Masaryk University
Faculty of Informatics

An Executable Formal Semantics of C++

Master's Thesis

Jan Tušil

Brno, Fall 2017

Replace this page with a copy of the official signed thesis assignment and a copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jan Tušil

Advisor: Jan Strejček

Abstract

«abstract»

Keywords

C++ semantics k-framework

Contents

List of Tables

List of Figures

1 Introduction

Writing correct software is hard. Although formal methods for software verification are being developed, there are few high quality tools on the market. One particular problem is that in order to create a production-ready tool, it is not enough to understand formal methods; the developers also need to understand the precise semantics of the selected programming language.

In recent years, a platform named „𝕂 framework" is gaining popuparity. The platform is based on the idea that formal, executable language semantics can be used to derive a large variety of tools, including interpreters, debuggers, model checkers or deductive program verifiers [8] (see Figure 1.1). Tool developers, skilled in a particular area of formal methods, can work inside their area of expertise and developing language independent tools, while leaving language details to someone else. Thus, a separation of concerns is achieved.

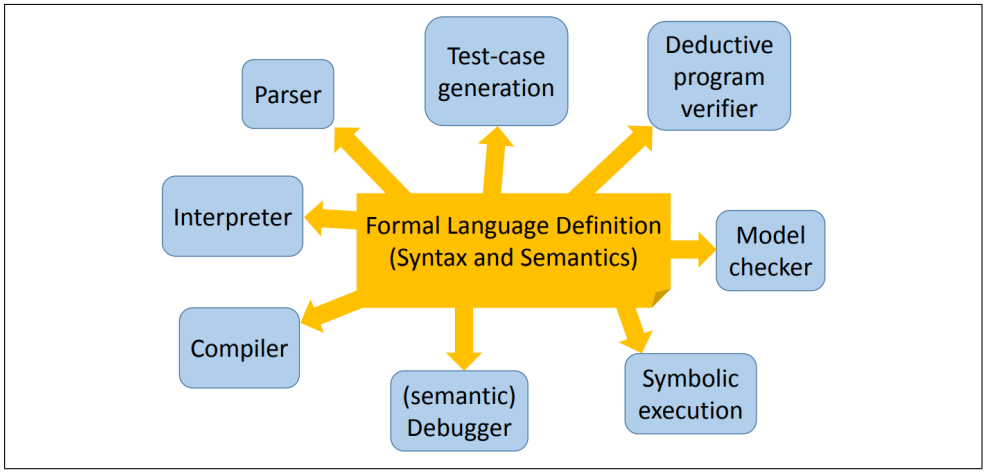

Figure 1.1: The idea behind 𝕂. Adopted from [7].

𝕂 framework has been successfully used to give formal semantics to a varienty of languages, including Java [2], Python, Javascript [6], and C [3, 5], all of which is publicly available. The C semantics has been used to create RV-Match, a „tool for checking C programs for undefined behavior and other common programmer mistakes" [4]. At

the time of writing, the C semantics is being extended to support C++; the C++ support is also the focus of this thesis.

The thesis originally aimed to implement whichever language features needed to be done, as the C++ language is complex and the C++ semantics is still highly incomplete. As the work progressed, two features were selected to be implemented: *enumerations* and *constant expressions*. It went out that the two features play together rather nicely: C++ allows enumerators to be initialized with constant expressions, so enumerations can be used to test the implementation of constant expressions.

Enumerations were chosen because of their relative simplicity: it is a purely compile-time feature, which does not interfere much with other language features; it exists in the language from its beginning, and *scoped enumerations* introduced in C++11 are even simpler both from language and user point of view (the legacy C-style enumerations still need to be supported, though).

Constant expressions, on the contrary, had undergone a deep change in C++11, which allowed a restricted set of runtime computations to happen in the compilation time; the future revisions released the restrictions to the point that in C++17, almost arbitrary side effect free computations can happen in the time of compilation. One could reasonably expect that in order to implement constant expressions, a more fundamental change to the semantics have to be done.

The purpose of this text is to describe the implementation of the aforementioned features. The rest of the document is organized as follows:

1. introduction of the project of C/C++ semantics
2. description of the involved concepts
3. outline of the general architecture of the project
4. discussion of implementation of enums
5. discussion of implementation of constant expressions

From this point on, we will write „Project" instead of „the C/C++ semantics in K".

2 Background

This chapter intends to give a brief overview of the $\mathbb{K}$ framework, the C++ language, and the Project. The level of detail here is necessary to understand the description of the Implementation section and does not go much deeper; an inquisitive reader is encouraged to go through the K tutorial.

2.1 K framework

In the $\mathbb{K}$ framework, languages are described in a style commonly known as *operational semantics*. For any language L, when L is given a particular definition D in $\mathbb{K}$ framework, then D assigns to every program in L a transition system $(Cfg, \rightarrow)$. Here Cfg denotes a set of program configurations and $\rightarrow$ is a binary relation over Cfg; the relation is called a „transition relation" and its elements are „transitions".

The configurations are not abstract, but they have an internal structure, which depends on the definition D. For imperative languages, the configurations may consists of a „program" part and a „data" part.

2.1.1 Terms

The program part can be represented as a term. $\mathbb{K}$ allows to define a multisorted algebraic signature (S, Σ); closed terms over this signature forms a (multisorted) term algebra. One may then choose a particular sort $s \in \Sigma$ and declare the set of all programs to be the set of all closed terms of the sort s.

Sorts are defined using `syntax` keyword. The definition

```
syntax H
syntax H ::= world()
syntax H ::= hello(H,H)
```

defines sort `H` and a nullary constructor `world` and a binary constructor `hello` of that sort. Uknown sorts on the left hand side of the operator `::=` are automatically defined, and when defining multiple constructors for one sort, the right hand sides can be chained with the operator `|`. The definition above is therefore equivalent to the following definition:

```
syntax H ::= world() | hello(H,H)
```

Furthermore, K allows the sort constructors to be given in an infix syntax and to contain „special" symbols. Therefore, instead of

```
syntax G ::= unit() | add(G, G) | inv(G)
```

it is possible to write

```
syntax G ::= ".G" | G "+" G | "-" G
```

to describe the signature of groups. K also supports subsorting; the following definition makes the sort `Int` a subsort of the sort `Real`:

```
syntax Real ::= Int
```

```
syntax AExp  ::= Int | Id
               | AExp "+" AExp
syntax BExp  ::= Bool
               | AExp "<=" AExp
               | "!" BExp
               | BExp "&&" BExp
               | "(" BExp ")"
syntax Block ::= "{" "}"
               | "{" Stmt "}"
syntax Stmt  ::= Block
               | Id "=" AExp ";"
               | "while" "(" BExp ")" Block
               | Stmt Stmt
syntax Pgm   ::= Stmt
```

Figure 2.1: A definition of an algebraic signature of an imperative language.

K is distributed together with a basic library; together they provide a number of pre-defined sorts, including `Id` `Int`, `Bool`, `List`, and `Map`. The `Id` is a sort of C-style identifiers, the other names are rather self-explanatory. With use of these sorts, an algebraic signature of an imperative language can be defined as in Figure 2.1. From the definition, *arithmetic expressions* (represented by the sort `AExp`) are integers, identifiers, and sums of other arithmetic expressions; the arithmetic expressions can be compared to create *boolean expressions*, and so on.

```
configuration <T>
                <k> $PGM:Pgm </k>
                <state> .Map </state>
              </T>
```

Figure 2.2: A definition of the initial configuration of the language Imp.

2.1.2 Configurations

A part of a program configuration can be stored in a *cell*, which can be thought of as a labeled multiset [5]. Cells may contain terms of a given sort or other cells. In a source code of a language definition in $\mathbb{K}$, cells are written in an xml-style notation.

Figure 2.2 contains a snippet of such source code. The keyword `configuration` here defines three cells (`T`, `k` and `state`), a single structure for all configurations, and an initial configuration. Every cell in the definition has some content: the `state` cell contains `.Map`, which has a constructor of sort `Map`, representing an empty map; the `k` cell contains a term of sort `Pgm` consisting of a variable with name `$PGM`, and the `T` cell contains the other two cells. The initial configuration for program `P` is just like that, except that the variable `$PGM` is replaced by a term, representing the program `P`.

We have said earlier that the programmer may choose a sort, whose terms will represent the „program" part of the configuration; that happens in the configuration definition. In this particular semantics of the language Imp, the „data" part of the configuration is represented by a term of sort `Map`.

2.1.3 Parsing

It can be seen from the Figure 2.1 that definitions of algebraic signatures in $\mathbb{K}$ looks similar to definitions of context-free grammars using a BNF notation. This is not a coincidence; in fact, $\mathbb{K}$ allows to specify the *concrete syntax* with use of special *attributes*. This subsection describes some of the parsing facilities $\mathbb{K}$ provides; the Project does not use them, but the description enables the reader to easily experiment with the provided examples.

```
module ARITH-SYNTAX
  syntax AExp  ::= Int
                 | AExp "/" AExp              [left]
                 > AExp "+" AExp              [left]
                 | "(" AExp ")"               [bracket]
  syntax Pgm ::= AExp
endmodule
module ARITH
  imports ARITH-SYNTAX
  configuration <T>
                  <k> $PGM:Pgm </k>
                  <ret exit="">0</ret>
                </T>
endmodule
```

Figure 2.3: File `arith.k`. Note the `ret` cell with `exit` attribute.

Before going further, let us note that in the time of writing there exist two implementations of $\mathbb{K}$: UUIC-$\mathbb{K}$, developed by University of Illinois in Urbana-Campaign, and RV-$\mathbb{K}$, developed by RuntimeVerification, inc. There are some differences between the two; one of them is that the RV-$\mathbb{K}$ requires the configuration to contain a cell with `exit` attribute: the cell contains the value returned by `krun`. All examples in this chapter from this point further work with both implementations of $\mathbb{K}$, modulo this „exit" difference.

Figure 2.3 contains a definition of a simple language of arithmetic expressions. The definition contains a few things we have not described yet. First, it is split into two modules; the `ARITH-SYNTAX` module contains everything which is needed to generate a parser, while the `ARITH` module contains everything else. Second, one of the sort constructors is separated from the previous one by a > sign; that causes the preceding productions in the concrete syntax grammar to have a higher priority than the following ones and thus to bind tighter. Third, some constructors have *attributes* attached; the `left` attributes causes a binary constructor to be left-associative, and the `bracket` attribute means that the corresponding unary constructor should be parsed as a pair of brackets.

When stored in a file with name `arith.k`, the definition may be compiled using UIUC-𝕂 (the first command) or using RV-𝕂 (the second one):

```
$ kompile arith.k
$ kompile -O2 arith.k
```

The `krun` command will use the module `ARITH-SYNTAX` to generate the parser and the module `ARITH` to generate an interpreter. The names of the modules used can be specified by command-line arguments; if they are not specified (as above), 𝕂 infers them from the filename. Both parser and interpreter are stored in a directory `arith-kompiled`.

The compiled definition can be used to parse and execute a program written in the arithmetic language.

```
$ cat addition.arith
1 + 2 + 3
$ krun addition.arith
<T> <k> 1 + 2 + 3 </k> <ret> 0 </ret> </T>
```

The command parses the given source file, creates an initial configuration, walks in the generated transition system until it reaches a terminal configuration, and pretty-prints it. The generated transition system contains only one configuration (the initial one), as the current language definition of Arith does not have any semantic rules.

The pretty-printing implies that the abstract syntax is *unparsed* back to the concrete one; therefore, from the output above one can not conclude that the file was parsed correctly, as changing the left associativity to the right one (by changing `left` attributes to `right` ones) would not alter the output. However, `krun` can be instructed to output a textual version of its internal representation:

```
$ krun --output addition.kast
<T>‘(‘<k>‘(‘_+__ARITH-SYNTAX‘(‘_+__ARITH-SYNTAX‘(
#token("1","Int"),#token("2","Int")),#token("3","Int"))),
‘<ret>‘(#token("0","Int")))
```

The output is easily readable for a machine; to parse it, the generated concrete syntax parser is not needed. It is less readable for humans, though. From the output one can easily get

```
+( +( 1, 2 ), 3 )
```

simply be keeping only the content of `k` cell, removing superfluous characters and adding spaces. One can see that this expression,

```
$ cat simple.arith
5/2 + (1 + 3) / 2
$ krun simple.arith
<T> <k> 5 / 2 + ( 1 + 3 ) / 2 </k> <ret> 0 </ret> </T>
$ krun --output kast simple.arith
<T>‘(‘<k>‘(‘_+__ARITH-SYNTAX‘(‘_/__ARITH-SYNTAX‘(
#token("5","Int"),#token("2","Int")),‘_/__ARITH-SYNTAX‘(‘
_+__ARITH-SYNTAX‘(#token("1","Int"),#token("3","Int")),
#token("2","Int")))),‘<ret>‘(#token("0","Int")))
```

Figure 2.4: Another example of a program in the language of arithmetic expressions. The last output corresponds to the expression `+( /( 5, 2 ), /( +( 1, 3 ), 2 ) )`

when interpreted as in prefix-notation, correspond to the content of `addition.c`. From a different example on Figure 2.4 one can see that the division has a priority over addition and the parenthesis bind the tightest.

2.1.4 Rules

So far, the definition of language Arith was able to generate only trivial transition systems with one configuration and no transitions. To generate configurations from the initial one, $\mathbb{K}$ provides a concept of *semantic rules*. In their simplest version, the rules have the form of $\varphi \Rightarrow \psi$, where φ, ψ are *patterns* - configurations with free variables, where every variable free in ψ have to be free in φ. We say that a pattern φ *matches* a concrete configuration *Cfg*, when φ may be turned into *Cfg* by substituting free variables of φ with concrete terms. In that case we say that the variables *bind* to the corresponding terms. When φ matches a configuration *Cfg*, a transition is generated into a new configuration Cfg', which is the result of substituting the free variables of ψ with the bound terms.

Basic rules

For example, the module `ARITH` of the language Arith may be extended with the following rule:

```
rule <T><k>I1:Int + I2:Int</k><ret>R:Int</ret></T>
```

```
  => <T><k>I1 +Int I2</k><ret>R</ret></T>
```

The left hand side of the rewriting operator (=>) contains three free variables (I1, I2, R), all of which required to have a sort Int. The variables I1, I2 are used as the two parameters of the constructor +, while the variable R represents the content of the ret cell. On the right hand side, I1 and I2 are given as parameters to built-in function +Int, which implements the addition of two integers; the variable R is used in the same place as on the left side, thus leaving the ret cell unchanged.

The compiled definition takes two numbers and adds them together:

```
$ kompile arith.k
$ cat simple.arith
1 + 2
$ krun simple.arith
<T> <k> 3 </k> <ret> 0 </ret> </T>
```

Here krun again printed the final configuration. It is possible to print an *i*-th configuration with use of the depth switch:

```
$ krun --depth 0 simple.arith
<T> <k> 1 + 2 </k> <ret> 0 </ret> </T>
$ krun --depth 1 simple.arith
<T> <k> 3 </k> <ret> 0 </ret> </T>
```

Local rewriting

The semantic rule above applies the rewriting operator (=>) to whole configuration, although it changes only the content of the k cell. $\mathbb{K}$ implements a concept called *local rewriting*, which allows language definition developers to use the rewriting operator inside a cell or inside a term. With use of local rewriting, the above rule can be written as:

```
rule <T> <k> I1:Int + I2:Int => I1 +Int I2 </k>
         <ret> R:Int </ret> </T>
```

The rule can be even more simplified with use of an anonymous free variable, denoted by an underscore (_):

```
rule <T> <k> I1:Int + I2:Int => I1 +Int I2 </k>
         <ret> _ </ret> </T>
```

```
rule <k> I1:Int + I2:Int => I1 +Int I2 </k>
```

Figure 2.5: A rule for addition of two integers.

It is also possible to use the rewriting operator multiple times in one semantic rule:

```
rule <T> <k> (I1:Int => 0) + (I2:Int => I1 +Int I2) </k>
         <ret> _ </ret> </T> requires I1 =/=Int 0
rule <T> <k> (0 + I:Int) => I </k> <ret> _ </ret> </T>
```

A `requires` clause causes a rule to apply only if a certain condition holds; in this particular example, the rule should not apply if the variable `I1` is bound to zero. Without the clause, the rule would be able to apply indefinitely, without any effect.

Configuration abstraction

Semantic rules usually need to be aware only of a few configuration cells. In $\mathbb{K}$, the semantics rules have to mention only the cells they really need to mention. The $\mathbb{K}$ tool then, from the definition of configuration, infers the context in which such local rewriting takes place. The rule for addition can be written as in Figure 2.5. Such semantics rules are not only shorter and easier to write, but they are also independent on most of the configuration; when the structure of configuration changes, the rules may remain the same. This feature is called *configuration abstraction*.

2.1.5 Computations

When creating a language definition, it is often needed to compute something first, and then use the result of the computation to compute something else. In $\mathbb{K}$, the notion of *first* and *then* is formalized in terms of *computations* and their *chaining*. Computations have the sort `K`; all user-defined sorts are automatically subsorted to `K`. Computations can be composed using the `~>` constructor, which is associative. The sort `K` has a nulary constructor `.K`, which represents an empty computation and acts as a unit with respect to `~>`. Thus `K` with `~>` and `.K` form a monoid.

In practice, the monoidal structure means that any term consisting of computations and the `~>` constructor behave as a *chain* of computations, with hidden empty computations everywhere inside. One can then insert a computation `c` to any position in the chain simply by rewriting an empty computation on that position `c`. It also allows replacing the rule 2.5 with

```
rule (<k> I1:Int + I2:Int => I1 +Int I2) ~> _</k>
```

without losing any existing behavior. If the old rule matches a configuration with a term `C` in the `k` cell, then the new rule matches the term `C ~> .K`, as the anonymous variable binds to the empty computation. But the new rule matches also the term `C`, because due the monoidal structure, configurations `C` and `C ~> .K` are equal. On the other hand, the new rule adds some behaviors, because it matches not only when the `k` cell contains exactly an addition of two integers, but also when it contains any sequence of computations, where the first computation is an addition of two integers. The first computation in the list is called *the top*. The parenthesis in the new rule are needed, as the constructor `~>` binds tighter then the operator `=>`.

The rules of the form

```
rule <k> SomeRewritingHere ~> _ </k>
```

can be also written as

```
rule SomeRewritingHere
```

For example, the rule

```
rule (<k> I1:Int + I2:Int => I1 +Int I2) ~> _</k>
```

can be also written as

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

2.1.6 Strictness and evaluation strategies

How to compute the sum of three numbers, say $1 + 2 + 3$? In the language Arith, the + constructor is left-associative, so the natural approach is to compute $1 + 2$ first, which yields a result r, and then to compute $r + 3$. Although it is possible to write such rules manually, $\mathbb{K}$ provides a number of tools, which enable the programmer to describe such computations on higher level of abstraction.

```
syntax KItem ::= holdAddR(AExp) | holdAddL(Int)
rule E1:AExp + E2:AExp => E1 ~> holdAddR(E2)
     requires notBool isInt(E1)
rule E1:Int + E2:AExp => E2 ~> holdAddL(E1)
     requires notBool isInt(E2)
rule E2:Int ~> holdAddL(E1:Int) => E1 + E2
rule E1:Int ~> holdAddR(E2) => E1 + E2
rule I1:Int + I2:Int => I1 +Int I2
```

Figure 2.6: Heating and cooling rules, written manually.

Sort predicates

For every sort `Srt` 𝕂 automatically generates a *sort predicate* `isSrt` of sort `Bool`. The predicate takes any term and returns `true` if the term is of sort `Srt`, otherwise returns `false`. So far, the definition of Arith language can not evaluate nested expressions, as the built-in function `+Int` can be applied only on terms of sort Int. Moreover, the left side of the rule requires the involved terms to be of sort `Int` With use of sort predicate, a rule

```
rule <l> E1:AExp + E2:AExp => -1 </l>
requires notBool isInt(E1) orBool notBool isInt(E2)
```

can be added into the language definition; the rule rewrites a sum of two terms of sort `AExp` to `-1`, unless both of the terms have the sort `Int`. The `notBool` and `orBool` are built-in functions; `isInt` is a sort predicate for the sort `Int`.

Heating/cooling

Using the information above, a programmer may use the piece of 𝕂 code in Figure 2.6 to evaluate nested expressions. The idea here is that the left addend is evaluated first, then the right addend is evaluated and finally the two integers are added using the built-in function `+Int`. One way to interpret the rules is that the first two rules extract an unevaluated expression out of the addition, which creates a hole in the term, the extracted expression is then evaluated, and the third and forth rule plug the evaluated expression back into the hole. The constructors `holdAddR` and `holdAddL` are used to represent the original term without the extracted subterm; the sort `KItem` is a bit special, but

it is subsorted to the sort `K` as any user-defined sort. The process of extracting a subterm is known as *heating*, while the opposite process is *cooling*.

Evaluation contexts

Heating and cooling rules are very common; they are also tedious to write manually. In $\mathbb{K}$, the idea of evaluating a certain subterm first can be expressed in terms of *evaluation context*. With use of a keyword `context`, the $\mathbb{K}$ code

```
syntax KItem ::= holdAddR(AExp)
rule E1:AExp + E2:AExp => E1 ~> holdAddR(E2)
     requires notBool isInt(E1)
rule E2:Int ~> holdAddL(E1:Int) => E1 + E2
```

can be equivalently expressed as:

```
context HOLE:AExp + E:AExp [result(Int)]
```

The `context` declaration means exactly that: whenever the top of a `k` cell contains an addition of two `AExp`s and the first one is not of sort `Int`, extract the first one, push it on the top of the `k` cell and replace the addition with some placeholder; when the extracted subterm gets evaluated to `Int`, plug it back to the original context.

With use of context, the code on Figure 2.6 can be equivalently expressed as

```
context HOLE:AExp + E:AExp [result(Int)]
context I:Int + HOLE:AExp [result(Int)]
rule I1:Int + I2:Int => I1 +Int I2
```

which significantly reduces the amount of code. When compiled, the generated interpreter correctly evaluates the expression $3 + 1 + 7$; the evaluation progress is shown in Figure 2.7.

Strictness attributes

$\mathbb{K}$ defines a sort `KResult`, which represents results of computations. When no `result` attribute is given to a context declaration, the declaration behaves as with `[result(KResult)]`. When writing a larger language definition, it is convenient to identify the sorts of desired results and subsort them to `KResult`.

```
3 + 1 + 7
( 3 + 1 ) ~> #freezer_+__ARITH-SYNTAX1_ ( 7 )
4 ~> #freezer_+__ARITH-SYNTAX1_ ( 7 )
4 + 7
11
```

Figure 2.7: The progress of evaluating the expression $3 + 1 + 7$. An ith line represents the configuration after i computational steps, starting from zero. For brevity, only the content of the `k` cell is shown.

When defining a sort constructor, the constructor may be tagged with an attribute `strict`. In that case $\mathbb{K}$ generates a context declaration for every parameter of the constructor. With use of the `strict` attribute, a definition of the language Arith can be given using only a few lines (Figure 2.8).

2.1.7 Functions

——————— TODO: Features to cover here:

- modules
- Konstruktory a atribut [function]

Differences between UIUC-K and RV-K

- In RV-K, the configuration needs to have a cell with an `exit` attribute.
- KServer. Only one instance can be running. Uses NailGun. What does work and what does not.
- kompile must be run with -O2 switch

2.2 C++

Standard, see [1].

```
module ARITH-SYNTAX
  syntax AExp  ::= Int
                 | AExp "/" AExp [left, strict]
                 > AExp "+" AExp [left, strict]
                 | "(" AExp ")"  [bracket]
  syntax Pgm ::= AExp
endmodule
module ARITH
  imports ARITH-SYNTAX
  configuration <T>
                  <k> $PGM:Pgm </k>
                  <ret exit=""> 0 </ret>
                </T>
  syntax KResult ::= Int
  rule I1:Int + I2:Int => I1 +Int I2
  rule I1:Int / I2:Int => I1 /Int I2 requires I2 =/=Int 0
endmodule
```

Figure 2.8: A definition of the language Arith.

3 $\mathbb{K}$ C/C++ semantics overview

3.1 Build notes

The project of $\mathbb{K}$ C/C++ semantics is hosted on GitHub[1]. It can be built easily by simply following the build instructions in the repository; however, the process deserves a few things to be mentioned here.

- At the time of writing, there are currently two implementations of $\mathbb{K}$ framework: UIUC-$\mathbb{K}$[2], developed by University of Illinois at Urbana–Champaign, and RV-$\mathbb{K}$[3], developed by RuntimeVerification Inc; the latter is the one used by the project. The RV-$\mathbb{K}$ builds and runs without problems, with two minor exceptions:
  - It does not support examples included in the $\mathbb{K}$ tutorial.
  - It requires flex[4] to be present in the system.
- The project uses clang as a library to parse C++ sources; however, the currently required version 3.9 is a bit outdated.
- The officially supported operating system is Ubuntu 16.4 LTS; however, it works without problems on Fedora 26.
- The build can take up to thirty minutes on this text's author's machine.

3.2 Basic usage

Main user interface of the project consists of a script `kcc` [5], which implements a compiler based on the C/C++ semantics. The script mimincs the interface of gnu gcc compiler and supports many of gcc's command-line parameters. It is therefore possible to use it to build programs instead of gcc; however, the generated executables are many times slower then the ones built using gcc.

---

1. `https://github.com/kframework/c-semantics`
2. `https://github.com/kframework/k`
3. `https://github.com/runtimeverification/k`
4. `https://github.com/westes/flex`

```
$ cat hello.C
extern "C" int puts(char const *s);
int main() {
        puts("Hello␣world");
}
$ kcc hello.C -o hello
$ ./hello
Hello world
```

Figure 3.1: A "hello world" program.

In 𝕂 framework, a semantics of programming language L assigns to every program in L a set of program configuration with a transition system over them. The executable file `hello` generated by `kcc` is a perl script, which walks through the transition system in a step-by-step manner. The walk starts in an initial configuration and ends in a configuration for which no further transition is defined. The script then examines the final configuration and stops, possibly printing an error message in case the walk ended abnormally.

It is possible to specify an exact number of computational steps to take by setting the variable `DEPTH` to the desired value. In this particular example, the executable is able to print only an incomplete portion of the text; then the error message is printed.

```
$ env DEPTH=675 ./hello
Hello woError: Execution failed.
```

The full list of accepted environment variables can be obtained by setting the environment variable `HELP`.

3.3 Under the hood

3.3.1 How `kcc` works?

But how do `kcc` and the generated perl script work internally? 𝕂 framework provides a tool `kompile` in order to compile a programming language semantics, and another tool `krun`, which is used to run

a program against the semantics of the program's language. More precisely, `krun`

1. takes a program and *compiled* programming language semantics as an input,
2. parses the program,
3. creates an initial configuration from the parsed program,
4. traverses the induced transition system from the initial configuration until a terminal configuration is reached,
5. and outputs the terminal configuration.

The `krun` tool can be also configured to traverse the transition system in a different manner, e.g. to perform a search for a specific *pattern*, or to stop the traversal after specified number of steps.

The project of C/C++ semantics internally consists of multiple K semantics, all of which need to be compiled with `kompile`. When `kcc` is invoked on a C++ program, a clang-based tool `clang-kast` is used to convert each source file into K's internal representation (K AST). Every converted file is then individually used as an input to `krun` with *static C++ semantics*; the resulting terminal configuration can be thought of as an equivalent of an object file. The outputs are then joined together with runtime library and the result is wrapped in a generated Perl script. The script then, when executed, runs the linked program using `krun` and *executable C/C++ semantics*, possibly passing its command line arguments to the program.

3.3.2 Structure of configurations

In K, a language semantics is defined by specifying an abstract syntax, a structure of configurations over the syntax, and rewrite rules over the configurations and the syntax.

Abstract syntax The abstract syntax is defined using `syntax` keyword and BNF-like notation. For example, the source file `semantics/c11/library/io.k` contains a syntax declaration

```
syntax KItem ::= sendString(Int, String)
```

```
configuration
<global/>
<result-value> 139:EffectiveValue </result-value>
<T><exec>
  <threads color="yellow" thread="">
    <thread multiplicity="*" color="yellow" type="Map">
      <thread-id color="yellow"> 0 </thread-id>
      <k color="green">
        loadObj(unwrapObj($PGM:K))
        ~> initMainThread
        ~> pgmArgs($ARGV:List)
        ~> callMain(/* left out */)
      </k>
      <thread-local/>
  </thread></threads>
</exec></T>
```

Figure 3.2: A source code of a simplified configuration definition (see `semantics/c11/language/execution/configuration.k` for full version).

which declares all terms with label `sendString`, one parameter of sort `Int`, and one of sort `String`, to be of sort `KItem`. Terms of that sort represent computational items; however, terms with label `sendString` are never parsed as a part of the program and their purpose is purely semantic.

Configurations Configurations are defined as shown in listing 3.2; from the example, a number of observations can be made:

- The definition consists of a `configuration` keyword followed by a list of nested cells; the cells does not need to be enclosed in a top cell.
- Configurations consist of multiple cells; a cell can be thought of as a labeled multiset [5]. Cells may contain other cells, integers, lists, maps and arbitrary terms (including program ASTs).
- A cell can be included in its supercell a multiple times; the multiplicity can be adjusted with the attribute `multiplicity`.

- Cells are usually defined in place of their use, but they may also be defined elsewhere, which is the case for `global` and `thread-local`.
- Computations are contained in the `k` cell.
- The content of a cell in its definition definition specifies the cell's initial value. For example, the `k` cell here initially contains a sequence of computations, parametrized by parsed program and command-line arguments.
- 𝕂 allows each cell to have a color, and provides a tool, `kdoc`, to generate a colorful documentation from a language definition. The tool is broken, though.

Rewriting rules Rewriting rules specify the transition relation on configurations. Rules usually consists of a `rule` keyword, followed by a list of configuration cells, and a `requires` clause. Inside the cells, a rewriting may take place, which is then denoted by „`=>`". If there are more rewritings inside one rule, they all happen at once; in the C/C++ semantics, this is often used when declaring an entity.

The following rule, which gives semantics to `sendString`, can serve as an example.

```
rule <k> sendString(FD::Int, S::String)
      => #putc(FD, ordChar(firstChar(S)))
      ~> sendString(FD, butFirstChar(S))
     ...</k>
     <options> Opts::Set </options>
     requires lengthString(S) >Int 0
     andBool notBool (NoIO() in Opts)
```

The rule says: „Every configuration, in which

1. there is an `options` cell containing a set not containing an `NoIO()` term, and in which
2. there is also a `k` cell having on its top a `sendString` item parametrized with an integer and a nonempty string `S`,

```
rule <k> sendString(FD::Int, S::String) => .K ...</k>
     <options> Opts::Set </options>
     requires lengthString(S) <=Int 0
     orBool (NoIO() in Opts)
```

can be rewritten to another configuration by rewriting the `sendString` item to `#putc` of the first character, followed by (`~>`) the same `sendString` item, but without the first character of the string." This way the rule encodes the following piece of semantic information: „To send a nonempty string means to send its first character and then to send the rest, unless the IO is disabled".

3.3.3 Can we see it?

When the executables generated by `kcc` are run in an environment with variable `VERBOSE` set, they produce the final configuration in text form to standard output. For the „hello world" program above (listing 3.1), the konfiguration produced by the command

```
$ env VERBOSE=1 DEPTH=675 ./hello
```

has about 600 kilobytes. The excerpt in the figure 3.3 contains a thread with two *computational items* on the top of its `K` cell. From that point, if the execution had not been stopped, the first item would have sent the rest of the `Hello world` string to stdout, then it would have been removed and the second item would have been proccessed.

```
‘<generatedTop>‘(...
  ‘<thread>‘(
    ‘<thread-id>‘(#token("0","Int")),
    ‘<k>‘(
      sendString(
        #token("1","Int"),
        #token("\"rld\\n\"","String")
      ) ~>
      sent(
        #token("1","Int"),
        #token("\"Hello world\\n\"","String")
      )...
    )...
  )...
)
```

Figure 3.3: An excerpt of generated configuration. Large portions of the configuration were replaced by elipsis (...) and the formatting (whitespaces) was added manually.

4 Implementation

This chapter focuses on those parts of the C/C++ semantics project, which are related to the goal and contribution of this thesis. The chapter describes the implementation of the main features, shows relations between the implementation, standard and general architecture of the semantics, and highlights some aspects of the C++ language one may perhaps oversee when using the language as a programmer. The last section of this chapter then gives a short evaluation of the implementation.

4.1 Enumerations

In order to implement enumerations, several parts of the semantics had to be modified. The translation tool clang-kast was slightly modified to produce AST nodes for enumeration declarations; to process the declarations in the semantics, a new file was added to static semantics and a new set of cells was added to common part of configuration. It was also needed to implement enumerator lookup, which required addition of a few cells to configuration and a slight modification of some of the name lookup rules. The semantics was to some extent already prepared to work with enumerations, and some of the relevant rules (e.g. for conversions) needed no change. To ensure correctness of implementation, several test cases were added to the test suite. Overall, most of the modifications were additive, and only little of the existing code needed to be changed. During the implementation process, a few minor bugs were discovered and fixed.

```
enum E { A = 5, B = A + 3 };
```

Figure 4.1: A declaration of an *unscoped enumeration* `E` with *unscoped enumerators* `A` and `B`.

```
enum class E { A = 5, B = A + 3 };
```

Figure 4.2: A declaration of a *scoped enumeration* `E` with *scoped enumerators* `A` and `B`.

```
enum E1 : int {};
enum class E2 : int {};
enum class E3 {};
```

Figure 4.3: Declarations of enumerations with fixed underlying type.

```
enum E1 : char;
enum class E2 : unsigned int;
enum class E3;
```

Figure 4.4: *Opaque-enum-declaration*s.

```
enum E;
```

Figure 4.5: Not a valid declaration, because „opaque-enum-declaration declaring an unscoped enumeration shall not omit the enum-base" (7.2:2)

```
enum class E : short { A, B = A + 2, C };
```

Figure 4.6: Prior the closing bracket, the enumerators `A`, `B` and `C` have a type `short`.

```
enum E { A, B, C = (char)255, D };
```

Figure 4.7: Prior the closing bracket, the enumerators `A`, `B` and `D` have an unspecified type, the enumerator `C` has type `char`.

```
enum E { A, B = A, C = +A };
```

Figure 4.8: Quiz: what are the types of B and C prior the closing bracket?

```
enum { A };
enum : char { B, C };
```

Figure 4.9: „The optional identifier shall not be omitted in the declaration of a scoped enumeration"(7.2:2), however, it may be omitted in the declaration of an unscoped enumeration.

Declaration

Enumeration declaration, including the *opaque declaration*, is implemented in module `CPP-DECL-ENUM` of the semantics. Every enumeration declaration is processed as follows:

1. A new `cppenum` cell is created in the current translation unit. An error is reported if there already exist an enumeration with the same name, unless the declaratation is opaque.
2. The enumeration being declared is added to environment, so that it could be later looked up.
3. For full declarations, the enumerators are processed in the order of their declarations. Processed enumerators are stored in sub-cells of the `cppenum` cell; for unscoped enumerations, the enumerators are also added to the scope surrounding the enumeration declaration.
4. For unscoped enumerations without a fixed underlying type, the set of enumeration values and the underlying type is computed and stored in the `cppenum` cell.

For declarations of enumerations with no fixed underlying type, the standard keeps one aspect of the declaration unspecified. Prior the closing bracket of the enumerator declaration [1], the type of an enumerator with initializer is the type of the initializer, and type of an enumerator without initializer is the type of previous enumerator, whenever possible. If there is no initializer specified for the first enumerator, the type of the enumerator is unspecified; it is also unspecified for enumerators (without initializers) whose value does not fit

1. And inside the enumeration declaration in particular.

into the type of previous enumerator. For example, in the declaration on figure 4.10, the type of enumerator `A` in the declaration of `B` is not specified, and therefore the value of `B` is not specified, too. Similarly, the value of enumerator `D` on most platforms does not fit to unsigned char, which is the type of enumerator `C`, and its type is thus unspecified. Note that the types of enumerators are unspecified only inside of the declaration of the enumeration, i.e. prior the closing bracket of the declaration. Type of every enumerator after the complete declaration is always the type of the enumeration, so this unspecified behaviour is usually not a problem in practice, unless one writes code similar to the one in image 4.10.

```
enum E {
        A, B=sizeof(A), C=(unsigned char)255, D
};
```

Figure 4.10: Declaration of an enumeration with unspecified values of enumerators.

However, the semantics should be aware of this behaviour. Many real-world programs use enumerations whose enumerators does not have initializers, since the value of the enumerators is by default numbered from zero. Earlier versions of the semantics caused the semantic-based compiler kcc to stop the compilation whenever an undefined or unspecified behaviour was encoutered, which would be an unforunate thing to do for such programs. For this reason, the project maintainer added an error-reporting and recovery support to the semantics [2]. The current version of the semantics issues a warning, whenever this unspecified behaviour occur. Ideally, the warning would be suppressed if the unspecified type is never used, but this enhancement was not implemented.

Enumerator lookup

The name of an enumerator can be referred to using the scope resolution operator applied to a name of the enumeration. This was

2. https://github.com/kframework/c-semantics/commit/584fa6ff4a90aca45de99d6b210177258ebd96d4

implemented easily using only a few rules in the (static) semantics. Furthemore, the enumerators of an unscoped enumeration are declared in the scope immediately containing the declaration of the enumeration. To implement this, I have decided to add a few new cells, which map names of enumerators of enums defined in the surrounding scope to their corresponding type. The lookup then reuses rules from the previous case. It might be possible to implement the lookup even without those extra cells, but the implemented solution seemed to be simplier.

The rules for enumerator lookup also have to consider the context in which the lookup is performed. As noted earlier, it is mandated by the standard that the types of declared enumerators are different inside the declaration then after it. One may find that surprising; however, this is needed in order to easily create enumerator initializers, which depends on values of previous enumerators of the same enumeration, as it is ilustrated in figure 4.11. If the type of the enumerator `A` in the initializer of the enumerator `B` was the type of the enumeration (as it is after the declaration), the initializer expression would be ill-formed, as in C++ enumerations are not implicitly convertible to arithmetic types. Thus, an enumerator prior the closing bracket has always an integral type.

```
enum F {
        A=1, B=A+2
};
```

Figure 4.11: An enumerator depends on value of previous enumerator.

4.2 Generalized constant expressions (constexpr)

TBD

Bibliography

[1] Working draft, standard for programming language c++, November 2014.

[2] Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015.

[3] Chucky Ellison. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois, July 2012.

[4] Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Rosu. Rv-match: Practical semantics-based program analysis. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *LNCS*, pages 447–453. Springer, July 2016.

[5] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of c. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 336–345. ACM, June 2015.

[6] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015.

[7] Grigore Roşu. From rewriting logic, to programming language semantics, to program verification. In *Logic, Rewriting, and Concurrency: Essays Dedicated to José Meseguer*, volume 9200 of *LNCS*, pages 598–616. Springer, September 2015.

[8] Grigore Rosu. K - a semantic framework for programming languages and formal analysis tools. In Doron Peled and Alexander Pretschner, editors, *Dependable Software Systems Engineering*, NATO Science for Peace and Security. IOS Press, 2017.