CLOSING THE GAP IN THE LLVM BACKEND OF $\mathbb{K}$

BY

MICHAEL LEE ABIR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Professor Grigore Roșu

ABSTRACT

In this thesis, we further develop part of the $\mathbb{K}$ framework, a framework for specifying and executing the formal semantics of languages. We dive into the LLVM backend, one of the engines for concrete execution, and implement key functionality that is present in the other concrete execution engine. We then add a new interface that is unique to the LLVM backend, making this backend diverge from the other backend. Finally, with the backend caught up and divergent, we implement and evaluate pattern matching optimization strategies.

To my family, whose love and support will always keep me going

ACKNOWLEDGMENTS

I would like to give a special thanks to my advisor, Professor Grigore Roșu, who believed in me and persevered through all my challenges. I would also like to thank Dwight Guth, who made my work possible. Lastly, I would like to thank the members of the Formal Systems Laboratory, who helped me with many issues I faced while working on this project.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND 3
2.1 $\mathbb{K}$ 3
2.2 $\mathbb{K}$ Definitions 3
2.3 The LLVM Backend 5

CHAPTER 3 INPUT AND OUTPUT INTERFACE 7
3.1 The IO Interface 7
3.2 Modifications To The Interface 10
3.3 IO with Linux 10
3.4 Execution and Locking 12
3.5 Logging 12
3.6 Example Usage 13

CHAPTER 4 FFI 15
4.1 The Foreign Function Interface 15
4.2 FFI Type Conversion 18
4.3 Implementing the FFI Call 19
4.4 FFI Pointer Types 20
4.5 Example Usage 21

CHAPTER 5 PATTERN MATCHING HEURISTICS 22
5.1 Pattern Matching Matrices 22
5.2 The Heuristics 23
5.3 Evaluation of Heuristic 25

CHAPTER 6 CONCLUSION 27

REFERENCES 28

APPENDIX A FULL IMPLEMENTATIONS 29

APPENDIX B IMP 30

CHAPTER 1: INTRODUCTION

There are two main approaches to developing programming languages. Researchers often define languages according to mathematical descriptions in the form of formal semantics, whereas industries take the concrete approach of implement first, define later. Even with languages that have a concrete implementation and a formal definition, there is a large distinction between the two.

The C programming language is a mature language with both a formal definition, the C standard as defined in a readable human language, and multiple implementations, including *gcc* and *clang*. Despite its maturity, the distinction between its definition and implementation poses many issues of correctness. One is often left to think, *what is C*? More importantly, what is the *correct behavior* of a program in C? Naturally, one would reference the C standard to understand behavior, but how can one understand the full behavior of a large program when the standard isn't executable? The next logical step would involve compiling and executing a program with a compiler such as *gcc* or *clang*. Since these implementations could have bugs, or their own implementation choices, how can we conclude which behavior would be correct?

To answer these questions, we combine the definition of a language with its implementation, and arrive at the *executable formal semantics* of a language. There are multiple efforts to define executable semantics of programming languages, and in this thesis we focus on one such effort. The $\mathbb{K}$ framework is a framework that allows one to not only define the formal semantics of a language, but also execute programs against these semantics. This allows one to execute a program according to the formal semantics, and the behavior of the program will therefore be *correct by definition*.

We focus on catching up one of the execution engines of $\mathbb{K}$ in this work, in an effort to solidify it as *the* execution engine of $\mathbb{K}$. We do this by first implementing the missing input and output functionality, and then developing a foreign function interface that allow languages defined in $\mathbb{K}$ to be usable in the real-world. Lastly, we implement new pattern matching heuristics to speed up pattern matching and help make the LLVM backend faster than the rest of the execution engines.

We first cover the necessary background information in chapter 2, where we define key

concepts of $\mathbb{K}$ and the LLVM backend, including basic $\mathbb{K}$ definitions and sort interpretation in the LLVM backend. In chapter 3, we discuss the previous interface for input and output, and both update this interface to handle errors and describe the implementation details. We then define the new foreign function interface in chapter 4, including the additions to the $\mathbb{K}$ builtins and the implementation in the backend. Lastly, we discuss optimizations of the LLVM backend through the pattern matching heuristics in chapter 5.

CHAPTER 2: BACKGROUND

$\mathbb{K}$ is a language and a rewriting engine for executable semantics definitions, which automatically generates a parser and interpreter from a language definition [1] [2] [3]. Programs written for languages defined in $\mathbb{K}$ can be verified and symbolically executed using tools that are automatically generated from the language definition. Using $\mathbb{K}$, one can define a wide range of concurrent programming languages, type systems, or other formal languages, and its execution engines are extremely powerful in bridging formal semantics with real systems. We explore $\mathbb{K}$, the LLVM backend of $\mathbb{K}$, and systems defined in $\mathbb{K}$ that utilize the highly optimized LLVM execution backend.

2.1 $\mathbb{K}$

Since $\mathbb{K}$ is a general-purpose framework for defining a wide range of languages and systems, one would hope that such a framework is usable for real applications. The power of $\mathbb{K}$ is not only in its ability to provide this bridge between definition and implementation, but also its ability to compete with real-world implementations of some systems. In fact, the **KEvm** project is one such system [4], which has the full formal specification of the Ethereum Virtual Machine, developed by the Formal Systems Laboratory at the University of Illinois and Runtime Verification Inc.

The $\mathbb{K}$ framework has made substantial contributions to the formal specification of C, as well as many other languages. The C Semantics project defines the "negative" semantics of C, which accepts correct programs and reject those with undefined behavior [5]. Such a tool is monumental in the space of formal verification, and its proprietary extension, *RV-Match*, is used commercially in mission-critical applications to discover undefined behavior.

2.2 $\mathbb{K}$ DEFINITIONS

$\mathbb{K}$ is, at its core, a semantics framework and programming language. The main components of $\mathbb{K}$ programs are the module definitions, where both syntactic terms and semantic rules are defined. In $\mathbb{K}$, one defines the grammar of a language, its configuration cells, and the semantics through rewrite rules.

2.2.1 Defining Grammars

$\mathbb{K}$ is order-sorted, allowing a much more natural definition of grammars. Grammars are defined with the *syntax* keyword. For example, to define a simple grammar for arithmetic expressions that use addition, one could write the following $\mathbb{K}$ definition:

```
syntax AExp  ::= Int
               | AExp "plus" AExp
```

Examples of terms that will be of sort *AEx* are 1, 3 *plus* 5, and 7 *plus* 0 *plus* 11. $\mathbb{K}$ will automatically generate a parser from this grammar definition.

2.2.2 Configurations

Configurations in $\mathbb{K}$ allow us to define complex states while maintaining auxiliary computation cells. In $\mathbb{K}$, matching is done on multiple configuration cells in parallel, and therefore our rewrite rules can be quite powerful.

To define a configuration that contains two cells, the *k* computation cell that begins with an *AExp* and the *state* cell, as well as a container cell, we use the **configuration** keyword:

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

This will allow us to write rules that match on both the program fragments and the current state of the program.

2.2.3 Rewrite rules

As mentioned, rewrite rules allow us to define the semantics of a language. Rewrite rules rely on pattern predicates that can match on any configuration fragment. For example, if we want the interpretation of the *plus* operator to be the addition operator over integers, we can add the following rule in our $\mathbb{K}$ definition:

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

Since our language definition is very simple, and does not have any state to store, we can define rules as above that will apply to the beginning of the computation cell. What we see

here is the power of the rewriting engine and pattern matching engine, since one rule can provide the full semantics that we had in mind for our *plus* operator. Note that certain sorts and functions are implemented as builtins, including the *+Int* operator, which are predefined for both convenience and optimization reasons in some cases.

2.3 THE LLVM BACKEND

We now explore one of the execution engines of $\mathbb{K}$, the LLVM backend. This backend was written from the ground up to be a replacement for the previous execution engine, the OCaml backend, and is designed to be the default execution engine for $\mathbb{K}$. The majority of the backend is written in C++, which handles the majority of the LLVM code generation and the implementation of frontend hooks; pattern matching logic is written in Scala, and, of course, some glue code and minor functions are written in LLVM assembly.

The core concepts relevant to this thesis are the translation between the $\mathbb{K}$ frontend, where certain functions and sorts are declared, and the LLVM backend, where these functions are implemented. There are three main object types that the backend utilizes in this thesis, including *block*, *string*, and *mpz_t*.

2.3.1 Integer Types

$\mathbb{K}$ supports the full set of integer types, whereas C++ and the underlying systems assume fixed size integers. For this purpose, we use the *gmp* library for arbitrary precision arithmetic to represent the integers, which gives us a very simple interface for arbitrary integer manipulation. The functionality described in this thesis mostly relies on fixed size integers that correspond to machine sizes, and therefore *gmp* is used for storing and extracting integers of a fixed size.

2.3.2 String Types

The *string* type can represent both arbitrary strings of sort **String**, as well as the **Bytes** type. This type is defined in the backend, and shares some similarities with the *block* type. A *string* is composed of a *blockheader*, which contains useful information such as size and the tag, and the data, which is a unbound *char* array. This type should be familiar to those who have used strings in other languages.

2.3.3 Block Types

Lastly, and perhaps most importantly, is the generic *block* type, which is the representation of most sorts in the LLVM backend. A block, like strings, is composed of a blockheader, together with an array of pointers called the *children*. The children of a block depend on the sort that the block represents. Symbols are blocks that have no children, which is indicated by a 1 in the lowest bit of the block pointer. As we will see in later sections, objects of nested types have other objects as children.

CHAPTER 3: INPUT AND OUTPUT INTERFACE

In order to make the LLVM backend comparable to the OCaml backend, we first had to catch up the LLVM backend's functionality to that of the OCaml backend. One of the most pressing functionality that was missing was an interface for input and output (IO), which is key for defining real-world languages in $\mathbb{K}$. Since the interface was already defined in the frontend, this chapter mostly focuses on the additions to the interface and the implementation of this interface in the LLVM backend.

3.1 THE IO INTERFACE

We'll begin by specifying the relevant interface for input and output in the $\mathbb{K}$ frontend, which has been implemented in the OCaml backend. We include the error types, as well as the function declarations *domains.k*, part of the $\mathbb{K}$ builtins library. This interface is responsible for Linux-style file operations, which allows languages defined in $\mathbb{K}$ to have their own IO and file operations libraries. This interface allows for most basic file operations, such as opening writing to files, as well as more advanced operations such as locking. Logging and command execution allows languages that use this interface to develop quite complex IO functionality, which is quite necessary for real-world languages.

3.1.1 IO Errors

In table 3.1, we show all the possible IO errors that the backend returns after an IO operation. These include the standard Linux System Errors from *errno.h*, the end of file **EOF** error, and our own error codes, such as **noparse** and **unknownIOError**.

Table 3.1: IO Errors

noparse	unknownIOError	EOF	E2BIG
EAGAIN	EBADF	EBUSY	ECHILD
EDEADLK	EDOM	EEXIST	EFAULT
EFBIG	EINTR	EINVAL	ESOCKTNOSUPPORT
EISDIR	EMFILE	EMLINK	ENAMETOOLONG
ENFILE	ENODEV	ENOENT	ENOEXEC
ENOLCK	ENOMEM	ENOSPC	ENOSYS
ENOTDIR	ENOTEMPTY	ENOTTY	ENXIO
EPERM	EPIPE	ERANGE	EROFS
ESPIPE	ESRCH	EXDEV	EWOULDBLOCK
EINPROGRESS	EALREADY	ENOTSOCK	EDESTADDRREQ
EMSGSIZE	EPROTOTYPE	ENOPROTOOPT	EPROTONOSUPPORT
EIO	EOPNOTSUPP	EPFNOSUPPORT	EAFNOSUPPORT
EADDRINUSE	EADDRNOTAVAIL	ENETDOWN	ENETUNREACH
ENETRESET	ECONNABORTED	ECONNRESET	ENOBUFS
EISCONN	ENOTCONN	ESHUTDOWN	ETOOMANYREFS
ETIMEDOUT	ECONNREFUSED	EHOSTDOWN	EHOSTUNREACH
ELOOP	EOVERFLOW		

3.1.2 IO Functions

We place all the relevant declarations and definitions of the IO interface in the $\mathbb{K}$ frontend module, **K-IO**, in listing 3.1. Those familiar with Linux system calls and library functions should see great similarities in this interface. While not specified, the majority of functions listed have the *hook* attribute that connects these functions to their corresponding implementation in the various backends. Some functions, such as *#open*, have an alternative calling method for convenience, and some objects syntactic sugars for comprehensible code, including the *#stdin* object.

Listing 3.1: K-IO Module

```
syntax Int ::= "#open" "(" String ")"
             | "#open" "(" String "," String ")"
             | "#tell" "(" Int ")"
             | "#getc" "(" Int ")"

syntax String ::= "#read" "(" Int "," Int ")"

syntax K ::= "#close" "(" Int ")"
           | "#seek" "(" Int "," Int ")"
           | "#seekEnd" "(" Int "," Int ")"
           | "#putc" "(" Int "," Int ")"
           | "#write" "(" Int "," String ")"
           | "#lock" "(" Int "," Int ")"
           | "#unlock" "(" Int "," Int ")"

rule #open(S:String) => #open(S:String, "r+")

syntax Int ::= "#stdin"
             | "#stdout"
             | "#stderr"

rule #stdin => 0
rule #stdout => 1
rule #stderr => 2

syntax KItem ::= "#system" "(" String ")"
       | "#systemResult" "(" Int "," String "," String ")"

syntax K ::= #logToFile(String, String)
```

Note that functions whose return type is *void* are of sort K in listing 3.1. Since every sort is a subsort of K, these functions either return the **.K** object, indicating an empty object of sort K, or they will return an *IOError*, which is also a subsort of sort K.

3.2 MODIFICATIONS TO THE INTERFACE

Since the previous IO interface, as well as its implementation in the OCaml backend, did not provide any error codes upon failure, this addition is necessary in the frontend interface. We accomplish this by introducing new return sorts, *IOString* and *IOInt*.

3.2.1 IO Sorts

To incorporate the errors into the interface, we add two new sorts to the **K-IO** module, namely *IOInt* and *IOString*. Note that the sort *IOError* existed previously. *IOInt* has *IOError* and *Int* as subsorts, and, of course, *IOString* has *IOError* and *String* as subsorts. Intuitively, an *IOInt* represents the result of an IO call, where an *Int* will be returned upon success, and a corresponding *IOError* will be returned upon failure. Similarly, *IOString* are returned from IO calls that return strings upon success.

Each IO call is modified to return an *IOInt* if it previously returned an *Int*, or an *IOString* if it previously returned a *String*. With these new returned sorts, we add two functions to retrieve the data from them. We accomplish this by adding the following two projection functions in $\mathbb{K}$:

```
syntax String ::= #projectString(IOString) [function]
syntax Int ::= #projectInt(IOInt) [function]
rule #projectString(S:String) => S
rule #projectInt(I:Int) => I
```

We have two functions, *#projectString* and *#projectInt*, which are partial functions defined on *IOString* and *IOInt*, respectively. If the IO sort contains data, rather than an *IOError*, then this data is returned. As these are partial functions, they will get stuck if an *IOError* is returned; since this provides poor error handling, user-defined functions could be used instead.

3.3 IO WITH LINUX

Implementing our interface requires large amounts of support from the operating system, and Linux provides such support. Since the current LLVM backend supports Linux and Linux-like systems, our choice of the Linux system calls and library as our sole support was justified.

The IO interface defined closely resembles that of the Linux system calls and library for IO, which treats each IO device as a file and provides an associated file descriptor as its handler, and therefore has a very intuitive implementation. Rather than describe the implementation of each function, we will explain certain functions and give an overview of the rest of the functions.

3.3.1 Implementing #open

The *#open* function is the most common entry point of the IO interface and sets the tone for the rest of the functions. This function, which is often called with access control flags, is responsible for parsing the control mode flags, opening the file using the system call *open*, and returning a processed file descriptor that can be used in the frontend.

For generality, we set the permission bits of all files to the following default access:

$$S\_IRUSR \mid S\_IWUSR \mid S\_IRGRP \mid S\_IWGRP \mid S\_IROTH \mid S\_IWOTH$$

These access bits indicate that the file can be read and written by the user, group of the user, and others, but note that this file is not executable by default.

Once the control mode flags are parsed, we set the flags to be passed to the system call using a scheme that follows the C *fopen* modes. If the mode contains a +, then we set the mode to **O_RDWR**; otherwise, we set it to **O_RDONLY** if the mode is r, and **O_WRONLY** if the mode is w or a. For both write and append modes, w and a respectively, we set the **O_CREAT** flag, which creates the file if it does not exist. If the mode specified is write, then we set the **O_TRUNC** flag to truncate, or clear, the file before it is opened; if the mode is append, then we set the **O_APPEND** flag. The mode x sets the **O_EXCL**, which ensures that opening this file will create it, and will fail otherwise. Lastly, if the mode is e, the **O_CLOEXEC** flag is set, specifying *close-on-exec*, which is critical for multithreaded programs.

Once the mode flags are set up, the *open* system call is called on the filename, mode flags, and access flags, and returns a corresponding error from table 3.1 if the call fails, or it returns the file descriptor as an *Int*, both of which are subsorts of *IOInt* as defined previously.

3.4 EXECUTION AND LOCKING

We provide two additional mechanisms, one for executing commands and one for locking and unlocking files. The locking and unlocking functions, like some of the other IO functions implemented, are a simple wrapper to the Linux calls. Both of these functions, after processing the arguments into a file descriptor and length to lock, create a lock whose attributes correspond to either locking or unlocking a section of the file. The locks for locking and unlocking are identical, except that the *l_type* is set to **F_WRLCK** for locking, and **F_UNLCK** for unlocking. With the lock and file descriptor, a call to *fcntl* is made, which performs the actual locking mechanism.

3.4.1 The #system Function

Executing commands requires a more complex setup, though its underlying method of execution is a call to *fork*, as well as calls to either *execl* or *system*. The *#system* function, which is our function for execution, accomplishes its task by creating a child process, setting up pipes for the output of the child, executing the corresponding command, and returning both the status and outputs of the command.

Creating the child process is simple through the use of *fork*, and a pipe is created for both *stdout* and *stderr*. The child process then connects its *stdout* and *stderr* to the corresponding pipes, and, if the command to execute is not empty, will execute that command through a call to *execl*, followed by an *exit* to terminate itself. If, however, the command is empty, then the child calls *system(NULL)*, which returns a nonzero value if a shell is available, or 0 otherwise; this value is passed into *exit*.

While the child process is executing, the parent process reads the output of the child process through the pipe, and calls *waitpid* to synchronize with the termination of the child. At this point, the parent process constructs the return object from the *#system* function, which is composed of the return status of the child process, the buffer containing the standard output of the child, and the buffer containing the standard error.

3.5 LOGGING

Logging functionality is crucial for many applications, and therefore is critical in the IO interface. In this section, we define the *#logToFile* function and its implementation in the

LLVM backend. There are two main components to logging, which are the registering of logs, and the writing of logs, which are accomplished by the *log* and *flush_logs* functions respectively.

3.5.1 Registering Logs

When a frontend application calls *#logToFile*, it translates to a call to *log*, or, more precisely, **hook_IO_log**. This function takes the full file path and a message to log. Our logging architecture uses a static map from file paths to a list of messages to record which messages must be logged, and provides a mechanism for flushing these logs to the actual log files. Our *log* function is responsible for registering the log files and messages in the log map, as well as setting up the flushing mechanism. By using the Linux call *atexit*, the first call to *log* will register the function *flush_logs* to be called when the process terminates. In effect, logs are recorded whenever they are sent to the backend, but logs are only written to files when the program exits.

3.5.2 Flushing Logs To Files

Once the program exits and the logs are ready to be flushed to their corresponding files, *flush_logs* is called. Logs are unique to a single process, and therefore require unique identifiers. We use the *pid*, or *process ID*, to distinguish logs files with identical names from different processes. For example, should two instances of a logging program be executed with the log file *output.log*, then two log files will be created, named pid_1*_output.log* and pid_2*_output.log*, where pid_1 and pid_2 are the *pid* of the two processes.

To flush all the logs, we iterate through all the log pairs that are registered in our log map. For each one of these logs pairs, we open up the file associated with the pair, with the addition of the process *pid* as the prefix for the filename. We open each file for appending, so that the logs accumulate in the file, and write each message out in the order it appears in. Since we use a list, our messages appear in the order that they arrived.

3.6 EXAMPLE USAGE

We can use the IO interface to build a custom interface for IO, or to add some minor functionality to a language. Though the *log* function is provided in the IO interface, we

could choose to implement our own logging mechanisms within a $\mathbb{K}$ program. We can easily accomplish this goal with the following addition to the **IMP** syntax in appendix B:

```
syntax Stmt ::= "writeMsgToLog" "(" String ")"
```

To add meaning to our new syntactic entity, we use the following rule, which opens the file *output.log* for appending, projects the file descriptor to an *Int*, and writes the message to the log.

```
rule writeMsgToLog(S) => #write(projectInt(#open("output.log", "a+")), S)
```

With just one syntax addition and a single rewrite rule, we have added our own custom logging mechanisms to our toy language. We can see how simple it is to add more complex functionality to a language, or we can just include the full IO functionality in our toy language. The IO interface defined in this chapter allows us to develop a rich set of libraries for IO operations in $\mathbb{K}$-based languages.

CHAPTER 4: FFI

With the addition of the IO interface implementation, the LLVM backend was nearly caught up to the OCaml backend in terms of functionality. However, an important piece was missing: the Foreign Function Interaface (FFI). In this chapter, we discuss the FFI in the $\mathbb{K}$ frontend and the corresponding implementation in the LLVM backend. The main attraction of this interface is the *ffiCall* function, which accepts both fixed and variadic arguments, and our support for primitive types, such as *void* and standard machine integer types, as well as *struct* types.

4.1 THE FOREIGN FUNCTION INTERFACE

The FFI is similar in many aspects to *libffi*, the common library for FFI calls [6], largely due to our usage of *libffi* as the underlying logic of the interface. We present the new interface through the types that are supported, as well as the functions that are implemented.

Table 4.1: Available FFI Types

void	uint8
sint8	uint16
sint16	uint32
sint32	uint64
sint64	float
double	uchar
schar	ushort
sshort	uint
sint	ulong
slong	longdouble
struct (List)	

4.1.1 FFI Types

The types in table 4.1 include basic types, such as *void* and *uint*, as well as nested *struct* types. Here, *struct* (List) is a type constructor that takes a list of types. For example, consider the following struct **Point** in C:

```
struct Point {
uint8_t x;
uint8_t y;
};
```

The corresponding FFI type for **Point** would be *struct(uint8, uint8)*.

4.1.2 Function Address

We also define a new function, *#functionAddress*, in the **FFI** module.

```
syntax Int ::= "#functionAddress" "(" String ")"
```

This function takes a function name and returns its corresponding address in the current address space. This function is necessary for the functions that perform the FFI call, which we will define shortly, as these functions take the address of a function.

4.1.3 FFICall

Lastly, we show the functions that provide the actual FFI functionality. These functions are responsible for taking a function, its arguments, and corresponding types, and performs the actual function call to the foreign function.

We first define the most common function that is used, which corresponds to a foreign function with fixed arguments.

```
syntax Bytes ::= "#ffiCall" "(" Int "," List "," List "," FFIType ")"
```

As mentioned previously, this function takes in the function address as an integer in the first argument. The next argument is a list of bytes, where each element, in order, is a byte representation of an argument to the foreign function. The second list argument is the list of types, which are of sort *FFIType*, where each list element corresponds to one of the foreign function arguments. Lastly, the return type of the foreign function is specified as an *FFIType*.

The next function we define performs a foreign function call as well, with the addition of variadic arguments. We accomplish this by passing in a list of types for the fixed arguments, as well as a list of types for the variadic arguments. Note that the list of values passed in contains all the arguments; the backend will have enough information to split the arguments based off of the lists of types.

```
syntax Bytes ::=
       "#ffiCall" "(" Int "," List "," List "," List "," FFIType ")"
```

Again, a function address and return type are necessary.

Lastly, we define the general method for calling both functions with fixed arguments, and functions with fixed and variadic arguments. It is defined as follows:

```
syntax Bytes ::=
       "#ffiCall" "(" Bool "," Int "," List "," List "," Int "," FFIType ")"
```

For reference, we show the declaration of this function with labeled arguments:

$$\#ffiCall\ (\ isVar, addr, args, types, NFixed, ret\ )$$

This function is both declared and completely defined in the $\mathbb{K}$ frontend module **FFI**. It is responsible for calling the corresponding fixed or variadic FFI call, which is accomplished with the following two $\mathbb{K}$ rules:

```
rule #ffiCall(false, Addr::Int, Args::List, Types::List, _, Ret::FFIType)
  => #ffiCall(Addr, Args, Types, Ret)

rule #ffiCall(true, Addr::Int, Args::List, Types::List,
              NFixed::Int, Ret::FFIType)
  => #ffiCall(Addr, Args, range(Types, 0, size(Types) -Int NFixed),
              range(Types, NFixed, 0), Ret)
```

When *isVar* is *false*, this means that the function being called is not taking variadic arguments, and therefore we call the non-variadic *#ffiCall* with the corresponding arguments, and ignore the *NFixed* parameter.

When *isVar* is *true*, and therefore we would like to call the variadic function, we process the list of types. Since *NFixed* represents the number of fixed arguments, we take the first *NFixed* types as a list of types, and then the remaining types as a list of variadic types, all of which we pass into the variadic form of the FFI call.

The generalized FFI call function allows us to have a uniform calling interface that isn't concerned with the presence or lack of variadic arguments. Though there is no difference in calling the generalized function or the specific function necessary, it may be useful to solely rely on the generalized function as the entry point to the interface.

4.2 FFI TYPE CONVERSION

In this section, we describe the implementation and use of FFI types in the LLVM backend, including type unwrapping and the corresponding calls to *libffi*. A link to the code can be found in appendix A.

4.2.1 FFI Type Conversion

We begin by discussing the conversion between objects of sort **FFIType** from the frontend, and the interpretation of that object in, which is eventually passed into *libffi*. This is done in the *getTypeFromBlock* function.

Every **FFIType** object is passed to the backend in the form of a **block pointer**, of which the header contains the sort **FFIType** and the first child of which is the symbol that corresponds to a type, if that child does not have its own children.

To convert a type to the corresponding *libffi* type, the type object is passed into *getTypeFromBlock*, which returns the final *ffi_type*. Each basic type, including *void*, has its own function that returns the tag that corresponds to the type. The symbol from the first child is then compared to each type tag, which is returned from the aforementioned function, until a match is found, and then the corresponding *ffi-type* is returned from *libffi*.

For example, if the first child of an object from the frontend has a symbol that matches the symbol of the *void* type, then the object *ffi_type_void*, defined in *libffi*, is returned.

However, if the child *does* have its own children, then the type is actually a *struct* type, and must be handled recursively. If a type object from the frontend is identified as a struct type, a new *ffi_type*, which is a struct that is defined in *libffi*, must be constructed. The fields that must be set are the type, and since we only deal with one non-primitive type, which is set to **FFI_TYPE_STRUCT**, and whose elements must be allocated and set. To get the elements, we recursively call *getTypeFromBlock* on each one of the children, and set the corresponding element in the *ffi_type*, making sure to *null*-terminated the list of elements. Since memory must be allocated for the struct types and its children, every reference is stored in a static vector, and the elements are freed when the *ffiCall* function succeeds or fails.

To understand the conversion, we refer to the **Point** structure defined in section section 4.1.1. Naturally, since it is a struct type, a new *ffi_type* structure is allocated, and its type is set to **FFI_TYPE_STRUCT**. Next, each child is recursively analyzed; since each child is the primitive type *uint8*, its simple primitive *libffi* type is returned, which is *ffi_type_uint8*. Therefore, the final *ffi_type* structure contains two elements, both of which are *ffi_type_uint8*.

4.3 IMPLEMENTING THE FFI CALL

We now focus on the actual meat of the FFI implementation, which is the *ffiCall* function. This function is responsible for interpreting the arguments from the frontend and executing the foreign function call. There are three main tasks completed in this function. In the backend, we implement one generalized function for both fixed and variadic arguments, and provide specialized hooks that call the generalized function. First, arguments are processed, including processing the types as mentioned above. Next, an FFI context is created and prepared. Lastly, the function is executed with the corresponding arguments through *libffi*, and the result is processed and returned to the frontend.

Processing the arguments involves checking the variadic arguments, if they exist, and ensuring that the number of arguments is equal to the number of fixed arguments and variadic arguments. We then process the list of **FFIType** objects to get a corresponding array of *ffi_types*, then process the list of data, which is received as **Bytes**, into an array of the data in the **Bytes** objects, and get the return type using *getTypeFromBlock*.

Once the arguments are processed, and we have the arrays of *ffi_types*, the return *ffi_type*, the number of arguments, and the number of fixed arguments if there are variadic arguments, we prepare the *Call InterFace* object, *CIF*, through a call to *ffi_prepare_cif*, which is defined in *libff*. Upon success, the *ffi_status* returned is **FFI_OK**; other return statuses are handled by throwing labeled exceptions.

The final call takes the newly created *CIF*, the address of the foreign function, the arguments to the function, and a pointer for the return value that will be filled. The function, *ffi_call*, is defined in *libffi*, and is responsible for performing the actual function call. When it completes, the final return value that has been filled in is processed and returned to the frontend application.

4.4 FFI POINTER TYPES

Though there currently is no reference to pointer types in the frontend interface, this would require only a minor addition to enable, as we have already implemented this in the LLVM backend. The interface, on the backend side, is defined as follows:

```
string * hook_FFI_alloc(block * kitem, mpz_t size);
block * hook_FFI_free(block * kitem);
block * hook_FFI_bytes_ref(string * bytes);
mpz_ptr hook_FFI_bytes_address(string * bytes);
bool hook_FFI_allocated(block * kitem);
```

We have the ability to allocate and free memory in the backend in a way that both abstracts the memory, yet provides concrete values for addresses. With *alloc*, defined in **hook_FFI_alloc**, we pass in *kitem*, which will be used as our abstracted reference of the pointer and is our handle for freeing the memory, and *size*, which, of course, is the number of bytes to allocate. Upon success, this function returns a **string pointer**, which is the backend representation of the frontend sort **Bytes**. For those familiar with languages like C, these bytes returned are analogous to the value returned by **malloc**, in which a pointer value is returned. In the process, a call to *alloc* will register *kitem* as the handle for this address if the reference does not exist yet, or will return the corresponding address to this reference.

When a reference is no longer needed, a call to *free*, or **hook_FFI_free**, will free any memory that is handled by *kitem*, similar to how the function **free** in C will free the memory associated with a reference, or pointer.

Often times it is useful to retrieve the bytes associated with a reference handle, as the handle itself does not contain this information. This is accomplished through a call to **hook_FFI_bytes_ref** with a valid bytes argument, but will throw an error if such a reference does not exist.

To convert between the **Bytes** representation of an address and an integer representation, we provide the function **hook_FFI_bytes_address**. We also provided the final function, **hook_FFI_allocated** to check if an address, represented as a **Bytes** sort, contains a reference handle. This is often called before a call to *ref*, as it is a way to handle the *no reference* error that could occur.

4.5 EXAMPLE USAGE

Suppose we have the following C function, defined in a linked library:

```
int returnFive(void) {
    return 5;
}
```

What if we wanted to build a wrapper to this function in our toy language in appendix B? Since we know this function returns an integer, we add it as an arithmetic expression, of which *Int* is a subsort.

```
syntax AExp ::= "foreignFive" "(" ")"
```

With this, we define a new term, *foreignFive()*. Now that we have a new syntactic entity, we would like to give it meaning, and we do so with the following rewrite rule:

```
rule foreignFive ( ) =>
    Bytes2Int(#ffiCall(#functionAddress("foreignFive"),
                       ListItem(.Bytes),
                       ListItem(#void),
                       #sint), LE, Signed)
```

This rule rewrites our newly defined token to an integer. The key point is the reference to *#ffiCall*, where we pass in the address of an external function *foreignFive*, with a list of empty **Bytes** and a list containing the **FFIType** corresponding to *void*, along with the return type of *#sint*. The rest of the rule converts the returned **Bytes** to an **Int**.

Assuming that the function symbol *foreignFive* is linked against the program, and that its type interface is the same as the one specified above, then the term *foreignFive()* will rewrite to the returned value of the external function *foreignFive*, which is 5. With this example, we can see how simple it is to provide custom wrappers to external functions, and even build our own interface for external libraries using the foreign function interface defined in this chapter.

CHAPTER 5: PATTERN MATCHING HEURISTICS

We now shift our focus from simply closing the gap between the LLVM backend and other execution backends, to optimizing and finely tuning our pattern matching engine. In this chapter, we will focus on implementing Maranget's pattern matching heuristics [7].

5.1 PATTERN MATCHING MATRICES

We follow Maranget's work by compiling pattern matching to decision trees. We think of pattern matching as a clause matrix of patterns and actions; if a row is matched, the corresponding action is taken. Patterns are built up from *constructor* patterns, pattern combinations, or *wildcard* patterns. A wildcard pattern will match any pattern, which is quite useful for our purposes. Recall that in $\mathbb{K}$, we can match on partial configurations, and we can think of matching the rest of the configuration as wildcard patterns.

5.1.1 Decomposing the Clause Matrix

The process of compiling pattern matching matrices to decision trees uses two decompositions of the clause matrix. The first decomposition is the specialization by a constructor, and the second is the default matrix.

Specializing by a constructor c allows us to decompose the clause matrix into a simpler matrix, which makes the assumption that the first pattern has c as a head constructor. This decomposition results in a matrix that retains the rows of the original clause matrix that admit c as a head constructor, and replaces the pattern with its subterms.

The default matrix is a decomposed clause matrix, which retains rows of the clause matrix that do not have a head constructor in the first column.

The specialized matrix S and default matrix D are, in a sense, simplified clause matrices, and can be used in the matching process instead of the original clause matrix. This concept is explained in lemma 1 of [7].

5.2 THE HEURISTICS

The pattern matching engine in the LLVM backend used some variation of the f heuristic in [7], but we wanted to provide a better alternative that would produce the best results. We implement all of Maranget's heuristics, allowing applications to finely tune which heuristics are used, and allow heuristics combinations for greater optimizations. We present the heuristics for column index choice in table 5.1, and explain the pseudo-heuristics in the implementation details.

Table 5.1: Maranget Heuristics

Alias	**Heuristic**	**Brief Explanation**
f	first row	favors columns whose first row pattern is a constructor
d	small default	based off the number of wildcard patterns
b	small branch factor	related to the number of head constructors in the column
a	arity	the negation of the sum of constructor arities
l	leaf edge	number of leaf nodes that are children of the switch node
r	rows	total number of rows in the decomposed matrices
n	needed columns	the number of rows such that the column is needed
p	needed prefix	size of largest prefix of rows where the column is needed
q	constructor prefix	approximation of p by checking constructor patterns

The implementation of these heuristics is done in Scala, which is exclusively used at compile-time for generating decision trees, since pattern matching is built into the language and it provides a simpler approach to such a task. Each named heuristic is its own object and implements the *computeScoreForKey* method. The pseudo-heuristics N, L, and R return a default score of 0, but implement the $breakTies$ method. We discuss our original heuristic as well, and provide the comparison in the evaluation section.

Heuristic Default

In our default heuristic, we iterate over the patterns in a column and return our accumulated result if we find a pattern in a new action priority. For each pattern, we provide a score that is determined by the pattern type, and add this score to the result.

Heuristic f: First Row

We iterate over the patterns in the column and return 1 if the action priority of the pattern is not equal to the head action priority, similar to the default heuristic, or return 0 if we find a *wildcard* pattern. By default, this method returns 0.

Heuristic d: Small Default

This heuristic returns the negative of the number of default patterns.

Heuristic b: Small Branch Factor

If the column has an incomplete signature, we return the negative of the size of the signature minus one; otherwise, we return the negative of the size of the signature.

Heuristic a: Arity

We iterate through constructors in the column signature and expand the column fringe by the constructor, and return the negative sum of the size of these expansions.

Heuristic l: Leaf Edge

We specialize on each constructor in the column's signature and count the number of specialized matrices that have a best row index. To this value, we add 1 if the default matrix has a best row index.

Heuristic r: Rows

In this heuristic, we iterate of the constructors in the signature, and then over the patterns, and count the number of patterns that are specialized on this constructor. We then iterate over the patterns if the signature is incomplete, and count the number of default patterns in the column. The resulting heuristic is the negation of this value.

Heuristic n: Needed Columns

We count the number of rows for which the column is necessary.

Heuristic p: Needed Prefix

We count the number of rows for which the column is necessary, but return if we reach a new priority, indicating that we passed the prefix.

Heuristic q: Constructor Prefix

This heuristic has a similar functionality to heuristic p, in that it returns if we reach a new priority; however, we approximate by counting the number of patterns which are not wildcards.

5.3 EVALUATION OF HEURISTIC

We evaluated the heuristics on the toy language **IMP**, which is a simple imperative programming language with arithmetic and boolean expressions, control flow statements, and loops. The full $\mathbb{K}$ definition of **IMP** can be found in appendix B. Our evaluation strategy looked at the average path length taken in the compiled decision tree, which is a good indicator of the efficiency of our heuristics. We show the 20 best results in table 5.2.

Table 5.2: Heuristics Results

Heuristic	**Size**	**Shared Size**	**Max Path Length**	**Average Path Length**
l_bR	9980.0	3544.0	109.0	35.71444759206799
lqbR	9980.0	3544.0	109.0	35.71444759206799
qblR	10208.0	3586.0	109.0	35.792953929539294
qbrN	10208.0	3586.0	109.0	35.792953929539294
qbrL	10208.0	3586.0	109.0	35.792953929539294
qbrR	10208.0	3586.0	109.0	35.792953929539294
qbnN	10208.0	3586.0	109.0	35.792953929539294
qbnL	10208.0	3586.0	109.0	35.792953929539294
qbnR	10208.0	3586.0	109.0	35.792953929539294
qbpN	10208.0	3586.0	109.0	35.792953929539294
qbpL	10208.0	3586.0	109.0	35.792953929539294
qbpR	10208.0	3586.0	109.0	35.792953929539294
_blR	10214.0	3587.0	109.0	35.75270562770563
_brN	10214.0	3587.0	109.0	35.75270562770563
_brL	10214.0	3587.0	109.0	35.75270562770563

In our evaluation, we see that most of the metrics are quite similar for the 20 best heuristics combinations; however, for the language defined in appendix B, we find that the combination l_bR and $lqbR$ perform identically. Therefore, were this a real language, we would have two rather good choices for our heuristic.

This type of evaluation is crucial in fine tuning applications when using the LLVM backend. Each language definition may benefit from a different combination of heuristics, and finding out the best combination requires evaluating all the possible heuristics combinations, or at least the ones that can be justified to perform well. $\mathbb{K}$ uses a default combination of heuristics, which appear to perform well for general applications; however, each application may benefit from its own analysis. Certain programs may benefit from different combinations despite running on the same $\mathbb{K}$ application, though such micro-optimizations may yield negligible improvements.

CHAPTER 6: CONCLUSION

The efforts put forth by the $\mathbb{K}$ framework allow us to define complex languages and systems, and the application of such languages can be used as the *de facto* execution platforms of large systems, such as the Ethereum Virtual Machine, or as a means to verify mission critical applications, as in the case of the C Semantics Project. We have the explicit understanding that definition and execution should go hand in hand, and the concrete execution in the LLVM backend provides us such an opportunity.

By implementing the enhanced IO interface, and both developing and implementing a new foreign function interface, we bring the LLVM backend of $\mathbb{K}$ one step closer towards deprecating the other concrete execution backends. Not only does this FFI give an edge to the LLVM backend, but it also allows projects that rely on external libraries, such as the C Semantics, to have a more general interface with which to interact.

We conclude by highlighting the current efforts of the LLVM backend, which largely focus on speed of bottlenecks, including garbage collection. The LLVM backend is an ongoing project that provides the core functionality required for concrete execution of large and complex languages, and provides performance tuning mechanisms through the pattern matching heuristics discussed. The work in this thesis, as well as the tremendous amount of work done by others before this thesis, allows the LLVM backend to be used by an increasing number of $\mathbb{K}$ projects as the sole engine for concrete execution.

REFERENCES

[1] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[2] X. Chen and G. Roşu, "A language-independent program verification framework," in *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*. Springer, November 2018, pp. 92–102.

[3] G. Rosu, "K - a semantic framework for programming languages and formal analysis tools," in *Dependable Software Systems Engineering*, ser. NATO Science for Peace and Security, D. Peled and A. Pretschner, Eds. IOS Press, 2017.

[4] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, B. Moore, Y. Zhang, D. Park, A. Ştefănescu, and G. Roşu, "Kevm: A complete semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium*. IEEE, 2018, pp. 204–217.

[5] C. Hathhorn, C. Ellison, and G. Roşu, "Defining the undefinedness of c," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, June 2015, pp. 336–345.

[6] "The libffi website," 2929. [Online]. Available: https://sourceware.org/libffi/

[7] L. Maranget, "Compiling pattern matching to good decision trees," in *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ser. ML 08. New York, NY, USA: Association for Computing Machinery, 2008. [Online]. Available: https://doi.org/10.1145/1411304.1411311 p. 3546.

APPENDIX A: FULL IMPLEMENTATIONS

The implementation of the IO interface, FFI, and pattern matching is located in the *llvm-backend* repository:

IO:

`https://github.com/kframework/llvm-backend/blob/master/runtime/io/io.cpp`

FFI:

`https://github.com/kframework/llvm-backend/blob/master/runtime/meta/ffi.cpp`

Heuristics

`https://github.com/kframework/llvm-backend/blob/master/matching/`
`src/main/scala/org/kframework/backend/llvm/matching/Heuristics.scala`

APPENDIX B: IMP

```
// Copyright (c) 2014-2019 K Team. All Rights Reserved.
module IMP-SYNTAX
  imports DOMAINS-SYNTAX

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

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

module IMP
  imports IMP-SYNTAX
  imports DOMAINS

  syntax KResult ::= Int | Bool
```

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


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

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

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

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

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

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

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

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

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