Executing Formal Semantics with the $\mathbb{K}$ Tool *

David Lazar[1], Andrei Arusoaie[2], Traian Florin Șerbănuță[1,2], Chucky Ellison[1], Radu Mereuta[2], Dorel Lucanu[2], and Grigore Roșu[1,2]

[1] University of Illinois at Urbana-Champaign
{lazar6,tserban2,celliso2,grosu}@illinois.edu
[2] University Alexandru Ioan Cuza of Iași
{andrei.arusoaie,traian.serbanuta,radu.mereuta,dlucanu}@info.uaic.ro

Abstract. This paper describes the $\mathbb{K}$ tool, a system for formally defining programming languages. Formal definitions created using the $\mathbb{K}$ tool automatically yield an interpreter for the language, as well as program analysis tools such as a state-space explorer. The modularity of $\mathbb{K}$ and the design of the tool allow one semantics to be used for several applications.

1 Introduction

Programming languages are the key link between computers and the software that runs on them. While syntax is typically formally defined for almost any programming language, semantics is most often given in natural language, and only rarely using mathematical language. However, without a formal semantics, it is impossible to rigorously reason about programs in that language. Moreover, a formal definition of a language is a specification offering its users and implementers a solid basis for agreeing on the meaning of programs. Unfortunately, tools for creating and working with formal definitions are poor and unfriendly, causing language designers to prefer writing reference manuals or reference implementations over formal definitions.

This paper presents a tool that makes it easy to write formal definitions for large languages and use them for analysis and verification. This tool, known as the $\mathbb{K}$ tool, is an executable implementation of the $\mathbb{K}$ framework [2], a formal specification language that is simultaneously expressive, modular, and analyzable. We extend an earlier implementation [4] with a mechanism for guided state-space search and an easy-to-use frontend that supports input and output. These key features allow users to experiment with language design and specification by means of testing and exhaustive non-deterministic behavior exploration.

Besides didactic and prototypical languages (such as System F and Agent), the $\mathbb{K}$ tool has been used to completely formalize C and Scheme. Several other languages are currently being defined using the $\mathbb{K}$ tool, including Haskell, Javascript, LLVM IR, and Python. The $\mathbb{K}$ tool has also been used in the development of several analysis tools, including a new program verification tool using program assertions based on matching logic, a model checking tool based on the CEGAR cycle, and several runtime verification tools. References and links to these tools and definitions can be found on the $\mathbb{K}$ tool website, `http://k-framework.org`.

* This work is supported by Contract 161/15.06.2010, SMISCSNR 602-12516 (DAK).

2 The 𝕂 Tool: Basics

𝕂 definitions of programming languages can be written in machine-readable ASCII. The 𝕂 tool provides facilities to manipulate such definitions, including typesetting them into their LaTeX mathematical representation, and generating execution and analysis tools. For example, Figure 1 gives the definition of a simple calculator language with variables, input, and output. We assume this definition is saved in a file called `exp.k` for the following examples.

```
module EXP
  configuration
    ⟨k⟩ $PGM:K ⟨/k⟩
    ⟨state⟩ $STATE:Map ⟨/state⟩
    ⟨streams⟩
      ⟨in stream="stdin"⟩ .List ⟨/in⟩
      ⟨out stream="stdout"⟩ .List ⟨/out⟩
    ⟨/streams⟩

  syntax KResult ::= Int

  syntax K ::= K + K [strict]
             | K / K [strict]

  rule I1:Int + I2:Int => I1 +Int I2
  rule I1:Int / I2:Int => I1 /Int I2
    when I2 =/=Int 0

  syntax K ::= Id

  rule ⟨k⟩ X:Id => I ⋯⟨/k⟩
       ⟨state⟩⋯ X |-> I:Int ⋯⟨/state⟩

  syntax K ::= read
             | print K [strict]

  rule ⟨k⟩ read => I ⋯⟨/k⟩
       ⟨in⟩ ListItem(I:Int) => .List ⋯⟨/in⟩

  rule ⟨k⟩ print I:Int => I ⋯⟨/k⟩
       ⟨out⟩⋯ .List => ListItem(I) ⟨/out⟩
end module
```

MODULE EXP

CONFIGURATION

$$\langle \$PGM \rangle_{\mathsf{k}} \; \langle \$STATE \rangle_{\mathsf{state}} \; \langle \langle \cdot \rangle_{\mathsf{in}} \; \langle \cdot \rangle_{\mathsf{out}} \rangle_{\mathsf{streams}}$$

SYNTAX $KResult ::= Int$

SYNTAX $K ::= K + K$ [strict] $\mid K$ / K [strict]

RULE $I_1 + I_2 \Rightarrow I_1 +_{Int} I_2$

RULE $I_1 \,/\, I_2 \Rightarrow I_1 \div_{Int} I_2$ when $I_2 \neq_{Int} 0$

SYNTAX $K ::= Id$

RULE $\langle \underset{I}{\underline{X}} \cdots \rangle_{\mathsf{k}} \; \langle \cdots X \mapsto I \cdots \rangle_{\mathsf{state}}$

SYNTAX $K ::=$ read $\mid$ print K [strict]

RULE $\langle \underset{I}{\underline{\mathtt{read}}} \cdots \rangle_{\mathsf{k}} \; \langle \underset{\cdot}{\underline{I}} \cdots \rangle_{\mathsf{in}}$

RULE $\langle \underset{I}{\underline{\mathtt{print}\ I}} \cdots \rangle_{\mathsf{k}} \; \langle \cdots \underset{I}{\underline{\cdot}} \rangle_{\mathsf{out}}$

END MODULE

Fig. 1. 𝕂 definition of a calculator language with variables and I/O (left: ASCII source; right: LaTeX generated by the tool)

For execution and analysis purposes, the definitions are translated into Maude rewrite theories. To obtain the rewrite theory associated to `exp.k`, we use the `kompile` tool:

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

Once the definition is compiled, it can be used for interpretation and analysis. Consider the program `p1.exp`:

```
print((read + read + read) / 3)
```

which reads three numbers and outputs their (truncated) mean. We can test this program using the `krun` tool:

```
$ echo "3 14 15" | krun p1.exp
10
```

Notice the use of the operating system's standard input/output streams.

Consider now the program `p2.exp`:

```
print(x + y)
```

which prints the sum of two externally defined variables x and y. If we forget to pass in a value for x at the start of the program's execution:

```
$ krun p2.exp --STATE="y |-> 2"
<k> x ~> □ + 2 ~> print □ </k>
<state> y |-> 2 </state>
```

the tool prints a configuration indicating that the execution got stuck. The contents of the k cell tells us that the next computation to perform is the lookup of x. Since x is not present in the state, the rule for variable lookup can not apply so the execution is unable to proceed. If we instead type `--STATE="y |-> 2  x |-> 3"`, the tool prints the expected result of 5.

These examples demonstrate a new and important feature of the $\mathbb{K}$ tool: the ability to associate cells in the configuration with data from the outside world. In the definition above, the in and out cells are linked to standard input/output (via the `stream` attribute) to achieve interactive I/O. This feature allows $\mathbb{K}$ definitions to easily be tested for correctness using existing test suites and test frameworks. Similarly, the state cell is initialized to the `$STATE` variable. This allows the contents of the cell to be manipulated from the command-line, as in the previous example. Incidentally, the k cell is also initialized to a variable, `$PGM`, which is always mapped to the input program.

3 The $\mathbb{K}$ Tool: Analysis

The $\mathbb{K}$ tool is more than an interpreter front-end. Consider the program `p3.exp`:

```
print(print(read) + print(read))
```

The definition of EXP in Figure 1 says the value returned by `print` is the printed number. Therefore, the program should read two numbers, print them, and then print their sum. If we just execute the program as before, we see what we expect:

```
$ echo "3 14" | krun p3.exp
31417
```

However, the definition also says that the evaluation order of + (specified by the `strict` annotation on its syntax) is nondeterministic. If we search for all possible behaviors of `p3.exp`, we obtain two final configurations with differing out cells: one where "3" is printed first and one where it is printed second:

```
$ echo "3 14" | krun p3.exp --search
Search results:
```

```
Solution 1, state 4:
<k> 17 </k>
<state> . </state>
<streams>
  <in> "" </in>
  <out> "31417" </out>
</streams>
```

```
Solution 2, state 9:
<k> 17 </k>
<state> . </state>
<streams>
  <in> "" </in>
  <out> "14317" </out>
</streams>
```

The state-space exploration functionality provided by `krun --search` can be used to explore all possible thread interleavings of multi-threaded programs. It is also used to find undefined behaviors in programs (particularly C programs).

In addition to the functionality shown in the examples above, the $\mathbb{K}$ tool gives $\mathbb{K}$ definitions access to the exploration, analysis, and proving tools available for all Maude rewrite theories [1], allowing programs written in the defined programming languages to also be debugged, traced, and model checked, all without modifying the definition.

4 Conclusion

The modularity and executability features provided by the $\mathbb{K}$ tool have made it possible to completely define large languages like C. These features also make it easy to experiment with language design in order to create new languages and make modifications to existing languages. Regardless of the language being defined, the same $\mathbb{K}$ definition that is tested by executing programs is used to do program analysis and is used to do proofs about the language. One semantics is used for all applications.

In this paper, we have shown only a subset of the features offered by the $\mathbb{K}$ tool. To learn more about it, or to start developing a programming language, download the $\mathbb{K}$ tool from our open source project page, `http://k-framework.googlecode.com`, and start by reading the $\mathbb{K}$ Primer [3].

References

1. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude, A High-Performance Logical Framework, LNCS, vol. 4350. Springer (2007)
2. Roșu, G., Șerbănuță, T.F.: An overview of the K semantic framework. J. Logic and Algebraic Programming 79(6), 397–434 (2010)
3. Șerbănuță, T.F., Arusoaie, A., Lazar, D., Ellison, C., Lucanu, D., Roșu, G.: The K primer (version 2.5). In: Proceedings of the 2nd Intl. K Wkshp. (K'11). Electronic Notes in Theoretical Computer Science, vol. to appear (2012)
4. Șerbănuță, T.F., Roșu, G.: K-Maude: A rewriting based tool for semantics of programming languages. In: 8th Intl. Wkshp. on Rewriting Logic and its Applications (WRLA'10). LNCS, vol. 6381, pp. 104–122 (2010)