CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Formal Semantics for Javalette in the $\mathbb{K}$ framework

Master's thesis in Computer Science - Algorithms, Languages, and Logic

BURAK BİLGE YALÇINKAYA

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

A Formal Semantics for Javalette in the $\mathbb{K}$ framework

BURAK BİLGE YALÇINKAYA

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

A Formal Semantics for Javalette in the $\mathbb{K}$ framework

BURAK BİLGE YALÇINKAYA

© BURAK BİLGE YALÇINKAYA, 2022.

Supervisor: Magnus Myreen, Department of Computer Science and Engineering
Advisor: Rikard Hjort, Runtime Verification
Examiner: Andreas Abel, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

A Formal Semantics for Javalette in the $\mathbb{K}$ framework

BURAK BİLGE YALÇINKAYA
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis is about developing an executable formal semantics for Javalette in the $\mathbb{K}$ framework. Javalette is an imperative programming language. Its syntax is formally specified using BNF (Backus-Naur form) notation, but it does not have a formal semantics. The semantics of the language is informally documented in English. Javalette has several extensions that enrich the language's syntax and semantics with new types, statements, and expressions. $\mathbb{K}$ is a toolset for programming language design and implementation. It provides a specification language for formally defining syntax and semantics. From these definitions, $\mathbb{K}$ automatically generates various tools such as parsers, interpreters, model checkers, and deductive verifiers. The purpose of this project is to develop a complete formal semantics for the Javalette language, design an architecture for extending the language modularly and implement language extensions, find and resolve undefined behaviors in the language, and use the formal semantics to develop an input fuzzer for testing Javalette programs and implementations.

Keywords: Formal semantics, K framework, programming languages, Javalette.

Acknowledgements

I would like to thank my supervisor Magnus Myreen and advisor Rikard Hjort for their endless support. Thank you for recommending this project idea and guiding me with interesting discussions throughout the project. I also would like to thank Runtime Verification and all the contributors for creating and open-sourcing the $\mathbb{K}$ framework. Lastly, I want to thank my family and friends for their support and encouragement.

Burak Bilge Yalçınkaya, Gothenburg, May 2022

Contents

List of Figures

1

Introduction

Programming is the act of precisely telling computers what to do using a programming language. By this precise nature of computing, programming languages have to be fully defined unambiguously, in contrast to human languages. Hence, writing specifications is a critical part of programming language design and development.

Programming language specifications consist of two main parts: syntax and semantics. These two components describe how programs look and behave, respectively. When designing a programming language, formally specifying the language syntax is a common practice. The syntax is usually defined using the conventional Backus-Naur form (BNF) notation, and this specification forms a basis for parsers and documentation. Using BNF is highly desirable since there are well-developed tools, such as GNU Bison and Yacc, for automatically generating parsers from BNF files[9][10]. On the other hand, instead of using formal semantics, the semantics of the language is usually described informally using a natural language or pseudo-code together with example programs.

A similar situation applies to the Javalette language. Javalette is an imperative programming language designed for teaching compiler construction [5]. Its syntax is formally specified using BNF notation, but the semantics is vaguely described in English. There is also a collection of test programs that contributes to the specification of the language. These programs are used to prevent ambiguities and misunderstandings as well as to test Javalette compilers. Compiler implementors refer to the test suite to better understand the language. Compilers that pass all the test cases are assumed to be correct.

Informal descriptions of the semantics of a language can not be precise enough and usually carry ambiguities due to the limitations of natural languages. In the absence of formal semantics, compiler and interpreter implementors have to rely on their understanding of the language. For this reason, there can be inconsistencies between different implementations of the same language, and reasoning about programs becomes harder. On the other hand, rigorously written formal semantics help developers to communicate clearly and understand the language better. Furthermore, it is possible to use formal specifications for the automatic generation of programming language tools.

Despite all these reasons, formal semantics is not widely used due to high devel-

opment cost and not having direct benefits as in BNF. The $\mathbb{K}$ framework aims to balance these cost and benefits. $\mathbb{K}$ is a framework for designing and implementing programming languages. Language designers can formalize their languages in $\mathbb{K}$ and then build various tools around the language, such as parsers, interpreters, compilers, and verifiers. A complete formal semantics written in $\mathbb{K}$ is enough to get an interpreter for free. The automatic generation of such tools closes the gap between defining and implementing a programming language [12].

This project aims to write a formal semantics for Javalette in the $\mathbb{K}$ framework, and to answer the following research questions:

1. What aid can we get from $\mathbb{K}$ when writing formal semantics for a language which has not been formalized before? What advantages does it offer for spotting unspecified behaviors?

2. How difficult is it to add extensions to a formalization in $\mathbb{K}$? Can we modularly grow a programming language with new syntax and semantics?

3. How can we benefit from $\mathbb{K}$ for testing implementations of a language? How can we generate interesting test cases using $\mathbb{K}$?

1.1 Contributions

The main contribution of this project is K-Javalette[1], an executable formal semantics for the Javalette language. Being executable, the semantics also yields a reference interpreter for the language. The interpreter is tested against the test suite and it passes all test cases. Chapter 3 explains the implementation of K-Javalette.

In addition, we provide language extension modules for arrays and pointers. Chapter 4 explains how language extensions are defined in $\mathbb{K}$, and shows that it is feasible to modularly extend a programming language defined in $\mathbb{K}$.

We also identify underspecifications in Javalette semantics. Chapter 5 documents how K-Javalette fixes these issues, and provides example programs to describe expected behaviors.

Lastly, we propose an application of K-Javalette in randomized testing of compilers. Chapter 6 describes how our formal semantics in $\mathbb{K}$ is used for input fuzzing.

[1] K-Javalette is publicly available on GitHub [2].

2

Background

This section aims to provide the background information needed to understand the following chapters. Section 2.1 is an overview of the Javalette language. Section 2.2 introduces the core concepts of the $\mathbb{K}$ framework that are used to develop K-Javalette.

2.1 Javalette

Javalette [5] is an imperative programming language with a C-like syntax. It was designed for educational purposes and is used in the Compiler Construction course at Chalmers University of Technology. The core language has only primitive data types, basic control structures and statements for imperative programming, and procedures. It does not have heap-allocated data or dynamic data structures. Figure 2.1 shows an example program that demonstrates variables, procedures, and loops. It prints numbers from 1 to 10.

```
1  int main () {
2      printString("Counting...");
3      printNumbers(10);
4      return 0;
5  }
6
7  void printNumbers (int n) {
8      int i;
9      i = 1;
10     while (i <= n) {
11         printInt(i);
12         i++;
13     }
14 }
```

Figure 2.1: A simple Javalette program

The core of the Javalette is a very simple programming language yet it can be extended with various features. Some possible extensions proposed in the course are arrays, dynamic data structures, pointers, and object orientation with or without dynamic dispatch [1].

2.2 The $\mathbb{K}$ Framework

$\mathbb{K}$ [12] is an executable semantics framework for designing and implementing programming languages. Given a language definition, $\mathbb{K}$ is capable of creating parsers, interpreters, and program verification tools automatically. These tools are use cases of formal semantics, but they also support the development of formal semantics, for instance, by running the generated interpreter to test the semantics. It has been used for specifying formal semantics of several major real-world languages such as C, JavaScript, and Java [7, 11, 6].

$\mathbb{K}$ specifications consist of *syntax*, *configuration*, and *rule* declarations. Configurations represent the program state and rule declarations describe how the language constructs are executed. The $\mathbb{K}$ compiler (`kompile`) takes a language specification and outputs an interpreter for the language.

$\mathbb{K}$ provides two backends for different purposes: LLVM and Haskell. The default backend is the LLVM backend, which is an efficient and optimized backend for fast concrete execution. It compiles the language definition and creates an interpreter using LLVM. The other backend is the Haskell backend. It provides more advanced features for symbolic execution and formal reasoning. The Haskell backend also supports concrete execution but it is not as performant as the LLVM backend [14, 3]. In this project, we use the LLVM backend to generate an interpreter.

In this section, we go through the basics of $\mathbb{K}$ by implementing a basic calculator language for integers.

2.2.1 Syntax declarations

For syntax declarations, $\mathbb{K}$ has a BNF-like notation extended with annotations. Figure 2.2 defines an expression syntax for arithmetic operations on integers.

```
1  module ARITH-SYNTAX
2      imports INT-SYNTAX
3
4      syntax Exp ::= Int
5                   | Exp "+" Exp      [left, seqstrict, add]
6                   | Exp "*" Exp      [left, seqstrict, mult]
7                   | Exp "/" Exp      [left, seqstrict, mult]
8
9      syntax priorities mult > add
10 endmodule
```

Figure 2.2: Syntax for the integer calculator

For parsing integers, the built-in `INT-SYNTAX` module is used. The `left` annotation makes the second production left-associative. The `seqstrict` annotation automatically creates rules to evaluate the sub-expressions. The user-defined `add` and `mult` labels are later used in the priority declaration. Since `add` has a lower priority than `mult`, an addition cannot be an immediate child of a multiplication or division.

2.2.2 Configurations

To give semantics to the language, we first need to define the state of a running program. Configurations are XML-like nested structures organized as *cells*. Each cell stores a semantic component such as environments, stores, call stacks, or threads.

Suppose, in our integer calculator, we also want to count the number of multiplication operations performed. Figure 2.3 is the configuration declaration for our calculator.

```
configuration
    <T>
        <k> $PGM:Exp </k>
        <counter> 0:Int </counter> // ":Int" is a cast to Int
                                   // to prevent ambiguity
                                   // between Int and Exp
    </T>
```

Figure 2.3: Configuration for the integer calculator

In the above configuration, `<k/>` is a special cell that contains a series of computations sequenced with the sequencing operator (`~>`). Its initial value will be an expression (`Exp`) parsed from the input. The `$PGM` variable is a special variable denoting the parsed input to the interpreter. The second cell in the configuration is `<counter/>`, which is initialized to 0. The `<T/>` cell is the top-level cell that holds other cells. Figure 2.4 shows the initial configuration for the input string "1 + -2 * 3". The `<k/>` cell contains the parsed expression, and the counter is initially 0. The `<k/>` cell ends with "`~> .`", which denotes the end of the computation.

```
<T>
    <k>
        1 + -2 * 3 ~> .
    </k>
    <counter>
        0
    </counter>
</T>
```

Figure 2.4: Initial configuration for "1 + -2 * 3"

Configurations can be deeply nested. We may want to have variables in our language. Then, we can add an environment map to the configuration using the built-in `Map` data structure (Figure 2.5).

2.2.3 Rules

Rule declarations give semantics to the language constructs by defining rewrite operations on configuration cells. Rules can refer to relevant configuration cells for reading or writing. Figure 2.6 shows a rule that states when there is an addition of

```
configuration
    <T>
        <k> $PGM:Exp </k>
        <counter> 0:Int </counter>
        <env> .Map </env> // .Map is an empty Map
    </T>
```

Figure 2.5: Configuration with an environment

two integers (`A` and `B`) on top of the `<k/>` cell, replace it with the sum of the numbers using the built-in addition operator (`+Int`). The `=>` operator denotes the rewrite operation. The pattern inside the parenthesis matches the first computation in the cell. The `Rest` variable matches the rest of the sequence and leaves it unchanged.

```
rule <k> (A:Int + B:Int => A +Int B) ~> Rest </k>
```

Figure 2.6: Addition of two integers

Since it is a common pattern, there is a syntactic sugar for matching the beginning of a sequence. The `"..."` pattern denotes that there can be more elements in the sequence.

```
rule <k> A:Int + B:Int => A +Int B ... </k>
```

Figure 2.7: Addition of two integers - simplified

```
<T>
    <k> 1 + 2 ~> . </k>
    <counter> 0 </counter>
</T>
```

```
<T>
    <k> 3 ~> . </k>
    <counter> 0 </counter>
</T>
```

Figure 2.8: State change after addition

Note that the addition rule only mentions the `<k/>` cell since addition does not involve the `<counter/>` cell. This feature is called *configuration abstraction*. On the other hand, the multiplication rule uses both cells. It performs the multiplication and also increments the counter. The rewrite operator can appear multiple times anywhere in the configuration.

```
rule <k> A:Int * B:Int => A *Int B ... </k>
     <counter> C => C +Int 1 </counter>
```

Figure 2.9: Multiplication of two *Int*s increments the counter

```
1 rule <k> A:Int / B:Int => A /Int B ... </k> requires B =/=Int 0
```

Figure 2.10: Division by non-zero

Which rule to apply is determined by pattern matching the configuration with rules. In addition to pattern matching, it is possible to use side conditions with the `requires` keyword. The rule in Figure 2.10 is only applied when `B` is not zero

All of the above rules assume that operands are `Int` literals, so they are not able to operate on more complex expressions. For more complex expressions, we need to define how to recursively evaluate sub-expressions. Strictness annotations (`strict` and `seqstrict`) in syntax definitions automatically generates necessary rules to take out operands (redexes) and plug them back when they are evaluated. When there are multiple operands, `seqstrict` is used to specify the evaluation order from left to right. To enable strictness annotations, we need to define the `KResult` sort to tell the compiler what sorts do not need to be evaluated further.

```
1 syntax KResult ::= Int
```

Figure 2.11: Integers are results

2.2.4 Functions

Sometimes it is more convenient to define a part of the semantics as a function. In $\mathbb{K}$, functions are defined by using the `function` attribute in the syntax declaration. When function symbols appear in the configuration, they are immediately evaluated using associated rules. Figure 2.12 shows an example function that takes a `List` and returns the first element.

```
1 syntax KItem ::= listHead(List)   [function]
2 rule listHead(ListItem(X) Xs:List) => X
3 // listHead(.List) is undefined
```

Figure 2.12: A non-total function

As in this example, functions do not always return a value. To define a total function, the `functional` attribute is used. The compiler performs a check for non-exhaustive patterns in total functions.

```
1 syntax Int ::= listLength(List)   [function,functional]
2 rule listLength(ListItem(_) Xs:List) => 1 +Int listLength(Xs)
3 rule listLength(.List)               => 0
```

Figure 2.13: A total function

3

K-Javalette: A formal semantics for Javalette

This chapter describes how the Javalette language is formalized in $\mathbb{K}$. It provides a detailed explanation of the main components of K-Javalette. Figure 3.1 shows the steps of running a program with the interpreter generated from K-Javalette.

Figure 3.1: Main components of K-Javalette

3.1 Syntax

The syntax module is based on the existing syntax definition of the language provided as a BNFC file. Since $\mathbb{K}$ uses a similar BNF notation, there is a close resemblance between the two specifications. See Figure 3.2 and Figure 3.3 for a comparison of syntax declarations in $\mathbb{K}$ and BNF. The main difference is the use of annotations in $\mathbb{K}$. Operator precedence and associativity rules are defined using annotations as explained in Section 2.2.1. Besides, $\mathbb{K}$ has strictness annotations, which are, in fact, a part of the semantics.

```
1  syntax Exp ::= // ...
2    | Int        [literal]
3    | Id
4    | Id "(" Args ")"   [funcall]
5    | "-" Exp    [strict, unary]
6    | Exp MulOp Exp   [left, seqstrict(1,3), binaryMult]
7    | Exp AddOp Exp   [left, seqstrict(1,3), binaryAdd]
8    | Exp RelOp  Exp  [left, seqstrict(1,3), binaryComp]
9    > Exp "&&" Exp    [right, strict(1)]
10   // more productions ...
11 syntax Args ::= List{Exp, ","}
12 syntax priorities literal > funcall > unary > binaryMult
13                   > binaryAdd > binaryComp
```

Figure 3.2: Javalette syntax in K

```
...
ELitInt. Expr6 ::= Integer ;
EVar. Expr6 ::= Ident ;
EApp. Expr6 ::= Ident "(" [Expr] ")" ;
Neg. Expr5 ::= "-" Expr6 ;
EMul. Expr4 ::= Expr4 MulOp Expr5 ;
EAdd. Expr3 ::= Expr3 AddOp Expr4 ;
ERel. Expr2 ::= Expr2 RelOp Expr3 ;
EAnd. Expr1 ::= Expr2 "&&" Expr1 ;
...
separator  Expr "," ;
coercions  Expr 6 ;
...
```

Figure 3.3: Javalette syntax in LBNF

Another major difference is in the assignment syntax. In the original specification, only identifiers are allowed on the left-hand side of an assignment. On the other hand, K-Javalette allows any expression in the syntax. Consequently, it requires additional checks in the type-checking. The purpose of this is to allow extensions in the future.

```
syntax Stmt ::=
  Exp "=" Exp ";" [strict(2)]
```

```
Ass. Stmt ::=
  Ident "=" Expr ";" ;
```

Figure 3.4: Assignment syntax

3.2 Configuration

```
<jl>
    <common>
        <program> $PGM:Program </program>
        <progress> .K </progress>
        <funs> .Map </funs>
    </common>
    <typecheck/>
    <exec/>
</jl>
```

Figure 3.5: Main configuration of K-Javalette

Figure 3.5 shows the main configuration of K-Javalette, which consists of three sub-cells:

- `<common/>`: Stores the information used by all phases of interpretation. The `<program/>` sub-cell keeps the input program. The `<progress/>` cell keeps a sequence of high-level steps to be executed. The initial value `".K"` denotes

an empty sequence. Figure 3.6 shows the initialization of this cell. The last sub-cell `<funs/>` is a map of globally declared functions in the program. It is populated during the `#processTopDefs` step.

- `<typecheck/>`: Stores the type checker-related information explained in Section 3.4.
- `<exec/>` Stores the execution-related information explained in Section 3.6.

```
1 rule
2     <program> Prg:Program </program>
3     <progress> . =>
4         #processTopDefs ~>
5         #typecheck ~>
6         #returncheck ~>
7         #execute ~>
8         #set_code
9     </progress>
```

Figure 3.6: Main configuration of K-Javalette

3.3 Processing Top-level Definitions

In core Javalette, function definitions are the only top-level forms. This step goes through all the top-level definitions in the program and creates a map of functions in the `<funs/>` cell. It also ensures that all function names are unique. The `processTopDef` helper is executed for each top-level definitions in the program.

```
1 rule
2     <progress> processTopDef(T I (Ps) Body) => . ... </progress>
3     <funs> FUNS => FUNS[I <- (T I (Ps) Body)] </funs>
4     requires notBool(I in_keys(FUNS))
```

Figure 3.7: Processing top-level definitions

3.4 Types

There are four basic types in Javalette and they are defined in the syntax module.

```
1 syntax Type ::= "int"
2               | "double"
3               | "boolean"
4               | "void"
```

Figure 3.8: Javalette types

The type-checking state consists of four elements as shown in Figure 3.9. It contains program fragments to type-check (`<tcode/>`), the return type of the surrounding

function (`<retType/>`), a map from variables in the current scope to their types (`<tenv/>`), and a set of variables declared in the current block (`<tenv-block/>`).

```
<typecheck>
    <tcode> .K </tcode>
    <retType> void </retType>
    <tenv> .Map </tenv>
    <tenv-block> .Set </tenv-block>
</typecheck>
```

Figure 3.9: Type-checking configuration

3.4.1 Functions

At the beginning of type-checking, the `</tcode>` cell is initialized with the whole program. Then, each top-level definition is type-checked separately. Following rules separate the top-level definitions in the input program. (A `Program` is a list of `TopDef`s, and `.Program` is an empty list.)

```
rule <tcode> TD:TopDef Prg:Program => TD ~> Prg ... </tcode>
rule <tcode> .Program => . ... </tcode>
```

Figure 3.10: Type-checking a program

Figure 3.11 shows the typing rule for functions. It validates the parameters, sets the environment and the return type, and finally leaves the function body in `<tcode/>` to be type-checked. The `paramMap` helper function creates a map from parameter names to their types, and `validParams` is a boolean function that checks the parameter names and types. Parameters cannot be `void`, and all parameter names must be unique.

```
rule <tcode> T:Type FName:Id ( Ps:Params ) Body
             => Body ...
     </tcode>
     <tenv> _         => paramMap(Ps) </tenv>
     <tenv-block> _ => .Set          </tenv-block>
     <retType> _     => T            </retType>
     requires notBool(FName in builtinFuns )
      andBool validParams(Ps)
```

Figure 3.11: Type-checking a function

3.4.2 Statements and scope rules

A variable declaration "`T V = E;`" is well-typed if `T` is not `void`, `V` is not previously declared in the block, and `E` has the type `T`. If these preconditions hold, the variable is added to the scope. Figure 3.12 shows the implementation of this rule in K-Javalette.

```
rule <tcode> ( T:Type V:Id = E:Exp ; ):Stmt => . ... </tcode>
     <tenv> ENV => ENV[V <- T] </tenv>
     <tenv-block> BLK => SetItem(V) BLK </tenv-block>
     requires T =/=K void
       andBool notBool(V in BLK)
       andBool checkExp(T, E)
```

Figure 3.12: Type-checking variable declaration

Note that the initializer expression `E` is type-checked before `V` is introduced to the scope.

Multiple variable declarations are split into separate statements.

```
rule <tcode>
        T:Type V:DeclItem , V2 , Vs:DeclItems ; )
            =>
        (T V;) ~> ( T V2 , Vs ; ) ...
     </tcode>
```

Figure 3.13: Multiple variable declarations

As stated in Section 3.1, assignments have a more permissive syntax in K-Javalette. Thus, there is a need to check if the left-hand side of an assignment is a valid expression. Figure 3.14 defines the typing rule for assignments using the `isLValue` function, which returns `true` if the given expression is an identifier. The `inferExp` and `checkExp` functions are used for type-checking expressions. If `V` is an identifier, `inferExp(V)` implicitly checks if it is in the scope.

```
rule <tcode> ( V = E ; ) => . ... </tcode>
     requires checkExp( inferExp(V) , E)
      andBool isLValue( V )

syntax Bool ::= isLValue(Exp) [function,functional]
rule isLValue(_:Id)    => true
rule isLValue(_:Bool)  => false
rule isLValue(_:Int)   => false
rule isLValue(_:Float) => false
// ... and more rules to return false for all other expressions
```

Figure 3.14: Type-check assignment

Block statements and bodies of control structures (`if/else`, `while`) create new scopes in the environment. After these statements, the environment must be restored to the previous state. Variables declared in the block must be removed from the environment. In order to implement this behavior, we introduce environment recovery actions to be used in `<tcode/>`. Figure 3.15 demonstrates the use of the `twithBlock` action, which takes a chunk of code and sets a reminder to recover the

environment after checking the given code. The `twithBlock` helper is also used for bodies of conditional statements.

```
rule <tcode> { Ss } => twithBlock( Ss ) ... </tcode>

syntax KItem ::= tenvReminder(Map, Set)
               | twithBlock(K)

rule <tcode> twithBlock(S) => S ~> tenvReminder(ENV,BLK) ...
     </tcode>
     <tenv> ENV </tenv>
     <tenv-block> BLK => .Set </tenv-block>

rule <tcode> tenvReminder(ENV,BLK) => . ... </tcode>
     <tenv> _ => ENV </tenv>
     <tenv-block> _ => BLK </tenv-block>
```

Figure 3.15: Type-checking block statements

3.4.3 Expressions

The typing rules for expressions are implemented as functions since they do not manipulate the state. The `inferExp` function infers the type of the given expression and `checkExp` uses `inferExp` to type-check an expression against the given type.

```
syntax InferRes ::= Type | "#typeError"
syntax InferRes ::= inferExp(Exp) [function, functional]

syntax Bool ::= checkExp(InferRes, Exp) [function, functional]
rule checkExp( T:Type, E )      => equalType(T, inferExp(E))
rule checkExp( #typeError, _ ) => false
```

Figure 3.16: Type-checking expressions

As an example, Figure 3.17 shows the type inference rule for the addition operation.

```
rule inferExp( E1:Exp + E2:Exp ) => inferArith(inferExp(E1), E2)

syntax InferRes ::= inferArith(InferRes, Exp) [function,functional]
rule inferArith(T:Type, E2:Exp) => T requires isNumeric(T)
                                        andBool checkExp(T, E2)
rule inferArith(#typeError, _) => #typeError
```

Figure 3.17: Type inference for addition

Inference rules for variable expressions and function applications need to look up from the configuration. $\mathbb{K}$ has a special syntax (double square brackets around the evaluation rule) to allow function rules to read the configuration. The rule in Figure 3.18 states that if the environment maps the identifier `V` to type `T`, then the expression `V` has type `T`.

```
rule [[ inferExp(V:Id) => T ]]
    <tenv> ... V |-> T ... </tenv> // pattern matching on a Map
```

Figure 3.18: Type inference for addition

3.5 Return checking

In Javalette, every non-void function must return a value. If there is a conditional statement in the function, both execution paths must return. The return checker is implemented as a function that goes through all functions in the program. Statements in a function body are checked sequentially. If none of them is a *terminating* statement, the function is rejected.

```
syntax Bool ::= retcheckProgram(Program) [function, functional]
rule retcheckProgram(.Program) => true
rule retcheckProgram(F:TopDef Rest) =>
    retcheckTopDef(F) andBool retcheckProgram(Rest)

syntax Bool ::= retcheckTopDef(TopDef) [function, functional]
rule retcheckTopDef(FD:FunDef) => retcheckFunDef(FD)

syntax Bool ::= retcheckFunDef(FunDef) [function, functional]
rule retcheckFunDef(T _ (_) { Body }) =>
    (T ==K void) orBool retcheckStmts(Body)

syntax Bool ::= retcheckStmts(Stmts) [function, functional]
rule retcheckStmts(.Stmts) => false
rule retcheckStmts(S Ss) =>
    retcheckStmt(S) orBool retcheckStmts(Ss)
```

Figure 3.19: Return checker implemented as a function

To be considered a terminating statement, a statement must be a return statement, a block statement with a terminating statement, or a conditional statement where both branches are terminating (Figure 3.20).

3.6 Execution

The execution configuration consists of four components:

- `<k/>`: the computation cell
- `<env/>`: a map from variable names in the scope to their locations in the memory
- `<store/>`: a map from memory locations to values
- `<next-loc>`: an integer denoting the next available location in the memory

```
syntax Bool ::= retcheckStmt(Stmt) [function, functional]
rule retcheckStmt(return _;) => true
rule retcheckStmt(return;)   => true

rule retcheckStmt(if(_) T else F) => retcheckStmt(T)
                                     andBool retcheckStmt(F)
rule retcheckStmt({ Ss })         => retcheckStmts(Ss)

rule retcheckStmt(;)                      => false
rule retcheckStmt(_:Type _:DclItems ;) => false
rule retcheckStmt(_ = _ ;)                => false
rule retcheckStmt(_ ;)                    => false
rule retcheckStmt(while(_)_)              => false
```

Figure 3.20: Terminating statements

- `<stack/>`: the call stack

```
<exec>
    <k>    .K       </k>
    <env>  .Map     </env>
    <store>  .Map   </store>
    <next-loc> 0    </next-loc>
    <stack>  .List  </stack>
</exec>
```

Figure 3.21: Execution configuration

The execution starts with a call to the `main` function in the `<k/>` cell. Initially, the environment, store, and stack are empty.

```
rule <progress> #execute => #executing ... </progress>
     <k>                 . => main(.Args)    </k>
```

3.6.1 Expressions and values

When there is an expression on top of the `<k/>` cell, it should be evaluated and replaced with its value. We define the sort of values using the built-in `Int`, `Float`, `Bool`, and `String` sorts. To represent the results of `void` expressions, we use "nothing" (Figure 3.22).

```
syntax Value ::= Int
               | Float
               | Bool
               | "nothing"
syntax KResult ::= Value
```

Figure 3.22: Values in core Javalette

Since we have strictness annotations in the syntax, most of the expressions are implemented as in Section 2.2.3. Figure 3.23 provides some of the rules for evaluating binary operations.

```
1 rule <k> I1:Int + I2   => I1 +Int   I2 ... </k>
2 rule <k> I1:Float + I2 => I1 +Float I2 ... </k>
3 rule <k> I1:Int >  I2  => I1 >Int   I2 ... </k>
```

Figure 3.23: Evaluating binary operations

Operators in the previous examples are strict in both arguments. In contrast, conjunction and disjunction operators are strict only on the first argument. Figure 3.24 shows the lazy semantics of the `&&` operator. If the first operand evaluates to `false`, the second operand is not evaluated.

```
1 rule <k> false:Value && _ => false:Value ... </k>
2 rule <k> true:Value  && E => E ... </k>
```

Figure 3.24: Short-cut evaluation

A variable expression is evaluated by reading its value from the store. Figure 3.25 describes reading the value of a variable. The top of the `<k/>` cell is a variable (`X`), the environment maps `X` to the location `L`, and the store maps `L` to the value `V`.

```
1 rule <k> X:Id => V ...</k>
2      <env> ... X |-> L ... </env>
3      <store> ... L |-> V ...</store>
```

Figure 3.25: Variable lookup

3.6.2 Environment and store

In K-Javalette, the memory is modeled as a map from integer locations to values. Each variable has an associated location in the store, and the value of the variable is kept in that location. Another approach would be to directly store values in the environment. Our indirect approach is more flexible and makes it easy to implement references.

Whenever a variable is declared, a fresh location is allocated in `<store/>` by using the `<next-loc/>` cell. A simplified version of the variable declaration rule is shown in Figure 3.26. The new variable's identifier is added to the environment and its initial value is stored at the next available location in the `<store/>` cell.

If the variable is declared without an initializer expression, the declaration statement is re-written using the default value. Figure 3.27 shows the definition of the default values.

```
1 rule <k> T Var = Val ; => . ...          </k>
2      <env>      ENV    => ENV[Var <- I] </env>
3      <store>    S      => S[ I <- Val ] </store>
4      <next-loc> I      => I+Int 1       </next-loc>
```

Figure 3.26: Variable declaration and allocation in the store

```
1 rule <k> (T:Type V:Id ;):Stmt => T V = defaultValue(T) ; ... </k>
2
3 syntax Value ::= defaultValue(Type) [function,functional]
4 rule defaultValue(int)     => 0
5 rule defaultValue(double)  => 0.0
6 rule defaultValue(boolean) => false
7 rule defaultValue(void)    => nothing
```

Figure 3.27: Variable declaration without initializer

Variables declared in blocks are allowed to shadow previous declarations. For this reason, block statements and control structures require an environment recovery mechanism similar to `twithBlock` in type-checking (Figure 3.28).

```
1 rule <k> { Ss } => withBlock(Ss) ... </k>
2
3 syntax KItem ::= envReminder(Map)
4                | withBlock(K)
5
6 rule <k> envReminder(ENV) => . ... </k>
7      <env> _ => ENV </env>
8 rule <k> withBlock(S) => S ~> envReminder(ENV) ... </k>
9      <env> ENV </env>
```

Figure 3.28: Executing block statements

Figure 3.29 is an example program with variable shadowing. The configuration on the right shows the program state after the variable declaration on line 7. The remaining statements in the block are listed on top of the `<k/>` cell, and an `envReminder` comes after that.

3.6.3 Conditional statements

If/else statements are strict on the condition expression. As a result, there is no need for defining additional rules for evaluating the condition. As shown in Figure 3.30, one of the two statements is chosen according to the condition value, and then executed in a block.

The condition in a while statements is not strict since it needs to be evaluated before every iteration. While loops are executed using the `freezeWhile` helper.

```
int main()
{
  int x = 1;
  int y = 2;

  {
    int x = 3;

    printInt(x);
  }

  return 0;
}
```

```
<jl>
  <exec>
    <k> printInt ( x ) ; ~> .Stmts ~>
        envReminder ( x|->0 y|->1 ) ~>
        return 0 ;  .Stmts ~>
        envReminder ( .Map ) ~> return; ~> .
    </k>
    <env>
      x |-> 2
      y |-> 1
    </env>
    <store>
      0 |-> 1
      1 |-> 2
      2 |-> 3
    </store>
    <next-loc>
      3
    </next-loc>
    // ... more cells
  </exec>
  // ... more cells
</jl>
```

Figure 3.29: An example program with variable shadowing

```
rule <k> if(true)  T else _ => withBlock(T) ... </k>
rule <k> if(false) _ else F => withBlock(F) ... </k>
```

Figure 3.30: Executing if statements

```
syntax KItem ::= freezeWhile(Exp, Stmt)
rule <k> while(E) S => E ~> freezeWhile(E,S) ... </k>
rule <k> true  ~> freezeWhile(E,S) => withBlock(S) ~> while(E) S
     ... </k>
rule <k> false ~> freezeWhile(_,_) => .               ... </k>
```

Figure 3.31: Executing while statements

4

Language Extensions

The modularity of $\mathbb{K}$ allows extending languages by adding new syntax and semantic rules without changing existing modules. It is possible to add new productions to an existing sort (e.g. a new form of expression) and define relevant semantic rules. In this way, we can modularly extend languages in both syntax and semantics.

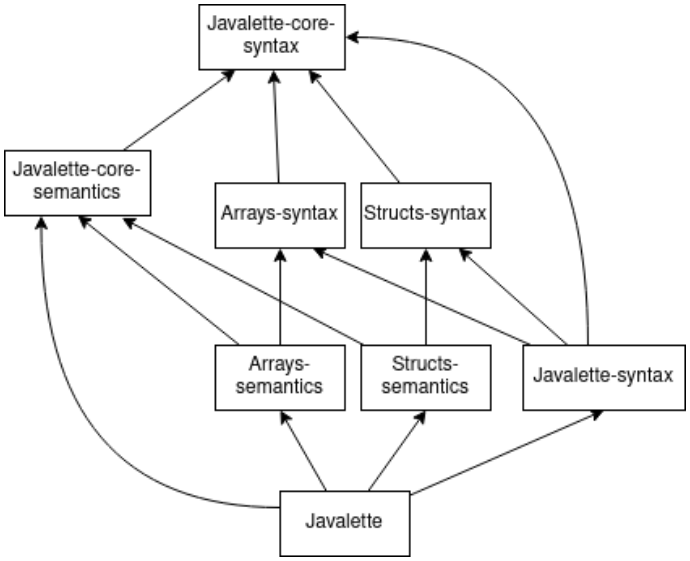

Figure 4.1: Dependencies with extensions

In the following, we develop the arrays and dynamic data structures language extensions as modules that can be easily enabled. Figure 4.1 shows the general structure of modules in the extended language. Syntax modules of the extensions import the core syntax module and add necessary productions to relevant sorts. Then, in the main syntax module, these additions are enabled by just importing the extension modules (Figure 4.2).

Semantic modules for the extensions add new rules needed for the new constructs added in the syntax module. Since some parts of the semantics are implemented as `functional` symbols, the compiler shows hints to implement necessary rules needed after the syntax extension, thanks to the non-exhaustive patterns check. Semantic extensions, like syntax extensions, are simply enabled by importing the extension from the main module if they do not require adding a new cell to the configuration

```
module JAVALETTE-SYNTAX
    imports JAVALETTE-SYNTAX-CORE
    imports JAVALETTE-ARRAYS-SYNTAX
    imports JAVALETTE-STRUCTS-SYNTAX
endmodule
```

Figure 4.2: Extending the syntax

or a new step to the `<progress/>` cell. Figure 4.3 describes how semantics for extensions are enabled.

```
module JAVALETTE

    imports JAVALETTE-ARRAYS  // import semantics for arrays
    imports JAVALETTE-STRUCTS // import semantics for structs
    // ... more imports
    configuration
        <jl>
            <common/>
            <typecheck/>
            <exec/>
            <structs/>        // add new cell for structs
        </jl>
    // ... more rules
endmodule
```

Figure 4.3: Extending the semantics

4.1 Arrays

This section explains the development of the arrays extension. The arrays extension consists of the array type, expressions for creating arrays and accessing array elements, and for-loop statements to iterate over arrays. The syntax module adds new productions for `Type`, `Exp`, and `Stmt`. Possible ambiguities in the syntax are resolved by assigning suitable priority labels (e.g. `literal` or `funcall`) from the core syntax.

After enabling this syntax extension, the compiler should complain about functions on `Type`, `Exp`, and `Stmt` such as `inferExp` and `isLValue`. The reason is that these symbols are defined as total functions and we have not covered the new cases yet.

Typing rules for array expressions are implemented by adding new rules for `inferExp`. Figure 4.5 is the typing rule for indexing expressions. If `Arr` is of type `T[]` and `Ix` is an `Int`, then `Arr [ Ix ]` is a `T`.

The extension includes `for`-loop statements to iterate through arrays. It consists of a loop variable `X` of type `T`, an array expression `Arr` of type `T[]`, and the loop body. Figure 4.6 describes the typing and scope rules of `for`-loops. The array expression is type-checked before `X` is added to the scope. The scope of the variable `X` is limited

```
module JAVALETTE-ARRAYS-SYNTAX
    imports JAVALETTE-SYNTAX-CORE

    syntax TypBox ::= r"\\[[ \\n\\t]*\\]" [token]
    syntax Type ::= Type TypBox

    syntax Exp ::= "new" Type Boxes        [unary]
                 | Exp Box                 [funcall]
                 | Exp "." Id              [strict(1), unary]

    syntax Id ::= "length"                 [token]

    syntax Box ::= "[" Exp "]"             [strict]
    syntax Boxes ::= NeList{Box, ""}

    syntax Stmt ::= "for" "(" Type Id ":" Exp ")" Stmt [strict(3)]

endmodule
```

Figure 4.4: Arrays syntax

```
rule inferExp( Arr [ Ix ] ) => arrayElement(inferExp(Arr))
     requires checkExp(int, Ix)

syntax InferRes ::= arrayElement(InferRes) [function, functional]
rule arrayElement(T []) => T
rule arrayElement(_) => #typeError [owise]
```

Figure 4.5: Type of an array element

to the loop body. The loop body is regarded as a new block, and variables declared in the loop body can shadow the loop variable.

```
rule <tcode> (for( T X : Arr) Body)
         => twithBlock(
             (T X ;):Stmt ~>
             twithBlock( Body )
         ) ...
     </tcode>
     requires checkExp(T[], Arr)
```

Figure 4.6: Typing rule for for-loops

To define the execution semantics of arrays, we need to extend the `Value` sort to represent array values. An array value is a pair of the starting location and size of the array. The default value for arrays is an empty array.

The `new` expression allocates consecutive locations in the store for each array element. For multi-dimensional arrays, sub-arrays are created recursively. The program in Figure 4.8 declares a 2-dimensional integer array.

```
syntax Value ::= array(Int, Int) // array(Location, Length)

rule defaultValue(_:Type []) => array(0,0)
```

Figure 4.7: An array is a pair of location and size

```
int main()
{
    int[][] matrix = new int[2][3];
    matrix[0][0] = 0;
    matrix[0][1] = 1;
    matrix[0][2] = 2;
    matrix[1][0] = 3;
    matrix[1][1] = 4;
    matrix[1][2] = 5;
    return 0;
}
```

Figure 4.8: Create a 2-dimensional array

Figure 4.9 shows the state after the assignments in the example program. In the variable declaration, the initializer expression is evaluated before the allocation for the variable. Hence, the variable points to `8`.

```
<jl>
  <exec>
    <k>
      return 0 ;  .Stmts ~> envReminder ( .Map ) ~> return ; ~> .
    </k>
    <env> matrix |-> 8 </env>
    <store>
      0 |-> array ( 2 , 3 )  // matrix[0]
      1 |-> array ( 5 , 3 )  // matrix[1]
      2 |-> 0                // matrix[0][0]
      3 |-> 1                // matrix[0][1]
      4 |-> 2                // matrix[0][2]
      5 |-> 3                // matrix[1][0]
      6 |-> 4                // matrix[1][1]
      7 |-> 5                // matrix[1][2]
      8 |-> array ( 0 , 2 )  // matrix
    </store>
    <next-loc> 9 </next-loc>
    // ... more cells
  </exec>
  // ... more cells
</jl>
```

Figure 4.9: Configuration after creating a 2-dimensional array

We can access elements of an array by their locations in the store. The indexing expression also checks for array bounds.

```
rule <k> array(Loc, Len) [ Ix:Int ] => X ...</k>
     <store> ... (Loc +Int Ix ) |-> X ... </store>
     requires Len >Int Ix
```

Figure 4.10: Indexing expression

4.2 Dynamic Data Structures

This section explains the development of the dynamic data structures extension. This extension introduces compound data types, allocating heap objects, and pointers to access heap objects. An example program using structs is given in Figure 4.11. Structures are defined using the `struct` keyword on the top-level. The `typedef` keyword defines a pointer type. The use of pointers is limited to structs. It is not allowed to define pointers to primitive types. The `new` keyword creates a heap object and returns a pointer.

```
struct Pair {
  int x;
  int y;
};
typedef struct Pair * Coord;

int main()
{
  Coord pt = new Pair;
  pt->x = 1;
  pt->y = 2;

  printInt(pt->x + pt->y); // prints "3"

  return 0;
}
```

Figure 4.11: Structs syntax

The syntax extends the `TopDef`, `Type`, `Exp`, and `Stmt` sorts.

We need to add new configuration cells (Figure 4.13) to keep struct definitions and pointer types declared in the program. The `<structMaps/>` cell is a map from struct names to struct definitions. The `typedefs` cell is a map from pointer type names to associated struct names. These maps are populated when processing the top-level definitions. Pointer type names and struct names must be unique but they use different name spaces. Field names in a struct must be unique.

For the example program in Figure 4.11, the `<structs/>` cell would be as shown in Figure 4.14. Note that struct definitions in `structMaps` are stored as maps that associate field names to their types and positions in the struct. In this example, the `Pair` struct has two fields called `x` and `y`. Field `x` of a `Pair` is an `int` and it is the first (0-indexed) field in the struct. Struct fields are stored in the memory according

```
module JAVALETTE-STRUCTS-SYNTAX
  imports JAVALETTE-SYNTAX-CORE

  syntax TopDef ::= StructDef
                  | TypeDef

  syntax StructDef ::= "struct" Id "{" FieldDefs "}" ";"
  syntax FieldDefs ::= List{FieldDef, ""}
  syntax FieldDef ::= Type Id ";"

  syntax TypeDef ::= "typedef" "struct" Id "*" Id ";"

  syntax Type ::= Id // a pointer type

  syntax Exp ::= Exp "->" Id                [strict(1), funcall]
               | "new" Id                   [unary]
               | "(" Id ")" "null"          [literal]
endmodule
```

Figure 4.12: Structs syntax

```
configuration
    <structs>
        <structMaps> .Map </structMaps>
        <typedefs>   .Map </typedefs>
    </structs>
```

Figure 4.13: Structs configuration

to these indexes.

```
<structs>
  <structMaps>
    Pair |-> ( x |-> fpair ( int , 0 )
               y |-> fpair ( int , 1 ) )
  </structMaps>
  <typedefs>
    Coord |-> Pair
  </typedefs>
<structs/>
```

Figure 4.14: Structs configuration: an example

At the type-checking phase, it is checked whether the struct names in the `typedef`s exist.

Figure 4.16 shows the type-checking rules for struct definitions. Struct fields must have valid data types. If the type of a field is an identifier, it must be a pointer type declared in a `typedef`. (`#ptr(T)` is the internal representation for pointer type to struct `T`)

```
1 rule <tcode> typedef struct SName * _ ; => . ... </tcode>
2      <structMaps> Structs </structMaps>
3      requires SName in_keys(Structs)
```

Figure 4.15: Check typedefs

```
1  rule <tcode> struct _SName { Fields }; => . ... </tcode>
2       requires validFields(Fields)
3
4  syntax Bool ::= validFields(FieldDefs) [function,functional]
5  rule validFields(.FieldDefs) => true
6  rule validFields((T _;) FDs) => validDataType(T) andBool
       validFields(FDs)
7
8  rule [[ validDataType(T:Id) => T in_keys(TD) ]]
9    <typedefs> TD </typedefs>
10 rule [[ validDataType(#ptr(T)) => T in_keys(TD) ]]
11   <structMaps> TD </structMaps>
```

Figure 4.16: Check struct fields

A pointer dereferencing `E -> F` is well-typed if the expression is `E` is a pointer to a struct that has a field `F`. Figure 4.17 shows the encoding of this rule in $\mathbb{K}$. (The `#let` expression is a syntactic sugar for binding names to intermediate values in RHS of rules.)

```
1  rule inferExp(E -> F) => inferField(inferExp(E), F)
2
3  syntax InferRes ::= inferField(InferRes, Id) [function,functional]
4
5  rule [[
6          inferField(TName:Id, F) =>
7            #let fpair(T,_) = SM[F]
8            #in T
9      ]]
10     <typedefs> ... TName |-> SName ... </typedefs>
11     <structMaps> ... SName |-> SM ... </structMaps>
12     requires F in_keys(SM)
```

Figure 4.17: Check pointer dereferencing

A pointer either points to a struct in the memory or it is a `null` pointer. We represent the value of a pointer as a pair of struct name and starting location of the struct (Figure 4.18). Elements of a struct are consecutively placed in the memory. Uninitialized pointers default to `null`.

When a struct object is created with `new` (Figure 4.19), a memory location is allocated in `<store/>` for each field (line 4). Then, each field is initialized to default values of its type (line 10). If a struct field is a pointer, it is initially `null`.

```
1  syntax Value ::= struct(Id, Int)
2                 | "#nullptr"
3
4  // syntax Value ::= defaultValue(Type) [function, functional]
5  rule defaultValue(#ptr(_)) => #nullptr
6  rule defaultValue(_:Id) => #nullptr
```

Figure 4.18: Pointer values

```
1  rule <k> new SName:Id => initFields(I, values(SM)) ~>
2                           struct(SName, I) ... </k>
3       <structMaps> ... SName |-> SM ... </structMaps>
4       <next-loc> I => I +Int size(SM) </next-loc>
5
6  syntax KItem ::= initFields(Int, List)
7  rule <k> initFields(_, .List) => . ... </k>
8  rule <k> initFields(I, ListItem(fpair(T,X)) FDs) =>
9             initFields(I, FDs) ... </k>
10      <store> ST => ST[I +Int X <- defaultValue(T)] </store>
```

Figure 4.19: Struct creation and memory allocation

Figure 4.20 is an example of creating nested structures. Inline comments show the content of the store. When creating an `Obj_s` on line 14, no allocation is made for a `Pair_s`, so `myObj->p` is initially `null`.

Figure 4.21 shows the evaluation rule for pointer dereferencing. To access a field of a struct, the address of the struct and the index of the field are used.

Since there is no rule for dereferencing null pointers, if the struct pointer is `null`, then the execution gets stuck.

```
1  struct Pair_s {
2    int x;
3  };
4
5  typedef struct Pair_s * Pair;
6
7  struct Obj_s {
8    int t;
9    Pair p;
10 };
11
12 typedef struct Obj_s * Obj;
13
14 int main() {
15   Obj myObj = new Obj_s;   // <store>
16                            //   0 |-> 0
17                            //   1 |-> #nullptr
18                            //   2 |-> struct ( Obj_s , 0 )
19                            // </store>
20
21   myObj->t = 123;          // <store>
22                            //   0 |-> 123
23                            //   1 |-> #nullptr
24                            //   2 |-> struct ( Obj_s , 0 )
25                            // </store>
26
27   myObj->p = new Pair_s;   // <store>
28                            //   0 |-> 123
29                            //   1 |-> struct ( Pair_s , 3 )
30                            //   2 |-> struct ( Obj_s , 0 )
31                            //   3 |-> 0
32                            // </store>
33   return 0;
34 }
```

Figure 4.20: Creating nested structures

```
1 rule <k> struct(SName, Loc) -> F =>
2             #let fpair(_,I) = SM[F]
3             #in ST[Loc +Int I] ... </k>
4      <store> ST </store>
5      <structMaps> ... SName |-> SM:Map ... </structMaps>
```

Figure 4.21: Accessing a struct field

5

Unspecified Behaviors

Formalizing the semantics requires being precise about the details and thinking about corner cases. While developing the formal semantics, we have discovered a number of undefined behaviors in the Javalette documentation and proposed necessary changes to resolve them. We also created relevant test cases to demonstrate the expected behavior. This chapter explains our findings in detail.

5.1 Core Javalette

5.1.1 Variable scope rules in conditional statements

In the following program, the body of the `if` statement is not a block. Therefore, whether `x` is defined or not after the `if` statement is ambiguous. In order to resolve this ambiguity, we decided to always regard bodies of conditional statements as new blocks. Therefore, it is a failing program in K-Javalette.

```
1 int main()
2 {
3     if(readInt() > 0)
4         int x = 0;
5
6     printInt(x);
7
8     return 0;
9 }
```

5.1.2 Multiple variable declarations in one statement and evaluation of initial values

The documentation does not fully specify the order of adding variables to the scope and evaluating initial values. It leads to undefined behavior when an initial value in a variable declaration has side effects or refers to other variables. In Figure 5.1, it is not clear whether `y` is initialized to `1` or `2`.

In K-Javalette we decided to regard multiple declarations as separate statements. Therefore, the program in Figure 5.1 is equivalent to the one in Figure 5.2, and it should print "2".

```
1  int main()
2  {
3      int x = 1;
4
5      {
6          int x = 2, y = x;
7
8          printInt(y);
9      }
10
11     return 0;
12 }
```

Figure 5.1: Ambiguous multiple declaration

```
1  int main()
2  {
3      int x = 1;
4
5      {
6          int x = 2;
7          int y = x;
8
9          printInt(y); // prints "2"
10     }
11
12     return 0;
13 }
```

Figure 5.2: An equivalent program to 5.1

5.1.3 Evaluation order of operands/arguments

The order of evaluation in binary operators and function calls was not specified. Thus, side effects in operands/arguments could lead to unexpected behavior. For example, the program in Figure 5.3 can output `"1\n2"` or `"2\n1"` depending on the evaluation order.

In order to eliminate this, we specified the evaluation order to be from left to right. Thus, the example program prints `"1\n2"`.

5.2 Arrays

5.2.1 Default values for arrays

When a variable is defined without an initial value, it is initialized to the default value of its type. We decided default values for arrays to be 0-length arrays, and not to introduce a `null` value for arrays. Therefore, the two programs in Figure 5.4 and Figure 5.5 are equivalent, and the output should be "0".

```
1  int main()
2  {
3      int t = foo(1) + foo(2));
4
5      return 0;
6  }
7
8  int foo(int x)
9  {
10     printInt(x);
11
12     return x;
13 }
```

Figure 5.3: Side effect in binary operation

```
1  int main()
2  {
3      int[] arr;
4
5      printInt(arr.length);
6
7      return 0;
8  }
```

Figure 5.4: Array declaration without initializer

```
1  int main()
2  {
3      int[] arr = new int[0];
4
5      printInt(arr.length);
6
7      return 0;
8  }
```

Figure 5.5: Declare an empty array

5.2.2 Evaluation order in array indexing expressions

Like binary operations, array indexing expressions contain two sub-expressions. Therefore, the order of evaluation needs to be specified. In K-Javalette, the order of evaluation is specified using `seqstrict` as shown in Figure 5.6.

```
1  syntax Exp ::= Exp "[" Exp "]"        [seqstrict, funcall]
```

Figure 5.6: Strictness of array indexing

The following program should print "`array\nindex`".

```
1  int main()
2  {
3      int x = mkArray()[index()];
4      return 0;
5  }
6
7  int mkArray()
8  {
9      printString("array");
10     return new int[5];
11 }
12
13 int index()
14 {
15     printString("index");
16     return 1;
17 }
```

Figure 5.7: Evaluation order in array indexing

Similarly, the evaluation order of length expressions in array creation should be specified as well. In K-Javalette, lengths are evaluated sequentially from left to right. The example in Figure 5.8 should output `"1\n2\n3"`

```
1  int main()
2  {
3      int[][][] x = new int[f(1)][f(2)][f(3)];
4      return 0;
5  }
6
7  int f(int x)
8  {
9      printInt(x);
10     return x;
11 }
```

Figure 5.8: Creating a multi-dimensional array

5.3 Precedence of pointer and array expressions

Precedences between array and pointer expressions are not specified. According to the test suite, the syntax should allow the first expression in Figure 5.9. It should also allow parsing (2) a chain of pointer dereferencing and array indexing, as well as (3) creating multi-dimensional pointer arrays.

The first expression requires precedence of pointer dereferencing to be lower than or equal to struct creation. Otherwise, a struct creation cannot be parsed as an immediate child of pointer dereferencing. The second case requires array indexing and pointer referencing to be at the same level. If these two conditions hold, the

```
1 new foo->x
2 foo->bar[x]->baz
3 new foo[1][2]
```

Figure 5.9: Problematic expressions

third expression becomes ambiguous. It can be parsed as an array indexing on struct creation, or two-dimensional array creation.

In order to keep the syntax consistent and reasonable to implement, we decided to disallow the first case, which is dereferencing a newly created pointer. This change does not bring any practical limitation to the language because the same behavior is achieved by using parenthesis.

6

Testing Javalette Implementations

Besides being a reference for the language, formal semantics in $\mathbb{K}$ has several applications for testing and verification. In this work, we use K-Javalette to develop an input generator for testing Javalette compilers. A test case in the test suite consists of a Javalette program, an input file for the program, and the expected output. Since we have a reference interpreter obtained from the semantics, it is possible to generate the expected output for a given pair of Javalette program and input. Thus, using an input generator as illustrated in Figure 6.1 eliminates the need for input files and leads to more effective use of existing test programs by covering as many execution paths as possible.

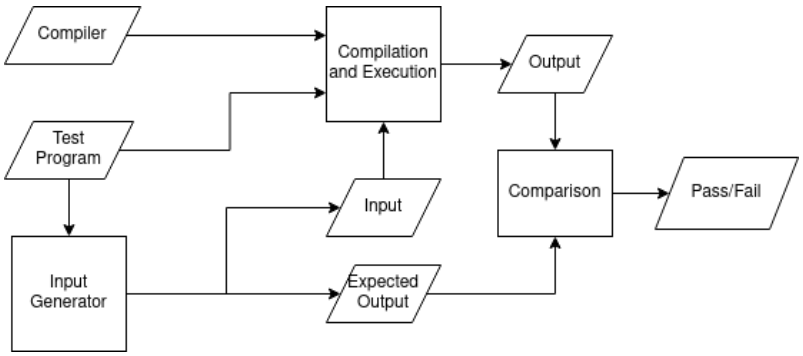

Figure 6.1: Testing compilers with random input

For generating random inputs, we created a new module that overrides the behavior of built-in input functions `readInt` and `readDouble` so that while evaluating these functions, instead of reading the standard input, random numbers are generated. The module extends the configuration with a seed variable for the random number generation (Figure 6.2). The seed is initialized via a command-line argument to `krun`.

Figure 6.3 shows the implementation for the `readInt` function. It generates a random integer using the built-in random number generator of $\mathbb{K}$, writes it to `stderr`, and leaves the number on top of the `<k/>` cell. The priority annotation next to the rule allows overriding existing rules. By default, all rules, including the core Javalette rules, are assigned the priority of 50. Setting a lower number (49 in this case) gives a higher priority to the rule.

```
configuration
     <randomize>
         <jl/>
         <rand-seed> $SEED:Int </rand-seed>
         <rand-initialized> false:Bool </rand-initialized>
     </randomize>
```

Figure 6.2: Override the behavior of input functions

```
rule <k>
          readInt(.Args) => #logInt(jlRandomInt()) ...
     </k> [priority(49)]

syntax KItem ::= #logInt(Int)
rule <k>
          #logInt(I) =>
            writeln(Int2String(I), #stderr) ~> I ...
     </k>
```

Figure 6.3: Override the behavior of input functions

Once the language is compiled with these definitions, the generated interpreter works as an input/output generator. The `krun` command is used to run the input generator (Figure 6.4). It generates a random input for the given source file (`<source-file>`). The generated input and output are stored in `<input-file>` and `<output-file>` respectively.

```
krun <source-file> \
     -cSEED=<rng-seed> \ # seed for the random number generator
     -cRUN=1 \
     --output-file /dev/null \ # do not print the final config.
     > <output-file>  # generated expected output
     2> <input-file>  # generated input
```

Figure 6.4: Use krun to generate a random input

The input generator can be used on the fly when running the test suite, or inputs can be pre-generated and stored in the test suite.

7

Discussion

In this chapter, we reflect on our development process and results with respect to the research questions.

7.1 Formal semantics in $\mathbb{K}$

K-Javalette is the first formal semantics for Javalette. Therefore, detecting ambiguities in the language was a considerable part of the development process. All problems we have found in the semantics are caused by underspecification. These issues were usually caught by carefully reading the documentation.

As for the benefits of $\mathbb{K}$, adopting the framework is easy since semantic rules in $\mathbb{K}$ usually resemble functional programs and it does not require advanced logic or programming knowledge. It also provides shortcuts for solving common problems such as associativity and strictness annotations. In addition, being able to execute the semantics allowed us to follow a test-driven approach throughout the development. Finally, compilation warnings for non-exhaustive checks helped us find unhandled cases such as default values for uninitialized arrays.

$\mathbb{K}$ allows having nondeterministic semantic rules. In other words, one can define rules with the same preconditions but different results. This feature enables defining nondeterministic languages and exploring possible execution paths. However, this can be an obstacle in our case. Ambiguities in the Javalette documentation may lead to unintentionally writing nondeterministic rules in the formal semantics. Unfortunately, $\mathbb{K}$ does not automatically check for nondeterminism.

7.2 Modules and extensions

$\mathbb{K}$ features such as modules, configuration abstraction, and rule priorities allow extending languages modularly. Chapter 4 explains how language extensions were defined in K-Javalette. Once implemented correctly, one can conveniently choose a subset of extensions and compile the language.

Although $\mathbb{K}$ allows developing extensions as separate modules, it is unrealistic to do it without examining the interaction between the extensions. Otherwise, there can

be conflicts between syntax or semantic rules from different modules. $\mathbb{K}$ is able to automatically diagnose conflicts in the grammar. However, there is not an easy way to check conflicts in the semantics.

7.3 Using the semantics for testing

Having executable semantics unlocks great potential for applications. Chapter 6 explains how we use K-Javalette for testing compilers. We developed a random input generator using the semantics with a small add-on. Together with the reference interpreter, it is used for testing compilers with random inputs.

7.4 Related work

Ellison et al. [7] developed an executable semantics in the $\mathbb{K}$ framework for C. It was considered to be the most comprehensive formal semantics for the C language with a 99.2% success ratio in the GCC test suite. The C language standard intentionally leaves some unspecified behaviors. For instance, argument evaluation order and integer sizes are implementation-specific. Ellison et al. parameterize the semantics by taking advantage of K's modularity, whereas we chose to completely specify the behavior in similar cases.

KWasm is a formal semantics of WebAssembly in $\mathbb{K}$ [8]. Elrond Semantics is a formal semantics in $\mathbb{K}$ for the Elrond Virtual Machine for blockchains [4]. Elrond semantics extends KWasm with Elrond-specific features for blockchain operations.

SpecTest [13] is a $\mathbb{K}$-based compiler testing tool for Java and Solidity compilers. It consists of an executable semantics, an input fuzzer, and a program mutator. The input fuzzer generates random inputs for programs generated by the program mutator. The program mutator takes a seed program and generates a new program by applying a set of mutations. It also analyzes the execution and tries to find less used rules to increase the testing coverage.

7.5 Future work

In this project, we chose to implement two of the extensions. One natural follow-up is to add more extensions to K-Javalette. Object orientation is one of the most preferred extensions in the Compiler Construction course. It would be an interesting avenue for future work since it brings new top-level definitions, types, expressions, and scope rules for methods.

Currently, K-Javalette relies on the LLVM backend for input/output functions, which prevents using the $\mathbb{K}$ framework's Haskell backend. Implementing the I/O functionality in a backend-independent way to enable the Haskell backend would be beneficial. After that, it would be possible to use K-Javalette for reasoning about programs or proving the properties of the semantics itself.

7.6 Conclusion

The tools used in programming language development are of great importance. In this project, we have demonstrated the capabilities of the $\mathbb{K}$ framework in defining programming languages. We have developed an executable formal semantics for the Javalette language using $\mathbb{K}$. As a by-product of the formal semantics, we have acquired a reference interpreter that supports all features of Javalette including input/output functions. We have shown that languages formalized using $\mathbb{K}$ can be extended modularly by adding new features implemented in separate $\mathbb{K}$ modules. On top of the core language, we have developed extension modules for supporting arrays, structs, and pointers. At last, we have demonstrated an application of executable semantics in randomized testing of compilers by developing a random input generator.

Bibliography

[1] Javalette - extensions. `https://github.com/myreen/tda283/blob/master/project/extensions.md`. Accessed: 2021-02-12.

[2] K-Javalette. `https://github.com/bbyalcinkaya/k-javalette`. Accessed: 2022-05-10.

[3] K User Manual. `https://github.com/runtimeverification/k/blob/master/USER_MANUAL.md`. Accessed: 2022-05-05.

[4] Semantics of Elrond and Mandos. `https://github.com/runtimeverification/elrond-semantics`. Accessed: 2022-05-12.

[5] The Javalette language. `https://github.com/myreen/tda283/blob/master/project/javalette.md`. Accessed: 2021-02-12.

[6] 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.

[7] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. *ACM SIGPLAN Notices*, 47(1):533–544, 2012.

[8] Rikard Hjort. Formally verifying webassembly with kwasm towards an automated prover for wasm smart contracts. 2020.

[9] John Levine. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.", 2009.

[10] John R Levine, John Mason, John R Levine, Tony Mason, Doug Brown, John R Levine, and Paul Levine. *Lex & yacc*. " O'Reilly Media, Inc.", 1992.

[11] Daejun Park, Andrei Stefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–356, 2015.

[12] Grigore Roșu and Traian Florin Șerbănuță. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[13] Richard Schumi and Jun Sun. Spectest: Specification-based compiler testing. In *International Conference on Fundamental Approaches to Software Engineering*, pages 269–291. Springer, Cham, 2021.

[14] Traian Florin Şerbănuţă, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Roşu. The K primer (version 3.3). *Electronic Notes in Theoretical Computer Science*, 304:57–80, 2014.