangle$</td>
<td>(out)</td>
<td>(out)</td>
</tr>
<tr>
<td>$\langle /out \rangle$</td>
<td>(out)</td>
<td>(out)</td>
</tr>
<tr>
<td>$\langle /\text{streams} \rangle$</td>
<td>(out)</td>
<td>(out)</td>
</tr>
</tbody>
</table>

As strictness is assumed to take care of the evaluation strategy, K rules are written with the assumption that the strict positions have already been evaluated (e.g., “print” was declared to be strict in its argument).

### 2.4 Executing programs with krun

Once the K definition of a language was written in the K tool and compiled successfully (using the kompile command), the krun command can be used to execute/interpret programs written in the defined language. By default, krun has the same restrictions as those mentioned for kast.

Figure 2 presents three programs and their runs using the krun tool. As mentioned in Section 2.3.3, the krun tool starts executing the program in the initial configuration, with the variable $\$PGM$ replaced by the K AST of the input program. The rules are then allowed to apply to the configuration, and the final configuration is obtained when no more rules apply. For example, the final configuration obtained after executing p0.exp contains 15 in the k cell, while the other cells are empty.

The “read and “print” instructions affect the in and out cells. Since these cells are annotated with the stream attribute, the krun tool requests input from the standard input stream whenever a rule needs an item from the in cell to match, and would flush whatever is entered in the out to the standard output stream. For example, when executing the program 2avg.exp, krun waits for the user to enter two numbers (the user entered 5 and 7 in the example), and then prints their arithmetic mean (6); then, the final configuration is printed.
Finally, the execution of `p4.exp` shows two things: that it is possible to pipe input to the program, and how a stuck final configuration looks like. As the rule for division is guarded by the condition that the divisor is not zero, no rule can advance the computation after `read` was replaced by 0. That is shown by the term `100/0` at the beginning of the computation.

We have shown here only the most basic uses of the `krun` tool. Section 5 shows how one can use options and configuration files to customize both the input and the output of `krun`.

### 3 More advanced K definitional features

Now, that the basics of writing a K definition have been covered, let us move on to discussing more advanced features that can be part of K definitions. In this section we will present several extensions to the EXP language and we will use them to exemplify several features not addressed in the previous section, including support for multithreading, modularity, advanced strictness, and advanced computations.

#### 3.1 Extending EXP with functional features

To make our language more expressive, let us start by adding λ and μ abstractions to it, turning it into a functional language. Instead of directly extending the existing definition in `exp.k`, we will also show how a definition can be spread across files and modules.

##### 3.1.1 Splitting definitions among files

Larger K definitions are usually spread in multiple files. A file can be included into another file using the `require` directive. Absolute `require` paths such as `/modules/substitution` are computed relatively to the path contained in the `$K_BASE` environment variable, which points to the installation directory of the K tool. The tree of `require` dependencies is followed recursively. If a file was already included through another `require` command, it will not be included twice. The
code shown in Figure 3 demonstrates these features. It is meant to be contained in a file `exp-lambda.k`, which resides in a directory `exp-lambda` within the directory containing `exp.k`.

### 3.1.2 Binders and substitution
Both “lambda” and “mu” are binders, binding their first argument to the second argument. We will use the builtin substitution operator to give semantics to these constructs. To guarantee that the substitution works correctly (avoids variable capturing), these constructs need to be marked with the `binder` annotation as shown on lines 8 and 11 of Figure 3. Currently, the `binder` annotation can only be applied to a two-argument production, of which the first must be an identifier.

With the substitution operator provided by the K tool’s SUBSTITUTION module, the semantics of function application and μ-unrolling are straightforward. The SUBSTITUTION module is completely defined in the K tool, leveraging the binder predicate mentioned above, and using the AST-view of the user-defined syntax to define a generic, capture-avoiding substitution. It exports a construct “syntax \( K ::= K[ K / #Id ] \)”, which substitutes the second argument for the third one in the first.

To guarantee a call-by-value evaluation strategy, the application operator is declared `seqstrict` in line 10, while the β-substitution operator is constrained to only happen at the top of the k cell (lines 28–29 of Figure 3). A special category “Val” is introduced to allow matching on both integers and λ-abstractions (lines 7–8 of Figure 3) and computation results are extended to include all the values (line 26). The μ-unrolling rule (lines 31–32) is also constrained to only apply at the top of the k cell. However, the reason here is avoid non-termination, by only unrolling μ in the evaluation position.

### 3.1.3 Desugaring syntax
We can show how one might desugar syntax in the K tool by adding two popular functional constructs to our language: “let” and “letrec”. Instead of directly giving them semantics, we will use “macro” capabilities to desugar them into λ and μ abstractions. The “let” operator desugars into an application of a λ-abstraction (lines 16–17 of Figure 3), while “letrec” desugars into the “let” construct binding a μ-abstraction (lines 18–19).

These macros are applied on programs at parse time; thus the K AST contains no traces of the desugared constructs. For example, the kast output for the program “letrec fact x = x ? x * (fact (x + -1)) : 1 in print (fact read)”, computing and printings the factorial of a number read from the console is:

```
''('''lambda''_._(# #id_('fact')(.List{K})),

'print'(1)(# #id_('factor')(List{K}), #read(List{K}))))),

'mu_._(# #id_('fact')(List{K}), 'lambda_._(# #id_('x')(.List{K})),

'?'(1)(# #id_('x')(List{K})),

'_'(# #id_('x')(List{K})),

'_'(# #id_('fact')(List{K}),'_'(# #id_('x')(List{K})),

'_'(# #id_('x')(List{K})),

'_'(# #id_('x')(List{K})),

'_'(# #id_('x')(List{K})),

'_'(# #id_('x')(List{K})),

'_'(# #id_('x')(List{K})),

'_'(# #id_('x')(List{K})),

# 1(List{K}))))
```

Technical Report, January 2012
3.2 Imperative features

In this section we will use the pretext of adding imperative features to our language to explain another set of advanced features of the K tool.

3.2.1 Statements

To begin, let us add a new syntactical category Stmt (for statements) and change the semicolon to be a statement terminator instead of a statement separator. This means splitting the production on line 9 in Figure 1 into two productions, a strict one injecting expressions into statements, and the other for composing statements:

\[
\text{syntax Stmt ::= Exp ';'} \quad \text{[strict]}
\ | \quad \text{Stmt Stmt}
\]

As we do not want statements to evaluate to values, we will not use strictness constraints to give their semantics. Instead, we replace the rule on line 43 in Figure 1 with two rules—one discarding the value for the expression statement, and the other sequencing the statements as computations:

\[
\text{rule 1: Int ; } \Rightarrow .
\]
\[
\text{rule St1:Stmt St2:Stmt } \Rightarrow \text{ St1 } \sim \text{ St2}
\]

3.2.2 Syntactic lists

K provides built-in support for generic syntactic lists: \(\text{List}\{\text{Nonterminal}, \text{terminal}\}\) stands for terminal-separated lists of zero or more Nonterminal elements. To instantiate and use the K built-in lists, you must alias each instance with a (typically fresh) nonterminal in your syntax. As an example, we can add variable declaration to our EXP language. The first production below defines the “Ids” alias for the comma-separated list of identifiers, while the second uses it to introduce variable declarations:

\[
\text{syntax Ids ::= List\{#,\Id, \},\}
\]
\[
\text{syntax Stmt ::= ‘var’ Ids ‘;’}
\]

Thus, both \text{var x, y, z;} and \text{var ;} are valid declarations.

For semantic purposes, these lists are interpreted as cons-lists (i.e., lists constructed with a head element followed by a tail list). Therefore when giving semantics to constructs with list parameters, we often need to distinguish two cases: one when the list has at least one element and another when the list is empty. To give semantics to “var”, we add three new cells to the configuration: \text{env}, to hold mappings from variables to locations, \text{store}, to hold mappings from locations to values, and \text{nextLoc}, to provide fresh locations. The following two rules specify the semantics of variable declarations:

\[
\text{rule (k) var (X:#Id,Xl:Ids } \Rightarrow \text{ XI) } : -(/k)
\]
\[
\text{(env) Rho:Map } \Rightarrow \text{ Rho[N/X] (/env)}
\]
\[
\text{(store)}-\quad \Rightarrow \text{ N } \rightarrow \text{ 0 } -(/\text{store})
\]
\[
\text{(nextLoc) N:#Int } \Rightarrow \text{ N +Int 1 (/nextLoc)}
\]
\[
\text{rule var .Ids ; } \Rightarrow . \quad \text{[structural]}
\]

The first rule declares the first variable in the list by adding a mapping from a fresh location to 0 in the \text{store} cell, and by updating the mapping of the name of the variable in the \text{env} cell to point to that location. The second rule terminates
the variable declaration process when there are no variables left to declare. Note how “.Ids” is used in the semantics to refer to the empty list. In general, a dotted-nonterminal represents the empty list for that nonterminal. We prefer to make the second rule structural, thinking of dissolving the residual empty var declaration as a structural cleanup rather than as a computational step (see Section 3.3 for more details about the types of rules).

3.2.3 Configuration abstraction

The addition of the env cell in the presence of concurrency requires further adjustments to the configuration. First, there needs to be an env cell for each computation cell, to avoid one computation shadowing the variables of the other one. Moreover, each environment should be tied to its computation, to avoid using another thread’s environment. This can be achieved by adding another cell, thread, on top of the k and env cells, and transferring the multiplicity tag of the k cell to the thread cell. Upon this transformation, the configuration will look like:

configuration
⟨thread, multiplicity="*"⟩
⟨k⟩ ¡PGM:K (/k)⟨env⟩ .Map (/env)
⟨/thread⟩
⟨nextLoc⟩ 0 ⟨/nextLoc⟩
⟨streams⟩
⟨in stream="stdin"⟩ .List (/in)
⟨out stream="stdout"⟩ .List (/out)
⟨/streams⟩
⟨store⟩ .Map (/store)

We also need to change the rule for “spawn” on lines 63–64 in Figure 1 to create the new thread:

rule (k) spawn E => 0 ⟨/k⟩ ⟨env⟩ Rho.Map (/env)
. => ⟨thread⟩− (k) E ⟨/k⟩ ⟨env⟩ Rho ⟨/env⟩ −⟨/thread⟩)

Changes in the configuration are quite frequent in practice, typically needed in order to accommodate new language features. K’s configuration abstraction process allows the users to not have to modify their rules when making structural changes to the language configuration. This is crucial for modularity, because it offers the possibility to write definitions in a way that may not require revisits to existing rules when the configuration is changed. Indeed, except for the “spawn” rule, none of the other rules (including the variable declaration rule above) need to change, despite there being the new thread cell involved. Instead, this cell is automatically inferred (and added by the K tool at compile time) from the definition of the configuration above. For our rule for “var” given in Section 3.2.2, it means that the k and env cells will be considered as being part of the same thread cell, as opposed to each being part of a different thread. The K tool can infer this context in instances when there is only one correct way to complete the configuration used in rules in order to match the declared configuration. To better understand what we mean by “one correct way”, we refer the interested reader to the K overview papers [13,16].

Multiplicity information is important in the configuration abstraction process, as it tells the K tool how to complete rules like “rendezvous” (lines 69–70 of Figure 1). As the k cell no longer has the multiplicity set to “*”, the tool can infer that each of the two computations resides in its own thread.

Continuing to add imperative features to our language, we can take the above information and add rules for reading and setting a variable in memory:
Note how these rules avoid mentioning the thread cell.

One limitation of the current implementation is that it does not allow multiple cells with the same name to appear in the initial configuration. This is not an inherent limitation of the configuration abstraction process, and will be corrected in later implementations of the K tool.

3.2.4 Advanced Strictness and Evaluation contexts

Suppose in our language we would like to be able to allow threads to share variables. We can achieve this goal using references. The syntax and semantics for adding references are:

\[
\text{syntax } \text{Exp} := \& \# \text{Id} \\
\text{rule } \langle k \rangle & X: \# \text{Id} = \text{I} : \# \text{Int} \Rightarrow \text{I} \cdot \langle /k \rangle \cdot \langle \text{env} \rangle \cdot X \cdot /\langle \text{store} \rangle \cdot \text{L} \cdot \langle /\text{env} \rangle \cdot /\langle \text{store} \rangle \cdot \text{L} \cdot \langle /\text{store} \rangle
\]

The above only provides read access to the references. To allow write access, we need to update the syntax and semantics for assignment as well:

\[
\text{syntax } \text{Exp} := \text{Exp} = \text{Exp} \cdot \text{strict}(2) \\
\text{rule } \langle k \rangle \ast L : \# \text{Int} = \text{I} : \# \text{Int} \Rightarrow \text{I} \cdot \langle /k \rangle \cdot \langle \text{store} \rangle \cdot \text{L} \cdot \langle /\text{store} \rangle
\]

However, this rule is not sufficient by itself, as it assumes the argument of “\(\ast\)” has been evaluated. Until now, we handled evaluation order using strictness annotations, but strictness cannot be used in this case because there are two syntactic productions involved (“\(\ast\)” and “\(\_ = \_\)” ) instead of one.

K contexts can be used to solve this problem, by generalizing strictness annotations. They allow the user to declare that a position in an arbitrary term should be evaluated. K contexts are similar to evaluation contexts [18]. For example, here is the context declaration needed above:

\[
\text{context } \ast \text{HOLE} = _
\]

where HOLE designates the position to be evaluated first. The context can be conditional, allowing side conditions on any of the variables of the term, including the HOLE. For example,

\[
\text{context } \ast \text{HOLE} = \text{I:} \# \text{Int} \text{ when } \text{I} < \text{I} \text{Int0}
\]

prohibits evaluating the HOLE position when writing a negative number, while

\[
\text{context } \ast \text{HOLE} = \text{I:} \# \text{Int} \text{ when } \text{HOLE} = /\text{==BoolNULL}
\]

prohibits evaluating the HOLE if it contains some “NULL” constant.

Finally, K contexts offer the possibility of performing additional operations on the term abstracted by HOLE. Suppose that in addition to the expression-dereferencing operator, we also had an array- or structure-dereferencing construct. Further suppose we had an increment operator in addition to the assignment one. Naively, we would need to specify contexts for any combination between the two categories. Instead,
we can extract the “l-value” of the expression to be assigned to or incremented. To do that, we use a special context syntax to state that the expression to increment should be wrapped by the auxiliary “lvalue” operation and then evaluated. Applying this technique for the context and rule above, we would obtain the following:

context (HOLE => lvalue(HOLE)) = _  
rule (k) loc(L)=V => V -> (/k)  
     ⟨store⟩- L|→ (_ => V) - ⟨/store⟩

where “lvalue” and “loc” are defined by:

syntax K ::= 'lvalue' K | 'loc' * K
context lvalue(* HOLE)
rule lvalue(* L: # Int) => loc(L)

Although this pattern is more verbose than the previous definition, it has the advantage of being more scalable, as it delegates the semantics of l-values to a single place.

3.2.5 Heating/cooling rules

Besides bringing more expressivity to K definitions, contexts are more general than the strictness annotations, allowing the latter to be expressed in terms of them. For example, the contexts associated to the strict annotation for “+” on line 5 in Figure 1 are:

context HOLE + _  
context _ + HOLE

while those for the seqstrict annotation for sequential composition (line 9) are

context HOLE ; _  
context R:KResult ; HOLE

However, contexts themselves are nothing else but syntactic sugar for so-called heating/cooling rules, which are nothing but a special kind of bidirectional rules. For example, the heating/cooling rules for the “* HOLE = _” context defined above area would be:

* K1:K = K2:K => K1:K ~> *HOLE= K2:K  
and  
K1:K ~> *HOLE= K2:K => * K1:K = K2:K

where the second rule is the inverse of the first, and *HOLE= is a special construct derived from the context term termed the “freezing context”. The heating rule for the more complex declaration (HOLE => lvalue(HOLE)) = _ is

K1:K = K2:K => lvalue(K1) ~> HOLE = K2

More details on how heating/cooling rules are implemented in the K tool can be found below (Section 3.3.3).

3.3 Controlling non-determinism

There are two main sources of non-determinism in programming languages—concurrency and order of evaluation. In this section we explain how the K tool can be used to explore both kinds of non-deterministic behavior.
### 3.3.1 Structural/Computational rules

At the theoretical level, \( \mathbb{K} \) rules are partitioned into structural rules and computational rules. Intuitively, structural rules rearrange the configuration so that computational rules can apply. Structural rules therefore do not count as computational steps. A canonical example of structural rules are the rules generated for strictness constraints. A \( \mathbb{K} \) semantics can be thought of as a generator of transition systems or Kripke structures, one for each program. Only the computational rules create steps, or transitions, in the corresponding transition systems or Kripke structures.

### 3.3.2 Transitions

Although desirable from a theoretical point of view, allowing all computational rules to generate transitions may yield a tremendous number of interleavings in practice. Moreover, most of these interleavings are behaviorally equivalent for most purposes. For example, the fact that a thread computes a step \( 8+3 \Rightarrow 11 \) is likely irrelevant for other threads, so one may not want to consider it as an observable transition in the space of interleavings. Since the \( \mathbb{K} \) tool cannot know (without help) which transitions need to be explored and which do not, our approach is to let the user say so explicitly with a "transition" attribute. The rules tagged with transition are the only ones considered as transitions when exploring a program's transition system.

For example, to explore the nondeterminism generated by the semantics of "print" for the EXP program \texttt{p5.exp} (\texttt{spawn(print 1); spawn(print 2); print 3;}), we can annotate the "print" rule on lines 54–55 in Figure 1 with the transition tag, and recompile with \texttt{kompile}. Figure 4 displays all behaviors observed upon use the \texttt{--search} option of the \texttt{krun} tool on \texttt{p5.exp}.

### 3.3.3 Non-deterministic strictness

The process of heating an argument to the front of the computation, reducing it one step, and cooling it back down may create an immense, often unfeasibly large space of possibilities to analyze. Therefore, for performance reasons, the \( \mathbb{K} \) tool chooses

![Fig. 4. Exploring the executions of a program with \texttt{krun --search}](image)
a default arbitrary, but fixed, order to evaluate the arguments of a strict language construct. Specifically, it eagerly heats and lazily cools the computation. This has the side effect of potentially losing behaviors due to missed interleavings.

To restore these missing interleavings, the K tool offers a superheat/supercooling process. These allow the user to customize the level of strictness-based nondeterminism available. They bear no theoretical signification, in that they do not affect the semantics of the language in any way. However, they have practical relevance, specific to our implementation of the K tool. More precisely, superheat is used to tell the K tool that we want to exhaustively explore all the non-deterministic evaluation choices for the strictness of the corresponding language construct. Similarly, whenever a rule tagged supercool is applied, the K tool will continue to apply (unrestricted) strictness cooling rules on the resulting computation, thus plugging everything back into its context, until no construct tagged superheat is left uncooled. Upon this step, the heating rules have the possibility to pick another evaluation order for the cooled fragment of computation.

This way, we can think of superheating/supercooling as marking fragments of computation in which exhaustive analysis of the evaluation order is performed. Used carefully, this mechanism allows us to explore more non-deterministic behaviors of a program, with minimal loss of efficiency.

For example, let us add the attribute superheat to the addition operator in EXP (line 5 in Figure 1):

```
syntax Exp ::= Exp + Exp [superheat strict]
```

After recompiling the K definition with kompile, we can run the program p3.exp ((print(1) + print(2)) + print(3)) using the --search option of krun. The tool finds four solutions, differing only in the contents of their out cell, containing strings "213", "123", "321", and "312". Why only four different outputs and not six? In the absence of any supercool rule, superheat only offers non-deterministic choice. That is, once an argument of a construct was chosen to be evaluated, it will be evaluated completely before being cooled. In order to observe full nondeterminism, the supercool attribute must be added to rules whose behavior we consider observable. If we add the supercool attribute to the output rule:

```
rule ⟨k⟩ print I:♯Int ⇒ 1 -(⟨k⟩) ⟨out⟩- . ⇒ ListItem(I) ⟨/out⟩ [supercool]
```

then, after recompiling, krun --search will exhibit all six expected behaviors of p3.exp, containing in the out cell the additional strings '132' and '231'.

It is worth noting that the use of these tags (transition, supercool, superheat) is sound with respect to the semantics of the definition in the sense they allow the K tool to partially explore the transition system defined by the semantics.

### 3.4 User builtins

K builtins are defined using the Maude language, to which all K definitions are compiled. In Figure 5 (left), we defined a new Maude module which declares a new sort #Frac and some operations over fractions. The module and sort names must be preceded with # to inform the K tool that these are builtins. Here we define
the operations we are interested in, \_\texttt{+Frac}_\_, \_\texttt{*Frac}_\_, and \_\texttt{/Frac}_\_, together with their “semantics” using Maude equations. Also, notice that we can use \texttt{K} builtins such as \_\texttt{+Int}_\_ and \_\texttt{*Int}_\_ when defining new builtins.

On the right side of Figure 5, we see how the newly defined module is used in a \texttt{K} definition. First, the builtin file is imported using the \texttt{require} directive, the same way we use it for \texttt{K} files. Second, the module \#FRAC is imported in EXP\texttt{–SYNTAX}, and we declare the sort \#Frac as being an expression. In the semantics module we make use of builtin operations declared in the \#FRAC module to write addition, multiplication and division rules.

Compilation changes slightly when using custom builtins, because we have to instruct \texttt{kompile} to use the additional builtin file:

\begin{verbatim}
$ kompile exp --lib frac.maude
Compiled version written in exp-compiled.maude.
\end{verbatim}

This will ensure that the compiled file will also include the new builtins and programs will parse correctly. After compilation, we can check if the builtin works:

\begin{verbatim}
$ cat frac.exp
 [4 / 5] + [6 / 5]
$ krun frac.exp
 ⟨kJ⟩
 ⟨/k⟩
\end{verbatim}

We can observe that the equations written in Maude applied successfully in the k cell and the program returns the expected output. It is also possible to use Maude hooks when declaring builtins, and such examples can be found in the default \texttt{K} builtins file: \texttt{pl-builtins.maude}.

### 3.5 \texttt{K} AST

In Section 2.2.3, we showed an example of how a \texttt{K} AST looks:

\begin{verbatim}
'print'_('''_('read(_.'List(_K))',''read(_.'List(_K))',''2(_.'List(_K))))
\end{verbatim}

The AST above is written in what we call “labeled form”. A \texttt{K} label (of sort \texttt{KLabel}) is applied to a list of elements of sort \texttt{K} separated by “\_\_”\_. Labels are generated from syntax declarations starting with a quote (‘), followed by the production where spaces are removed, sorts are replaced with underscores (_), and terminals stay
unchanged. For example, addition operator in Figure 1, line 5 has the corresponding label ‘_+_.

\( \mathbb{K} \) allows one to write rules using the labeled form. For example the addition rule (Figure 1, line 34) can be rewritten in labeled form:

\[
\text{rule} \quad '+_-'(\langle \mathbb{I}_1: \mathbb{I} \rangle, \langle \mathbb{I}_2: \mathbb{I} \rangle) = \mathbb{I}_1 + \mathbb{I} \mathbb{I}_2
\]

This option can be helpful when there are parsing issues. One can solve parsing ambiguities or sorting problems by writing the rules using the labeled form, which often avoids ambiguities due to being infix and is essentially monosorted. For instance, the \( \mathbb{K} \) tool will not be able to apply the rule below because the sort of M:Map[X] is K, while the addition operator expects an Expression:

\[
\text{rule} \quad (k) \ (X: \mathbb{E} \mathbb{p} \mathbb{r} \mathbb{t} \mathbb{a} = \mathbb{M}: \mathbb{M} \mathbb{p} \mathbb{a} \mathbb{p} \mathbb{t} [X]), Y: \mathbb{E} \mathbb{p} \mathbb{r} \mathbb{t} \mathbb{a} (\langle \mathbb{k} \rangle (\mathbb{M}: \mathbb{M} \mathbb{p} \mathbb{a} \mathbb{p} \mathbb{t} [X]), (\mathbb{k} \rangle (\mathbb{M}: \mathbb{M} \mathbb{p} \mathbb{a} \mathbb{p} \mathbb{t} [X]))
\]

This issue can be solved by writing the rule in the labeled form as follows:

\[
\text{rule} \quad (k) \ '+_-'((X: \mathbb{E} \mathbb{p} \mathbb{r} \mathbb{t} \mathbb{a} = \mathbb{M}: \mathbb{M} \mathbb{p} \mathbb{a} \mathbb{p} \mathbb{t} [X]), Y: \mathbb{E} \mathbb{p} \mathbb{r} \mathbb{t} \mathbb{a} (\langle \mathbb{k} \rangle \mathbb{M}: \mathbb{M} \mathbb{p} \mathbb{a} \mathbb{p} \mathbb{t} [X]))
\]

Finally, builtin datatypes “List”, “Map”, “Set”, and “Bag” can be written in label form by using the corresponding wrappers “wlist”, “wmap”, “wset”, and “wbag” to inject these types into KLabels.

### 3.6 Literate Programming

\( \mathbb{K} \) definitions adhere to the literate programming paradigm [10] by allowing an inter-mixture of formal rules and human-language descriptions to serve as documentation. In addition to the normal comments mentioned in Section 2.1.3, the \( \mathbb{K} \) tool also supports “compilable” comments, which are used in generating .pdfs. These special comments are allowed to contain any \LaTeX\ command. Section 4.3 shows how the kompile tool can be used to customize and typeset these definitions.

In Figure 1 at line 3 we have an example of a single line \LaTeX\ comment:

```
//@ Arithmetic Syntax
```

Multiline \LaTeX\ comments are also declared using the same pattern:

```
/*@ This is a multiline explanation
   of what was or what comes next. */
```

In addition to the special \LaTeX\ comments, the \( \mathbb{K} \) tool allows the user to associate \LaTeX\ commands for typesetting terminals in productions. For example, the \texttt{latex} attribute of the production:

```
syntax Exp ::= Exp "<=" Exp [seqstrict latex("\{#1\}\leq\{#2\})]
```

instructs that a term \( E \leq E' \) should be displayed as \( E \leq E' \) everywhere when typesetting the definition.

For large configurations, the user can specify where the top level configuration of the sub-cells of a cell should be split on multiple lines, by using the HTML-like \texttt{<br/>} element. The \texttt{<br/>} element can also be used at the top level of large rules with many cells to split them on multiple lines provided that the rule is annotated with the \texttt{large} attribute.
4 The kcompile tool

The kcompile tool is used to compile a K definition into a Maude rewrite theory. The K compiler has two components: the preprocessor and the compiler itself. Since we use Maude capabilities to parse and to compile a definition, the preprocessor acts as a wrapper over Maude and plays the user interface role providing different compilation options. The tool offers four categories of command line options: options for general purposes, options concerning \LaTeX generation, options for controlling non-determinism, and options for advanced features. All available options can be displayed using `kcompile --help`, which also provides more information about the compilation commands being executed.

In the remainder of this section, we will use the K definition of EXP presented in Figure 1 to explain the kcompile command line options.

4.1 Pre-processing and compilation

Before passing a K definition to the compiler, the preprocessor tokenizes the definition, appends some location metadata, and joins the definition into a single file.

kcompile creates a ./k/ directory to contain tool generated files and Maude input, output, and error files. On successful compilation, ./k/ contains only the “Maude-ified” version of the definition, e.g., `exp.maude` for `exp.k`, and a file named `all_tokens.tok`, which stores all the tokens from the definition. These tokens are exported by kcompile because they are needed by the krun tool to parse expressions/programs. Upon encountering an error in a definition, compilation fails and the ./k/ directory contains some additional files containing error messages. kcompile reports these errors and attempts to provide detailed instructions for finding their causes.

4.2 Name of the main module

By default, kcompile uses the the upper case version of the main file name as the name of the main module for the definition. If using another name for the main module, this must be indicated as the argument of the -l (or --lang) option. For example, “kcompile exp -l TEST” will instruct the K tool to use TEST as the main module for definition in file `exp.k`.

4.3 \LaTeX generation

To generate \LaTeX from the definition in Figure 1, we use the --latex option:

```
$ kcompile exp --latex
Generated exp.tex which contains the language definition.
```

If a list of module names is passed as argument to --latex, the tool generates \LaTeX only for those modules. If no module is given as argument, then kcompile typesets all user-defined modules and the \LaTeX comments between them.

Since \LaTeX files can be used to generate multiple different formats, kcompile provides shortcuts for some of them to generate .pdf, .ps, .eps, and .png files.
These other formats can be obtained by replacing the --latex option with either --pdf, --ps, or --eps, or --png, respectively.

By using a special comment in a definition, introduced by \!, the user can provide additional \LaTeX\ commands to go into the preamble of the generated \LaTeX\ file:

```latex
/*!
\title{EXP}
\author{Author (\texttt{author@mail.com})}
\organization{Organization}
*/
```

The tool puts the contents of this comment just before the \begin{document} corresponding to the whole definition and automatically appends \maketitle.

There are two different styles available for typesetting definitions. The \texttt{kompile} option --style has two possible values \texttt{mm}, denoting the mathematical style, and \texttt{bb} denoting “bubble”, more graphical style. These options must be used in conjunction with a \LaTeX\ option. The default value of --style option is \texttt{bb}.

The major difference between mathematical and bubble styles is related to how cells are displayed. In Figure 1, the definition is displayed using the bubble style. Below is shown the EXP configuration using the mathematical style:

```
configuration:

\langle$PGM$\rangle_k \langle\langle\cdot\rangle in \langle\cdot\rangle out\rangle streams
```

### 4.4 Exploring non-determinism

Section 3.3 showed how the \texttt{K} tool allows the exploration of non-deterministic features of a definition using the transition, superheat, and supercool annotations. In this section, we show how the user can set those attributes more generically.

The \texttt{kompile} tool allows the user to experiment with various settings for which productions to be superheating and which rules to be transitions/supercooling, without having to alter the definition at each iteration. This is achieved by allowing the user to redefine any of the superheat, supercool, and transition categories, by passing a string containing the space-separated list of tags which should be used instead of the default tag. For example, if the user had tagged certain rules with “tag1”, “tag2”, and “tag3”, then this compilation command could be used:

```
$ kompile exp --supercool "tag1 tag2 tag3"
```

to instruct the tool to treat all rules annotated with any of these tags as if they were annotated with supercool. One can also clear categories that were tagged directly. This is useful to increase the speed of interpretation, if transition analysis is not desired. This can be achieved with the command:

```
$ kompile exp --transition "" --superheat "" --supercool ""
```

By default, calling the \texttt{kompile} tool without setting any of these options has the same effect as:

```
$ kompile exp --transition "transition" --supercool "supercool" --superheat "superheat"
```
4.5 Advanced options

This category of options was added to allow advanced users to exploit some of the existent capabilities of K tool which were not otherwise accessible.

The option --prelude can be used for changing the default k-prelude.maude file. Advanced users have the freedom to replace the K prelude file with an user-defined prelude, which usually extends the original k-prelude.maude with more predefined sorts and operations.

As K definitions for complex languages grow bigger, the compilation time also increases. To address that, the --unquote option was added to kompile. This option short-circuits the formatting of the output module, using an external tool to pretty-print the Maude meta-modules in an attempt to increase compilation speed. Although this option is still experimental, it has been successfully used to compile the semantics of the C language [6].

For debugging reasons, the kompile tool provides the -m (or --maudify) option which only does preprocessing. For example, “kompile exp -m” only generates the .k/exp.maude file, which is exactly the compiler input. Another related option is the --flat option, which only flattens a K definition spread across multiple files into a single .k file located in the working directory.

5 The krun tool

The krun tool is used to execute and explore behaviors of programs for languages defined in K. The execution process consists of several steps, including parsing, initializing the configuration, the actual execution, and facilitating communication between the program and the OS. This section complements the already presented usages of krun by detailing some of the options krun offers for controlling each of the execution stages.

5.1 Parsing the input program

By default, the K tool takes advantage of Maude’s parsing capabilities to parse programs. At compile time, the syntax declarations in a K definition are translated into Maude operator declarations. The K tool allows the usage of the prec and gather syntax annotation attributes (which are taken over from Maude) for solving parsing ambiguities. For example, prec(...) defines a precedence for the corresponding syntactic construct, which is taken into account when parsing programs and rules (the lower the precedence the tighter the binding). Also, gather(...) is used for grouping reasons; for example, gather(E e) means left-associative and gather(e E) means right-associative. Note that these annotations are unrelated to the evaluation order of the constructs they apply to—they are only used to resolve parsing ambiguities. A left-associative operator may still evaluate right-to-left. For more information about these attributes we refer the interested reader to the Maude manual [3, Section 3.9].
5.2 Setting configuration variables

From a high-level perspective, a $K$ definition is executed by reducing an initial configuration in the Maude module containing the compiled definition. Creating this initial configuration is an important job of the $krun$ tool.

The configuration declaration in a definition specifies initial values for all cells. A cell may be initialized to a value like 42, List, or $x \mapsto 12$, or to a variable like $\$STATE:Map$ or $\$PGM:K$, where the string following the "$" is the name of the variable and the string following the ";" is its sort. Variables in the configuration can be set at runtime via command-line options passed to $krun$. The syntax for this is $--VARNAME="VALUE"$. For example, to initialize the variable $\$IN:List$ with a list containing "3", "1", "4", $krun$ could be invoked as follows:

$krun \ldots --IN="ListItem(3) ListItem(1) ListItem(4)" \ldots$

Certain configuration variables have a special meaning and are automatically initialized by $krun$. For example, the $\$PGM:K$ variable is automatically set to the $K$ term obtained from running $kast$ on the input program. Most often, the initial value of the $K$ cell is the variable $\$PGM:K$. All variables from the initial configuration not set by $krun$ are initialized with the empty value "." for their corresponding type.

5.3 Configuring $krun$

The $krun$ tool has many options to configure its execution. Options are $key \mapsto value$ pairs that may be set via command-line flags or configuration files. A list of available command-line flags can be displayed by typing $krun --help$. Options that require a string or a file path as a value translate into command-line flags naturally: $--main-module "foo$" is equivalent to $--main-module="foo"$, which is equivalent to $main-module \mapsto foo$. Options that require a boolean value are exposed as two command-line flags: $--color$ is equivalent to $color \mapsto true$ and $--no-color$ is equivalent to $color \mapsto false$. Configuration files are YAML files (typically ending with the .desk extension) where the YAML syntax "key: value" maps to an option naturally. The set of options $krun$ ultimately uses is a right-biased union (later options take precedence) of the following:

- The configuration file $\$K_BASE/tools/global-defaults.desk$
- The configuration file specified by $--desk-file$ or a configuration file with the .desk extension in the current directory
- All of the options (and option groups) specified on the command-line

In addition to specifying options, configuration files may declare option groups, which are a list of options aliased to a single name. To enable an option group, the name of the group is simply passed as a command-line flag to $krun$. The $--search$ flag is implemented as an option group which is declared in the global-defaults.desk configuration file.

5.4 Available options

This section describes the options available for configuring $krun$. Each description below includes the name of the option and the type of value the option requires.
**k-definition: file**
Sets the path to the K definition to execute. This option behaves similarly to the kast option with the same name.

**main-module: string**
Sets the main K module (see Section 2.1.2). If not set explicitly, the value of this option will be inferred from the k-definition option as follows:

\[
value(main-module) = \text{uppercase}(\text{basename}[value(k-definition), ".k"]).
\]

**syntax-module: string**
Sets the syntax module (see Section 2.2.3). If not set explicitly, the value of this option will be inferred from the k-definition option as follows:

\[
value(syntax-module) = \text{uppercase}(\text{basename}[value(k-definition), ".k"])-\text{SYNTAX}.
\]

**parser: string**
This option specifies the parser used to parse the input program. By default, this option is set to kast, meaning the kast tool is used to parse programs (see Section 2.2.3). Another parser can be specified by setting this option to a command (perhaps including command-line options) that accepts a filename as its last argument and prints a K AST to standard out.

**color: boolean**
The krun tool pretty-prints final K configurations using ANSI color sequences. This option enables/disables color in krun’s output. This is useful when piping the output to another tool or file.

**parens: boolean**
If set to false, krun will not add any additional parentheses to its output, meaning that resulting syntax may be ambiguous. If set to true, krun will add parentheses so that the result syntax is unambiguous. krun attempts to avoid adding unnecessary parentheses, but typically the result will have excess parentheses when this option is set to true.

**statistics: boolean**
Enables/disables printing the rewrite statistics generated by Maude.

### 6 Conclusions

The K tool demonstrates that a formal specification language for programming languages can be simultaneously simple, expressive, analyzable, and scalable. Executability, modularity, and state-space search are key features of the K tool, allowing to easily experiment with language design and changes by means of testing and even exhaustive non-deterministic behavior exploration.

In this paper, we have shown a subset of the features offered by the K tool. To learn more about it, or to start developing your own programming language, please download the K tool from our open source project page [8,9].

### References


