Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 304 (2014) 57–80

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

The $\mathbb{K}$ Primer (version 3.3)

http://k-framework.org

Traian Florin Șerbănuță

University of Bucharest
traian.serbanuta@fmi.unibuc.ro

Andrei Arusoaie

University Alexandru Ioan Cuza of Iași
andrei.arusoaie@info.uaic.ro

David Lazar

Massachusetts Institute of Technology
lazard@mit.edu

Chucky Ellison

University of Illinois at Urbana-Champaign
celliso2@illinois.edu

Dorel Lucanu

University Alexandru Ioan Cuza of Iași
dlucanu@info.uaic.ro

Grigore Roșu

University of Illinois at Urbana-Champaign
grosu@illinois.edu

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

http://dx.doi.org/10.1016/j.entcs.2014.05.003

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, but also because of the poor scalability of many frameworks, both in terms of modularity at a definitional level and in terms of simulation, execution and/or analysis time.

To address this difficulty in writing language semantics, the $\mathbb{K}$ framework [28] 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 $\mathbb{K}$ tool [13] version 3.2, an implementation of the $\mathbb{K}$ 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 behavior exploration.

$\mathbb{K}$ definitions are written in machine-readable ASCII, which the $\mathbb{K}$ tool accepts as input. For execution and analysis purposes, the definitions are translated into Maude [3] rewrite theories. For visualization and documentation purposes, definitions are typeset into their LaTeX mathematical representation. Fig. 4 gives the $\mathbb{K}$ definition (both ASCII and mathematical representations) of a simple calculator language with input and output. This language is described in detail in Section 2.

Besides didactic and prototypical languages (such as λ-calculus, System F, and Agents), the $\mathbb{K}$ tool has been used to formalize C [5], Python [9], Scheme [21] and OCL [1,29]; additionally, definitions of Java 7, Haskell and JavaScript are underway. With respect to analysis tools, the $\mathbb{K}$ tool has been used to define type checkers and type inferencers [6], and is currently being used in the development of a new program verification tool using Hoare-like assertions based on matching logic [24,25], in a model checking tool [2] based on predicate abstraction, and in researching runtime verification techniques [26,30]. All of these definitions and analysis tools can be found on the $\mathbb{K}$ tool website [12].

2 Writing the first $\mathbb{K}$ definition

In this section we guide the reader through the process of writing a simple language definition using the $\mathbb{K}$ tool. Upon completing this section, the reader should be able to easily write and typeset a definition like that of the EXP language presented

in Fig. 4, and to compile it and use it as an interpreter. As some of the more advanced features of $\mathbb{K}$ are not used by the definition in Fig. 4, we will provide several extensions from it, each exhibiting a feature of $\mathbb{K}$. The snippets of code presented in this paper highlight the keywords recognized by the $\mathbb{K}$ tool and typesets predefined syntactic categories using italics; the $\mathbb{K}$ tool provides similar syntax highlighting capabilities for several popular text editors, including vi and Emacs.

2.1 Basic ingredients

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

2.1.1 Modules

The $\mathbb{K}$ tool provides modules for grouping language features (syntax, evaluation strategies, configuration, and execution rules). A module is defined by the syntax:

```
module <NAME>
  ...
endmodule
```

where `<NAME>` is a name identifying the 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; however, it is generally considered good practice to isolate the (program) syntax from the semantics using two modules: `<NAME>` and `<NAME>-SYNTAX`. Separating the syntax from the rest of the definition minimizes program parsing ambiguities. To import modules, one needs to add at least one `imports <NAME>` directive after the header of a module. Multiple modules can be imported using the same `imports` directive by summing their names with the '`+`' symbol.

Modules need to be explicitly imported within other modules before referring to (parts of) them. For example, the `EXP-SYNTAX` module needs to be imported by the `EXP` module before using the syntax declared in it.

```
module EXP-SYNTAX
endmodule

module EXP
  imports EXP-SYNTAX
endmodule
```

2.1.2 Compiling definitions

$\mathbb{K}$ definitions are usually stored into files with extension "`.k`". Assume a file `exp.k` containing modules `EXP-SYNTAX` and `EXP` defined above. To compile this definition (and, indirectly, check its validity), one can execute the following command:

```
$ kompile exp
```

`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*⟩. Upon successful completion, `kompile` prints nothing. One can change these defaults with `kompile` options; type `kompile --help` for details.

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 [18] capabilities via LaTeX-specific comments (introduced by '`//@`' and '`/*@`'), which can be used to generate ready-to-publish definitions. We will explain these later in Section 5.

2.2 Language syntax

Any formal semantics of a language requires first a formal syntax. Although the $\mathbb{K}$ parser can parse many non-trivial languages, it is not (currently) 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 (using the `-parser` option of `krun`).

2.2.1 User-defined syntax

Syntax in $\mathbb{K}$ is defined using a variant of the familiar BNF notation, with terminals enclosed in quotes and non-terminals starting with capital letters. For example, the syntax declaration:

```
syntax Exp ::= Int
             | Exp "+" Exp          [seqstrict]  // addition
             | Exp "*" Exp          [seqstrict]  // multiplication
             | Exp "/" Exp          [seqstrict]  // division
             | "read"                            // read integer from console
             | "print" "(" Exp ")" [strict]      // writing evaluation results to console
             | "(" Exp ")"          [bracket]
```

defines a syntactic category `Exp`, containing the built-in integers and three basic arithmetic operations on expressions, as well as I/O operations. Each production can have a space-separated list of attributes which can be specified in square brackets at the end of the production. For example, the attribute `bracket` used in the last production specifies that the brackets are only used for grouping reasons—they will be omitted from the abstract syntax tree.

`seqstrict` and `strict` are attributes bearing semantical information, being used to define the evaluation strategy of the corresponding construct. For example, declaring addition `seqstrict` simply means that we would like arguments of plus to be evaluated from left to right before evaluating the addition itself. Other uses of attributes are to help the parser with operator precedences and grouping, or to instruct the PDF

generator how to display the various constructs. We will discuss these in more detail later in the paper.

Unless otherwise mentioned, all subsequent `syntax` declarations should be added inside the `EXP-SYNTAX` module.

2.2.2 Built-ins

In addition to the predefined semantic categories (computations, lists, sets, bags, maps), the $\mathbb{K}$ tool provides a number of built-in syntactic categories/datatypes, and semantic operations for them. The dynamically evolving list of these built in operations can be found in the files from the `include/builtins` directory from the distribution. Among the currently supported built-ins are Booleans (`Bool`), unbounded integers (`Int`), floats (`Float`), strings (`String`), and identifiers (`Id`).

Operations on built-in categories (such as those giving semantics for the basic arithmetic operators) are postfixed with the name of the category: for example, '`5 +Int 7`' applies the built-in integer addition on integers `5` and `7`, while '`5 <=Int 7`' produces the `Bool true` constant.

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`. Suppose also there exists an EXP program `2avg.exp`:

```
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 $\mathbb{K}$ abstract syntax tree. By default, `kast` requires to be run from the directory containing the $\mathbb{K}$ definition file, say ⟨*name*⟩`.k`, and the compiled definition; please use the `--help` option to see more details on how to call `kast` from other places.

Moreover, all tokens not declared in the user syntax are assumed to be identifiers.

The $\mathbb{K}$ AST is simply a tree having as nodes ($\mathbb{K}$) labels, either associated to syntactic productions (e.g., '`_+_`'), or injections of the predefined datatypes' constants as tokens.

2.3 Language semantics

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

2.3.1 Evaluation strategies: strictness

Evaluation strategies serve as a link between syntax and semantics, by specifying how 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, which is specified by the '`strict(1)`' attribute:

```
syntax Exp ::= Exp "?" Exp ":" Exp [strict(1)]
```

Although the order in which summands are evaluated might not matter, it matters crucially for a sequential composition operator (e.g., '`_;_`'); `seqstrict` requires a left-to-right order of evaluation.

```
syntax Exp ::= Exp ";" Exp [seqstrict]
```

Since rules for specifying evaluation strategies are tedious to write in most definitional formalisms, the $\mathbb{K}$ 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 non-terminal positions in the syntax declaration.

The $\mathbb{K}$ 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)`' above 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 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 nondeterminism. 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 `seqstrict` annotation for the '`+`' constructor says that we want to evaluate all arguments of the '`+`' operator *from left to right* before giving its semantics. In contrast, a `strict` annotation would also specify that both subexpressions must be evaluated, but does not constrain the evaluation strategy.

The $\mathbb{K}$ tool distinguishes a category of terms, `KResult`, which is used to determine which terms are values, or *results*. For example, the `syntax` declaration below specifies that integers are values in the EXP language.

```
syntax KResult ::= Int
```

Unlike syntactic productions declaring language constructs, the place of this declaration is in the EXP module, as it semantically defines integers as results.

More advanced details about strictness (including the more generalized notion of evaluation contexts) as well as the use of the $\mathbb{K}$ tool to explore the nondeterminism associated with strictness will be discussed later in the paper.

2.3.2 Computations

The sequencing of evaluation is made possible in $\mathbb{K}$ by *computation structures*. Computation structures, called "computations" for short, extend the abstract syntax of a language with a list structure using the separator $\curvearrowright$ (read "followed by" or "and

then", and written `~>` in ASCII). $\mathbb{K}$ 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 `syntax` keyword. The *KResult* sort described in the previous section is a subsort of K.

The intuition for computation structures of the form $t_1 \curvearrowright t_2 \curvearrowright \cdots \curvearrowright t_n$ 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.

The rules generated from strictness annotations are the main users of this sequencing constructor; they "heat" the computation by sequencing the evaluation of the arguments before the evaluation of the construct itself, and then "cool" it back once a value was obtained. For example, the heating rule for the conditional construct would look something like:

```
E1:Exp ? E2:Exp : E3:Exp => E1 ~> HOLE ? E2 : E3
```

with the side condition the `E1` is not a result. The "cool" rule would be the opposite:

```
E1:Exp ~> HOLE ? E2:Exp : E3:Exp => E1 ? E2 : E3
```

with the side condition that `E1` is a result this time.

2.3.3 Configurations, initial configuration

In $\mathbb{K}$, the state of a running program/system is represented by a *configuration*. Configurations are structured as nested, labeled cells containing various computation-based data structures. Within the $\mathbb{K}$ 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. For example, the configuration of EXP is:

```
configuration
  <k> $PGM:K </k>
  <streams>
    <in stream="stdin"> .List </in>
    <out stream="stdout"> .List </out>
  </streams>
```

The lines above describe both the initial configuration and the general structure of a configuration for `EXP` and should be included in the EXP module (usually in the first few lines). The configuration declaration is introduced by the `configuration` keyword, and consists of a cell labeled `k` which is meant to hold the running program (denoted here by the variable `$PGM` of type `K`), and a cell `streams` holding the `in` and `out` cells, which model an executing program's input/output streams (abstracted as lists). The `in` and `out` cells in the configuration declaration contain the XML attribute `stream` which is used for enhancing the interpreter generated from the definition with interactive I/O, with current possible values `stdin` and `stdout` (see Section 2.4). Variables in the initial configuration, e.g., '`$PGM:K`', are place-holders. They are meant to be initialized at the beginning of program execution (see Section 2.4).

The only types of cell contents currently allowed by the $\mathbb{K}$ tool are computations and lists/bags/sets/maps of computations with their sorts being `List`, `Bag`, `Set`, and `Map`, respectively. The elements of these structures are obtained by injecting computations

(of sort K) into these sorts through the constructors `ListItem`, `BagItem`, `SetItem`, and $\mapsto$ (written in ASCII as '`|->`'), respectively. As the K sort encompasses all built-in datatypes and user-defined syntax, this allows for items like '`ListItem(I:Int)`', '`SetItem(5)`', and '`x |-> 5`' to be written. Multiple items are space-separated, for example: '`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 in cell in the configuration.

The configuration declaration serves as the backbone for the process of *configuration abstraction* (later described in Section 4.1) 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

$\mathbb{K}$ *rules* describe how a running configuration evolves by advancing the computation and potentially altering the state/environment. In the $\mathbb{K}$ tool, semantic rules are introduced by the `rule` keyword. 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. The rules below are to be introduced in the EXP module.

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. For example, the semantics of addition and multiplication are completed by:

```
rule I1:Int + I2:Int => I1 +Int I2
rule I1:Int * I2:Int => I1 *Int I2
```

As strictness is assumed to take care of the evaluation strategy, $\mathbb{K}$ rules are written with the assumption that the strict positions have already been evaluated (e.g., '`+`' and '`*`' were declared to be strict in all arguments).

Furthermore, all rules can have side conditions which are introduced by the `requires` keyword, and are expected to be Boolean predicates. The following rule:

```
rule I1:Int / I2:Int => I1 /Int I2 requires I2 =/=Int 0
```

specifies that division should only be attempted when the denominator is non-zero.

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 a variable named `X` of sort `K`. The $\mathbb{K}$ tool does not require a variable to be typed everywhere it is used, but it must be typed at least once per rule. There are attempts to infer types automatically, but we here assume that variables are explicitly typed.

Anonymous variables are represented in the $\mathbb{K}$ tool by the underscore '`_`' symbol. They allow omitting the name of a variable when that is not used elsewhere in a rule. For example, here are the rules for evaluating the conditional expression:

```
rule 0 ? _ : E:Exp => E
rule I:Int ? E:Exp : _ => E requires I =/=Int 0
```

In the first rule, if the condition evaluates to 0, then the entire expression evaluates to the third argument, while the second (matched by the anonymous variable'_') is discarded. Otherwise, if the condition evaluates to something other than 0, the conditional expression evaluates to the second argument, the third being discarded.

In $\mathbb{K}$, rewriting is extended using what we call *local rewriting*. By pushing the rewrite actions 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 `=>`), 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 $\Rightarrow$), as shown by the following two rules for performing I/O:

```
rule <k> print(I:Int) => I ...</k>
     <out>... . => ListItem(I) </out>
```

```
rule <k> read => I:Int ...</k>
     <in> ListItem(I) => . ...</in>
```

For example, the `print` rule 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). Similarly, the `read` rule replaces one element in the input cell by the empty list (i.e., deletes it from that list) and uses its value as a replacement for `read` in the computation cell.

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 `read` and `print` rules mentioned above to omit the streams cell, as it can be easily inferred. More details about configuration abstraction are discussed in Section 4.1.

As an additional notation 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) to the left or to the right end of a cell. With this convention, we now have the full semantics for the `print` and `read` rules:

print 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;

read if `read` is found at the beginning of the computation cell, then the first element in the input list is removed, and its value is used as a replacement for `read`.

Note that the rules corresponding to strictness annotations are used both to ensure that a `print` or `read` expression would eventually reach the top of the computation if it is in an evaluation position, and also that once they are evaluated, their values will be plugged back into their corresponding context.

The above techniques make $\mathbb{K}$ rules simple and modular. By keeping rules compact and redundancy low, it is less likely that a rule will need to be changed as the configuration is changed or new constructs are added to the language.

2.4 Executing programs with `krun`

Once the $\mathbb{K}$ definition of a language is written in the $\mathbb{K}$ tool and it is compiled successfully (using the `kompile` command), the `krun` command can be used to execute/interpret programs written in the defined language.

```
$ cat p1.exp
(3 * (4 + 6)) / 2

$ krun p1.exp

<k>
  15
</k>
<streams>
  <in>
    .
  </in>
  <out>
    .
  </out>
</streams>
```

```
$ cat 2avg.exp
print((read + read) / 2)

$ krun 2avg.exp

5
7
6
<k>
  6
</k>
<streams>
  <in>
    .
  </in>
  <out>
    .
  </out>
</streams>
```

```
$ cat p2.exp
print(100 / read)

$ echo "0" | krun p2.exp

<k>
  100 / 0 ~> print HOLE
</k>
<streams>
  <in>
    .
  </in>
  <out>
    .
  </out>
</streams>
```

Fig. 1. Three EXP programs and their interactive executions using `krun`

Fig. 1 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 $\mathbb{K}$ 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 `p1.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 `p2.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.

3 More advanced $\mathbb{K}$ definitional features

In this section we present several extensions to the EXP language and we use them to exemplify several features not addressed in the previous section.

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.

```
1  require "../exp.k"
2  require "modules/substitution.k"

4  module EXP-LAMBDA-SYNTAX
5    imports EXP-SYNTAX

7    syntax Val ::= Int
8                 | "lambda" Id "." Exp [binder]
9    syntax Exp ::= Val
10                | Exp Exp [seqstrict] //application
11                | "mu" Id "." Exp [binder]
12 endmodule

14 module EXP-LAMBDA
15   imports EXP + EXP-LAMBDA-SYNTAX
16   imports SUBSTITUTION

18   syntax KResult ::= Val

20   rule (lambda X:Id.E:Exp) V:Val
21        => E[V / X]

23   rule mu X:Id.E:Exp
24        => E[mu X:Id.E:Exp / X]
25 endmodule
```

Fig. 2. `exp-lambda.k`: Extending EXP with functional features

3.1.1 Splitting definitions among files

Larger $\mathbb{K}$ definitions are usually spread in multiple files, each potentially containing multiple modules. As a file-equivalent of the `import` command on modules, a file can be included into another file using the `require` directive. `require` will first look up the path in the current directory; if not found, it will also look it up in the `include` directory from the $\mathbb{K}$ distribution. The tree of `require` dependencies is followed recursively. If a file was already included through another `require` command, it will not be included twice.

3.1.2 Binders and substitution

Both `lambda` and `mu` are binders, binding their first argument in the second argument. We use the built-in 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 Fig. 2. 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 $\mathbb{K}$ tool's `SUBSTITUTION` module, the semantics of function application and μ-unrolling are straightforward. The `SUBSTITUTION` module is completely defined in the $\mathbb{K}$ 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 / K ]`', 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. Moreover, since all rules which don't explicitly mention a cell are assumed to apply at the top of the computation cell, the evaluation strategy is also outermost (the β-substitution, lines 20–21 of Fig. 2). A special category `Val` is introduced to allow matching on both integers and λ-abstractions (lines 7–8 of Fig. 2) and computation results are extended to include all the values (line 18). Constraining rules not mentioning a cell to happen at the top of the computation

cell also helps with avoiding non-termination for μ-unrolling rule (lines 23–24), by only unrolling μ in an evaluation position.

$\mathbb{K}$ does not commit to substitution-based definitions only. Environment-based definitions are quite natural, too; in this setting, the environment is typically just another cell in the configuration.

3.1.3 Desugaring syntax

We can show how one might desugar syntax in the $\mathbb{K}$ tool by adding two popular functional constructs to our language: `let` and `letrec`.

```
syntax Exp ::= "let" Id "=" Exp "in" Exp
             | "letrec" Id Id "=" Exp "in" Exp
```

Instead of directly giving them semantics, we use "macro" capabilities to desugar them into λ and μ abstractions.

```
rule (let X:Id = E1:Exp in E2:Exp)
  => (lambda X.E2) E1 [macro]
```

```
rule (letrec F:Id X:Id = E1:Exp in E2:Exp)
  => (let F = mu F.lambda X.E1 in E2) [macro]
```

The `let` operator desugars into an application of a λ-abstraction, while `letrec` desugars into the `let` construct binding a μ-abstraction. Since they won't be part of the AST when giving the semantics, their corresponding productions don't need to be annotated as binders.

These macros are to be applied on programs at parse time; therefore both the syntax declaration and the macros themselves belong in the `EXP-LAMBDA-SYNTAX` module.

3.2 Imperative features

In this section we use the pretext of adding imperative features to our language to explain another set of advanced features of the $\mathbb{K}$ 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 an expression separator.

```
syntax Stmt ::= Exp ";" [strict]
              | Stmt Stmt
```

As we do not want statements to evaluate to values, we will not use strictness constraints to give their semantics. Instead, we give semantics for the expression statement and sequential composition with the following two rules:

```
rule V:Val ; => .
rule St1:Stmt St2:Stmt => St1 ~> St2
```

The fists rules discards the value for the expression statement; the second sequences statements as computations.

Again, we add syntax productions to EXP-SYNTAX and $\mathbb{K}$ rules to EXP.

3.2.2 Syntactic lists

$\mathbb{K}$ provides built-in support for generic syntactic lists: *List*{*Nonterminal*, *terminal*} stands for *terminal*-separated lists of zero or more *Nonterminal* elements. To instantiate and use the $\mathbb{K}$ built-in lists, you must alias each instance with a (typically fresh) non-terminal in your syntax. As an example, we can add variable declarations 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:

```
syntax Ids ::= List{Id, ","}
syntax Stmt ::= "var" Ids ";"
```

Thus, both '`var x, y, z;`' and '`var ;`' are valid declarations.

For semantic purposes, these lists are currently 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 two new cells to the configuration: env, to hold mappings from variables to locations, and store, to hold mappings from locations to values. The following two rules specify the semantics of variable declarations:

```
rule <k> var (X:Id,Xl:Ids => Xl) ; ...</k>
     <env> Rho:Map => Rho[N/X] </env>
     <store>... . => N |-> 0 ...</store>
   requires fresh(N:Int) .
rule var .Ids ; => . [structural]
```

The first rule declares the first variable in the list by adding a mapping from a fresh location `I`, specified by the `fresh` predicate in the side condition, to 0 in the store cell, and by updating the mapping of the name of the variable in the env cell to point to that location. The second rule terminates the variable declaration process when there are no variables left to declare. '`.Ids`' is used in the semantics to refer to the empty list. In general, a dotted-non-terminal represents the empty collection for that non-terminal. 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 4.3 for more details about the types of rules).

4 Concurrency and nondeterminism

This section shows how one can define nondeterministic features both related to concurrency and to the under-specification (e.g., order of evaluation). Moreover, mechanisms to control the state explosion due to nondeterminism are also presented.

4.1 Configuration abstraction

Assume we would like to extend EXP with concurrency features. 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, using the `multiplicity` XML attribute to indicate that the thread cell can occur multiple times. `multiplicity` can be used to specify how many copies of a cell are allowed: either 0 or 1 ('?'), 0 or more ('*'), or one or more ('+'). Upon this transformation, the configuration changes as follows:

```
configuration
  <thread multiplicity="*">
    <k> $PGM:K </k>
    <env> .Map </env>
  </thread>
  <store> .Map </store>
```

```
<streams>
  <in stream="stdin"> .List </in>
  <out stream="stdout"> .List </out>
</streams>
```

A possible syntax and semantics for thread spawning is:

```
syntax Stmt ::= "spawn" "{" Stmt "}"

rule <k> spawn { St:Stmt } => . ...</k> <env> Rho:Map </env>
     (. => <thread>... <k> St </k> <env> Rho </env> ...</thread>)
```

Changes in the configuration are quite frequent in practice, typically needed in order to accommodate new language features. $\mathbb{K}$'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 $\mathbb{K}$ 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 $\mathbb{K}$ 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 $\mathbb{K}$ overview papers [27,28].

Multiplicity information is important in the configuration abstraction process, as it tells the $\mathbb{K}$ tool how to complete rules like that for a `rendezvous` construct:

```
syntax Exp ::= "rendezvous" Exp [strict]

rule <k> rendezvous I:Int => I ...</k>
```

```
<k> rendezvous I => I ...</k>
```

As the k cell does not have the `multiplicity` set to '*' and thread does, 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:

```
syntax Exp ::= Id

rule <k> X:Id => I ...</k>
     <env>... X |-> L:Int ...</env>
     <store>... L |-> I:Int ...</store>
```

```
syntax Exp ::= Id = Exp [strict(2)]

rule <k> X:Id = I:Int => I ...</k>
     <env>... X |-> L:Int ...</env>
     <store>... L |-> (_ => I) ...</store>
```

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 future implementations of the $\mathbb{K}$ tool.

4.2 Advanced strictness and evaluation contexts

Suppose that in our language we would like to be able to allow threads to share any variable, not only those shared by the thread spawning semantics, but also some created afterwards. Assuming the two threads already share one variable, we can achieve this goal using references. The syntax and semantics for adding references are:

```
syntax Exp ::= "&" Id

rule <k> & X:Id => L ...</k>
     <env>... X |-> L:Int ...</env>
```

```
syntax Exp ::= "*" Exp [strict]

rule <k> * L:Int => I ...</k>
     <store>... L |-> I:Int ...</store>
```

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:

```
syntax Exp ::= Exp "=" Exp [strict(2)]

rule <k> * L:Int = I:Int ...</k>
     <store>... L |-> (_ => I) ...</store>
```

However, this rule is not sufficient by itself, as it assumes the argument of '`*`' 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 ('`*`' and '`_=_`') instead of one.

$\mathbb{K}$ `context`s 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. $\mathbb{K}$ contexts are similar to evaluation contexts [8,16]. For example, here is the context declaration needed above:

```
context * HOLE = _
```

where `HOLE` designates the position to be evaluated first. As contexts are essentially semantics, we add them to the EXP module.

The context can be conditional, allowing side conditions on any of the variables of the term, including the `HOLE`. For example,

```
context _ / HOLE
context HOLE / I:Int requires I =/=Int 0
```

specifies an evaluation strategy for division where the denominator must be evaluated first, and the numerator is evaluated only if the value for the denominator is non-zero.

The hole itself can be constrained. For example, in an object-oriented language,

```
context HOLE . X:Id requires HOLE =/=K super
```

specifies that the expression in the left side of a selection construct should be evaluated (to an object closure) only if the expression is not "super"; this to allow static lookup when using super, for example.

4.3 Controlling nondeterminism

There are two main sources of nondeterminism in programming languages—concurrency and order of evaluation. In this section we explain how the $\mathbb{K}$ tool can be used to explore both kinds of nondeterministic behavior.

4.3.1 Transitions

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.

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 usually behaviorally equivalent. For example, the fact that a thread computes a step $2+5 \Rightarrow 7$ 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, allowing one to declare rules bearing certain attributes as "transitions". The rules bearing attributes tagged as transitions are the only ones considered as transitions when exploring a program's transition system.

For example, suppose we would like to explore the nondeterminism generated by the semantics of `print` for the EXP program `p5.exp` containing the program

```
spawn (print 1);
spawn (print 2);
print 3
```

To do that, we can annotate the `print` rule with the `print` tag (to be able to refer to it), and recompile with `kompile --transition "print"`.

To contain the output and better make our point, we consider the simpler definition of EXP as a multithreaded calculator language presented in Fig. 4; nevertheless the technique presented extends to more complex languages. Fig. 3 displays all behaviors observed using the `--search` option of the `krun` tool on `p5.exp`.

4.3.2 Nondeterministic strictness

Although good in theory, the unrestricted use of heating/cooling rules may create an immense, often unfeasible large space of possibilities to analyze. Therefore, for performance reasons, the $\mathbb{K}$ tool chooses a default arbitrary, but fixed, order to evaluate the arguments of a strict language construct. Specifically, similarly to refocusing semantics [4], once a path to a redex is chosen, no other path is chosen until all redexes on that path are evaluated. This has the side effect of potentially losing behaviors due to missed interleavings.

```
$ krun p5.exp --search
Search results:

Solution 1:     Solution 2:     Solution 3:     Solution 4:     Solution 5:     Solution 6:

<k>             <k>             <k>             <k>             <k>             <k>
  1               1               1               1               1               1
</k>            </k>            </k>            </k>            </k>            </k>
<k>             <k>             <k>             <k>             <k>             <k>
  2               2               2               2               2               2
</k>            </k>            </k>            </k>            </k>            </k>
<k>             <k>             <k>             <k>             <k>             <k>
  3               3               3               3               3               3
</k>            </k>            </k>            </k>            </k>            </k>
<streams>       <streams>       <streams>       <streams>       <streams>       <streams>
  <in>            <in>            <in>            <in>            <in>            <in>
    .               .               .               .               .               .
  </in>           </in>           </in>           </in>           </in>           </in>
  <out>           <out>           <out>           <out>           <out>           <out>
    "132"           "123"           "312"           "321"           "231"           "213"
  </out>          </out>          </out>          </out>          </out>          </out>
</streams>      </streams>      </streams>      </streams>      </streams>      </streams>
```

Fig. 3. Exploring the executions of a program with `krun --search`

To restore these missing interleavings, the $\mathbb{K}$ tool offers a `superheat`/`supercool`ing 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 $\mathbb{K}$ tool. More precisely, productions whose attributes are specified as `superheat` suring compilation is used to tell the $\mathbb{K}$ tool that we want to exhaustively explore all the nondeterministic evaluation choices for the strictness of the corresponding language construct. Similarly, whenever a rule tagged with an attribute specified as `supercool` during compilation is applied, the $\mathbb{K}$ tool will reset the current context, restarting the search for a redex. Upon this step, the rules defining the descent towards the next redex have the possibility to pick another evaluation order.

This way, we can think of `superheat`ing/`supercool`ing as marking fragments of computation in which exhaustive analysis of the evaluation order is performed. Used carefully, this mechanism allows us to explore more nondeterministic behaviors of a program, with minimal loss of efficiency.

For example, let us name the addition operator in EXP:

```
syntax Exp ::= Exp "+" Exp [plus, strict]
```

After recompiling the $\mathbb{K}$ definition with `kompile -superheat "plus"`, 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 nondeterministic choice. That is, once an argument of a construct was chosen to be evaluated, it will be evaluated completely. In order to observe full nondeterminism, the rules whose behavior we consider observable, e.g., the output rule, must be specified as supercooling:

```
rule <k> print I:Int => I ...</k>
     <out>... . => ListItem(I) </out> [output]
```

then, after recompiling using `kompile --superheat "plus" --supercool "output"`, `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 compilation options is sound with respect to the semantics of the definition in the sense that they allow the $\mathbb{K}$ tool to partially explore the transition system defined by the semantics.

5 Literate programming

$\mathbb{K}$ definitions adhere to the literate programming paradigm [18] by allowing an intermixture 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 "literate" comments, which are used in generating `.pdf`s by using the "-pdf" option of the `kompile` tool. These special comments are allowed to contain any LaTeX command.

Single line LaTeX comments are introduced by '`//@`':

```
//@ Arithmetic Syntax
```

Multi-line LaTeX comments are also declared using the same pattern:

```
/*@ This is a multi-line 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 `latex` attribute of the production:

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

instructs that a term '`E <= E`' should be displayed as '$E \leq E'$' everywhere when typesetting the definition.

In the LaTeX notation, in-place replacement is displayed on the vertical axis, using a horizontal line instead of the rewrite symbol. Moreover, cells are displayed as labeled containers (or bubbles) holding their content, perhaps having their side edges looking jagged or torn, to specify that some content was abstracted away.

To help with better distinguishing cells and their contents, the XML attribute `color` can be used in the configuration to specify the color of each cell (using the standard colors from the LaTeX package `xcolor`).

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 '`<br/>`' element. The '`<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 `large` attribute.

The $\mathbb{K}$ tool can generate LaTeX from definitions, for documentation purposes. For example, the graphical depiction of the ASCII definition EXP presented in Fig. 4 was obtained with the command:

```
$ kompile exp --backend pdf
Generated exp.pdf which contains the language definition.
```

```
1  module EXP-SYNTAX

3    //@ Arithmetic Syntax
4    syntax Exp ::= Int
5                 | "(" Exp ")" [bracket]
6                 | Exp "+" Exp [seqstrict] //addition
7                 | Exp "*" Exp [seqstrict] //multiplication
8                 | Exp "/" Exp [seqstrict] //division
9                 | Exp "?" Exp ":" Exp [strict(1)]
10                | Exp ";" Exp [seqstrict]

12   //@ Input / Output Syntax

14   syntax Exp ::= "read"
15                | "print" "(" Exp ")" [strict]


18   //@ Concurrency features
19   syntax Exp ::= "spawn" Exp
20                | "rendezvous" Exp [strict]
21 end module

23 module EXP
24   imports EXP-SYNTAX
25   syntax KResult ::= Int
26   configuration
27     <k color="green" multiplicity="*"> $PGM:K </k>
28     <streams>
29       <in color="magenta" stream="stdin"> .List </in>
30       <out color="Fuchsia" stream="stdout"> .List </out>
31     </streams>

33  //@ Arithmetic Semantics

35   rule I1:Int + I2:Int
36     => I1 +Int I2

38   rule I1:Int * I2:Int
39     => I1 *Int I2

41   rule I1:Int / I2:Int => I1 /Int I2
42                                 requires I2 =/=Int 0

44   rule 0 ? _ : E:Exp => E

46   rule I:Int ? E:Exp : _ => E  requires I =/=Int 0

48   rule _:Int ; I2:Int => I2


51   //@ Input / Output Semantics


54   rule <k> read => I:Int ...</k>
55        <in> ListItem(I) => . ...</in>



59   rule <k> print(I:Int) => I ...</k>
60        <out>... . => ListItem(I) </out>



64   //@ Concurrency Semantics


67   rule <k> spawn E => 0 ...</k>
68        (. => <k> E </k>)



73   rule <k> rendezvous I => I ...</k>
74        <k> rendezvous I => I ...</k>

77 end module
```

MODULE EXP-SYNTAX

Arithmetic Syntax

SYNTAX *Exp* ::= *Int*
| (*Exp*) [bracket]
| *Exp* + *Exp* [strict]
| *Exp* ∗ *Exp* [strict]
| *Exp* / *Exp* [strict]
| *Exp* ? *Exp* : *Exp* [strict(1)]
| *Exp* ; *Exp* [seqstrict]

Input / Output Syntax

SYNTAX *Exp* ::= `read`
| `print` (*Exp*) [strict]

Concurrency features

SYNTAX *Exp* ::= `spawn` *Exp*
| `rendezvous` *Exp* [strict]

END MODULE

MODULE EXP

SYNTAX *KResult* ::= *Int*

CONFIGURATION:

Arithmetic Semantics

RULE $\frac{I1 + I2}{I1 +_{Int} I2}$

RULE $\frac{I1 * I2}{I1 *_{Int} I2}$

RULE $\frac{I1 \,/\, I2}{I1 \div_{Int} I2}$ requires $I2 =/=_{Int} 0$

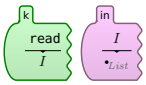
RULE $\frac{0\ ?\ —\ :\ E}{E}$

RULE $\frac{I\ ?\ E\ :\ —}{E}$ requires $I =/=_{Int} 0$

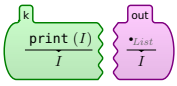
RULE $\frac{—\ ;\ I2}{I2}$

Input / Output Semantics

RULE

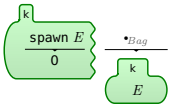


RULE

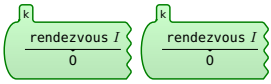


Concurrency Semantics

RULE



RULE



END MODULE

Fig. 4. $\mathbb{K}$ definition of a calculator language with I/O (left: ASCII source; right: generated LaTeX)

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:

```
/*!
\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`.

6 Related Work

The theoretical foundations of $\mathbb{K}$ have been laid out in [28]. A more up-to-date overview of the $\mathbb{K}$ technique is presented in the current volume [27].

There have been several reports on previous prototypes of the $\mathbb{K}$ tool [11,31]. Although they might provide historic insights on the evolution of the $\mathbb{K}$ tool, this report describes a substantially improved version of the $\mathbb{K}$ tool which resembles the previous only in the sense that they have the same goals.

We would like to mention several other platforms and tools to help the language researcher in experimenting with various designs and features which are closely related to our research. Rascal [10,17,32] and Spoofax [14,15] target developing IDEs for domain specific languages with Eclipse integration and code generation features. PLanCompS (Programming Language Components and Specifications) [22,23] focuses on modularity and on the development of a component-based framework for the design, specification and implementation of programming languages. Redex [7,19,20] focuses on specifying and debugging operational semantics and allows interactive exploration of term reductions and using randomized test generation.

7 Conclusions

The $\mathbb{K}$ 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 $\mathbb{K}$ tool, allowing to easily experiment with language design and changes by means of testing and even exhaustive nondeterministic behavior exploration.

In this paper, we have shown a subset of the features offered by the $\mathbb{K}$ tool. To learn more about it, or to start developing your own programming language, please download the $\mathbb{K}$ tool from our open source project page [12,13].

Acknowledgment

The work presented in this paper was supported in part by the (Romanian) DAK project Contract 161/15.06.2010, NSF grant CCF-1218605, the NSA grant H98230-10-C-0294, the DARPA HACMS program as SRI subcontract 19-000222, and the

Rockwell Collins contract 4504813093. We would like to also thank the anonymous reviewers for their in-depth comments and suggestions.

References

[1] Arusoaie, A., D. Lucanu and V. Rusu, *Towards a K semantics for OCL*, this volume.

[2] Asavoae, I. M. and M. Asavoae, *Collecting semantics under predicate abstraction in the K framework*, in: *WRLA*, LNCS **6381**, 2010, pp. 123–139.

[3] Clavel, M., F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet and C. Talcott, "All About Maude, A High-Performance Logical Framework," LNCS **4350**, Springer, 2007.

[4] Danvy, O. and L. R. Nielsen, *Refocusing in reduction semantics*, Technical Report RS-04-26, BRICS, DAIMI, Department of Computer Science, University of Aarhus (2004).

[5] Ellison, C. and G. Roşu, *An executable formal semantics of C with applications*, in: *POPL* (2012), pp. 533–544.

[6] Ellison, C., T. F. Şerbănuţă and G. Roşu, *A rewriting logic approach to type inference*, in: *WADT'08*, LNCS **5486**, 2009, pp. 135–151.

[7] Felleisen, M., R. B. Findler and M. Flatt, "Semantics Engineering with PLT Redex," MIT Press, 2009.

[8] Felleisen, M. and R. Hieb, *The revised report on the syntactic theories of sequential control and state*, Theoretical Computer Science **103** (1992), pp. 235–271.

[9] Guth, D., "A Formal Semantics of Python 3.3," Master's thesis, University of Illinois at Urbana-Champaign (2013).

[10] Hills, M., P. Klint, T. van der Storm and J. J. Vinju, *A one-stop-shop for software evolution tool construction*, ERCIM News **2012** (2012).

[11] Hills, M., T. F. Şerbănuţă and G. Roşu, *A rewrite framework for language definitions and for generation of efficient interpreters*, in: *WRLA'06*, ENTCS **176 (4)**, 2007, pp. 215–231.

[12] *K semantic framework website* (2010).
URL https://k-framework.org

[13] *Source code for K semantic framework (version 3.3)* (2013).
URL https://github.com/kframework/k/tree/v3.3

[14] Kats, L. C. L., R. Vermaas and E. Visser, *Integrated language definition testing: enabling test-driven language development*, in: *OOPSLA*, 2011, pp. 139–154.

[15] Kats, L. C. L. and E. Visser, *The Spoofax language workbench: rules for declarative specification of languages and IDEs*, in: *OOPSLA*, 2010, pp. 444–463.

[16] Klein, C., J. A. McCarthy, S. Jaconette and R. B. Findler, *A semantics for context-sensitive reduction semantics*, in: *APLAS*, LNCS **7078** (2011), pp. 369–383.

[17] Klint, P., T. van der Storm and J. J. Vinju, *EASY meta-programming with Rascal*, in: *GTTSE*, LNCS **6491**, 2009, pp. 222–289.

[18] Knuth, D. E., *Literate programming*, The Computer Journal **27** (1984), pp. 97–111.

[19] Matthews, J. and R. B. Findler, *Operational semantics for multi-language programs*, TOPLAS **31** (2009).

[20] Matthews, J., R. B. Findler, M. Flatt and M. Felleisen, *A visual environment for developing context-sensitive term rewriting systems*, in: *RTA*, LNCS **3091**, 2004, pp. 301–311.

[21] Meredith, P., M. Hills and G. Roşu, *An executable rewriting logic semantics of K-Scheme*, in: *SCHEME*, 2007, pp. 91–103, Laval University TR DIUL-RT-0701.

[22] Mosses, P. D., *Component-based description of programming languages*, in: *VoCS*, 2008, pp. 275–286.

[23] Mosses, P. D. and M. J. New, *Implicit propagation in structural operational semantics*, in: *SOS'08*, ENTCS **229 (4)**, 2009, pp. 49–66.

[24] Roşu, G. and A. Ştefănescu, *Matching logic: A new program verification approach*, in: *ICSE (NIER Track)*, 2011, pp. 868–871.

[25] Roşu, G., C. Ellison and W. Schulte, *Matching logic: An alternative to Hoare/Floyd logic*, in: *AMAST'10*, LNCS **6486**, 2010, pp. 142–162.

[26] Roşu, G., W. Schulte and T. F. Şerbănuţă, *Runtime verification of C memory safety*, in: *RV'09*, LNCS **5779**, 2009, pp. 132–152.

[27] Roşu, G. and T. F. Şerbănuţă, *𝕂: Overview and progress report*, this volume.

[28] Roşu, G. and T. F. Şerbănuţă, *An overview of the K semantic framework*, Journal of Logic and Algebraic Programming **79** (2010), pp. 397–434.

[29] Rusu, V. and D. Lucanu, *A K-based formal framework for domain-specific modelling languages*, in: *FoVeOOS'11*, LNCS **7421**, 2012, pp. 214–231.

[30] Şerbănuţă, T. F., "A Rewriting Approach to Concurrent Programming Language Design and Semantics," Ph.D. thesis, University of Illinois at Urbana-Champaign (2010).

[31] Şerbănuţă, T. F. and G. Roşu, *K-Maude: A rewriting based tool for semantics of programming languages*, in: *WRLA*, LNCS **6381**, 2010, pp. 104–122.

[32] van den Bos, J., M. Hills, P. Klint, T. van der Storm and J. J. Vinju, *Rascal: From algebraic specification to meta-programming*, in: *AMMSE*, EPTCS **56**, 2011, pp. 15–32.

A Syntax of 𝕂 in 𝕂

The module below describes the syntax of 𝕂 in a format similar to the current 𝕂 syntax specification formalism.

This syntax cannot be currently used by the 𝕂 tool to parse 𝕂; it is only meant to serve as a documentation tool.

Syntax definitional features used below which are not currently supported by the 𝕂 tool:

- Optional arguments in productions, specified using ? (and brackets)
- Parameterized non-terminals (used for Variable)

Special syntactic categories

- Text for arbitrary text (parsing is assumed greedy)
- Path for specifying paths

```
module K-SYNTAX

  syntax File ::= List{FileItem,""}
  syntax FileItem ::= "require" Path
                    | Module
                    | LatexComment
                    | LatexPreamble

  syntax LatexPreamble ::= "/*!" Text "*/"
  syntax LatexComment ::= "//@" Text "\n"
                        | "/*@" Text "*/"

  syntax Module ::= "module" Imports Sentences "endmodule"

  syntax Imports ::= List{Import,""}
  syntax Import ::= "imports" ModuleIds

  syntax ModuleIds ::= List{Id,"+"}
  syntax Sentences ::= List{Sentence,""}
  syntax Sentence ::= "syntax" Sort "::=" Syntax
    | "context" Term ("requires" KTerm)? ("[" Attrs "]")?
```

```
  | "configuration" BagTerms
  | "rule" Term ("requires" KTerm)? ("[" Attrs "]")?
  | LatexComment

syntax Ids ::= List{Id,""}
syntax Sort ::= Id | "K" | "KLabel" | "KList" | "List" | "ListItem"
             | "Set" | "SetItem" | "Bag" | "BagItem" | "Map" | "MapItem"
             | "CellLabel" | "KResult"

syntax Sorts ::= List{Sort,""}

syntax Terminal ::= String
syntax CFGSymbol ::= Sort | Terminal
syntax ProdTerm ::= List{CGFSymbol,""}
syntax Production ::= ProdTerm ("[" Attrs "]")?

syntax Syntax ::= List{Production,"|"}

syntax Term ::= ListTerms | BagTerms | MapTerms | ListTerms | SetTerms
             | KLabelTerm | CellLabelTerm | KListTerms

syntax CellLabelTerm ::= Id | "k" | "T"
                  | CellLabelTerm "=>" CellLabelTerm
                  | Variable{CellLabel}

syntax CellTerm ::= "<" CellLabelTerm CellAttrs? ">" Term "</" CellLabelTerm ">"

syntax BagTerm ::= "BagItem" "(" KTerm ")"
                | CellTerm
                | BagTerms "=>" BagTerms
                | "." | ".Bag"
                | Variable{Bag} | Variable{BagItem}
syntax BagTerms ::= List{BagTerm,""}



syntax SetTerm ::= "SetItem" "(" KTerm ")"
                | SetTerms "=>" SetTerms
                | "." | ".Set"
                | Variable{Set} | Variable{SetItem}
syntax SetTerms ::= List{SetTerm,""}

syntax MapTerm ::= KTerm "|->" KTerm
                | MapTerms "=>" MapTerms
                | "." | ".Map"
                | Variable{Map} | Variable{MapItem}
syntax MapTerms ::= List{MapTerm,""}

syntax ListTerm ::= "ListItem" "(" KTerm ")"
                | ListTerms "=>" ListTerms
                | Variable{List} | Variable{ListItem}
                | "." | ".List"
syntax ListTerms ::= List{ListTerm,""}


syntax KLabelTerm ::= Id
                  | KLabelTerm "=>" KLabelTerm
                  | Variable{KLabel}

syntax KItemTerm ::= KLabelTerm "(" KListTerms ")"
             | KTerm "=>" KTerm
             | "HOLE"
             | ".K"
             | Variable{K}
            // | Int | Bool | Float | String
            // | User defined syntax

syntax KTerm ::= List{KItemTerm,"~>"}

syntax KListTerm ::= KTerm
                | ".KList"
                | KListTerms "=>" KListTerms
```

```
                  | Variable{KList}
  syntax KListTerms ::= List{KListTerm,",,"}


   syntax Attrs ::= List{Attr,","}

   syntax Attr ::= AttrKey "(" Text ")"
   syntax AttrKey ::= Id | "strict" | "seqstrict" | "hybrid" | "function"
                  | "structural" | "transition" | "superheat" | "supercool"
                  | "latex" | "large"
                  | "binder"

   syntax CellAttrs ::= List{CellAttr,""}
   syntax CellAttr ::= CellAttrKey "=" String

   syntax CellAttrKey ::= "multiplicity" | "color" | "stream"

   syntax Variable{##X##} ::= "_" | "_:##X##" | Id ":##X##"
endmodule
```