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

1 Introduction **1**

2 Background **5**
- 2.1 *K framework* 5
  - 2.1.1 Terms 5
  - 2.1.2 Configurations 7
  - 2.1.3 Implementations of 𝕂 7
  - 2.1.4 Parsing 8
  - 2.1.5 Rules 10
  - 2.1.6 Computations 12
  - 2.1.7 Strictness and evaluation strategies 14
  - 2.1.8 Functions 16
- 2.2 C++ 18
  - 2.2.1 Enumerations 18
  - 2.2.2 Constant expressions 24

3 Project overview **27**
- 3.1 *Basic usage* 27
- 3.2 *Under the hood* 28
  - 3.2.1 Can we see it? 29

4 Implementation **33**
- 4.1 *Enumerations* 33
- 4.2 ———————— 33
- 4.3 *Generalized constant expressions (constexpr)* 36

5 Evaluation **39**

List of Tables

List of Figures

1.1 The idea behind 𝕂. Adopted from [7]. *1*

2.1 A definition of an algebraic signature of an imperative language. *6*
2.2 A definition of the initial configuration of the language Imp. *7*
2.3 File `arith.k`. Note the `ret` cell with an `exit` attribute. *8*
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 ) )` *10*
2.5 A rule for addition of two integers. *12*
2.6 Heating and cooling rules, written manually. *15*
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. *16*
2.8 A definition of the language Arith. *17*
2.9 An `eval` function. *17*
2.10 A declaration of an *unscoped enumeration* `E` with *unscoped enumerators* `A` and `B`, followed by a declaration of a variable `e` of type `E`, initialized by the enumerator `B`. This code is valid in both C++14 and C99. *18*
2.11 A declaration of an enumeration with the same enumerator as in an already existing enumeration. *19*
2.12 A declaration of a *scoped enumeration* `E` with *scoped enumerators* `A` and `B`. *20*
2.13 The type of the variable `x` is the underlying type of `E`. *20*
2.14 Explicitly specified underlying type. *20*
2.15 A packed enum. *21*
2.16 The reachability of the last line depends on whether `E` has a fixed underlying type or not. *22*
2.17 Unscoped enumerations can be implicitly converted to `int`. *23*
2.18 C++ does not allow an implicit conversion from int to enum. However, this is a valid C code. *23*
2.19 *Opaque-enum-declaration*s. *24*

2.20 Not a valid declaration, because „opaque-enum-declaration declaring an unscoped enumeration shall not omit the enum-base” [1, §7.2:2] *24*
2.21 A „wild” expression in a context that requires a constant expression. *25*
2.22 ... *25*
3.1 A "hello world" program. *28*
3.2 Running a compiled program for an exact number of computational steps. *28*
3.3 An excerpt of a generated configuration. Large portions of the configuration were replaced by elipsis (. . .); the formatting (whitespaces) was added manually. *30*
3.4 A semantics rule from file(...). *31*
4.1 Declaration of an enumeration with unspecified values of enumerators. *35*
4.2 An enumerator depends on value of previous enumerator. *36*

1 Introduction

Writing correct software is hard. Although formal methods for software verification are being developed, there are only a few high quality tools on the market. In order to create a production-ready tool based on a particular set of formal methods and targeting a particular language, it is necessary to understand the formal methods, the precise semantics of the target language, and possibly some other things. Researchers and engineers often work in teams, and various teams often implement different techniques for the same language; because of that, a platform, which would enable component reuse and separation of concerns, might significantly improve the productivity of the tool developers.

𝕂 framework is one such platform. It 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.

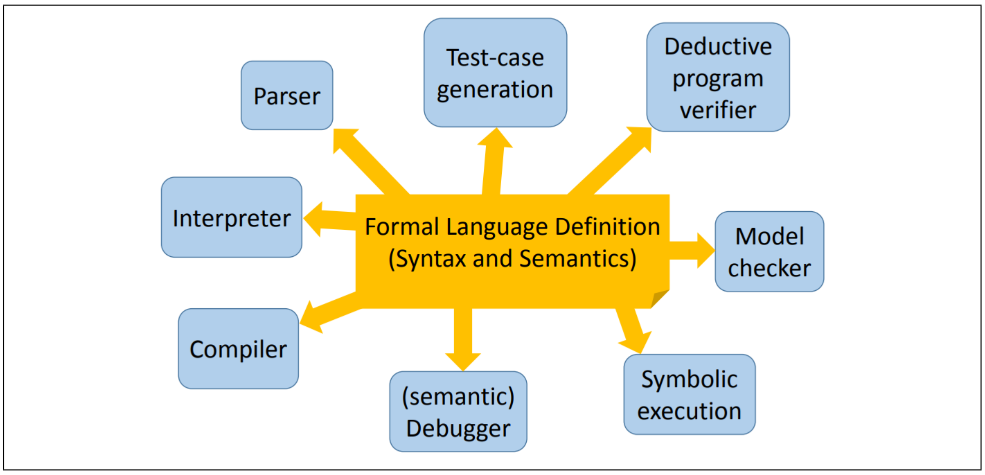
Parser
Test-case generation
Deductive program verifier
Interpreter
Formal Language Definition (Syntax and Semantics)
Model checker
Compiler
(semantic) Debugger
Symbolic execution

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 the C++ programming language. The C++ support is also the focus of this thesis. From this point on, when we write „Project", we mean this project of defining C/C++ language semantics in 𝕂.

The thesis aimed to implement two language features: *enumerations* and *constant expressions*. When we decided to implement the enumerations, we thought that it is a relatively simple language feature: they do not have any special runtime behavior, do not interfere much with other language features, and they exist in the language from its beginning. The *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). However, careful reading of the standard [1] revealed us that the enumerations behave in many ways which we considered to be surprising, as we did not know them from common programming practice.

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. Because of that, we expected from the beginning that in order to implement the constant expressions, a more fundamental change to the semantics have to be done.

The two features were successfully implemented; the implementation of enumerations was merged into the upstream, the implementation of constant expression is likely to be merged soon. It went out that the they play together rather nicely: C++ requires enumerators to be initialized with constant expressions; therefore, the implementation of enumerations was used to test the implementation of constant expressions. In addition, we fixed several bugs in the project, including one hard-to-debug nondeterministic behavior, and implemented a *zero initialization* of class types.

The purpose of this text is to describe the implementation of the aforementioned features. It is organized as follows. First, the 𝕂 framework is described in such level of detail, which is enough for the reader to understand the implementation of the features and which enables him to experiment with simple language definitions. Second, basic concepts of the C++ language are outlined, followed by more in-depth discussion of the selected language features. Third, the general architecture of the Project is described; fourth, the implementation of the language features is described. The final sections of the text contains an evaluation of the implementation, with a discussion of possible future work.

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 $(Cfg, \to)$. Here Cfg denotes a set of program configurations and $\to$ 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 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 language definition, the „data" part of the configuration is represented by a term of sort `Map`.

2.1.3 Implementations of 𝕂

Before going further, let us note that in the time of writing there exist two implementations of 𝕂: UUIC-𝕂[1], developed by University of Illinois in Urbana-Campaign, and RV-𝕂[2], developed by RuntimeVerification, inc. The former is intended to be the reference implementation, the latter is optimized for RV's tools based on the c-semantics. RV-𝕂

---

1. `https://github.com/kframework/k`
2. `https://github.com/runtimeverification/k`

```
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 an `exit` attribute.

differs from UIUC-𝕂 in a few aspects. One of the differences is that the RV-𝕂 requires the configuration to contain a cell with an `exit` attribute: the cell contains the value returned by `krun`. Because of that, it does not supports the examples included in the official 𝕂 tutorial. However, all examples in this chapter from this point further work with both implementations.

2.1.4 Parsing

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

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")))
```

```
$ 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 ) )`

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, 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.

2.1.5 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. 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 <k> I1:Int + I2:Int => I1 +Int I2 </k>
```

Figure 2.5: A rule for addition of two integers.

```
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>
```

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 𝕂, the semantics rules have to mention only the cells they really need to mention. The 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*.

2.1.6 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 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
```

2.1.7 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, 𝕂 provides a number of tools, which enable the programmer to describe such computations on higher level of abstraction.

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

```
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.

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 𝕂, the idea of evaluating a certain subterm first can be expressed in terms of *evaluation context*. With use of a keyword `context`, the 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
```

```
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.

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

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`.

When defining a sort constructor, the constructor may be tagged with an attribute `strict`. In that case 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).

2.1.8 Functions

In the preceding text, a number of built-in functions was used (e.g. `andBool`, `=/=Int` and `/Int`). Functions can appear in side conditions (the `requires` clause) or right hand sides of the rewriting operator. Their application does not create any transitions in the transition system and unlike constructors, functions cannot be pattern-matched. Custom functions can be defined with `syntax` keyword and a `function` attribute; their behavior is defined using the rewriting operator. See Figure 2.9 for an example.

```
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.

```
syntax Int ::= eval(AExp) [function]
rule eval(I:Int) => I
rule eval(E1:AExp + E2:AExp) => eval(E1) +Int eval(E2)
rule eval(E1:AExp / E2:AExp) => eval(E1) /Int eval(E2)
```

Figure 2.9: An `eval` function.

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

Figure 2.10: A declaration of an *unscoped enumeration* `E` with *unscoped enumerators* `A` and `B`, followed by a declaration of a variable `e` of type `E`, initialized by the enumerator `B`. This code is valid in both C++14 and C99.

2.2 C++

Standard, see [1].

2.2.1 Enumerations

Many languages implement enumerated types as a means for programmers to define a finite set of related values. Enumerations in the C++ language are based on enumerations from the C language. In this section we give a brief presentation of the C++ enumerations, as defined in [1]. Note that this text is concerned with what the language allows, rather than what is consider to be a good practice. For a discussion about the latter we point the reader to the relevant section of CppCoreGuidelines[3].

Unscoped enumerations

Figure 2.10 shows a basic C-style declaration of an enumeration `E`, with subsequent declaration of an object of the enumeration's type. Both declarations are valid in both C and C++. In C++14, the C-style enumerations are called *unscoped enumerations*. An enumeration may contain an unbounded number of named values, called *enumerators*. Here, `E` contains two enumerators: `A` and `B`. In C++14, enumerators of an unscoped enumeration are called *unscoped enumerators* and they are declared in the scope which contains the enumeration declaration (or, in the terms of the standard, the *enum-specifier* [1, §7.2:11]). Therefore, it is possible to refer to them simply as in this example.

---

[3] `http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`

```
enum E{B};
// enum F{B}; // an error
int main() {
    enum E{A,B};
    return (int)B;
}
```

Figure 2.11: A declaration of an enumeration with the same enumerator as in an already existing enumeration.

Because of this simplicity it is not possible to declare two unscoped enumerations with the same enumerator, in the same scope. It is still possible to create the second enumeration in a different scope, though; the enumeration may even have the same name, as in the Figure 2.11. If that happens, the old enumerator is *shadowed* with the new one, but it is still possible to access the old one using the scope resolution operator (i.e. `::B` in this example).

When a name of an enumeration (or a struct) is used in C, for example in a declaration of a variable of that enumeration type, it is necessary to use the `enum` keyword (as in Figure 2.10). In C++, this is not necessary, as a name of an enumeration (`enum-name`) denotes a type (it is a `type-name`, see [1, Annex A.1]). It is still possible to do it the C way, though; the type names accompanied by the keyword `enum` or `struct` (or similar) are called *elaborated type specifiers*[1, §3.4.4].

Scoped enumerations

The revision C++11 introduced `scoped enumerations`. They are declared using the phrase `enum class` or `enum struct`, and enumerators of a scoped enumeration are called *scoped enumerators*. Scoped enumerations differ from the plain enumerations in a number of aspects; the most significant difference is what they have their name for: scoped enumerators are declared in the scope of their enumeration[1, §7.2:11]. Because of that, the enumerators have to be referred to using the scope resolution operator (at least outside the enumeration declaration, see Figure 2.12). However, unscoped enumerators can be referred to this way, too ([1, §5.1.1/11]).

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

Figure 2.12: A declaration of a *scoped enumeration* `E` with *scoped enumerators* `A` and `B`.

```
#include <type_traits>
enum E {A, B, C};
std::underlying_type_t<E> x = 5;
```

Figure 2.13: The type of the variable `x` is the underlying type of `E`.

Underlying type

Every enumeration type has associated an `underlying type`. It is an integral type, which can represent all values of the enumeration; it also defines the size of the enumeration[4]. The standard library provides the programmer a way to find out the underlying type of an enumeration (Figure 2.13).

An enumeration's underlying type can be explicitly specified [1, §7.2:5]. This is achieved by appending a colon followed by a name of an integral type right after the `enum-name` in the declaration, as shown in Figure 2.14. The most obvious reason why would programmer want to specify an enum's underlying type is in order to limit the enumeration's size, perhaps because of hardware constrains. Before C++11, this was often achieved through an extension of a particular compiler; for example, the GNU gcc compiler provides a special attribute `packed`, which causes an enumeration's size to be as small as possible (Figure 2.15).

---

[4] Technically, it is more complicated: `https://stackoverflow.com/q/47444081/6209703`

```
enum E1 : char {A,B};
enum class E2 : short {C,D};
```

Figure 2.14: Explicitly specified underlying type.

```
enum E { A, B } __attribute__((packed));
```

Figure 2.15: A packed enum.

If the underlying type of a scoped enumeration is not explicitly specified, it is automatically defined to `int`. All scoped enumerations, as well as unscoped enumerations with explicitly specified underlying type, are said to have a *fixed underlying type*. This stands in contrast to unscoped enumeration without explicitly specified underlying type. Their underlying type is not fixed, but it is an *implementation-defined* integral type, which can represent all values of the enumeration [1, §7.2:7]. That also suggests that the underlying type is not known prior the closing brace of the declaration.

The fact that the standard does not specify the exact underlying type for enumerations whose underlying type is not fixed is also the reason why in pre-C++11 (including C11), the enumerations whose size matters to the programmer have to be declared with use of compiler-specific language constructs (as in Figure 2.15). Without that, the compiler could simply choose the underlying type (and thus the size) of the enumeration (with respect to some standard-given restrictions); in practice, the chosen underlying type is often `int`.

Values of an enumeration

The standard also defines the set of values of an enumeration; those are the values, which can be written or read from an object of the enumeration's type. For an enumeration whose underlying type is not fixed, the set of values limits the choice of the underlying type; it is defined such that the values of all its enumerators and all values between the enumerators are also values of the enumeration. The definition is rather technical, though; an inquisitive reader shall find it in [1, §7.2:8]. For an enumeration with fixed underlying type, the values of the enumeration are exactly the values of its underlying type.

This behavior may be a source of confusion for programmers who do not know this; in past, it certainly was for the author of this text. The programmer may write a code similar to the one in Figure 2.16, which assumes that the value of `e` has to be either `E::A` or `E::B`. This assump-

```
enum class E { A, B };
E giveMeAnEnum();
int foo(E e) {
        E e = giveMeAnEnum();
        switch(e) {
                case E::A: return 3;
                case E::B: return 42;
        }
        /* Unreachable? Well... */
}
```

Figure 2.16: The reachability of the last line depends on whether `E` has a fixed underlying type or not.

tion may be justified, if the programmer knows the set of values which may be returned from the function `giveMeAnEnum`. The programmer may try to go one step further and say: „If the function `giveMeAnEnum` tries to return something different than `E::A` or `E::B`, than the behavior is already undefined in *that* point; therefore, we do not need to handle that situation in `foo`." Such a reasoning is flawed, because the function `giveMeAnEnum` may return e.g `static_cast<E>(-117)`, which is guaranteed to be well-defined, because `-117` is a valid value for `int`, which is the underlying type of `E` here. The programmer's reasoning would be valid if the enumeration `E` was a plain unscoped enumeration without explicitly specified underlying type (i.e if we removed the `class` keyword). We will not go into detail here, as the reasoning would depend on the technical parts of [1, §7.2:8].

Enumerators

Declaration of an enumerator may contain an optional *initializer*. If the initializer is not specified, the value of the enumerator is zero if it is the first enumerator, otherwise it is the value of the previous enumerator plus one. When an enumeration is fully defined („following the closing brace of an enum-specifier"), the type of each enumerator is the type of the enumerator [1, §7.2:5]. For example, in the declaration of variable `e` in the Figure 2.12, the type of the expression `E::B` is `E`.

This is different from C, where enumerators have the type `int`. Still, unscoped enumerations can be implicitly converted to `int` [1, §7.2:10],

```
enum E { A };
void foo(int){}
void goo() {
    foo(A);
    foo(E::A);
    foo(E::A+1};
}
```

Figure 2.17: Unscoped enumerations can be implicitly converted to `int`.

```
enum E { A };
void foo(int x) { E e = x; }
```

Figure 2.18: C++ does not allow an implicit conversion from int to enum. However, this is a valid C code.

so they behave somewhat similarly to C. This conversion does not work for scoped enumerations. The other conversion (from integer types to enumerations) is not defined neither for scoped nor unscoped enumerations (Figure 2.18).

An initializer of an enumerator may refer to previously declared enumerators of the same enumeration (see 2.12). Inside the enumeration declaration (prior its closing brace), the type of its enumerators is not the type of the enumerator, but always an integral type[5]. Because of the stronger-than-C type system, in some cases it would be even inconvenient if the types of enumerators were types of their enumerations. For example, the declaration in Figure 2.12 declares an enumerator `B` with the initializer `A + 3`. If the type of `A` in this expression were `E`, then the expression would be ill-formed, because the scoped enumeration does not convert to int, so it cannot be added to an int.

Incomplete types

Some types cannot be used to define objects; those are called *incomplete* [1, §3.1:5]. A type may be incomplete at one point and complete at other point in a translation unit. Incomplete types are still useful; they

---

[5] We discuss this in more detail in Chapter 4

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

Figure 2.19: *Opaque-enum-declaration*s.

```
enum E;
```

Figure 2.20: Not a valid declaration, because „opaque-enum-declaration declaring an unscoped enumeration shall not omit the enum-base" [1, §7.2:2]

may be used in function declarations to denote its parameters and return values, to define objects of pointer type or references. There are two categories of incomplete types: void types (e.g. `volatile void`) and incompletely-defined types. It is not allowed to apply the `sizeof` operator to an incomplete type. A programmer may choose to use incompletely-defined types to limit the number of dependencies between header files. An enumeration whose underlying type is not fixed is incomplete until the closing brace of its declaration.

Opaque enumerations

Enumerations with fixed underlying type can be declared *opaque*, that is, without the list of enumerators (see Figure 2.19). An enumeration declared using an *opaque-enum-declaration* is a complete type (it is not an incomplete type); therefore, it can be used almost as a fully declared enumeration, except that its enumerators are not available. Such an enumeration needs to have a fixed underlying type (see Figure 2.20). An opaque enumeration can be fully redeclared later in the program [1, §7.2:3].

2.2.2 Constant expressions

In C++, expressions are used almost everywhere. However, not every kind of expression can be used everywhere: one can hardly imagine the code in Figure 2.21 to be valid. The expressions used in an enum declaration or as a `case` labels of a `switch` statement have to be

```
enum class E { A = getchar() };
```

Figure 2.21: A „wild" expression in a context that requires a constant expression.

```
constexpr int const & id(int const &x) { return x; }
void test() {
        int const a = 7;
        //constexpr int const & b = id(a); // 1
        constexpr int c = id(a); // 2
        static int const d = 8;
        constexpr int const &e = id(d); // 3
}
```

Figure 2.22: ...

computable in the compilation time. Those kind of expressions are generally known as *constant expressions*; they are defined in [1, §5.20].

The intuition behind the standard

The standard defines a couple terms with respect to constant expressions; this subsection attempts to give the reader an intuition which is behind them. The term *core constant expression* ([1, §5.20.2]) is used to represent an expression, which can be evaluated easily, using only the information the compiler has in the time of compilation. The term *constant expressions* ([1, §5.20:5]) denotes a core constant expression, whose value „make sense" in the compile time.

The Figure 2.22 can serve as an example here. The function `id` takes one parameter, which is a reference to a constant integer, and returns it back. The expression `id(a)` is then a *core constant expression*, as it can be easily evaluated (its value is equivalent to the value of the expression `a`). It is not a *constant expression*, because its value refers to a non-static local variable, and the memory location of a (non-static) is unknowable in the compile time (such memory location does not even exist in the execution time, or the variable may be instantiated multiple times in recursive calls). Therefore, its value cannot be used as the initializer of a constexpr reference (1).

How is then possible that it may be used in the declaration (2)? Note that the variable is not initialized simply by the expression `id(a)`, but with the rvalue-to-lvalue conversion of the result of that `id(a)`; such composed expression is a `constant expression`. In other words, the declaration of `c` does not concerned with the identity of the object it is initialized with, but only with its value - and that value is known in the compile time, as the variable `a` is initialized with the constant expression `7`. The declaration (3) works because the variable `d` was declared `static`.

3 Project overview

The project of C/C++ semantics in 𝕂 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.

- The projects depends on RuntimeVerification's implementation of the 𝕂. The RV-𝕂 builds and runs without problems, with a minor exception: it requires the lexical parser flex[2] to be present in the system.
- The project uses clang as a library to parse C++ sources; the currently required version 3.9 is a bit outdated. It is not a problem, as clang can be built easily and the Project can be configured to use clang installed in any directory.
- The officially supported operating system is Ubuntu 16.4 LTS, which already contains prebuilt clang libraries in the required version. But the projects works on Fedora 26 without any problems.
- The building process can take up to thirty minutes on this text's author's machine.

3.1 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 mimics the interface of the 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 than the ones built by gcc.

It has been said in Chapter 2 that in the 𝕂 framework, a semantics of a 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` (as shown in Figure 3.1) is

---

[1] `https://github.com/kframework/c-semantics`

[2] `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.

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 whenever 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 an error message is printed (Figure 3.2). The full list of accepted environment variables can be obtained by setting the environment variable `HELP`.

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

Figure 3.2: Running a compiled program for an exact number of computational steps.

3.2 Under the hood

The Project internally consists of a clang-based tool `clang-kast` and many 𝕂 modules. The modules are composed into three language definitions: the definition of C11 translation semantics, the definition of C++14 translation semantics and the definition of C11/C++14 execution semantics. The language definitions has to be compiled

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

by `kompile`; the compilation of both `clang-kast` and the language definitions is driven by a Makefile.

When `kcc` is invoked on a C++ program, a clang-based tool `clang-kast` is used to convert each source file into the 𝕂's internal representation (𝕂 AST). Every converted file is then individually used as a program, which is interpreted (using the 𝕂 tool `krun`) by the *C++ translation 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. When the script is executed, it interprets (with `krun`) the linked program using the *C11/C++14 execution semantics*, possibly passing the script's command line arguments to the program.

3.2.1 Can we see it?

When an executable generated by `kcc` is run in an environment with the variable `VERBOSE` set, the executable prints the final configuration in a text form to the standard output. For the „hello world" program above (Figure 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 computational item (`sendString`) would have sent the rest of the `Hello world` string to the standard output, then it would have been removed and the second computational item (`sent`) would have been processed.

The project contains some rules, which give semantics to the `sendString` term. The rule in the Figure 3.4 is one of them; it basically 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". The Chapter 2 contains everything needed to understand

```
`<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 a generated configuration. Large portions of the configuration were replaced by elipsis (...); the formatting (whitespaces) was added manually.

this rule, except a small detail: the elipsis (...) in the `k` cell behaves exactly as `~> _` and expresses the notion of „we do not care what is in the rest of this cell". We provide an explanation of the rule only as a means for the reader to check his understanding. 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`,

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."

```
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)
```

Figure 3.4: A semantics rule from file(...).

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

4.2 ——————

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 the C++ translation 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.

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 into the type of previous enumerator. For example, in the declaration on figure 4.1, 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 4.1.

However, the semantics should be aware of this behaviour. Many real-world programs use enumerations whose enumerators does not

---

[1] And inside the enumeration declaration in particular.

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

Figure 4.1: Declaration of an enumeration with unspecified values of enumerators.

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 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, we 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 the rules from the previous case.

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 4.2. 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.

---

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

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

Figure 4.2: An enumerator depends on value of previous enumerator.

4.3 Generalized constant expressions (constexpr)

The Project consists of the definition of translation semantics, and the execution semantics. Both have different configurations. Before we started the work, the translation semantics was already able to evaluate simple arithmetic or boolean C++ expressions (like `1 + 1 == 2`. However, the translation semantics was not able to evaluate more complex expresions, and function calls in particular. To evaluate function calls, the configurations of execution semantics contains a set of cells which represents *call stack*, but the configuration of translation semantics does not contain anything like that.

When designing the support for generalized constant expressions, several design alternatives seemed to be viable. One of them was to extend the translation semantics with new rules and configuration cell which would allow to evaluate the not-supported-yet expressions; another alternative was to reuse some of the rules from the execution semantics and modify the configuration accordingly. We decided to first try to reuse as much of existing code as possible, with the perspective that if that fails, we will fall back to other design alternative or try to explore the design space more carefully. It turned out that we could reuse almost everything; although the changes were quite extensive, there were only a few newly added rules and cells, and most of them had the character of „move this there", „use this file from the execution semantics in the static semantics" and „add a `require` rule here".

5 Evaluation

What works and what does not. The bugs we have found in compilers. Ambiguity or contradictions in the standard. The one thing we implement the same way gcc and clang does, but technically is incorrect.

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.