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 [7] (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.

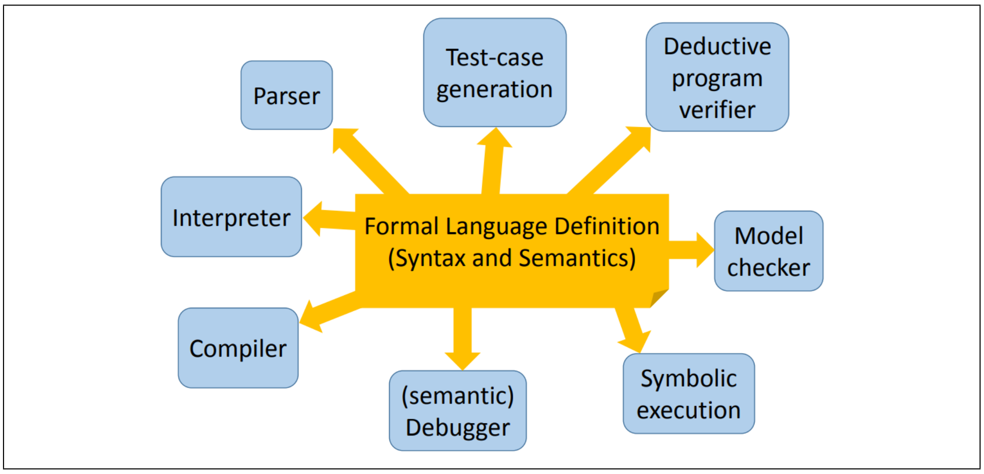

Figure 1.1: The idea behind 𝕂. Adopted from [6].

𝕂 framework has been successfully used to give formal semantics to a varienty of languages, including Java [1], Python, Javascript [5], and C [2, 4], 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" [3]. 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

K And Operational semantics 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 a simple imperative language, similar to the language IMP defined in the K tutorial [**?**], the configurations may consists of a „program" part and a „data" part.

Terms The program part can be represented as a term. $\mathbb{K}$ allows to define a multisorted algebraic signature (S, Σ) using a BNF-like notation; 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)
```

one may write

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

to describe a sort of groups.

𝕂 is distributed together with a basic library; together they proved a number of pre-defined sorts, including `Int`, `Bool`, `Id`, `List`, and `Map`.

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

Figure 2.1: A definition of an algebraic signature of the IMP.

As can be seen from Figure 2.1, definitions of the algebraic signatures in 𝕂 looks similar to definitions of grammars. The first two lines define

Of course, the programmer may choose to use a configuration with only one cell and use

Configurations In 𝕂, a part of the program configuration can be stored in a *cell*, which can be thought of as a labeled multiset [4]. In

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

Figure 2.2: A definition a configuration of a language Imp.

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 sort `Map` an represents a 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`.

Program configurations for small-step semantics can consist of a „program" part and a „data" part. Consider simple imperative language IMP. In textbook small-step SOS, the set of program configurations *Cfg* can be defined as $Pgm \times \Sigma$, where *Pgm* is a set of programs and Σ is the set of functions from variable names to (syntactic) integers; thus, configurations are tuples. In $\mathbb{K}$, configurations are made of semantic components (called *cells*) nested in a tree-like manner; the cells contain values of various predefined (numbers, lists, ...) or user-defined syntactic sorts. Every configuration is then a (ground) term.

The configuration for IMP can be defined as in figure 2.3. The `k` cell contains the „program" part - a list of computations yet to be computed. The `state` cell holds a map from variable names to values; the syntactic sorts of its keys and values are not specified. The `state` cell initially holds an empty map, while `k` cell initially contains the abstract syntax tree of whole program.

The abstract syntax is defined using BNF notation inside `syntax` declarations (see figure 2.4). $\mathbb{K}$ also supports attaching various attributes; some of them (e.g associativity attributes) are needed to sup-

configuration ⟨T⟩ ⟨k⟩ \$PGM:Pgm ⟨/k⟩ ⟨state⟩ .*Map* ⟨/state⟩ ⟨/T⟩

Figure 2.3: Definition of program configuration in the syntax of 𝕂.

syntax AExp ::= *Id* | *Int*

syntax BExp ::= *Bool*
| AExp "⟨=" AExp [**seqstrict**]

syntax Stmt ::= "skip"
| *Id* "=" AExp
| "if" BExp "then" Stmt "else" Stmt "fi" [**strict** (1)]

Figure 2.4: Part of IMP syntax in 𝕂. The syntactic categories *Id*, *Int* are predefined as well as some operations on them.

port automatic generation of concrete syntax parser, some of the other have a semantic impact (strictness attributes etc). The user-defined syntactic constructs can be used anywhere in configuration, not just in `k` cell.

The transition system is induced by rewriting rules; those have the form of $l \Rightarrow r$ `if` b, where l and r are matching-logic patterns [1], and b is a first-order formula over free variables of l. Whenever the pattern l *matches* a configuration such that b evaluates to *true*, the rule gets applied and generates a new configuration. One such rule in the IMP language may be

rule ⟨T⟩⟨k⟩ X:*Id* => I ···⟨/k⟩
⟨state⟩··· X |−> I ···⟨/state⟩⟨/T⟩

Many rules care about only a small fraction of the configuration. 𝕂 framework therefore implements a mechanism called „configuration abstraction", which allows rules to mention only the necessary configuration cells; 𝕂 then desugars them. For example, in the above rule it is possible to remove the $< T >$ cell.

1. Patterns are basically terms with free variables. Details about matching logic are given in **??**.

Those three kinds of $\mathbb{K}$ declarations are sufficient to give a semantics to given language; however, $\mathbb{K}$ provides some additional construct, which makes writing the semantics easier. Those will be discussed in latter sections

Features to cover here:

- heating/cooling rules
- modules
- contexts
- KResult
- Konstruktory a atribut [function]
- Atribut [strict]

2.2 C++

3 𝕂 C/C++ semantics overview

3.1 Build notes

The project of 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 𝕂 framework: UIUC-𝕂[2], developed by University of Illinois at Urbana–Champaign, and RV-𝕂[3], developed by RuntimeVerification Inc; the latter is the one used by the project. The RV-𝕂 builds and runs without problems, with two minor exceptions:
  - It does not support examples included in the 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` [4], 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 on 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 [4]. 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] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.