$\mathbb{K}$: A Semantic Framework for Programming Languages and Formal Analysis Tools

Grigore Rosu [a,1]
[a] *University of Illinois at Urbana-Champaign, USA*

Abstract. We give an overview of the $\mathbb{K}$ framework, following the lecture notes presented by the author at the Marktoberdorf Summer School in year 2016.

Keywords. formal semantics, program verification, rewriting, $\mathbb{K}$

Introduction

Recently, operational semantics of several real languages have been proposed, e.g., of C [10,15], Java [5], JavaScript [4,26], Python [13,27], PHP [11], CAML [25], EVM [17,16] thanks to the development of semantics engineering frameworks like PLT-Redex [20], Ott [33], Lem [22], $\mathbb{K}$ [29,30], etc., which make defining an operational semantics for a programming language almost as easy as implementing an interpreter, if not easier. Operational semantics are comparatively easy to define and understand, require little formal training, scale up well, and, being executable, can be tested. Thus, they are typically used as trusted reference models for the defined languages.

Despite their advantages, operational semantics are rarely used directly for program verification; the general belief is that proofs tend to be low-level, as they work directly with the corresponding transition system. Hoare [18] or dynamic [14] logics are typically used, as they allow higher level reasoning. These come at the cost of (re)defining the language semantics as a set of abstract proof rules, which are harder to understand and trust. The state-of-the-art in mechanical program verification is to develop and prove such language-specific proof systems sound w.r.t. a trusted operational semantics [23,19,1], but that needs to be done for each language separately and is labor intensive.

Defining even one complete semantics for a real language like C or Java is already a huge effort. Defining multiple semantics, each good for a different purpose, is at best uneconomical, with or without proofs of soundness w.r.t. the reference semantics. It is therefore not surprising that many practical program verifiers forgo defining a semantics altogether, and instead they implement ad-hoc verification condition (VC) generation, sometimes via (unverified) translations to intermediate verification languages like Boogie [2] or Why3 [12]. For example, program verifiers for C like VCC [6] and Frama-C [12], and for Java like jStar [9] take this approach. Also, *none* of the 35 verifiers that

[1] E-mail: grosu@illinois.edu

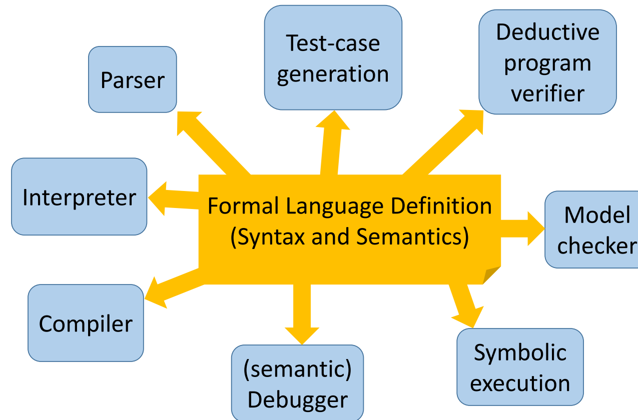

Figure 1. Architecture of the $\mathbb{K}$ framework, powered by matching logic

participated in the 2016 software verification competition (SV-COMP) [3] appear to be based on a formal semantics of any kind. The consequence is that such tools cannot be trusted. We would like program verifiers, ideally, to produce proof certificates whose trust base is only an operational semantics of the target language, same as mechanical verifiers based on Coq [21] or Isabelle [24] do, but without the effort to define any other semantics of the same language, either directly as a separate proof system or indirectly by extending the operational semantics with language-specific lemmas. We would like program verifiers, ideally, to take an operational semantics of a language as input and to yield, as output, a verifier for that language which is as easy to use and as efficient as verifiers specifically developed for that language.

The $\mathbb{K}$ framework [29,30] (`http://kframework.org`), illustrated in Figure 1. was born from our belief that programming languages must have formal definitions, and that tools for a given language, such as interpreters, compilers, state-space explorers, model checkers, deductive program verifiers, etc., can be derived from just *one* reference formal definition of the language, which is executable. No other semantics for the same language should be needed. This is the ideal scenario and there is enough evidence that it is within our reach in the short term. For example, [7] presents a program verification module of $\mathbb{K}$, based on matching logic [28], which takes the respective operational semantics of C [15], Java [5], and JavaScript [26] as input and yields automated program verifiers for these languages, capable of verifying challenging heap-manipulating programs at performance comparable to that of state-of-the-art verifiers specifically crafted for those languages. A precursor of this verifier, MatchC [31], has an online interface at `http://matching-logic.org` where one can verify dozens of predefined or new programs.

This paper gives a short overview of author's lectures notes presented at the Marktoberdorf Summer School in 2016.

1. Simple Language Definitions in $\mathbb{K}$

In this section we illustrate $\mathbb{K}$ by means of defining two canonical languages, one functional and another imperative. Many other languages, some significantly more complex,

were presented during the Marktoberdorf seminars and can be found in the 𝕂 tutorial, available online at `http://kframework.org`.

1.1. LAMBDA

Here we show a simple functional language definition in 𝕂, called LAMBDA, using a substitution style.

Substitution

We need the predefined substitution module[2], so we require it with the command below. Then we import its module called SUBSTITUTION in our LAMBDA module below.

```
require "substitution.k"

module LAMBDA
  imports SUBSTITUTION
```

Basic Call-by-value λ-Calculus

We first define a conventional call-by-value λ-calculus, making sure that we declare the lambda abstraction construct to be a binder, the lambda application to be strict, and the parentheses used for grouping as a bracket. Syntax in 𝕂 is defined using the familiar BNF notation, with terminals enclosed in quotes and nonterminals starting with capital letters. 𝕂 actually extends BNF with several attributes, some of which described in the sequel. The `strict` constructs can evaluate their arguments in any (fully interleaved) order. Here is the initial syntax of our λ-calculus:

```
  syntax Val ::= Id
               | "lambda" Id "." Exp  [binder]
  syntax Exp ::= Val
               | Exp Exp              [left, strict]
               | "(" Exp ")"          [bracket]
  syntax KVariable ::= Id
  syntax KResult ::= Val
```

β-reduction

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

Integer and Boolean Builtins

The LAMBDA arithmetic and Boolean expression constructs are simply rewritten to their builtin counterparts once their arguments are evaluated. The operations with sort suffixes in the right-hand sides of the rules below are builtin and come with the corresponding builtin sort. Note that the variables appearing in these rules have integer sort. That means that these rules will only be applied after the arguments of the arithmetic constructs are fully evaluated to 𝕂 results; this will happen thanks to their strictness attributes declared as annotations to their syntax declarations (below).

[2]Substitution can be defined fully generically in 𝕂 (not shown here), and then used to give semantics to various constructs in various languages.

```
syntax Val ::= Int | Bool
syntax Exp ::= Exp "*" Exp          [strict, left]
             | Exp "/" Exp          [strict]
             > Exp "+" Exp          [strict, left]
             > Exp "<=" Exp         [strict]
rule I1 * I2 => I1 *Int I2
rule I1 / I2 => I1 /Int I2  requires I2 =/=Int 0
rule I1 + I2 => I1 +Int I2
rule I1 <= I2 => I1 <=Int I2
```

Conditional

Note that the `if` construct is strict only in its first argument.

```
syntax Exp ::= "if" Exp "then" Exp "else" Exp    [strict(1)]
rule if true  then E else _ => E
rule if false then _ else E => E
```

Let Binder

The let binder is a derived construct, because it can be defined using λ.

```
syntax Exp ::= "let" Id "=" Exp "in" Exp
rule let X = E in E':Exp => (lambda X . E') E                        [macro]
```

Letrec Binder

We prefer a definition based on the μ construct. Note that μ is not really necessary, but it makes the definition of letrec easier to understand and faster to execute.

```
  syntax Exp ::= "letrec" Id Id "=" Exp "in" Exp
               | "mu" Id "." Exp                  [binder]
  rule letrec F:Id X:Id = E in E' => let F = mu F . lambda X . E in E' [macro]
  rule mu X . E => E[(mu X . E) / X]
endmodule
```

Compiling the Definition and Executing LAMBDA Programs

The $\mathbb{K}$ definition of LAMBDA is now complete. We can compile it using the command

```
$ kompile lambda.k
```

Then we can execute programs using the `krun` command. For example, if the file `factorial.lambda` contains the LAMBDA program

```
letrec f x = if x <= 1 then 1 else (x * (f (x + -1)))
in (f 10)
```

then the command

```
$ krun factorial.k
```

yields the expected result 3628800.

1.2. IMP

Below is the $\mathbb{K}$ semantic definition of the classic IMP language. IMP is considered a folklore language, without an official inventor, and has been used in many textbooks and papers, often with slight syntactic variations and often without being called IMP. It includes the most basic imperative language constructs, namely basic constructs for arithmetic and Boolean expressions, and variable assignment, conditional, while loop and sequential composition constructs for statements.

Syntax

```
module IMP-SYNTAX
```

This module defines the syntax of IMP. Note that `<=` is sequentially strict and has a LaTeX attribute making it display as $\leq$, and that `&&` is strict only in its first argument, because we want to give it a short-circuit semantics.

```
  syntax AExp  ::= Int | Id
                 | AExp "/" AExp             [left, strict]
                 > AExp "+" AExp             [left, strict]
                 | "(" AExp ")"              [bracket]
  syntax BExp  ::= Bool
                 | AExp "<=" AExp            [seqstrict, latex({#1}\leq{#2})]
                 | "!" BExp                  [strict]
                 > BExp "&&" BExp            [left, strict(1)]
                 | "(" BExp ")"              [bracket]
  syntax Block ::= "{" "}"
                 | "{" Stmt "}"
  syntax Stmt  ::= Block
                 | Id "=" AExp ";"           [strict(2)]
                 | "if" "(" BExp ")"
                   Block "else" Block        [strict(1)]
                 | "while" "(" BExp ")" Block
                 > Stmt Stmt                 [left]
```

An IMP program declares a set of variables and then executes a statement in the state obtained after initializing all those variables to 0. $\mathbb{K}$ provides builtin support for generic syntactic lists: *List*$\{$*Nonterminal*, *terminal*$\}$ stands for *terminal*-separated lists of *Nonterminal* elements.

```
  syntax Pgm ::= "int" Ids ";" Stmt
  syntax Ids ::= List{Id,","}
endmodule
```

We are done with the definition of IMP's syntax.

Semantics

```
module IMP
  imports IMP-SYNTAX
```

This module defines the semantics of IMP. Before you start adding semantic rules to a $\mathbb{K}$ definition, you need to define the basic semantic infrastructure consisting of definitions for *results* and the *configuration*.

Values and results

IMP only has two types of values, or results of computations: integers and Booleans. We here use the 𝕂 builtin variants for both of them.

```
syntax KResult ::= Int | Bool
```

Configuration

The configuration of IMP is trivial: it only contains two cells, one for the computation and another for the state. For good encapsulation and clarity, we place the two cells inside another cell, the "top" cell which is labeled T.

```
configuration <T color="yellow">
                <k color="green"> $PGM:Pgm </k>
                <state color="red"> .Map </state>
              </T>
```

The configuration variable $\$PGM$ tells the 𝕂 tool where to place the program. More precisely, the command "`krun program`" parses the program and places the resulting 𝕂 abstract syntax tree in the k cell before invoking the semantic rules described in the sequel. The `.Map` in the state cell is 𝕂's way to say "nothing". Technically, it is a constant which is the unit, or identity, of all maps in 𝕂 (similar dot units exist for other 𝕂 structures, such as lists, sets, multi-sets, etc.).

Arithmetic expressions

The 𝕂 semantics of each arithmetic construct is defined below.

Variable lookup

A program variable X is looked up in the state by matching a binding of the form $X \mapsto I$ in the state cell. If such a binding does not exist, then the rewriting process will get stuck. Thus our semantics of IMP disallows uses of uninitialized variables. Note that the variable to be looked up is the first task in the k cell (the cell is closed to the left and open to the right), while the binding can be anywhere in the state cell (the cell is open at both sides).

```
rule <k> X:Id => I ...</k> <state>... X |-> I ...</state>
```

Arithmetic operators

There is nothing special about these, but note that 𝕂's *configuration abstraction* mechanism is at work here! That means that the rewrites in the rules below all happen at the beginning of the k cell.

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

Boolean expressions

The rules below are straightforward. Note the short-circuited semantics of `&&`; this is the reason we annotated the syntax of `&&` with the 𝕂 attribute `strict(1)` instead of `strict`.

```
rule I1 <= I2 => I1 <=Int I2
rule ! T => notBool T
rule true && B => B
rule false && _ => false
```

Blocks and Statements

There is one rule per statement construct except for `if`, which needs two rules.

Blocks

The empty block `{}` is simply dissolved. The · below is the unit of the computation list structure *K*, that is, the empty task. Similarly, the non-empty blocks are dissolved and replaced by their statement contents, thus effectively giving them a bracket semantics; we can afford to do this only because we have no block-local variable declarations yet in IMP. Since we tagged the rules below as "structural", the 𝕂 tool structurally erases the block constructs from the computation structure, without considering their erasure as computational steps in the resulting transition systems. You can make these rules computational (dropping the attribute `structural`) if you do want these to count as computational steps.

```
rule {} => .   [structural]
rule {S} => S  [structural]
```

Assignment

The assigned variable is updated in the state. The variable is expected to be declared, otherwise the semantics will get stuck. At the same time, the assignment is dissolved.

```
rule <k> X = I:Int; => . ...</k> <state>... X |-> (_ => I) ...</state>
```

Sequential composition

Sequential composition is simply structurally translated into 𝕂's builtin task sequentialization operation. You can make this rule computational (i.e., remove the `structural` attribute) if you want it to count as a computational step. Recall that the semantics of a program in a programming language defined in 𝕂 is the transition system obtained from the initial configuration holding that program and counting only the steps corresponding to computational rules as transitions (i.e., hiding the structural rules as unobservable, or internal steps).

```
rule S1:Stmt S2:Stmt => S1 ~> S2  [structural]
```

Conditional

The conditional statement has two semantic cases, corresponding to when its condition evaluates to `true` or to `false`. Recall that the conditional was annotated with the attribute `strict(1)` in the syntax module above, so only its first argument is allowed to be evaluated.

```
  rule if (true)  S else _ => S
  rule if (false) _ else S => S
```

While loop

We give the semantics of the `while` loop by unrolling. Note that we preferred to make the rule below structural.

```
  rule while (B) S => if (B) {S while (B) S} else {}  [structural]
```

Programs

The semantics of an IMP program is that its body statement is executed in a state initializing all its global variables to 0. Since 𝕂's syntactic lists are internally interpreted as cons-lists (i.e., lists constructed with a head element followed by a tail list), we need to distinguish two cases, one when the list has at least one element and another when the list is empty. In the first case we initialize the variable to 0 in the state, but only when the variable is not already declared (all variables are global and distinct in IMP). We prefer to make the second rule structural, thinking of dissolving the residual empty `int;` declaration as a structural cleanup rather than as a computational step.

```
  rule <k> int (X,Xs => Xs);_ </k> <state> Rho:Map (.Map => X|->0) </state>
    requires notBool (X in keys(Rho))
  rule int .Ids; S => S  [structural]
endmodule
```

Compiling the Definition and Executing IMP Programs

After compilation with the command `kompile imp.k`, we can execute programs. If `sum.imp` contains the following program:

```
int n, sum;
n = 100;
sum = 0;
while (!(n <= 0)) {
  sum = sum + n;
  n = n + -1;
}
```

then `krun sum.imp` yields the final configuration

```
<T>
  <k> . </k>
  <state>
    n |-> 0
    sum |-> 5050
  </state>
</T>
```

2. Program Verification

One of the very basic design decisions of 𝕂 was to naturally support program verification for the defined languages. That is, to provide a language-independent reasoning in-

frastructure in addition to its language-independent execution infrastructure, which can then be used with any programming language by simply plugging in its semantics as a set of axioms. And, more importantly, to achieve the above in an ideal, mathematically grounded manner. That is, to have a fixed logic with a fixed sound and (relatively) complete proof system, where all the various programming languages become *theories*, about which we can reason using the fixed proof system, execution corresponding to just one particular proof for a reachability property. By "fixed logic" we therefore mean that it does not depend on any particular programming language. This is in sharp contrast to many logics used for program verification, such as Hoare logic, dynamic logic, or separation logic, which are in fact more like "design patterns": you have a "Hoare logic for Java", a "dynamic logic for C", a "separation logic for JavaScript", etc.

The logical foundation of $\mathbb{K}$ is *reachability logic* (see [7] for a recent reference) for dynamic properties, which uses *matching logic* (see [28] for a recent reference) for static properties. We do not discuss these logics here; instead we encourage the interested reader to consult the above-mentioned references. Here we only show how to do program verification with $\mathbb{K}$. The first step is to specify what you want to prove. Specifications can be written using the already existing $\mathbb{K}$ rule syntax; both the rules used for giving the language semantics and the rules used to specify program properties are *reachability rules* in reachability logic, which is the only type of sentence that reachability logic supports. One useful way to think when writing specifications is the following: "supposing that I want to add the program or fragment of program as a new construct to my language, what would its semantics be?". Let us consider the IMP language and its `sum.imp` program discussed in Section 1.2. We claim that the following $\mathbb{K}$ rule properly captures the intended semantics of the sum program:

```
rule
  <k>
    int n, sum;
    n = N:Int;
    sum = 0;
    while (!(n <= 0)) {
      sum = sum + n;
      n = n + -1;
    }
  =>
    .
  </k>
  <state>
    .Map
  =>
    n   |-> 0
    sum |-> ((N +Int 1) *Int N /Int 2)
  </state>
requires N >=Int 0
```

The rule above says that if the sum program is put in the `<k>` cell, with program variable `n` assigned a mathematical (or symbolic) variable N of sort *Int*, constrained to be larger than or equal to zero (the rule condition), and with an empty state, then the program rewrites to nothing and the state rewrites to one containing bindings of the program variables to their expected mathematical values. This is a partial correctness argument, in the sense that the program is allowed to not terminate [7]. We urge the reader to note that

the above is nothing but a normal $\mathbb{K}$ rule, containing only one variable[3], N of integer sort, and a requires clause; for comparison, the rule for division in Section 1.2 also contained a requires clause, but two variables and a smaller term in its left-hand side.

$\mathbb{K}$ does not currently do any automatic specification inference, so the user is required to supply all the helping specifications. In our case here, we need to abstract the behavior of the while loop. The usual approach, following the Hoare logic style, is to provide a *loop invariant*, that is, a tight property that holds at each iteration of the loop and implies the postcondition once the loop condition becomes negative. In our case, such an invariant would state that `n` is bound to some symbolic value N' with $N' \geq 0$ and `sum` to symbolic value $(N-N')*(N+N'+1)/2$. Hoare logic can be mechanically translated to reachability logic [32], so we can follow the same approach if we choose to. But we can often do proofs more elegantly and intuitively with reachability logic, by summarizing the effect of the entire loop with one reachability rule instead of focusing on the invariant. For example, we can add the following specification for the while loop:

```
rule
  <k>
    while (!(n <= 0)) {
      sum = sum + n;
      n = n + -1;
    }
  =>
    .
  ...</k>
  <state>...
    n   |-> (N:Int => 0)
    sum |-> (S:Int => S +Int ((N +Int 1) *Int N /Int 2))
  ...</state>
requires N >=Int 0
```

This rule states the obvious: when executed with `n` bound to[4] N and `sum` bound to S, when the loop exits `n` is bound to 0 and `S` is bound to the properly incremented value.

To verify the above using $\mathbb{K}$, put both rules above in a module extending the IMP semantics and save it in a file, say `sum-spec.k`. Then invoke the `krun` tool with the option `-prove` on this file. It should return `true`. There is also an option to see all the queries that $\mathbb{K}$ makes to Z3 [8] during the verification process.

3. More Complex Language Definitions in $\mathbb{K}$

$\mathbb{K}$ scales. Several real-life languages have been defined following a style similar to that in Section 1, such as: C [10,15], Java [5], JavaScript [26], Python [13,27], PHP [11], and more recently, EVM [16]. Then properties about programs in these languages have been verified using a style similar to that in Section 2. We cannot discuss any of these large language definitions here, but we can add a few more features to IMP to show how the modularity of $\mathbb{K}$ facilitates the process of defining large languages.

[3] The program variables `n` and sum are technically constants of sort `Id`.

[4] The N in this rule has nothing to do with the N in the previous rule, for the same reason for which the same variable names used in the different rules in Section 1 had nothing to do with each other. Technically, each rule is assumed universally quantified over its free variables.

IMP++ extends the IMP language with the features listed below:

Strings and concatenation of strings. Strings are useful for the `print` statement, which is discussed below. For string concatenation, we use the same `+` construct that we use for addition (so we overload it).

Variable increment. We only add a pre-increment construct: `++x` increments variable `x` and evaluates to the incremented value. Variable increment makes the evaluation of expressions have side effects, and thus makes the evaluation strategies of the various language constructs have an influence on the set of possible program behaviors.

Input and output. IMP++ adds a `read()` expression construct which reads an integer number and evaluates to it, and a variadic (i.e., it has an arbitrary number of arguments) statement construct `print(e1,e2,...,en)` which evaluates its arguments and then outputs their values. Note that the $\mathbb{K}$ tool allows to connect the input and output cells to the standard input and output buffers, this way compiling the language definition into an interactive interpreter.

Abrupt termination. The `halt` statement simply halts the program. The $\mathbb{K}$ tool shows the resulting configuration, as if the program terminated normally. We therefore assume that an external observer does not care whether the program terminates normally or abruptly, same like with `exit` statements in conventional programming languages like C.

Dynamic threads. The expression construct `spawn s` starts a new concurrent thread that executes statement `s`, which is expected to be a block, and evaluates immediately to a fresh thread identifier that is also assigned to the newly created thread. The new thread is given at creation time the *environment* of its parent, so it can access all its parent's variables. This allows for the parent thread and the child thread to communicate; it also allows for races and "unexpected" behaviors, so be careful. For thread synchronization, IMP++ provides a thread join statement construct "`join t;`", where `t` evaluates to a thread identifier, which stalls the current thread until thread `t` completes its computation.

Blocks and local variables. IMP++ allows blocks enclosed by curly brackets. Also, IMP's global variable declaration construct is generalized to be used anywhere as a statement, not only at the beginning of the program. As expected, the scope of the declared variables is from their declaration point till the end of the most nested enclosing block.

Syntax

```
module IMP-SYNTAX
```

IMP++ adds several syntactic constructs to IMP. Also, since the variable declaration construct is generalized to be used anywhere a statement can be used, not only at the beginning of the program, we need to remove the previous global variable declaration of IMP and instead add a variable declaration statement construct.

We do not re-discuss the constructs which are taken over from IMP, except when their syntax has been subtly modified (such as, e.g., the syntax of the previous "statement" assignment which is now obtained by composing the new assignment expression and the new expression statement constructs). For execution purposes, we tag the ad-

dition and division operations with the `addition` and `division` tags. These attributes have no theoretical significance, in that they do not affect the semantics of the language in any way. They only have practical relevance, specific to our implementation of the $\mathbb{K}$ tool. Specifically, we can tell the $\mathbb{K}$ tool (using its `superheat` and `supercool` options) that we want to exhaustively explore all the non-deterministic behaviors (due to strictness) of these language constructs. For performance reasons, by default the $\mathbb{K}$ tool chooses an arbitrary but fixed order to evaluate the arguments of the strict language constructs, thus possibly losing behaviors due to missed interleavings. This aspect was irrelevant in IMP, as its expressions had no side effects, but it becomes relevant in IMP++.

The syntax of the IMP++ constructs is self-explanatory. Note that assignment is now an expression construct. Also, `print` is variadic, taking a list of expressions as argument. It is also strict, which means that the entire list of expressions, i.e., each expression in the list, will be evaluated. Note also that we now defined sequential composition of statements as a whitespace-separated list of statements, aliased with the nonterminal `Stmts`, and block as such a (possibly empty) statement sequence surrounded by curly brackets.

```
  syntax AExp  ::= Int | String | Id
                 | "++" Id
                 | "read" "(" ")"
                 > AExp "/" AExp              [left, strict, division]
                 > AExp "+" AExp              [left, strict]
                 > "spawn" Block
                 > Id "=" AExp                [strict(2)]
                 | "(" AExp ")"               [bracket]
  syntax BExp  ::= Bool
                 | AExp "<=" AExp             [seqstrict, latex({#1}\leq{#2})]
                 | "!" BExp                   [strict]
                 > BExp "&&" BExp             [left, strict(1)]
                 | "(" BExp ")"               [bracket]
  syntax Block ::= "{" Stmts "}"
  syntax Stmt  ::= Block
                 | AExp ";"                   [strict]
                 | "if" "(" BExp ")"
                   Block "else" Block         [strict(1)]
                 | "while" "(" BExp ")" Block
                 | "int" Ids ";"
                 | "print" "(" AExps ")" ";"
                 | "halt" ";"
                 > "join" AExp ";"            [strict]

  syntax Ids   ::= List{Id,","}
  syntax AExps ::= List{AExp,","}
  syntax Stmts ::= List{Stmt,""}
endmodule
```

Semantics

```
module IMP
  imports IMP-SYNTAX
```

We next give the semantics of IMP++. We start by first defining its configuration.

Configuration

The original configuration of IMP has been extended to include all the various additional cells needed for IMP++. To facilitate the semantics of threads, more specifically to naturally give them access to their parent's variables, we prefer a (rather conventional) split of the program state into an *environment* and a *store*. An environment maps variable names into *locations*, while a store maps locations into values. Stores are also sometimes called "states", or "heaps", or "memory", in the literature. Like values, locations can be anything. For simplicity, here we assume they are natural numbers. Moreover, each thread has its own environment, so it knows where all the variables that it has access to are located in the store (that includes its locally declared variables as well as the variables of its parent thread), and its own unique identifier. The store is shared by all threads. For simplicity, we assume a sequentially consistent memory model in IMP++. Note that the thread cell has multiplicity "*", meaning that there could be zero, one, or more instances of that cell in the configuration at any given time. This multiplicity information is important for $\mathbb{K}$'s *configuration abstraction* process: it tells $\mathbb{K}$ how to complete rules which, in order to increase the modularity of the definition, choose to not mention the entire configuration context. The in and out cells hold the input and the output buffers as lists of items.

```
configuration <T color="yellow">
                <threads color="orange">
                  <thread multiplicity="*" color="blue">
                    <k color="green"> $PGM:Stmts </k>
                    <env color="LightSkyBlue"> .Map </env>
                    <id color="black"> 0 </id>
                  </thread>
                </threads>
                <store color="red"> .Map </store>
                <in color="magenta" stream="stdin"> .List </in>
                <out color="Orchid" stream="stdout"> .List </out>
              </T>
```

We can also use configuration variables to initialize the configuration through `krun`. For example, we may want to pass a few list items in the in cell when the program makes use of `read()`, so that the semantics does not get stuck. Recall from IMP that configuration variables start with a *$* character when used in the configuration (see, for example, *$PGM*) and can be initialized with any string by `krun`; or course, the string should parse to a term of the corresponding sort, otherwise errors will be generated. Moreover, $\mathbb{K}$ allows you to connect list cells to the standard input or the standard output. For example, the attribute `stream="stdin"` to the in cell tells `krun` to prompt the user to pass input when the in cell is empty and any semantic rule needs at least one item to be present there in order to match. Similarly but dually, the attribute `stream="stdout"` to the out cell, tells that any item placed into this cell by any rule will be promptly sent to the standard output. This way, krun can be used to obtain interactive interpreters based directly on the $\mathbb{K}$ semantics of the language. For example:

```
bash$ krun sum-io.imp --no-config
Add numbers up to (<= 0 to quit)? 10
Sum = 55
Add numbers up to (<= 0 to quit)? 1000
```

```
Sum = 500500
Add numbers up to (<= 0 to quit)? 0
bash$
```

The option `--no-config` instructs `krun` to not display the resulting configuration after the program executes. The input/output streaming works with or without this option, although if you don't use the option then a configuration with empty in and out cells will be displayed after the program is executed. You can also initialize the configuration using configuration variables and stream the contents of the cells to standard input/output at the same time. For example, if you use a configuration variable in the in cell and pass contents to it through `krun`, then that contents will be first consumed and then the user will be prompted to introduce additional input if the program's execution encounters more `read()` constructs.

The old IMP constructs

The semantics of the old IMP constructs is almost identical to their semantics in the original IMP language, except for those constructs making use of the program state and for those whose syntax has slightly changed. Indeed, the rules for variable lookup and assignment in IMP accessed the state cell, but that cell is not available in IMP++ anymore. Instead, we have to use the combination of environment and store cells. Thanks to $\mathbb{K}$'s implicit configuration abstraction, we do not have to mention the thread and threads cells: these are automatically inferred (and added by the $\mathbb{K}$ tool at compile time) from the definition of the configuration above, as there is only one correct way to complete the configuration context of these rules in order to match the configuration declared above. In our case here, "correct way" 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. Configuration abstraction is crucial for modularity, because it gives us the possibility to write our definitions in a way that may not require us to revisit existing rules when we change the configuration. Changes in the configuration are quite frequent in practice, typically needed in order to accommodate new language features. For example, imagine that we initially did not have threads in IMP++. There would be no need for the thread and threads cells in the configuration then, the cells k and env being simply placed at the top level in the T cell, together with the already existing cells. Then the rules below would be exactly the same. Thus, configuration abstraction allows you to not have to modify your rules when you make structural changes in your language configuration.

Below we list the semantics of the old IMP constructs, referring the reader to the $\mathbb{K}$ semantics of IMP for their meaning. Like we tagged the addition and the division rules above in the syntax, we also tag the lookup and the assignment rules below (with tags `lookup` and `assignment`), because we want to refer to them when we generate the language model (with the `kompile` tool), basically to allow them to generate (possibly non-deterministic) transitions. Indeed, these two rules, unlike the other rules corresponding to old IMP constructs, can yield non-deterministic behaviors when more threads are executed concurrently. In terms of rewriting, these two rules can "compete" with each other on some program configurations, in the sense that they can both match at the same time and different behaviors may be obtained depending upon which of them is chosen first.

```
  syntax KResult ::= Int | Bool
```

Variable lookup

```
rule <k> X:Id => I ...</k>
     <env>... X |-> N ...</env>
     <store>... N |-> I ...</store>  [lookup]
```

Arithmetic constructs

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

Boolean constructs

```
rule I1 <= I2 => I1 <=Int I2
rule ! T => notBool T
rule true && B => B
rule false && _ => false
```

Variable assignment

Note that the old IMP assignment statement "$X = I$;" is now composed of two constructs: an assignment expression construct "$X = I$", followed by a semicolon ";" turning the expression into a statement. Here is the semantics of the two constructs:

```
rule _:Int; => .
rule <k> X = I:Int => I ...</k>
     <env>... X |-> N ...</env>
     <store>... N |-> (_ => I) ...</store>  [assignment]
```

Sequential composition

Sequential composition has been defined as a whitespace-separated syntactic list of statements. Recall that syntactic lists are actually syntactic sugar for cons-lists. Therefore, the following two rules eventually sequentialize a syntactic list of statements "s1 s2 ... sn.. into the corresponding computation "s1 ~> s2 ~> ... ~> sn".

```
rule .Stmts => .
rule S:Stmt Ss:Stmts => S ~> Ss  [structural]
```

Conditional statement

```
rule if (true)  S else _ => S
rule if (false) _ else S => S
```

While loop

The only thing to notice here is that the empty block has been replaced with the block holding the explicit empty sequence. That's because in the semantics all empty lists become explicit corresponding dots (to avoid parsing ambiguities)

```
rule while (B) S => if (B) {S while (B) S} else {.Stmts}  [structural]
```

The new IMP++ constructs

We next discuss the semantics of the new IMP++ constructs.

Strings

First, we have to state that strings are also results. Second, we give the semantics of IMP++ string concatenation (which uses the already existing addition symbol + from IMP) by reduction to the built-in string concatenation operation.

```
syntax KResult ::= String
rule Str1 + Str2 => Str1 +String Str2
```

Variable increment

Like variable lookup, this is also meant to be a supercool transition: we want it to count both in the non-determinism due to strict operations above it in the computation and in the non-determinism due to thread interleavings. This rule also relies on $\mathbb{K}$'s configuration abstraction. Without abstraction, you would have to also include the thread and threads cells.

```
rule <k> ++X => I +Int 1 ...</k>
     <env>... X |-> N ...</env>
     <store>... N |-> (I => I +Int 1) ...</store>  [increment]
```

Read

The `read()` construct evaluates to the first integer in the input buffer, which it consumes. Note that this rule is tagged `increment`. This is because we will include it in the set of potentially non-deterministic transitions when we kompile the definition; we want to do that because two or more threads can "compete" on reading the next integer from the input buffer, and different choices for the next transition can lead to different behaviors.

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

Print

The `print` statement is strict, so all its arguments are eventually evaluated (recall that `print` is variadic). We append each of its evaluated arguments, in order, to the output buffer, and structurally discard the residual `print` statement with an empty list of arguments. We only want to allow printing integers and strings, so we define a *Printable* syntactic category including only these and define the `print` statement to only print *Printable* elements. Alternatively, we could have had two similar rules, one for integers and one for strings. Recall that, currently, $\mathbb{K}$'s lists are cons-lists, so we cannot simply rewrite the head of a list (P) into a list ($\cdot$). The first rule below is tagged, because we want to include it in the list of transitions when we kompile; different threads may compete on the output buffer and we want to capture all behaviors. The second rule is structural because we do not want it to count as a computational step.

```
syntax Printable ::= Int | String
syntax AExp ::= Printable

context print(HOLE:AExp, AEs:AExps);

rule <k> print(P:Printable,AEs => AEs); ...</k>
     <out>... .List => ListItem(P) </out>  [print]
rule print(.AExps); => .  [structural]
```

Halt

The `halt` statement empties the computation, so the rewriting process simply terminates as if the program terminated normally. Interestingly, once we add threads to the language, the `halt` statement as defined below will terminate the current thread only. If you want an abrupt termination statement that halts the entire program, then you need to discard the entire contents of the threads cell, so the entire computation abruptly terminates the entire program, no matter how many concurrent threads it has, because there is nothing else to rewrite.

```
rule <k> halt; ~> _ => . </k>
```

Spawn thread

A spawned thread is passed its parent's environment at creation time. The `spawn` expression in the parent thread is immediately replaced by the unique identifier of the newly created thread, so the parent thread can continue its execution. We only consider a sequentially consistent shared memory model for IMP++, but other memory models can also be defined in $\mathbb{K}$. Note that the rule below does not need to be tagged in order to make it a transition when we kompile, because the creation of the thread itself does not interfere with the execution of other threads. Also, note that $\mathbb{K}$'s configuration abstraction is at heavy work here, in two different places. First, the parent thread's k and env cells are wrapped within a thread cell. Second, the child thread's k, env and id cells are also wrapped within a thread cell. Why that way and not putting all these four cells together within the same thread, or even create an additional threads cell at top holding a thread cell with the new k, env and id? Because in the original configuration we declared the multiplicity of the thread cell to be "$*$", which effectively tells the $\mathbb{K}$ tool that zero, one or more such cells can co-exist in a configuration at any moment. The other cells have the default multiplicity "one", so they are not allowed to multiply. Thus, the only way to complete the rule below in a way consistent with the declared configuration is to wrap the first two cells in a thread cell, and the latter two cells under the "$\cdot$" also in a thread cell. Once the rule applies, the spawning thread cell will add a new thread cell next to it, which is consistent with the declared configuration cell multiplicity. The unique identifier of the new thread is generated using the "fresh" side condition.

```
rule <k> spawn S => !T:Int ...</k> <env> Rho </env>
     (.Bag => <thread>... <k> S </k> <env> Rho </env> <id> !T </id> ...</thread>)
```

Join thread

A thread who wants to join another thread `T` has to wait until the computation of `T` becomes empty. When that happens, the join statement is simply dissolved. The terminated thread is not removed, because we want to allow possible other join statements to also dissolve.

```
rule <k> join(T); => . ...</k> <thread>... <k>.</k> <id>T</id> ...</thread>
```

Blocks

The body statement of a block is executed normally, making sure that the environment at the block entry point is saved in the computation, in order to be recovered after the block

body statement. This step is necessary because blocks can declare new variables having the same name as variables which already exist in the environment, and our semantics of variable declarations is to update the environment map in the declared variable with a fresh location. Thus, variables which are shadowed lose their original binding, which is why we take a snapshot of the environment at block entrance and place it after the block body (see the semantics of environment recovery at the end of this module). Note that any store updates through variables which are not declared locally are kept at the end of the block, since the store is not saved/restored. An alternative to this environment save/restore approach is to actually maintain a stack of environments and to push a new layer at block entrance and pop it at block exit. The variable lookup/assign/increment operations then also need to change, so we do not prefer that non-modular approach. Compilers solve this problem by statically renaming all local variables into fresh ones, to completely eliminate shadowing and thus environment saving/restoring. The rule below can be structural, because what it effectively does is to take a snapshot of the current environment; this operation is arguably not a computational step.

```
rule <k> {Ss} => Ss ~> Rho ...</k> <env> Rho </env>  [structural]
```

Variable declaration

We allocate a fresh location for each newly declared variable and initialize it with 0.

```
rule <k> int (X,Xs => Xs); ...</k>
     <env> Rho => Rho[X <- !N:Int] </env>
     <store>... .Map => !N |-> 0 ...</store>
rule int .Ids; => .  [structural]
```

Auxiliary operations

We only have one auxiliary operation in IMP++, the environment recovery. Its role is to discard the current environment in the env cell and replace it with the environment that it holds. This rule is structural: we do not want them to count as computational steps in the transition system of a program.

```
rule <k> Rho => . ...</k> <env> _ => Rho </env>    [structural]
```

If you want to avoid useless environment recovery steps and keep the size of the computation structure smaller, then you can also add the rule

```
rule (_:Map => .) ~> _:Map  [structural]
```

This rule acts like a “tail recursion” optimization, but for blocks.

```
endmodule
```

On Kompilation Options

We are done with the IMP++ semantics. The next step is to kompile the definition using the `kompile` tool, this way generating a language model. Depending upon for what you want to use the generated language model, you may need to kompile the definition using various options. We here discuss these options.

To tell the $\mathbb{K}$ tool to exhaustively explore all the behaviors due to the non-determinism of addition, division, and threads, we have to kompile with the command:

```
kompile imp.k --transition="addition division lookup assignment increment read print"
```

As already mentioned, the syntax and rule tags play no theoretical or foundational role in $\mathbb{K}$. They are only a means to allow `kompile` to refer to them in its options, like we did above. By default, `kompile`'s transition option is empty, because this yields the fastest language model when executed. Transitions may slow down the execution, but they instrument the language model to allow for formal analysis of program behaviors, even for exhaustive analysis.

Theoretically, the heating/cooling rules in $\mathbb{K}$ are fully reversible and unconstrained by side conditions as we showed in the semantics of IMP. For example, the theoretical heating/cooling rules corresponding to the `strict` attribute of division are the following:

$$\begin{array}{l} E_1/E_2 \ \Rightarrow\ E_1 \curvearrowright \square/E_2 \\ E_1 \curvearrowright \square/E_2 \ \Rightarrow\ E_1/E_2 \\ E_1/E_2 \ \Rightarrow\ E_2 \curvearrowright E_1/\square \\ E_2 \curvearrowright E_1/\square \ \Rightarrow\ E_1/E_2 \end{array}$$

The other semantic rules apply *modulo* such structural rules. For example, using heating rules we can bring a redex (a subterm which can be reduced with semantic rules) to the front of the computation, then reduce it, then use cooling rules to reconstruct a term over the original syntax of the language, then heat again and non-deterministically pick another redex, and so on and so forth without losing any opportunities to apply semantic rules. Nevertheless, these unrestricted heating/cooling rules may create an immense, often unfeasibly large space of possibilities to analyze. The `-transition` option implements an optimization which works well with other implementation choices made in the current $\mathbb{K}$ tool. Recall from the detailed description of the IMP language semantics that (theoretical) reversible rules like above are restricted by default to complementary conditional rules of the form

$$\begin{array}{l} E_1/E_2 \ \Rightarrow\ E_1 \curvearrowright \square/E_2 \ \text{ if } E_1 \notin \mathit{KResult} \\ E_1 \curvearrowright \square/E_2 \ \Rightarrow\ E_1/E_2 \ \text{ if } E_1 \in \mathit{KResult} \\ E_1/E_2 \ \Rightarrow\ E_2 \curvearrowright E_1/\square \ \text{ if } E_2 \notin \mathit{KResult} \\ E_2 \curvearrowright E_1/\square \ \Rightarrow\ E_1/E_2 \ \text{ if } E_2 \in \mathit{KResult} \end{array}$$

Therefore, our tool eagerly heats and lazily cools the computation. In other words, heating rules apply until a redex gets placed on the top of the computation, then some semantic rule applies and rewrites that into a result, then a cooling rule is applied to plug the obtained result back into its context, then another argument may be chosen and completely heated, and so on. This leads to efficient execution, but it may and typically does hide program behaviors. Using the `-transition` option allows you to interfere with this process and to obtain all possible non-deterministic behaviors as if the theoretical heating/cooling rules were applied. Optimizations of course happen under the hood, but you need not be aware of them. Used carefully, this mechanism allows us to efficiently explore more of the non-deterministic behaviors of a program, even all of them (like here). For example, with the semantics of IMP++ given above, the `krun` command with the `--search` option detects all five behaviors of the following IMP++ program (`x` can be 0, 1, 2, 3, or undefined due to division-by-zero):

```
int x,y;
x = 1;
y = ++x / (++x / x);
```

Besides non-determinism due to underspecified argument evaluation orders, which the current $\mathbb{K}$ tool addresses as explained above, there is another important source of non-determinism in programming languages: non-determinism due to concurrency/parallelism. For example, when two or more threads are about to access the same location in the store and at least one of these accesses is a write (i.e., an instance of the variable assignment rule), there is a high chance that different choices for the next transition lead to different program behaviors. While in the theory of $\mathbb{K}$ all the non-structural rules count as computational steps and hereby as transitions in the transition system associated to the program, in practice that may yield a tremendous number of step interleavings to consider. 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 the other threads, so one may not want to consider it as an observable transition in the space of interleavings. Since the 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 using the `transition` option of `kompile`.

4. Conclusion

We gave a very high-level overview of the $\mathbb{K}$ framework, as presented at the Marktoberdorf Summer School in 2016. The interested reader is strongly encouraged to consult the K Tutorial reachable from the main $\mathbb{K}$ webpage at `http://kframework.org` and to contact the author. Besides many other languages covering various paradigms, the tutorial also shows how to define type systems in $\mathbb{K}$, including type inference.

The author would like to warmly thank the organizers, Doron Peled and Alexander Pretschner, for inviting him to lecture at the summer school and for their support and infinite patience during the editing of this paper. The work presented in this paper was supported in part by the Boeing grant on "Formal Analysis Tools for Cyber Security" 2016-2017, the NSF grants CCF-1318191 and CCF-1421575, and an IOHK gift.

References

[1] A. W. Appel. Verified software toolchain. In *ESOP'11*, volume 6602 of *LNCS*, pages 1–17, 2011.

[2] M. Barnett, B. yuh Evan Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO'05*, volume 4111 of *LNCS*, pages 364–387, 2006.

[3] D. Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In *TACAS'16*, volume 9636 of *LNCS*, pages 887–904, 2016.

[4] M. Bodin, A. Chargueraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In *POPL'14*, pages 87–100. ACM, 2014.

[5] D. Bogdănaş and G. Roşu. K-Java: A Complete Semantics of Java. In *POPL'15*, pages 445–456. ACM, January 2015.

[6] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 23–42, 2009.

[7] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu. Semantics-based program verifiers for all languages. In *OOPSLA'16*, pages 74–91. ACM, 2016.

[8] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340, 2008.

[9] D. Distefano and M. J. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA'08*, pages 213–226. ACM, 2008.

[10] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544. ACM, 2012.

[11] D. Filaretti and S. Maffeis. An executable formal semantics of php. In *ECOOP'14*, LNCS, pages 567–592. Springer, 2014.

[12] J. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In *ESOP'13*, volume 7792 of *LNCS*, pages 125–128, 2013.

[13] D. Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, July 2013. `https://github.com/kframework/python-semantics`.

[14] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604, 1984.

[15] C. Hathhorn, C. Ellison, and G. Roşu. Defining the undefinedness of C. In *PLDI'15*, pages 336–345. ACM, 2015.

[16] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu. Kevm: A complete semantics of the ethereum virtual machine. Technical Report http://hdl.handle.net/2142/97207, University of Illinois, Aug 2017.

[17] Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. WSTC17, International Conference on Financial Cryptography and Data Security, 2017.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 12(10):576–580, 1969.

[19] B. Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *The Journal of Logic and Algebraic Programming*, 58(1-2):61–88, 2004.

[20] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: On the effectiveness of lightweight mechanization. In *POPL'12*, pages 285–296. ACM, 2012.

[21] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0, `http://coq.inria.fr`.

[22] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *ACM SIGPLAN Notices*, volume 49, pages 175–188. ACM, 2014.

[23] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[24] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[25] S. Owens. A sound semantics for OCaml$_{light}$. In *ESOP'08*, volume 4960 of *LNCS*, pages 1–15, 2008.

[26] D. Park, A. Ştefănescu, and G. Roşu. KJS: A complete formal semantics of JavaScript. In *PLDI'15*, pages 346–356. ACM, 2015.

[27] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full monty. In *OOPSLA'13*, pages 217–232. ACM, 2013.

[28] G. Roşu. Matching logic. *Logical Methods in Computer Science*, to appear, 2017.

[29] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[30] G. Rosu and T. F. Serbanuta. K overview and simple case study. In *Proceedings of International K Workshop (K'11)*, volume 304 of *ENTCS*, pages 3–56. Elsevier, June 2014.

[31] G. Rosu and A. Stefanescu. Checking reachability using matching logic. In *OOPSLA'12*, pages 555–574. ACM, 2012.

[32] G. Rosu and A. Stefanescu. From hoare logic to matching logic reachability. In *FM'12*, volume 7436 of *LNCS*, pages 387–402, 2012.

[33] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: effective tool support for the working semanticist. In *ICFP'07*, pages 1–12. ACM, 2007.