K-LLVM: A Relatively Complete Semantics of LLVM IR

Liyi Li
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
liyili2@illinois.edu

Elsa L. Gunter
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
egunter@illinois.edu

Abstract

LLVM [20] is designed for the compile-time, link-time and run-time optimization of programs written in various programming languages. The language supported by LLVM targeted by modern compilers is LLVM IR [28]. In this paper we define **K-LLVM**, a reference semantics for LLVM IR. To the best of our knowledge, **K-LLVM** is the most complete formal LLVM IR semantics to date, including all LLVM IR instructions, intrinsic functions in the LLVM documentation and Standard-C library functions that are necessary to execute many LLVM IR programs. Additionally, **K-LLVM** formulates an abstract machine that executes all LLVM IR instructions. The machine allows to describe our formal semantics in terms of simulating a conceptual virtual machine that runs LLVM IR programs, including non-deterministic programs. Even though the **K-LLVM** memory model in this paper is assumed to be a sequentially consistent memory model and does not include all LLVM concurrency memory behaviors, the design of **K-LLVM**'s data layout allows the **K-LLVM** abstract machine to execute some LLVM IR programs that previous semantics did not cover, such as the full range of LLVM IR behaviors for the interaction among LLVM IR casting, pointer arithmetic, memory operations and some memory flags (e.g. `readonly`) of function headers. Additionally, the memory model is modularized in a manner that supports investigating other memory models. To validate **K-LLVM**, we have implemented it in $\mathbb{K}$ [40], which generated an interpreter for LLVM IR. Using this, we ran tests including 1,385 unit test programs and around 3,000 concrete LLVM IR programs, and **K-LLVM** passed all of them.

2012 ACM Subject Classification Theory of computation → Operational semantics

Keywords and phrases LLVM, formal semantics, K framework, memory model, abstract machine

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.7

Funding This material is based upon work supported in part by NSF Grant CCF 13-18191. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

1 Introduction

The Low Level Virtual Machine (LLVM) is designed for the compile-time, link-time and run-time optimizations of programs written in unspecified programming languages. An LLVM-based compiler, such as Clang, relies on a translation from a high-level source language to an intermediate representation (LLVM IR) that hides details about the specific target execution platform and acts as an interface for LLVM. Then, users are able to use the LLVM tools to perform program optimizations, transformations, and static analyses based on LLVM IR, which can also be translated into target architectures such as x86, PowerPC, and ARM. Hence, LLVM IR acts as a "central station" for translating high-level languages to target architectures, with a fixed set of language syntax, instructions, library functions, and a memory model [28].


© Liyi Li and Elsa L. Gunter;
licensed under Creative Commons License CC-BY

When using LLVM IR in Clang, the correctness of executing programs is a big concern. Previous work [43, 29] has identified more than 200 LLVM compiler bugs. To verify Clang, we first need to know what the correct behavior of LLVM IR is. However, there are several issues about the currently existing language specifications related to LLVM IR. This paper provides an overview of a relatively complete semantics addressing these issues. One challenge to giving a complete semantics is its sheer size. To the best of our knowledge, VeLLVM [44] is the only notable attempt to give LLVM IR a formal semantics, and it only provides a limited subset of LLVM IR features, which does not include the LLVM library functions, a multi-threaded memory model, or the standard-C library functions. A second challenge is finding the right balance between mathematical abstractions and real world concrete details about the LLVM IR semantics. Most of the previous work [44, 17, 16] has utilized mathematical abstractions for the LLVM IR semantics so that theorems could be proved in an elegant and simple way. However, LLVM IR is not a high enough level of language that such abstractions can reflect the full semantics with total precision. Its semantics contains a lot of detailed information that a real implementable semantics needs to explain. For example, even though VeLLVM allows memory alignments, it does not allow memory operations to have alignment information. In LLVM IR, if the alignment value for a memory operation is not set properly, the behavior can be undefined. The lack of such information means that VeLLVM lacks the definition of important features of LLVM IR. Filling in all these details is challenging but important for defining the whole LLVM IR semantics. Third, even if one has the details in their semantics, we still need a good way to combine them together to form a unified framework with simplicity and modularity. For example, the C semantics in $\mathbb{K}$ [10] has considered many details of the memory layouts necessary for executing C programs. However, its execution and memory models are so basic that any extension of the semantics requires a major change in them, such as the extension of atomic memory operations. In this paper, a rigorous executable specification is formalized for the LLVM IR language to overcome these problems. Our **K-LLVM** semantics defines almost all of the features in LLVM IR that are listed in the LLVM IR documentation (see Limitations at the end of this section), which has more than 219 pages. **K-LLVM** also offers a unified framework as an abstract machine that executes LLVM IR programs. The framework allows us to cover all corner case semantics of LLVM IR operations. The full details of our semantics can be found in the **K-LLVM** implementation [25]. This paper highlights an interesting portion of **K-LLVM** to show how one can possibly find a balance between abstractions and real world programming to provide a better, clearer, and more useful language semantics. First, we introduce some benefits, features, and a limitation of **K-LLVM**.

The Most Complete LLVM IR Semantics. **K-LLVM** is the most complete LLVM IR semantics to date, and provides a reference for people to use when exploring LLVM IR behaviors, including threading behaviors. The semantics is complete relative to a byte-wise, sequentially consistent memory model. **K-LLVM** defines corner cases for all LLVM IR operations, some of which have not been defined by previous work.

A Unified and Rigorously Mathematical Framework. We provide a unified and rigorously mathematical framework where people can observe the semantic behaviors in a single interface and also prove properties of compilers, with a focus on LLVM IR and LLVM IR compilers. Transforming programs from a high-level language to a low-level machine code requires a lot of phases, each of which might cause correctness concerns. For example, the infamous out-of-thin-air problems can arise at every level of intermediate AST as a result of a transformation

or compiler optimization. They can even appear when some old processors try to execute certain programs [34]. **K-LLVM** provides a way for users to reason about the behaviors of these translations based on the rigorous executable semantics of LLVM IR.

A Conceptual Device and a Virtual Machine. **K-LLVM** is implemented as a virtual machine that runs LLVM IR codes, that are interpretable by users. Instead of having to understand axiomatized memory events, they deal with central processors, threads, memory caches, etc. **K-LLVM** accomplishes this by providing an abstract machine that combines its runtime system, executions and memory models (in byte-wise sequential consistency). It implements the executable LLVM IR semantics for version 6.0.0. The abstract machine is also scalable. With simple changes to the current **K-LLVM**, the machine can allow the LLVM IR instructions to be executed out of order, handle speculative executions, and simulate a real-world memory environment that allows for features such as memory caches.

Detailed LLVM IR Low-level Structure. LLVM IR is a low enough language that one cannot define the semantics without explicitly incorporating aspects of the underlying architecture. It is important to deal with low-level data values like integers, floats, and pointers in a more detailed format based on bits and bytes, instead of pure mathematical concepts (see Section 3.2).

Parametric Behavior. **K-LLVM** has been implemented in a direct and transparent manner in $\mathbb{K}$, resulting in an interpreter for LLVM IR. **K-LLVM** is parameterized by important information needed for implementing defined behaviors. Users can configure the parameters of the semantics based on specific architectures or compilers, and then proceed to see executable behaviors formally in the implementation in $\mathbb{K}$.

Undefined Behavior. We classify three different types of undefinedness in LLVM IR. The first one is `undef`, which represents an unspecified value for a program position; the program should proceed no matter what the value is. In some cases, `undef` also means that the program has ill-defined behavior, such as representing a race in the memory. There are two ways to deal with `undef` in **K-LLVM**: ***krun*** can be used to execute a program with `undef` and get a fixed deterministic behavior by assuming one path, or ***ksearch*** can be used to search for all different behaviors by executing the program non-deterministically. Sometimes, the non-deterministic search space caused by `undef` values in LLVM IR is too large. In such cases, the symbolic execution engine in ***ksearch*** with the $\mathbb{K}$ equivalence checker can be used to determine if two programs return the same results. Additional discussion can be found in Section 5. The second kind of undefinedness is an undefined behavior represented by a poison value, because LLVM IR does not have a defined symbol for it. Its meaning is similar to `undef`, but it has certain undefined behaviors associated with it. **K-LLVM** will carry the poison value and continue computation until a non-deterministic point is reached, then give an error message saying that there is a poison value in the program, and stop the continuation of the computation. If no non-deterministic point is found, **K-LLVM** can finish the computation successfully. The third kind results from underspecification in the LLVM IR documentation. We named this as **unspecified behaviors** in this paper. When facing the third kind, **K-LLVM** immediately labels the computation an error state, saying there is an unspecified behavior in the system. More information can be found in Section 5.

Independent of $\mathbb{K}$. The implementation in $\mathbb{K}$ gives **K-LLVM** the power to have an interpreter automatically, and have tools for state space searching and symbolic executions. Essentially, $\mathbb{K}$ [41] is an executable semantic framework based a rewriting logic [33]. Once a language semantics is defined in $\mathbb{K}$, it automatically turns it into a logical form by turning each semantic rule into an axiomatic rule with pre and post-conditions; thus, it creates an axiom set for the language. Additionally, there are many tools available in $\mathbb{K}$. For example, ***kompile*** can be used to see if the semantics has static type problems and to generate an interpreter, so that ***krun*** can be used with the interpreter to test their semantics by actual concrete programs. ***ksearch*** allows searches of traces of multi-threaded programs based on the interpreter. The symbolic engine in ***ksearch*** and the program equivalence checker in $\mathbb{K}$ can allow for two sets of traces to be compared by symbolically executing two different multi-threaded programs and seeing if the two sets produce the same output. Even though we have defined **K-LLVM** in $\mathbb{K}$, the semantics is independent of its implementations in $\mathbb{K}$. In fact, we have defined the **K-LLVM** abstract machine in Isabelle [38] for manually proving theorems about **K-LLVM**. Additional discussion is presented in Section 5.

Limitations. In this paper, the **K-LLVM** memory model is based on byte-wise sequential consistency. LLVM IR specifies a range of behaviors for memory operations with different orderings and for `volatile` memory accesses, while **K-LLVM** does not support the full range. In **K-LLVM**, every memory location is mapped to a single byte datum; there is only one memory cache to deal with all memory operation requests from the different threads. Single thread instruction execution is in the program order. Based on this model with the **K-LLVM** abstract machine, we provide an observation in Section 4.4. We implemented the full LLVM IR concurrency model (based on the `glibc` and `pthread_create` libraries) in $\mathbb{K}$ with all of the memory ordering behaviors of the atomic memory operations, but we have not yet finished proving the properties of that model; so it will be a future extension of **K-LLVM**. The work is described in the technical report [25].

2 Related Work

The only other formal executable semantics of LLVM IR is VeLLVM. Here we discuss the relationships between VeLLVM and **K-LLVM**, and also review large language specifications related to our design.

VeLLVM. VeLLVM was the first and only attempt to define a complete specification for the core of LLVM IR prior to **K-LLVM**. It was defined in the theorem prover Coq [8] and covered a limited set of LLVM IR instructions. VeLLVM formalizes a mechanized semantics for LLVM IR, its type system, and the properties of its SSA form. It also has an interpreter extracted from Coq that ran 145 test programs and passed 134 of them. With VeLLVM, users can prove properties about translations defined in LLVM IR. Many papers have been published about compiler correctness, memory models, and verifications of compiler schemes using VeLLVM [23, 16]. Here, we compare the latest code update (April 1st, 2019, LLVM IR version 3.8) of VeLLVM and its documentation [44] with **K-LLVM**.

Figure 1 provides a comparison between VeLLVM and **K-LLVM** of all LLVM IR version 3.8 and 6.0.0 common features, for which VeLLVM and **K-LLVM** are defined. Our **K-LLVM** defines significantly more features than VeLLVM, and basically covers all features of LLVM IR; whereas there are a lot of features missing in VeLLVM. One of the important differences between VeLLVM and **K-LLVM** is that **K-LLVM** actually splits the semantics

Feature	VL	KL	Feature	VL	KL
Modules and Layouts	●	●	Function Definitions and Declarations	●	●
Global Variables and Thread Locals	◐	●	Aliases	○	●
Type System with Address Space	◐	●	Well-formedness and Validity Checks	◐	●
Ensure Local Vars as count values	○	●	Undefined Behavior Recognition	◐	●
Integers, Arrays, Structs	◐	●	Floating points and Constant Exps	◐	●
Undefined Values	●	●	Packed Struct Values, Labels	○	●
Char list, Block Addresses and Vectors	○	●	Poison Values	◐	●
Arithmetic Expressions with flags	◐	●	Function Calls with different flags	◐	●
Floating-point Expressions with flags	◐	●	Logical Expressions with flags	◐	●
icmp and fcmp operators	◐	●	getelementptr	◐	●
br, phi, select, ret and unreachable	●	●	switch and indirectbr operators	○	●
Casting operators	◐	●	Vector operators	○	●
Aggregate Type operators	◐	●	Exception Handlings	○	●
Non-atomic Memory operators	◐	●	Atomic Memory ops (seqcst only)	○	●
RMW operators (seqcst only)	○	●	va_arg operators	○	●
Full Intrinsic Library Functions	◐	●	C Standard Memory Allocations	◐	●
C Standard I/O Functions	○	●	Threading Library Functions	○	●
Mutex Library Functions	○	●			

Support level: ● = Full ◐ = Partial ○ = None

VL represents VeLLVM[44] and KL is our work.

Figure 1 VeLLVM vs **K-LLVM**.

into static and dynamic ones, so that actions that occur at compilation time can be separated from those happening at execution time. To illustrate why this is important, both VeLLVM and **K-LLVM** allow instructions to contain constant expressions. The semantics of constant expressions in **K-LLVM** respects the LLVM IR documentation, in which a constant expression cannot contain any local variable, and it must be reduced to a value such as an integer, pointer value, etc., at compilation time. However, VeLLVM allows constant expressions to contain local variables, so evaluating them becomes an execution time event, which directly contradicts the LLVM IR design. There are also some features for individual LLVM IR entities missing in the current implementation of VeLLVM. For example, even though VeLLVM has floating-point values, it does not perform some important checks, so there are some floats allowed in VeLLVM and not allowed in LLVM IR. Some key features are missing for exploring correct program behaviors. For example, in `Program-A` (Fig. 2 in Sec. 3.1), VeLLVM is not able to recognize $\%u1$ (line 5) as a poison value because its `getelementptr` semantics lacks the `inbounds` flag; and if the value `100` for $\%r9$ in line 20 happens to accidentally be a valid memory address, the VeLLVM `store` semantics is not able to rule out the program (line 22). This is the type of legitimate security issue that someone would want to ask an LLVM IR semantics to deal with or at least warn of. **K-LLVM** supports all of these missing features.

The key difference between VeLLVM and **K-LLVM** is the memory layout structure. VeLLVM relies on the CompCert memory layout structure [2, 24], with some extensions to provide the structures of the pointers, memory locations, and data in the main memory. CompCert's memory structure is for C. Although it is a good approximation of the LLVM IR memory layout structure, they are not the same. The memory layouts in VeLLVM have two features that keep them from implementing the full range of memory behaviors. First, VeLLVM equates pointers and memory addresses. A pointer is a language construct associated with a memory address, but it is not the actual memory address. Equating these two in defining the LLVM IR semantics has shortcomings. For example, The LLVM IR pointer-aliasing rule defines that a valid pointer cannot come from an integer constant. In line 20 of `Program-A` (Fig. 2 in Sec. 3.1), if the value `100` happens to be a valid memory address, by using VeLLVM's memory pointer representation and applying a `getelementptr` operation with `inbounds`, it is hard to recognize the variable $\%r9$ is not a valid pointer in the next line. More details are in our implementation of pointers in Sec. 4.3.

The second limiting memory-layout feature is that VeLLVM implements the concept of typed memory chunks. In this system, a `malloc` creates a large chunk, and at this point each byte in the chunk has no type. Then, a `store` causes a small area of the large chunk to become a typed memory chunk with the type of the stored data, and the memory address of the typed chunk is represented by a base value and offset. A `load` can happen only if the `load`'s type is consisted with the type of the loading small chunk. However, LLVM IR specifically says that their values stored in memory have no types. VeLLVM's implementation not only contradicts the design of LLVM IR but also rules out some interesting executions. For example, lines 6 to 11 in `Program-A` (Fig. 2 in Sec. 3.1) load two `i8` values from consecutive `i8*` locations. VeLLVM produces an error state for this program because the two `i8*` locations were previously stored by two different `i32` integers in an array in line 3. Moreover, the typed memory chunk concept even makes it hard for VeLLVM to define interesting executions such as the following: store an `i48` at the beginning of a `[3 x i32]` large chunk (partitioned into three small `i32` typed memory chunks), and then try to load the middle `i32` integer from the chunk.

Other Work Related to the LLVM IR Semantics. There are other pieces of work that are not meant to directly define the LLVM IR semantics but influence **K-LLVM**. First, Lee et al. [23] investigated the LLVM IR undefined behaviors with no concrete semantics for all undefined behaviors. Kang et al. [16] provided a model in C to support the `inttoptr`/`ptrtoint` casting operations. Their work enlightened **K-LLVM**. However, their definition focused on the aspect of a memory model, leaving the execution of programs as a black box. Thus, their casting operation semantics does not work with the real LLVM IR semantics. Ellison and Rosu [10] defined the full C semantics with a simplified version of CompCert's model. Chakraborty and Vafeiadis [6] provided a concurrent abstracted memory model for LLVM IR that focused on an abstraction of the concurrent LLVM IR memory behaviors. Lee et al. proposed a novel LLVM memory model including a data layout and memory pointer provenance model [22]. They claimed to provide a better LLVM memory model that was sound and performed better. However, their model targeted a very small set of LLVM IR memory related instructions, and their abstract machine was simple. It is unclear how their model can be extended to include the behaviors of other LLVM IR instructions, especially the side-effects caused by interactions between different instructions, such as the additional behaviors caused by having the `readonly` flag or the thread creation instruction in the system. Compared to Lee et al.'s model, **K-LLVM** has a much simpler data layout and a concrete abstract machine to support different semantic behaviors including corner cases and side effects caused by the interaction of different instructions. Memarian et al. [35] provided two pointer provenance models for C/C++ languages and reconciled the ISO C standard. Similar to Lee et al.'s work, Memarian et al. focused on creating better pointer provenance models for C instead of investigating different C instruction behaviors through a concrete abstract machine. Without great effort, it is unclear how to build an abstract machine to support all LLVM IR instructions based on their model.

Other Large Language Specifications. **K-LLVM** is a formal and executable specification, of which many have been defined recently. Standard ML by Milner, Tofte, Harper, and Macqueen [36] is one of the most prominent and mathematical programming language specifications, whose formal and executable specifications were added to by Lee, Crary, and Harper [21], VanInwegen and Gunter [14], and Maharaj and Gunter [30]. Blazy and Leroy [2] verified an optimizing compiler based on CLight in CompCert. Large language specifications

have been defined in $\mathbb{K}$, including C [10], PHP [12], JavaScript [37], and Java [4]. A lot of work has been done on formalized specifications in Java and C#: Eisenbach's formal Java semantics [9] and Syme's HOL semantics [42] for Drossopoulou, the C# standard by Börger et al. [5], which is formally executable and uses abstract state machines [13], and the executable Java specification by Farzan et al. [11]. We cannot list all of the interesting examples of formalized language specifications in this paper for space reasons.

Our mechanized specification of **K-LLVM** shares many of the difficult challenges faced by the works described above and involves many new ones, due to the complex and dynamic nature of **K-LLVM**. They are detailed in later sections.

3 Background and Challenges

Below we discuss the major challenges that needed to be faced when developing **K-LLVM**. Additionally, we introduce briefly LLVM IR programs, **K-LLVM** and $\mathbb{K}$.

3.1 A Taste of LLVM IR Programs and Assumptions on LLVM IR

The LLVM language (LLVM IR) is a statically and strongly typed, assembly-like, Static Single Assignment (SSA) based language. It has undefined behaviors but the undefinedness is well documented. The LLVM language itself does not have operations or libraries to support multi-threaded behaviors, but LLVM IR's structure is highly related to the C/C++ library. LLVM IR basically assumes a runtime environment of C++. LLVM IR also contains a set of functions comprising an intrinsic library, in which part of the standard C library is included. It also relies on other functions in the `stdlib.h` header. For example, it needs dynamic memory management functions such as `malloc`, `realloc` and `free` to provide heap memory access, as well as functions dealing with the environment such as `abort`, `exit` and `system`. Furthermore, it needs functions listed in the `stdio.h` header to provide I/O support, as well as library functions from the Pthread and Pthread-mutex libraries to provide threading and mutual exclusion behaviors. These functions are not strictly part of the LLVM IR listed in the documentation but we define them anyway.

The current LLVM IR can be viewed as "C- -". Except function bodies, most features in C can be found in LLVM IR, such as global variables, `struct` datatypes, function headers and different flags for global variables or functions, etc. The main difference between LLVM IR programs and C programs are the function bodies, a.k.a. expressions. The LLVM IR expressions are register-based, SSA based and assembly-like. These features eliminate the undefinedness of the evaluation order in an LLVM IR program. We show some examples of LLVM IR expressions in Figure 2 to provide a taste of LLVM IR. These expressions are used throughout the whole paper. We believe that these expressions are enough to show the key features of LLVM IR and the construction of LLVM IR programs based on these expressions and other components (function headers, global variables and modules, etc) can be easily found in the LLVM documentation. This is also the reason we refer to these expressions as "programs" in the rest of the paper.

LLVM IR distinguishes local variables from global variables. Variables starting with the character % are local ones, while those starting with the character @ are global. Global variables can only have a pointer type. Any number following the character `i` in LLVM IR, such as `i32` or `i1`, means an integer type declaration with the size of the bits. `i32*` refers to a 32-bit integer pointer type declaration. Instructions starting with the keyword `icmp` are the integer comparison operators. With the keyword `eq`, the instruction *%r8* `= icmp eq i64`

```
Program-A :
 1    %r1  = call i8* @malloc (i64 12)
 2    %r2  = bitcast i8* %r1 to [3 x i32]*
 3    store [3 x i32] [i32 0, i32 0, i32 0], [3 x i32]* %r2
 4    %r3  = getelementptr inbounds [3 x i32], [3 x i32]* %r2 , i64 0, i32 1
 5    %u1  = getelementptr inbounds [3 x i32], [3 x i32]* %r2 , i64 -1, i32 4  ;poison value.
 6    %u2  = getelementptr inbounds i8, i8* %r1 , i64 3
 7    %u3  = load i8, i8* %u2
 8    %u4  = ptrtoint i8* %u3  to i64
 9    %u5  = add i64  %u4 , 1
 10   %u6  = inttoptr i64 %u5  to i8*
 11   %u7  = load i8, i8* %u6
 12   %r4  = bitcast i32* %r3 to [2 x i32]*
 13   store [2 x i32] [i32 11, i32 11], [2 x i32]* %r4
 14   %r5  = ptrtoint [2 x i32]* %r4  to i64
 15   %r6  = inttoptr i64 %r5  to i64*
 16   %r7  = load i64, i64* %r6 ;read back the two i32 array as an i64 value 47244640267.
 17   %r8  = icmp eq i64  %r7 , 47244640267
 18   br i1  %r8 , label %next , label %exit
 19   next :
 20   %r9  = inttoptr i64 100 to i32*
 21   %r10  = getelementptr inbounds i32, i32* %r9 , i64 0  ;poison value.
 22   store i32 42, i32* %r9 ;unspecified behavior due to invalid pointer.
 23   exit :
 ...
Program-B :
 Thread-1 :                                          Thread-2 :
 ...                                                 ...
  store atomic i32 42, i32* @x  monotonic, align 1    store atomic i32 1, i32* @y  monotonic, align 1
  %a  = load atomic i32, i32* @y  monotonic, align 1  %b  = load atomic i32, i32* @x  monotonic, align 1
 ...                                                 ...
 Program-C :
  Thread-1 :                  Thread-2 :
  ...                         ...
   store i32 42, i32* @x       store i32 1, i32* @y
   %a  = load i32, i32* @y     %b  = load i32, i32* @x
  ...                         ...
Program-D :
 Thread-1 :                                          Thread-2 :
 ...                                                 define i32 () @f  {
  %a  = load i32, i32* @x                             store i32 1, i32* @x
  %r  = call i32 @pthread_create (i32 ()* @f ,...)    return 0
 ...                                                 }

Program-E :
  %r1  = call i8* @malloc (i64 12)
  %r2  = ptrtoint i8* %r1  to i32
  %r3  = call i8* @printf(@x , i32 %r2)
```

Figure 2 LLVM IR Example Programs.

`%r7 , 47244640267` tests whether the value in the variable *%r7* and `47244640267` are the same and stores the result to the variable *%r8*. The "`;`" operation allows users to put comments after a line of code.

`Program-A` does several pointer arithmetic operations and memory operations. Several key observations about LLVM IR are made here. First, `getelementptr` is a memory address calculation operation and has an `inbounds` flag. No previous work has formally defined the behavior of flags of `getelementptr`. The definition of `inbounds` is hard because it not only affects the final result but also affects every intermediate result of computing the memory address. For example, in `Program-A`, *%u1* (line 5) is a poison value because we have `inbounds` in the `getelementptr`, and the second index is `i64 -1`, which makes the intermediate result out-of-bounds. Even though the final result is in bounds because we add back numbers, the `inbounds` still makes the final result a poison value. We talk about our definition of the `getelementptr` operation in Section 4.4. Second, as we mentioned in Section 2, LLVM IR views the main memory as having no type. We can store an array `[11, 11]` (line 13) and magically get back the `i64` value 47244640267 (line 16). This has effects on defining the **K-LLVM** type system, which will be explained in Section 4.1. Finally, executing `Program-A` in **K-LLVM** stops at the line 22. It is an unspecified behavior in LLVM IR to read data from a memory location pointed to by a pointer that was not properly created. This has not been properly defined by previous work, especially the definition of a memory operation

combined with casting and pointer arithmetic operations. More details are in Section 4.4. `Program-B` and `Program-C` distinguish between a non-atomic and atomic memory operation. Thanks to our **K-LLVM** virtual machine definition, we are able to produce the race caused by two non-atomic operations in two different threads. Additional details are in Section 4. While maintaining sequential consistency, the execution of `Program-D` could result in a race on $@x$ because of the special instruction execution order of LLVM, which the **K-LLVM** abstract machine models. More details are in Section 4.2. `Program-E` is an example for showing the usage of the $\mathbb{K}$ symbolic execution engine in Section 5.

After reading the programs in Figure 2, questions about the memory locations and memory alignments may come to mind. Memory implementation is very complicated in real world programming languages. LLVM IR does not actually fix a special implementation of memory addresses. For simplicity, we assume in this paper that there is a one-to-one mapping from natural numbers to memory addresses, and a memory chunk is always in a range that can be defined between a left and a right integer bound. The memory addresses refer to conceptual memory byte data. Conceptual memory bytes are not actual byte data – details are in Section 4.3. LLVM IR also allows setting up alignments for different types, memory endianness and address space information by using `target datalayout`. Although we have implemented these features in **K-LLVM**, for simplicity, we assume in this paper that alignments, paddings for `structs` and address spaces never cause a problem in calculating memory addresses or type checking, and we assume little-endian byte-order. Finally, we assume that the heap size is infinite while the stack for each function is finite and has a maximum bound, and if a stack overflows in a thread, the whole system reaches an error state. We believe that assuming a max bound on the stack is an advantage of **K-LLVM** over previous formal semantics of LLVM IR. In the LLVM documentation, some stack intrinsic functions and function flags (`probe-stack/safestack`) indicates that function stacks has max bounds. The implementation of **K-LLVM** stacks is introduced in the descrition of the abstract machine (Sec. 4.2).

3.2 Challenges

Some challenges were introduced in Sec. 2 (versus VeLLVM). Others are listed below.

Sheer Size of LLVM IR. The first challenge is the sheer size and precision of LLVM IR. With respect to instructions, LLVM IR has more than 60 operators and 100 intrinsic library functions. Some operators have complex rules or different requirements according to the input. For example, `store` operators can be either non-atomic or atomic, and atomic `store` operators have six different orderings. All of these require different semantic rules. The previous work only defined some of the operators, or some of their features. No previous work has defined the massive number of intrinsic library functions. **K-LLVM** defines all the LLVM operations and intrinsic functions. We handle this challenge through a special heavily testing strategy to define **K-LLVM** described in Sec. 5.

Subtlety of Well-formedness. In LLVM IR, the subtlety of various instructions and the well-formedness of instructions are often directly connected with the semantics of the instructions in a particular place in a given program. The syntactic nature of even a single instruction is determined by the semantic context. For example, the `getelementptr` operator allows indices to be integer local variables if the pointer input is an `array` pointer. However, if it is a `struct` pointer, LLVM IR requires the indices to be integer constants that can be statically reduced to integer values. These two types can be mixed together in a single usage

of `getelementptr` in an LLVM IR program. Another example is that the input containing a decimal representation of a floating-point constant needs to be exact. This means that the value `1.1` cannot be a valid constant for floating-point operators in LLVM IR because it cannot be precisely represented by a finite floating point number, and LLVM IR requires the compilers to LLVM IR to round the float to a hexadecimal format. This is an error in both Clang (the LLVM compiler) and VeLLVM.

Detailed Low-Level Features. As we mentioned in Section 1, it is not feasible to gloss over the details of LLVM IR's low-level features, such as how to represent integers, floats and pointers. The effects are easily felt when we combine casting operations with memory operations. It is a common source of confusion among LLVM IR users, and thus, a common source of bugs. We also need to admit the fact that memory locations are highly related to integer behaviors; so converting pointers to integers, doing certain arithmetic on them, and converting them back to pointers are valid program exercises within a memory chunk created by a `malloc` operation. This brings us a big challenge. For example, in `Program-A` (Fig. 2), we cannot use pointer $\%r9$ to store data to the main memory (line 22), even if it is accidentally at the right range of a memory chunk, because $\%r9$ is not a valid pointer according to the LLVM IR pointer-aliasing rule. Defining a data structure to capture the behaviors covering all corner cases is one of the key contributions of **K-LLVM**. In addition, it is important to admit that the low-level structure of LLVM IR is based on bits and bytes; as well as the integer, float and pointer calculations are based on two's compliments, IEEE 754, and integer pointer calculations.

Instructions Having Side-effects on Subsequent Instructions. Some instructions may cause side-effects on subsequent instructions depending on their behaviors. For example, in `Program-A` (Fig. 2), one can use the pointer `%r4` to access memory because it was a subsequent computation result of the pointer `%r1` from a `malloc` function, while `%r9` cannot be used to access memory data because it is from an integer constant. Defining these complicated side-effects requires new ideas. In addition, LLVM IR instructions can have very different requirements for different computer components. This complicates the design of different components of the **K-LLVM** abstract machine.

As we solved these challenges, we tried our best to define all language features in **K-LLVM**.

3.3 The $\mathbb{K}$ Framework

$\mathbb{K}$ [41] is a rewrite-based, executable semantic framework in which programming languages, type systems and formal analysis tools can be defined. **K-LLVM** is independent of $\mathbb{K}$. However, the implementation of **K-LLVM** in $\mathbb{K}$ follows the mathematical definition closely, and some $\mathbb{K}$ tools are useful for supporting the usage of **K-LLVM**. Once a language semantics is built, one can use ***kompile*** to see if it has static type problems and to generate an interpreter, so that users can use ***krun*** with the interpreter to test their semantics by actual concrete programs. ***ksearch*** allows users to search traces of multi-threaded programs based on the interpreter. The symbolic engine of ***ksearch*** and the program equivalence checker in $\mathbb{K}$ allow users to compare two sets of traces from two different symbolically executed multi-threaded programs to see if their outputs are the same. Additional discussion is presented in Section 5.

4 K-LLVM Semantics

In this section, we define the semantics of **K-LLVM**. It is divided into two parts: the **K-LLVM** static semantics (Sec. 4.1) and the **K-LLVM** dynamic semantics (Sec. 4.2 to Sec. 4.4). In this paper, we focus on parts of the descriptions of the static and dynamic semantics. We mainly discuss the general process and the type checking stage of the static semantics; as well as the general (sequentially consistent) abstract machine structure and memory operation specifications of the dynamic semantics. Other interesting details are in our technical report and $\mathbb{K}$ formalization [25]. Some important features of **K-LLVM** are based on Zhao et al.'s work [44]. For example, the **K-LLVM** formalization of SSA and `Phi` functions is very similar to the one in Zhao et al.'s work. The comparison of the work and **K-LLVM** is in Sec. 2.

4.1 K-LLVM Static Semantics

When giving the semantics of LLVM IR, **K-LLVM** uses two different ASTs, a front-end AST (FAST) and a back-end AST (BAST). The syntax of LLVM IR 6.0.0, which is documented in the website `http://releases.llvm.org/6.0.0/docs/LangRef.html`, is directly parsed into the FAST. We have formally defined the LLVM IR 6.0 syntax in $\mathbb{K}$, and it parses any LLVM IR program into the FAST format. **K-LLVM** static semantics refers to the LLVM IR behaviors that happen at compilation time. For an LLVM IR program, parsing is not enough to rule out unqualified programs. After parsing, a series of checks need to be performed on an LLVM IR program, including well-typedness, static single assignment, and well-formedness. The **K-LLVM** static semantics implementation applies these checks and rule out unqualified programs. It also translates a FAST program into a representation in the BAST format, which is passed to the dynamic semantics for execution. Figure 3 depicts the phases in the **K-LLVM** static semantics.

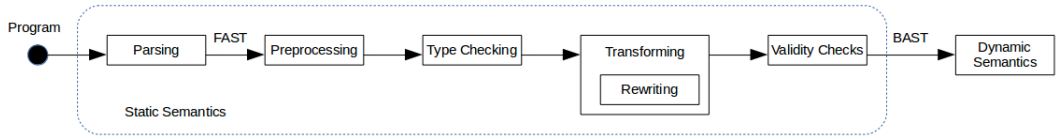

Figure 3 Static Semantics of **K-LLVM**.

Here we first sketch the functionality of each phase, and then focus on the type checking phase. More information can be found in the technical report [25]. The purpose of the preprocessing phase is to simulate the LLVM compilation steps that happen in the linkage time, including joining all modules from different files and dealing with global variables. The constant expression rewriting phase reduces LLVM IR constant expressions to values. After type checking, the transformation phase translates a program in FAST to a form in BAST. The validity checks phase applies well-formedness checks to the BAST program code, such as ensuring the code is in Static Single Assignment (SSA) form. We have proved the following theorem about the **K-LLVM** type system.

Type Checking. This step emulates the behaviors of LLVM IR type checking for the functions in LLVM IR modules. LLVM IR is a relatively strongly-typed language, and its type system is very straightforward. The **K-LLVM** type checking process is a complete implementation of the LLVM IR type system listed in its documentation. The input for the **K-LLVM** type checking function is a term and its type; the function outputs `true` if

the term has been type checked and has the input type, and `false` otherwise. "Relatively strongly-typed" here means that the type system of LLVM IR guarantees the type prevention property, i.e., a typed value produced from a typed LLVM IR expression is compatible with the size of the value in runtime, and any later usage of the value will not result in a type error or size error if there is a move (usage). However, the program still has the chance of going wrong in the case of other problems, such as division by zero. in `Program-A` in Figure 2, every line of code except `store` and `br` instructions assigns a value to a variable. After type checking, each variable has a type. *%r1* has type `i8*` (line 1) and *%r2* has type `[3 x i32]*` (line 2). If we eliminate line 2 and replace the variable *%r2* in line 3 with *%r1*, the line results in a type error. In addition, there is also a chance that a correctly typed LLVM IR program is never executed since the execution of an LLVM IR program depends on the runtime environment setting. For example, The abstract machine in **K-LLVM** (Sec. 4.2) is parameterized by the function stack size. Users are free to set the size to `0`, in which no program can be executed in any step.

%struct.RT `= type {i8, [10 x [20 x i32]], i8}`
`getelementptr inbounds` *%struct.RT* `,` *%struct.RT* `*` *%u* `,i64 0,i32 add (i32 1,i32 0),i32` *%x*

Figure 4 A Type Example.

There are some tricky cases of the type system. In Figure 4, we show a `getelementptr` instruction on a `struct` type. For a `struct`, the value of the index for the `getelementptr` affects the type result of the final value of an instruction, because every position in a `struct` can have different type. Type checking a `getelementptr` relies on executing part of the semantics of the `getelementptr` arguments. That is why some index values of `getelementptr` that are associated to `struct` type positions are required to be inferred statically. This means that such positions can contain neither local nor global variables, even if a constant expression (no variables inside) is allowed. For other non-`struct` index positions, variables are allowed, such as the *x* `getelementptr` in Figure 4.

LLVM IR takes the view that values stored in the memory have no types, and that memory instructions will always produce values of the prescribed types. In fact, LLVM IR does not have a clear idea of main memory. It does not even have a built-in memory allocation instruction, instead, it relies on Standard-C library to provide such instructions. It basically assumes that the memory machine as a black box, and every memory request is **valid** as long as the size of the requested data matches the size of its type, the memory pointer is not out-of-range, and there is no race. In addition, one can have a correctly typed program where the result value produced by the program does not make sense. For example, loading an `i31` value from a heap field that is previously stored as an `i30` value is unspecified. To support the type system, **K-LLVM** assumes that each of the poison value and undefined value is implemented as a family of constructs, one for each type (ASTs as `undef (`*Type*`)` and `poisonValue (`*Type*`)`). Combining all these features of LLVM IR type system, we have shown the following type prevention theorem (the proof sketch is in the technical report [25]):

▶ **Theorem 1.** Assuming every load returning a value in a type prescribed in the load instruction, the program is well-typed by the **K-LLVM** type system, and the program executes at least one step, then every register and every return value of the program will be of the type assigned during the type checking.

The statement about the loading value in the theorem refers to that every load instruction reads a value that is previously stored with a proper type matching the load instruction, i.e., no having the case such as loading an `i31` value from a heap field that is previously stored as an `i30` value. The theorem assumes that every LLVM IR program can make a move, and it does not guarantee that the execution of a type-correct program has at least one move. After a program has been checked and transformed through the **K-LLVM** static semantics, the transformed BAST program is ready for execution by the **K-LLVM** dynamic semantics.

4.2 The K-LLVM Abstract Machine

As we mentioned in Section 1, the semantics of the execution of the LLVM IR programs in **K-LLVM** described is via an abstract machine. There are three reasons for this. First, it is a concise way to define all features and aspects of the LLVM IR semantics. LLVM IR is a programming language that connects different computer resources through many different instructions. The best way to model these different features is to design a computer-like mathematical entity which simulates them. Second, the abstract machine is designed to emulate real world computer components. Often, mathematical abstract machines are complicated and confusing. The **K-LLVM** abstract machine execution is easy for users to follow since they can relate it to real world computer components. Third, our abstract machine is modular; as a consequence, it is also extendable. In previous language semantics, designers either only define straight-line single-threaded instruction behaviors or only define a subset of all instructions with complete concurrent behaviors. Once concurrent behaviors are introduced, a single instruction's semantics can affect the whole semantic universe forcing designers to update all existing instruction semantics to handle any side-effects. The design of the **K-LLVM** abstract machine allows us to focus on designing one feature at a time in isolation. Additionally, because of the modular design, the abstract machine can be easily updated to support progressive concurrent features. For example, we update the byte-wise sequential consistency model in this paper to a model containing the full LLVM IR concurrency features in our corresponding technical report [25], without modifying a *single* instruction semantic rule, and only changing transition rules for describing how to maintain the execution order in the `continuation` and `toCommit` component of each thread.

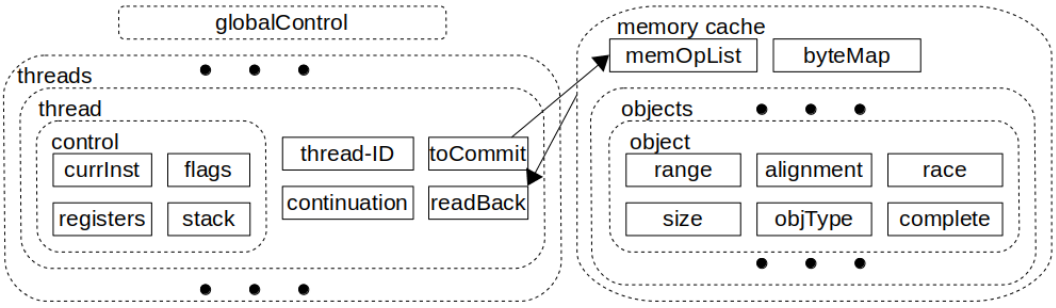

Figure 5 Component Relations in the **K-LLVM** Memory Layout Structure.

Figure 5 describes the overall structure of the **K-LLVM** abstract machine and the interactions of different components. The arrows show the direction of messages passing between the main components. A rounded dashed component means a program state entity that might contain other component structures, while a square component means a program state entity whose content is an integer, list, set, map, etc. The **K-LLVM** abstract machine is independent of the platform in which we implement the machine. At the top level, the

abstract machine can be thought of as a set of threads communicating with a set of memory caches, and a global control unit provides global information for threads. As a simplifying assumption to achieve byte-wise sequential consistency, we assume the memory cache set is a singleton set, so we will refer to this cache as the memory cache in the rest of the paper. The `globalControl` component represents the global control unit containing several sub-components storing information about threads, such as thread identifier calculation, thread final states and mutex lock information. We will see an example of using this information in Section 5. There are several components in each thread as shown in the left side of Figure 5. In Section 4.1, we said that a LLVM IR program is compiled to a list of BAST control flow graphs (CFGs) for execution. The `continuation` component represents a dynamically executing CFG; it contains a sequence of dynamic basic blocks of instructions to be executed. A thread executes one instruction at a time, i.e., the first instruction in the first block. Thread execution is modeled by consuming instructions as they are executed and possibly inserting a new basic block after the current block during loop execution.

For each thread, the `control` component includes `registers`, a `stack`, `flags`, and `currInst` components. The `registers` component is a map from local variables to values. We introduce how we represent values in the next section. The `stack` component records function call stack frames for context switching in LLVM IR based on `call` and `return` instructions. Each stack also contains fields for local memory allocations in a function directed by the `alloca` instruction. The **K-LLVM** stack implementation is not a simple mapping. Each **K-LLVM** function stack has a maximum allowed allocation space, and stack overflow leads to an error state. Every time when a function is called, a memory range is actually created in the heap for storing the function stack information. This implementation allows us to implement some LLVM IR flags such as "inalloca", and also some extra tools built on top of **K-LLVM** (as a future work) to track stack buffer overflows such as `AddressSanitizer` and `SAFECode`. The `flags` component contains the set of function header flags describing the function that is currently executing. For example, the `readonly` flag tells the LLVM compiler that the function will never produce memory write operations, and this need to be reflected in the execution semantics; see Section 4.4 for a complete semantics. The `currInst` component contains a dynamic block number and instruction number pair representing the unique identifier for the currently executing instruction. Dynamic block numbers are basically timestamps and uniquely identify each execution of a basic block; when a new basic block of instructions is put into the `continuation` component, a new such number is associated with the block. Instruction numbers can be assigned statically, e.g., using textual order in the LLVM IR file. For example, the numbers on the left side of `Program-A` (Fig. 2) are a possible instruction numbering.

The `currInst` pair allows us to modularly add new concurrent behaviors to the **K-LLVM** abstract machine. Even though our model assumes byte-wise sequential consistency in this paper, the machine has potential for additional concurrent behaviors. When dispatching a memory instruction, a thread need not wait for the instruction commit before proceeding. For example, a thread will not wait for a `load` instruction to write values to `registers`. Instead, it moves on and marks the specific register as unavailable. If the next instruction needs the register value, the thread component blocks. Otherwise, the thread continues to execute instructions. The `currInst` pair identifies a specific instruction and corresponding register during write back. The example `Program-D` (Fig. 2) shows how this feature can affect program execution in practice. Without this mechanism, the `load` instruction in `Thread-1` always happens before the `store` in `Thread-2`. With this mechanism, an observer can observe the value `1` or even a race on $@x$. This example is the motivation of having the abstract machine

in **K-LLVM** even though its memory model assumes byte-wise sequential consistency in this paper. **K-LLVM** is mainly used to verify LLVM compiler steps, and verifying programs containing library functions is a key verification component. The *pthread_create* function in `Program-D` is a library function and its functionality should contain fences to prevent the behavior of executing `Program-D` described above. The abstract machine mechanism in **K-LLVM** allows to prove that a particular implementation of *pthread_create* does not have harmful behaviors like the one above; whereas otherwise we do not have a mechanism to verify such library function usage in a program.

The `toCommit` and `readBack` components in a thread are to deal with memory instructions, and they also act as interface communicating with the memory cache. From the memory point of view, all it knows about memory requests from each thread are from the two components. In this sense, they belong to the memory model of the **K-LLVM** abstract machine, even though they are located in each thread. The `toCommit` component is implemented as a queue that receives memory operations from `continuation` and then sends them to the memory cache in order. The `readback` component is implemented as a map and represents the intermediate step of getting back a value from a memory-read from the memory cache and assigning the value to registers. These components are needed to distinguish between memory instructions and their corresponding execution. Another reason is the need to simulate the difference between the non-atomic and atomic memory operations in LLVM IR. LLVM IR assumes that each non-atomic memory write or read operation accesses a single byte of data in the memory cache at a time, while an atomic operation accesses several bytes at once. By breaking down the execution of non-atomic `store` and `load` instructions into possibly several memory operations, we are able to capture potential races in a multi-threaded program.

The memory cache has a fixed structure in **K-LLVM**, which is listed on the right side of Figure 5. The `memOpList` component stores the memory operations from different threads, in order to allow the interleaving of memory operations from different threads. The `byteMap` component is a function that maps a memory location to a byte of data. A memory write operation in **K-LLVM** stores an array of bytes in the `byteMap` component. While `byteMap` represents the entire memory cache, a memory chunk refers to a continuous memory region in `byteMap` and is allocated by a global memory initialization or local memory allocation. An `object` component stores metadata for a specific memory chunk. Each `object` contains a `range` component indicating the range of the chunk in the whole memory domain (as keys of `byteMap`), an `alignment` component with alignment information, a `size` component with the size of the chunk in bytes, and an `objType` component indicating if the memory chunk is static or not. The `complete` and `race` components are used to record the status of the operations accessing the memory chunk. According to the LLVM IR documentation, non-atomic memory operations should access a memory range one byte at a time. When a non-atomic memory operation is accessing a memory chunk at the same time as another memory write operation, a race occurs, and the result is `undef`. The `complete` and `race` components are used to record this status and give the result. The implementations of the `byteMap` and `object` components are used to represent the low-level memory layout structures in LLVM IR, whose requirements are stated in Section 2 where we discuss VeLLVM layout structure. We believe that storing metadata on a per-chunk basis is the best way to implement the LLVM IR memory layout model to maximize the concurrent memory access behaviors allowed by LLVM IR.

We have briefly described the different components of the **K-LLVM** abstract machine above. The details of the implementations of each component can be found in our implementation [25]. In the following sections, we will introduce some detailed implementation aspects related to memory accesses. The full LLVM IR concurrency model can also be found in the technical report [25].

4.3 K-LLVM Data Layout

In this and the next sections, we introduce a portion of the **K-LLVM** abstract machine in depth, especially, the components and rules related to executing memory related instructions. The manner in which data layout and memory layout are implemented in **K-LLVM** facilitates the precise semantics of many language features of LLVM IR while maintaining a concise abstract machine for the execution semantics. In this section, we introduce the implementation of register and memory location values in **K-LLVM** and example rules using these values. The need for two different kinds of values arises as from the fact that memory only sees values as a sequence of bytes, while instructions see registers as holding compound data. We describe these two kinds in Figure 6, and we also show some example rules using these data. In Figure 6, rules connected by a ⇒ operator mean that the transition from the left hand side to the right hand side happens in the beginning of a `continuation` component. There is an implicit rule saying that every transition happening in the beginning of a `continuation` also happens globally. More complex transition rules are introduced in Fig. 8. The `add` and `icmp eq` are instructions in LLVM IR appearing here in the concrete syntax.

The `undef` value for a *Bit* datum exists due to undefinedness of LLVM IR. In LLVM IR, if an integer that is not a multiple of the length of a byte (like a 23-bit integer), and is stored to the memory, then the values for the extra bits generated during the process are undefined (`undef`). A memory location value is implemented by the *Byte* type. In addition to having eight *Bit* data, each *Byte* datum contains a range attribute (*Range Option*) and a flag attribute (*Range State*). If a *Byte* datum represents a part of a pointer, the range attribute is the left and right edges of the memory range to which the pointer points, and if not then `none`. If a *Byte* datum represents a part of a pointer, and the pointer is the result from a `getelementptr` instruction with a `inrange` flag, the flag attribute is the left and right edges of the memory range that the `inrange` flag defines. If the pointer does not come from a `getelementptr` instruction, the flag attribute is `none`. If a `getelementptr` generates an error due to mixing of `inrange` flags, the flag attribute records the error. We will see more about the `inrange` flag of a `getelementptr` in the paragraph describing `store` instruction semantics below. We want to have these two attributes associated with a *Byte* datum because we want to provide pointer provenance, so that when a pointer is cast to an integer or stored to the memory cache, it does not lose side-effect information, such as what is the memory field the pointer points to. The real data structure of `Byte` data in **K-LLVM** has more fields including information about block address information, endianness, and if a pointer datum is pointing to a heap, stack, or static constant memory chunk. For simplicity, we do not include them here, and assume the bytes are in little-endian format. We also assume no distinction between heap and stack pointers here, even though we have distinct implementations for each in **K-LLVM**.

$$
\begin{array}{l}
\mathit{Bit} ::= \texttt{1} \mid \texttt{0} \mid \texttt{undef} \quad \mathit{Range} ::= \texttt{range}(\mathit{Nat}\ \texttt{,}\ \mathit{Nat}) \quad {}'a\ \mathit{State} ::= \mathit{Error} \mid {}'a\ \mathit{Option} \\
\mathit{Byte} ::= \texttt{byte}(\mathit{Bit\ List}\ \texttt{,}\ \mathit{Range\ Option}\ \texttt{,}\ \mathit{Range\ State}) \\
\mathit{Loc} ::= \texttt{loc}(\mathit{Bit\ List}\ \texttt{,}\ \mathit{Type}\ \texttt{,}\ \mathit{Range\ Option}\ \texttt{,}\ \mathit{Range\ State}) \\
\mathit{Int} ::= \texttt{intLoc}(\mathit{Bit\ List}\ \texttt{,}\ \mathit{Type}\ \texttt{,}\ \mathit{Range\ Option}\ \texttt{,}\ \mathit{Range\ State}) \\
\mathit{Float} ::= \texttt{floatLoc}(\mathit{Bit\ List}\ \texttt{,}\ \mathit{Type}\ \texttt{,}\ \mathit{Range\ Option}\ \texttt{,}\ \mathit{Range\ State}) \\
\texttt{(a)}\quad \texttt{add}\ T\ \texttt{intLoc}(X, A1, B1, C1)\texttt{,}\ \texttt{intLoc}(Y, A1, B2, C2) \\
\qquad \Rightarrow\ \texttt{intLoc(bitAdd(}T, X, Y\texttt{)}\texttt{,}\ A1\texttt{,}\ \texttt{judge(}B1, B2\texttt{)}\texttt{,}\ \texttt{judge(}C1, C2\texttt{))} \\
\texttt{(b)}\quad \texttt{icmp eq}\ T\ \texttt{loc}(X, A1, B1, C1)\texttt{,}\ \texttt{loc}(Y, A1, B2, C2)\ \Rightarrow\ \texttt{intLoc([}X = Y\texttt{]}\texttt{, i1,none,none)}
\end{array}
$$

Figure 6 Memory Data Structure.

For register values, we only introduce integer, float and pointer values here. The description of other register values can be found in the **K-LLVM** semantics implementation [25]. Any of the integer (*Int*), float (*Float*) or pointer (*Loc*) data contains a *Bit* list, a *Type* field representing the type of the datum, a range attribute and a flag attribute. The *Bit* list represents the binary format of the value for the datum being either an integer, float (in the IEEE 754 format) or memory address. The size of the list is equal to the size of the integer/float/pointer type defined for the data (the pointer size is parameterized in **K-LLVM**). We assume that all integer, float and pointer arithmetic is based on the computation of binary representations, even though we might show decimal representations in some examples in this paper for presentation purpose. The range and flag attributes have meaning that is closely related to the ones in a *Byte* datum, as we will explain below.

The reason for making the register and memory data structure so complicated is that **K-LLVM** covers the relatively complete semantics of LLVM IR including corner cases of not only the individual instruction semantics but also the interactions between casting, arithmetic and memory related instructions in LLVM IR. Hence, the pointer provenance information needs to be available both in the threads and the memory cache. In **K-LLVM**, the provenance information is stored in the value representation to enable three features of LLVM IR that require execution decisions based on the past history of the value. First, there are flags (`inrange`), which require the possibility of turning the transition state to an error state in executing a memory instruction long after the computation of a `getelementptr` with the flags. Second, a pointer is valid for accessing a memory datum if and only if it is created from a non-free memory allocation, or it is the result of a finite number of memory computations based on a non-free memory allocation pointer, and its pointing memory field is within the memory range of the allocated chunk. Third, an error should be detected when an execution is accessing memory data by a pointer cast from an integer value whose calculation never involves values cast from pointers, even if the integer has the same value as the memory address of a valid pointer.

The two rules `(a)` and `(b)` in Fig. 6 give an example describing how an arithmetic instruction is executed in **K-LLVM** based on the data structure described above. In evaluating an LLVM IR `add` instruction (rule `(a)`), the value computation happens between the *Bit* lists of two data (`bitAdd` adding two binary numbers together). The function `judge` merges two range or flag attributes from possibly two different data that possibly come from two pointer sources. The `judge` details are in the actual **K-LLVM** semantics implementation [25]. Here, we give some interesting examples. If a pointer is cast to a integer constant (with the range attribute $[L, R]$) and added to another integer constant (with the range attribute `none`), the `judge` produces a memory range from the pointer in the range attribute of the result datum. If the two range attributes of two `intLocs` have two different memory ranges (like $[L, R] \neq [S, T]$), the judged result is `none`. If two flag attributes of two data have two different memory ranges, in this case, `judge` produces an error state in the flag attribute of the result datum; and if the result datum is further turned into a pointer, and is used to read memory data, the program results in unspecified behavior. Rule `(b)` gives an example of a comparison instruction that discards the pointer information and produces a pure 1-bit integer constant. Depending on the instruction, including the nature of its arguments, pointer information might or might not be transmitted along with the result of the calculation.

4.4 Sample Instruction Semantics

In this section, we introduce semantic rules supporting memory related instructions in **K-LLVM**. The set of memory related instructions we select to describe here contains LLVM IR casting, address calculation (`getelementptr`) and memory instructions, as well as memory

related flags on the function headers. **K-LLVM** is the first formal semantics to cover all behavioral aspects (under byte-wise sequential consistency) of this set, including the side-effects due to interactions between different instructions inside or outside of the set. Under the byte-wise sequential consistency assumption, the behaviors of different orderings in an atomic memory operation collapses to the behavior of the sequentially consistent (`seqcst`) ordering. It is worth noting that there are cases when an instruction can go to an unspecified behavior or other error states. We will not list all of those rules here, although we have defined them in **K-LLVM**. Interested readers may get more details from the **K-LLVM** semantics [25].

Casting Instructions. Here we describe the semantics of `inttoptr` and `bitcast` as the highlights of the **K-LLVM** semantics of casting instructions in Figure 7. The other casting instructions are implemented in a similar manner. Before **K-LLVM**, no complete interpretation for the LLVM IR casting operations existed, especially one supporting casting between integers or floats and pointers. These casting instructions are hard to define because the resulting values can vary depending on the program context for the values of the instructions.

(a) $\texttt{inttoptr(intLoc(}X, T1, B, C\texttt{),}T2\texttt{)} \Rightarrow \texttt{loc(trunc(}X\texttt{,sizeof(}T1\texttt{) - sizeof(}T2\texttt{)),}T2, B, C\texttt{)}$
$\quad if\ \texttt{sizeof(}T1\texttt{)} \geq \texttt{sizeof(}T2\texttt{)}$

(b) $\texttt{inttoptr(intLoc(}X, T1, B, C\texttt{),}T2\texttt{)} \Rightarrow \texttt{loc(addZero(sizeof(}T2\texttt{) - sizeof(}T1\texttt{))@}X, T2, B, C\texttt{)}$
$\quad if\ \texttt{sizeof(}T1\texttt{)} < \texttt{sizeof(}T2\texttt{)}$

(c) $\texttt{bitcast(}Label\texttt{(}X, T1, B, C\texttt{),}T2\texttt{)} \Rightarrow \texttt{rebuild(}X, T2, B, C\texttt{)}$
$\quad if\ \neg\texttt{isPointerType(}T1\texttt{)} \wedge\ Label \in \texttt{\{intLoc, floatLoc\}}$

(d) $\texttt{bitcast(loc(}X, T1, B, C\texttt{),}T2\texttt{)} \Rightarrow \texttt{loc(}X, T2, B, C\texttt{)}$

Figure 7 Casting Rules.

In Figure 7, rules `(a)` and `(b)` describe the semantics of `inttoptr`. The main idea is to replace the type attribute of the source `intLoc` with the target type. If the target type size is smaller than (or equal to) the source one, the semantics truncates (using the `trunc` function) the bits (represented by X as a list) by the difference of the sizes of the two types starting from the most significant bit. Otherwise, we create a list of `0` bits, whose size is the difference between the two type sizes, by using the `addZero` function. We place the bits in front of the source bit list (variable X). For example, in `Program-A` (Fig. 2), we assume that the code is running in a 32-bit machine and variable $\%r5$ has the value represented by $\texttt{intLoc(}X\texttt{, i64, }B\texttt{, }C\texttt{)}$ in line 15. The code tries to convert the $\%r5$ value to a pointer. The final result pointer can be represented by $\texttt{loc(}X'\texttt{, i64*, }B\texttt{, }C\texttt{)}$ by taking the right-most 32-bits from X and changing the constructor from `intLoc` to `loc`.

Rules `(c)` and `(d)` describe the much simpler dynamic semantics of `bitcast` instructions. Besides the memory data layout, the **K-LLVM** type system also contributes to the simplicity. Once we find out that $T1$ is not a pointer, we can immediately infer that $T2$ is also not a pointer because LLVM IR only allows pointer to pointer or non-pointer to non-pointer `bitcast`. Thus, the rule `(c)` should take the bits (variable X) with additional attribute information and distribute them to form a corresponding value with respect to the type $T2$, which is what the function `rebuild` does. For example, if we `bitcast` an `i24` integer (as $\texttt{intLoc(}X\texttt{, i24, }B\texttt{, }C\texttt{)}$) to a three `i8` integer array `[3 x i8]`, the 24-bit list X is cut into three equal parts ($X1$, $X2$ and $X3$), so we have an array with three elements of the format $\texttt{intLoc(}Y\texttt{, i8, }B\texttt{, }C\texttt{)}$ where Y can be either $X1$, $X2$ or $X3$. Alternatively, if a `bitcast` sees a *Loc* datum, it is immediately inferred that the casting is between two pointers, and the only effect is the updating of the source type $T1$ with the target type $T2$.

The Semantics of `getelementptr`. A `getelementptr` instruction is a memory address calculation whose main idea is to calculate a memory address value based on a sequence of indices. Section 3.1 touches on one of the special cases of `getelementptr` semantics. The main idea of `getelementptr` is similar to the one in Zhao's work [44]. It uses a sequence of indices of different types to walk incrementally into a data structure layout to calculate a pointer to the sub-component found at the end of the path the indices describe. Here, we focus on one particularly important feature of the instruction, the keyword `inbounds`, which is a flag applied on the computation results of a `getelementptr` instruction. For this flag, LLVM IR requires all the intermediate and final computation results on the address of the input pointer are within a valid range of the allocated object pointed to by the address. In **K-LLVM**, we implement this with the address computation function `calGEP`. The function calculates an new address value by adding multiplication results of the index and type size to the input address, one adding at a time. In each step, before the calculation, the function first checks if the input address is within the range indicated by the range attribute of the input pointer. After we compute the final address result, we also check if the memory chunk pointed to by the input pointer still exists. For example, line 4 of `Program-A` (Fig. 2) is a `getelementptr` instruction, and it is executed successfully in **K-LLVM**. However, if a memory-`free` for the input pointer *%r2* is added before the `getelementptr`, the `inbounds` flag makes the instruction result in a poison value, because the memory chunk pointed to by *%r2* does not exist anymore. As another example of an `inbounds` flag, executing line 5 of `Program-A` highlights how a poison value can be produced from a `getelementptr`. The index `i64 -1` makes an intermediate computation result out-of-bound, so variable *%u1* gets a poison value. Another example is to execute line 21 of `Program-A`. The execution of this `getelementptr` fails the `inbounds` check because its input pointer has range attribute `none`, so variable *%r10* results in a poison value. There is also an `inrange` flag in a `getelementptr` instruction. This flag has subsequent effects on memory instructions after the `getelementptr`. The flag information is carried as the flag attribute in the pointer derived from the `getelementptr` so that the succeeding memory instructions can use it. We will introduce its semantics in the next section.

The `store` Semantics. We only introduce the **K-LLVM** `store` memory instructions here; the other memory instructions are implemented in a similar manner. **K-LLVM** fully implements the semantics of `stores` under the byte-wise sequential consistency assumption. Specifically, **K-LLVM** distinguishes the non-atomic and atomic `store` instructions by breaking the execution of an memory instruction into three different stages, as shown in Figure 8. As we mentioned, we do not list negative rules, such as configurations going to an error state when a `store` is performing a `write` operation in the memory cache, when the memory chunk has already been freed by another thread. The rules in Figure 8 are simplified versions of the actual **K-LLVM** rules. The information and handling about address spaces and memory alignments is not mentioned here. In fact, the construct `write` has several fields than one shown in the figure. On the other hand, these rules are non-trivial, and they have enough functionality to show manner in which the **K-LLVM** abstract machine distinguishes between the behaviors of atomic and non-atomic `store` instructions.

In Figure 8, the *Exp* type represents an instruction that involves in the computation in a `continuation` component (Ψ in Fig. 8). We uses `store` and `atomicStore` constructs in Figure 8 that are different from the LLVM IR concrete syntax. They are BAST format transformed from an LLVM IR `stores` instruction in their simplified form here. Each of them has three fields. The first represents the type of the value; the second is the value

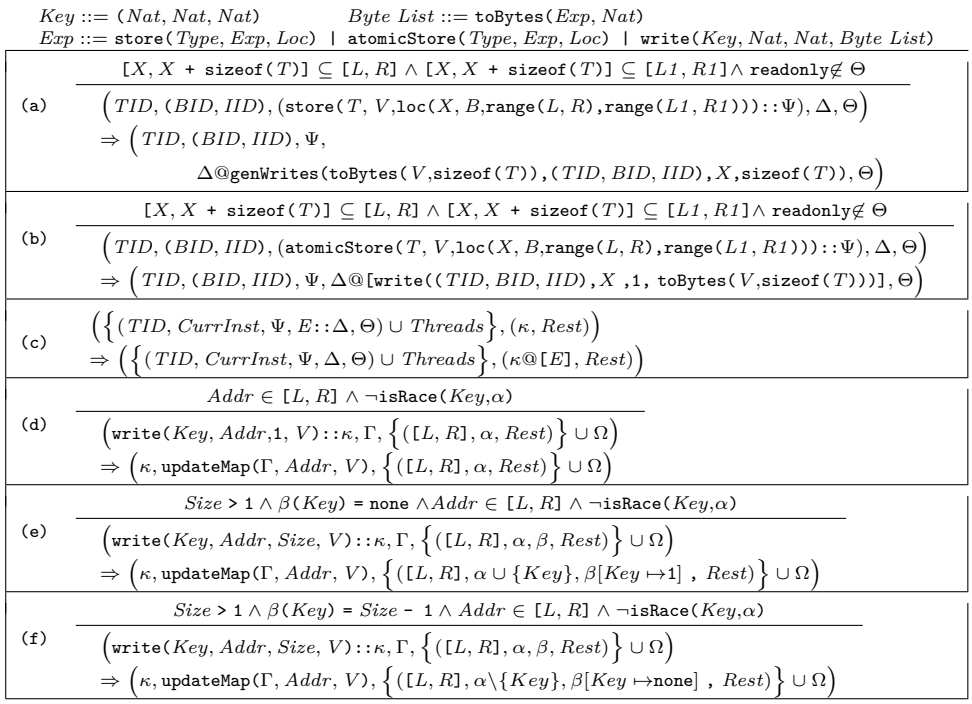

Figure 8 Memory Store Rules.

to store in the memory cache and the third is the memory pointer. The `write` construct represents the memory operation that a thread uses to communicate with the memory cache and the memory cache uses to perform memory events. When a `store` is executed in the `continuation` component (Ψ), a list of `writes` are generated in the `toCommit` component (Δ in Fig. 8). They have the same group ID represented as a *Key* type that is a triple of the thread ID, dynamic block number, and instruction number of the `store`. A `write` also has other fields: a natural number representing the memory address value, another natural number representing the total size of `writes` from the same *Key*, and a list of bytes to write to the memory. The total size is the same for different `write` operations with the same *Key*. It is both the size of the list of `writes` generated by a non-atomic `store` and the size of bytes of the value to write to the memory cache. An `atomicStore` generates a singleton `write`.

Before we describe the rules in Figure 8, some conventions are worth noting. Without special greek letter illustrations on different components, a name of a component with its first character capitalized is the variable representing the component in all rules (e.g. *CurrInst* for the `currInst` component, and *Threads* for the `threads` component). The variable *Rest* appearing in some rules in Figure 8 (and Fig. 9) represents the rest of components in a `thread` or `object` component, which do not involve in the computation of the rules. As we have said in Section 4.2, the **K-LLVM** abstract machine is for a set of threads communicating with a single memory cache. The `globalControl` component is omitted in the computation here, since we do not need it. Based on these assumptions, we define a transition state to be a pair of a set of threads and a memory cache: $(Threads, Memory)$. A single thread contains five components related to memory instructions: `thread-ID`, `currInst`, `continuation` (Ψ), `toCommit` (Δ) and `flag`. For simplicity, we assume that a thread only contains these five

components in this section; Also, we assume that the memory cache only contains three components: the `memOpList` (κ), `byteMap` (Γ) and `objects` (Ω) components. The `objects` component (Ω) contains a set of `object`. Only three sub-components (`range`, `race` (α) and `complete` (β)) in the `object` are related in defining the semantics of `store`. The math inclusive range $[A, B]$ represents a set of natural number sequencing from the number A to the number B inclusively. Finally, there are implicit rules omitted in Figure 8, suggesting that transitions happening in a thread or the memory cache also happens globally.

Rules `(a)` and `(b)` in Figure 8 describe how an atomic `store` and non-atomic `store` generate a list of `write` operations that are pushed to the `toCommit` component (Δ) whose job is to convey memory operations to the memory cache. The basic idea is to create a list of `writes` at the end of `toCommit` (Δ) when we have a `store` in the head of `continuation` (Ψ). The two rules describe the cases when an `inrange` flag is present in the flag attribute of the input pointer. In such cases, to execute a `store` not only requires for the address value to be within the range indicated in the pointer but also for it to be in the range carried by the `inrange` flag; otherwise, the whole system results in an unspecified behavior state. In **K-LLVM**, there are rules similar to rules `(a)` and `(b)` dealing with pointers without `inrange` flags derived by removing the checks for the `inrange` edges from rules `(a)` and `(b)`. Since we will use these rules in an example, we call them rules `(ax)` and `(bx)` to distinguish them from rules `(a)` and `(b)`. The function `toBytes` splits a value into a list of bytes (AST in Fig. 8). The list size is defined by its natural number argument. Its functionality is similar to the `rebuild` function to turn a value into a list of elements. The only difference is that `toBytes` creates a list of bytes instead of values in the case of `rebuild`. The function `genWrites` takes a list of bytes, a *Key* datum, a memory address, and the size of the byte list, then generates a list of `writes` by distributing a byte at a time from the byte list, and associates each byte with a memory address and other attributes. The address value is selected in sequence from the address range between the address and the address plus the size. Rule `(b)` is for dealing with atomic `stores`. The key difference is that it only generates a singleton `write` containing the full value to be stored instead of a list. Rule `(c)` allows the head element in the `toCommit` component (Δ) of a thread to move to the tail position of the `memOpList` (κ) in the memory cache.

Rules `(d)`, `(e)` and `(f)` deal with different situations of correctly committing a `write` to the `byteMap` (Γ). The `complete` component (β) in `(e)` and `(f)` is a map from a *Key* to a natural number indicating how many `writes` have been committed to `byteMap` (Γ) since the first `write` with the *Key*. The *Key* marks a single instruction and `complete` allows tracking the process of the writes entailed by the instruction. To detect races, the `race` component (α) contains *Keys* indicating every *Key* occupying the memory chunk (`object`) represented by the `range` of the `object` component. The variable *Size* represents the number of `writes` from the same *Key*, i.e. the same `store` instruction. All rules `(d)`, `(e)` and `(f)` need to satisfy two side conditions. The first one is the condition $Addr \in [L, R]$ to locate a specific `object` in the `objects` component (Ω) by comparing *Addr* with the range of the `object` (L and R). In **K-LLVM**, an `object` is created by a memory allocation; thus, the ranges of `objects` (Ω) are always disjoint. Any address (e.g. *Addr*) within a range (e.g. $[L, R]$) can be a key to locate the range, which in turn locates an `object`. The second condition is to check if a *Key* is in race with other *Keys* in `race` (α) by the function `isRace`. The function `isRace` checks if the `race` component (α) for the `object` pointed to by the memory address value (*Addr*) has been occupied by another *Key*. If *Size* is `1` (rule `(d)`), the `write` represents an atomic memory `store`, and writes a list of bytes (V) to `byteMap` (Γ) using the function `updateMap`. The function updates a range of bytes to corresponding range of

addresses in `byteMap` (Γ). Rule `(e)` is executed if two other conditions are satisfied: the *Size* is not `1` and no `write` for this *Key* has yet completed. In such case, rule `(e)` writes a list of bytes to `byteMap` (Γ) and updates the information in the `race` (α), and initializes the *Key* in the `complete` component (β). Rule `(f)` represents the finish of the execution of a non-atomic `store` in the memory cache. In such cases, we remove the appearance of the entities represented by variable *Key* in the `race` (α) and `complete` (β) components. We also need to update `byteMap` (Γ) with the final `write` term. Besides rule `(e)` and `(f)`, another rule not listed here deals with the case when $\beta(Key)$ does exist and is less than *Size* `- 1`. In this rule, we continue to write a byte to the `byteMap` component (Γ) without touching the `race` component (α) and incrementing the `complete` component (β) for *Key*.

$$
\begin{aligned}
&\texttt{(s)}\ \Big(\big\{ \big(\varphi,\texttt{(1,13)},(\texttt{store([2 x i32],[11,11], loc(100,[2 x i32]*, range(96, 108), none))}:: \Psi),\texttt{[]},\emptyset\big) \big\} \cup \Xi, \big(\texttt{[]}, \Sigma, \big\{ \big(\texttt{[96,108]},\emptyset,\emptyset,\Upsilon\big) \big\} \cup \Omega \big) \Big) \\
&\Rightarrow \Big(\big\{ \big(\varphi,\texttt{(1,13)},\Psi,(\texttt{write}((\varphi,\texttt{1,13}),\texttt{100,8,byte}(B,\texttt{none,none}))::\Delta),\emptyset\big) \big\} \cup \Xi, \big(\texttt{[]}, \Sigma, \big\{ \big(\texttt{[96,108]},\emptyset,\emptyset,\Upsilon\big) \big\} \cup \Omega \big) \Big) \\
&\Rightarrow \Big(\big\{ \big(\varphi,\texttt{(1,13)},\Psi,\Delta,\emptyset\big) \big\} \cup \Xi, \big(\texttt{[write((}\varphi\texttt{ ,1,13),100,8,byte(}\ B\ \texttt{,none,none))]}, \Sigma, \big\{ \big(\texttt{[96,108]},\emptyset,\emptyset,\Upsilon\big) \big\} \cup \Omega \big) \Big) \\
&\Rightarrow \Big(\big\{ \big(\varphi,\texttt{(1,13)},\Psi,\Delta,\emptyset\big) \big\} \cup \Xi, \big(\texttt{[]}, \Sigma[\texttt{100}\mapsto \texttt{byte}(B,\texttt{none,none})], \big\{ \big(\texttt{[96,108]},\{(\varphi,\texttt{1,13})\},\{(\varphi,\texttt{1,13}) \mapsto \texttt{1}\},\Upsilon\big) \big\} \cup \Omega \big) \Big) \Rightarrow \cdots \\
&\texttt{(t)}\ \Big(\big\{ \big(\varphi,\texttt{(1,22)},(\texttt{store(i32, 42, loc(100, i32*, none, none))}:: \Psi),\texttt{[]},\emptyset\big) \big\} \cup \Xi, \mathit{Rest} \Big) \\
&\quad \Rightarrow \big(\texttt{error unspecifiedBehavior} \big)
\end{aligned}
$$

Figure 9 Memory Store Example Configuration Transitions.

As an example of applying the `store` rules, we focus on the `store` instruction in line 13 of `Program-A` (Fig. 2). Group `(s)` in Figure 9 represents the computations for executing the first few steps of the `store` instruction. In these diagrams, we show the computations as transitions from one state to another. Each transition state is a tuple of a thread set and the memory cache. In threads, Ξ represents all threads that are not involved in the computation. The thread we care about has a thread ID φ. We assume that the `(1,13)` in the first state after the label `(s)` represents the `currInst` pair. In the `continuation` component, we have the `store` instruction of line 13 (Fig. 2) on the top of the computation, and Ψ represents the rest of the computations in `continuation`. For simplicity, we assume that the `toCommit` and `flags` components are empty for thread φ, so they have the values `[]` and $\emptyset$, respectively. In the memory cache, for simplicity, we assume that `memOpList` is empty, `byteMap` is represented by the variable Σ. The memory cache contains some objects. The Ω in Fig. 9 represents objects not related to this `store` computation, and there is an object with range value `[96,108]` that matters in this computation (Let's assume that `[96,108]` is the memory range created previously). We also assume that the current `race` and `complete` components are both empty (an empty set and empty map). Υ represents the rest of the components in the object that is not involved in the computation. The to-store data for the `store` operation is an array of type `[2 x i32]` and value `[11,11]`. Here, we show these data in decimal formats. In the real **K-LLVM** abstract machine, they should be in the binary format. In this example, we assume that the memory pointer address is a natural number `100`, and the range of the memory chunk pointed to by the pointer is in the range `[96,108]`.

By applying rule `(ax)` above, we get a new transition state after the first "⇒" (Fig. 9). Rule `(ax)` generates a list of eight bytes in the `toCommit` component. The first one is the `write` term shown in the state, and the other seven bytes are represented by variable Δ.

The variable B inside the `byte` construct is an eight bit list with all of `0` bits because we are getting the left-most eight bits of the `[11,11]` array. By applying rule `(c)`, we get the resulting state after the second "⇒". This rule moves the `write` operation from the component `toCommit` in thread φ to the empty component `memOpList` in the memory cache. Next, rule `(e)` is executed and we get another new state after the third "⇒". We can see that the components `race`, `complete`, and `byteMap` (Γ) are updated, and the `memOpList` component becomes empty. This process keeps going until all items in `toCommit` have reached `byteMap` (Γ).

Another example is group `(t)` in Figure 9. It represents the computations of the `store` instruction at line 22 of `Program-A` (Fig. 2). In the initial state, the pointer has the range attribute `none`, so the state is transitioned to an error state with the `unspecifiedBehavior` indicator.

Notice that in some states in Group `(s)`, the system might have non-deterministic choices over transition rules. For these non-deterministic choices, we have the following important observation, which is clearly true in **K-LLVM** because the `toCommit` and `memOpList` components are in FIFO order, and each thread executes instructions in the program order in the `continuation` component.

▶ **Observation 2.** Assume that a trace of memory operations is generated by observing the order of memory operations committed to the `byteMap` in the memory cache. For a valid LLVM IR program, no matter which rule the **K-LLVM** abstract machine chooses to apply in a transition state if such rule correctly pattern matches the state, the memory trace generated by executing the program is byte-wise sequentially consistent.

The `readonly` Function Flag. LLVM IR allows users to set flags on the function headers that suggest that the function has certain features over memory instructions. The `readonly` flag is a representative. It means that the execution of the function with the flag should not use any memory write operations, e.g. a `store` instruction. If executing a function does use a write operation, it is unspecified behavior. In Figure 5, there is a `flags` component in the `control` component of a thread. During the static semantics step in Section 4.1, all functions from a LLVM IR program are compiled to BAST format and stored in a database, including function header flag information. During executing in the **K-LLVM** abstract machine, when a function is called, **K-LLVM** context switches the `control` component for the function, including the flag information called from the database and stored in the `flags` component. When **K-LLVM** is executing a `store` operation, according to the `store` rules (Fig. 8), **K-LLVM** checks if the `flags` contain a `readonly` flag. If not, the `store` operation can proceed; otherwise, the whole transition state is transitioned to an error state of `unspecifiedBehavior`.

We have given a general idea of how **K-LLVM** implements different semantic aspects of LLVM IR here. The full details from the real **K-LLVM** semantics in $\mathbb{K}$ and another **K-LLVM** abstract machine in the Isabelle implementation [25].

5 Evaluation and Applications

Evaluating **K-LLVM** took more than half of the development time. We used $\mathbb{K}$ to generate an interpreter for **K-LLVM** and ran LLVM IR programs in it. We mainly used the testing process as a tool to validate the correctness of our semantics, comprised of individual instruction semantics and our memory models. We also developed several tools to show the usage of **K-LLVM**.

Testing Process of K-LLVM. The validation of language semantics is usually accomplished through the use of external test suites [3, 10, 12], which was also part of our strategy. We use a large test suite to test the output of **K-LLVM** against Clang/Clang++. The tests are split into two sets. We have a set of unit test cases containing totally 1,385 medium size test programs, and they were made in the process of defining K-LLVM. They are made to test each individual instruction or intrinsic function listed in the LLVM documentation with the consideration of all corner cases. We also have a set of regression test suite. There are 2,156 programs from the GCC-torture test suites. They are compiled from C to LLVM directly without optimizations. They are used as a regression test suite to validate K-LLVM. Besides, we also use the test suite (around 900 test cases) from previous K-Java semantics [4] as a regression test suite to test K-LLVM. We compiled the Java test cases from Java to LLVM without optimizations. For all of these cases, we first get the output from Clang/Clang++ for compiling a test program to machine code and executing it, and then compare the output with the output of executing the same program by **K-LLVM**. For validating the threading libraries (including mutex ones), we use the K state space exploration tool that will be introduced later in the section. In the unit test suite, we had 128 multi-threaded programs. We first execute them by the state space exploration tool, and get all traces (including all syscalls/memory operations) of each individual program, and examine manually if they are correct. The test cases and Clang bugs have been documented in the **K-LLVM** implementation [25], and the bugs have been reported to the LLVM community.

The methodology for developing **K-LLVM** was based on a strategy named Test Driven Development (TDD), whose basic idea is to develop tests before implementing the actual features. LLVM IR has an official test suite, but it is hard to break it down into individual pieces. In developing **K-LLVM**, the test principle is to test individual features while coordinating new features with old defined ones. When we defined a new feature in **K-LLVM**, we followed four steps. First, we read the details about the feature in the LLVM IR documentation, and thought about how to define the static and dynamic semantics of it. Next, we wrote out unit test cases to test the feature in the current LLVM IR implementation (Clang/Clang++). We made sure that we covered enough corner cases by designing a good set of new unit tests. We then defined the feature and tested it with the new unit tests, making sure it could pass them all. Third, we added the new feature to all of the defined unit tests to see if it caused any new problems. Finally, we tested the whole semantics with the regression test suite (the GCC-torture and K-Java test suites) and made sure that it passed more test cases than before and did not introduce new problems. When we developed **K-LLVM**, we started by defining the static semantics for each individual feature in LLVM IR, and made sure that all static features were validated for every variable, expression, instruction, function and module. After that, we defined the **K-LLVM** memory model and validated the correctness of the model. Following the definition of the model, we incrementally defined the semantics of the instructions, working from those that interacted least with other instructions and the memory such as the arithmetic and conversion instructions, through to the branching instructions and finally those that affected the memory. Lastly, we defined different memory operations. The distinction between the atomic and non-atomic memory operations is particularly complicated due to the fact that we define the non-atomic memory system to be based on reading/writing one byte at a time.

While searching for undesirable behaviors in Clang was not an objective of this project, we found some in the process of defining the **K-LLVM** semantics. Mainly, we ran test programs, and compared their outputs with those listed in the LLVM documentation. Undesirable behaviors happened in very diverse circumstances. A large number of them related to the

fact that Clang does not place enough checks to validate what the LLVM IR documentation suggests. In other cases, Clang has missing features. For example, one cannot cast an `fp128` constant to a `ppc_fp128` constant, which should be allowed. In some cases, the description of the LLVM documentation is not clear. For example, in describing the `fptrunc` and `fpext` instructions, LLVM IR uses the idea of large floating point types, and allows a comparison of two of them. However, it does not give a precise description of how to make this comparison. In fact, we found that the two types `fp128` and `ppc_fp128` are not comparable, so there is no way in LLVM IR to cast from one to the other, contrary to the documentation.

Finally, we use 128 multi-threaded programs to test the **K-LLVM** thread library with ***ksearch***. **K-LLVM** produced a set of behaviors that are all expected according with respect to our thread and byte-wise sequentially consistent memory model. There are other multi-threaded programs used for testing the full memory concurrency behaviors, which is out-of-scope of the paper.

Morpheus on K-LLVM. We built the Morpheus tool [32] on top of **K-LLVM** to support correct specifications of compiler optimizations of LLVM IR programs. The Morpheus core language is a domain-specific one for formal specifications of program transformations. It describes program transformations as rewrites on control flow graphs with temporal logic (CTL) side conditions. Morpheus allows users to specify comprehensible program optimizations including those in data flow analysis and data dependence graph analysis. Its executable semantics allows these specifications to act as prototypes for the optimizations themselves, so that candidate optimizations can be tested and refined before including them in a compiler. We built Morpheus on top of **K-LLVM** in $\mathbb{K}$, so that users are able to specify program optimizations in LLVM IR, and test the optimizations by using $\mathbb{K}$ tools for LLVM IR programs. Through the **IsaK** and **TransK** tools [27, 26], we translate **K-LLVM** into a transition system in Isabelle, and merge it with the Morpheus tool in Isabelle. With this system, we are able to prove the correctness of the optimizations in Isabelle under the assumption that programs are executed in the **K-LLVM** abstract machine and a choice of memory model. As an instance, we are able to define redundant store elimination properties on LLVM IR programs in Isabelle under sequential consistency. With the **K-LLVM** abstract machine, we have a framework for proving the correctness of the optimization for all programs in LLVM IR in Isabelle. The finalization of the proof will be an interesting future work of **K-LLVM**. The detailed semantics of Morpheus, and its union with a transition semantics for a fragment of LLVM for use in proving properties of program transformations is in [31], but **K-LLVM** came after the paper.

Detecting Undefined Behaviors. When an undefined behavior happens, **K-LLVM** outputs an error state. This is particularly useful for programmers to reveal unexpected behaviors to programmers, especially memory access errors. For example, in `Program-A` (Fig. 2), the execution of the program results in a transition state with an `error` component containing an `unspecifiedBehavior` construct (Fig. 9). This is because pointer $\%r9$ comes from a non-valid source. By using ***krun***, we can see the following error message for the `Program-A` execution:

```
$ krun program-a.ll
ERROR while executing the program.
Description: The argument pointer points to an illegal location.
Line-number: 22
```

For some undefined behaviors in LLVM IR, the ***ksearch*** space exploration method cannot list all outputs. `Program-E` (Fig. 2) is such an example. The program is to create a memory field, get a memory pointer, then turn the pointer to an integer and print it. The output is a non-deterministic value with infinite many possible values. When using ***krun*** (the single-thread execution engine in $\mathbb{K}$) to execute the program, it prints out a random integer value depending on the runtime memory address allocation in **K-LLVM**. A better way to analyze the program is to use the $\mathbb{K}$ symbolic execution engine. One can use ***ksearch*** with the `-symbolic` flag to execute this program, and the final result is a variable representing a integer value. One can also use the $\mathbb{K}$ symbolic equivalence checker to check if the executions of two similar programs printing out variables representing the same range of integers. The equivalence checker relies on the Z3 SMT solver to calculate if two variables representing the same range of values.

State Space Exploration. A trivial utility of **K-LLVM** is state space exploration through the ***ksearch*** tool. Users can use ***ksearch*** (actual command: `krun -search`) to see all possible final results and traces of multi-threaded programs based on the automatically generated interpreter for **K-LLVM** in $\mathbb{K}$. This can be useful for detecting out-of-thin-air behaviors. For example, by assuming sequential consistency, if we execute `program-B` (Fig. 2) with the initial values of zero in both memory fields for pointers $@x$ and $@y$, the final results of $\%a$ and $\%b$ can never both be zero. We can also detect undefined values of a race. According to the documentation of LLVM IR, when a non-atomic `store` happens, and another memory operation from another thread is trying to access the same field, a race happens, and the two memory operations both get `undef`. By using ***ksearch*** to execute `program-C` (Fig. 2), we can see `undef` for variables $\%a$ and $\%b$ in some final results.

Additionally, the option `-pattern` allows us to filter the traces generated by executing a multi-threaded program. This option can be used to detect some interesting behaviors. For example, in **K-LLVM**, the `globalControl` component has a sub-component named `waitJoinThreads` that is used to store the states when a thread is waiting to join its child threads. If two threads in **K-LLVM** use the Pthread library function `pthread_join` to wait for each other in a multi-threaded program, the result is a deadlock. We can use the `-pattern` option with the pattern $\langle M$ `(` X `|->` `EDEADLK)` $\rangle_{\texttt{waitJoinThreads}}$ to detect if any trace of the multi-threaded program results in a deadlock. The key word `EDEADLK` is a flag in the Pthread library meaning that a thread has ended in a deadlock. Variable X represents any thread with an unspecified thread ID.

6 Conclusion and Future Work

In this paper, we propose **K-LLVM**, a formal semantics of LLVM IR in $\mathbb{K}$. The main advantages of **K-LLVM** is its relatively completeness and its implementation via a novel abstract machine for LLVM IR. To the best of our knowledge, **K-LLVM** is the most complete formal semantics of LLVM IR. We fully define the static semantics and dynamic semantics of LLVM IR relative to a sequentially consistent memory model. To validate its completeness, we ran 1,385 unit testing and around 3,000 concrete test programs, all of which **K-LLVM** successfully executed. **K-LLVM** provides guidance and reference to future compiler developers on exactly what are permissible behaviors in running LLVM IR programs. It also provides important piece of a framework for proving properties of compilers to or from LLVM IR. The **K-LLVM** abstract machine is a concise way of specifying how each LLVM IR instruction interacts with different computer components. In particular, **K-LLVM**

covers corner cases and side-effects of instruction semantics that previous work does not have, such as the different cases of the `getelementptr` operators, casting operators, and memory operators. **K-LLVM** also supports multi-threaded behaviors and provides users a collection of tools, including a state-space searching tool to explore traces of their LLVM IR programs under the assumption of sequential consistency. While this was not the main focus of this work, we also found more than 20 bugs in the current LLVM implementation, Clang.

In follow-on work to this paper, we have two on-going studies of **K-LLVM**. First, we are trying to finalize the full LLVM IR memory model in **K-LLVM**, including the behaviors of different atomic memory orderings and `volatile` memory accesses, with heavy testings and proofs of its relationship with existing C++ memory models [1, 18, 19, 15, 39, 7]. Second, we are defining a formal semantics for Haskell and verifying the correctness of the compiler from Haskell to LLVM IR, which requires both the semantics of Haskell and the semantics of LLVM IR as given in this paper.

References

1 Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ Concurrency. *SIGPLAN Not.*, 46(1):55–66, January 2011. doi:10.1145/1925844.1926394.

2 Sandrine Blazy and Xavier Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. doi:10.1007/s10817-009-9148-3.

3 Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A Trusted Mechanised JavaScript Specification. *SIGPLAN Not.*, 49(1):87–100, January 2014. doi:10.1145/2578855.2535876.

4 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. doi:10.1145/2676726.2676982.

5 Egon Börger, Nicu G. Fruja, Vincenzo Gervasi, and Robert F. Stärk. A High-level Modular Definition of the Semantics of C#. *Theor. Comput. Sci.*, 336(2-3):235–284, May 2005. doi:10.1016/j.tcs.2004.11.008.

6 Soham Chakraborty and Viktor Vafeiadis. Formalizing the Concurrency Semantics of an LLVM Fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 100–110, Piscataway, NJ, USA, 2017. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=3049832.3049844.

7 Soham Chakraborty and Viktor Vafeiadis. Grounding Thin-air Reads with Event Structures. *Proc. ACM Program. Lang.*, 3(POPL):70:1–70:28, January 2019. doi:10.1145/3290383.

8 The Coq development team. *The Coq Proof Assistant Reference Manual*. LogiCal Project, 2019. Version 8.9.0. URL: http://coq.inria.fr.

9 Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java Type System Sound? *Theor. Pract. Object Syst.*, 5(1):3–24, January 1999. doi:10.1002/(SICI)1096-9942(199901/03)5:1<3::AID-TAPO2>3.0.CO;2-T.

10 Chucky Ellison and Grigore Rosu. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, New York, NY, USA, 2012. ACM. doi:10.1145/2103656.2103719.

11 Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal Analysis of Java Programs in JavaFAN. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 501–505, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

12 Daniele Filaretti and Sergio Maffeis. An Executable Formal Semantics of PHP. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 567–592, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

13 Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995. URL: http://dl.acm.org/citation.cfm?id=233976.233979.

14 Myra Van Inwegen and Elsa L. Gunter. HOL-ML. In *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, HUG '93, pages 61–74, London, UK, UK, 1994. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=646520.694367.

15 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A Promising Semantics for Relaxed-memory Concurrency. *SIGPLAN Not.*, 52(1):175–189, January 2017. doi:10.1145/3093333.3009850.

16 Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A Formal C Memory Model Supporting Integer-pointer Casts. *SIGPLAN Not.*, 50(6):326–335, June 2015. doi:10.1145/2813885.2738005.

17 Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. Crellvm: Verified Credible Compilation for LLVM. *SIGPLAN Not.*, 53(4):631–645, June 2018. doi:10.1145/3296979.3192377.

18 Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming Release-acquire Consistency. *SIGPLAN Not.*, 51(1):649–662, January 2016. doi:10.1145/2914770.2837643.

19 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing Sequential Consistency in C/C++11. *SIGPLAN Not.*, 52(6):618–632, June 2017. doi:10.1145/3140587.3062352.

20 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

21 Daniel K. Lee, Karl Crary, and Robert Harper. Towards a Mechanized Metatheory of Standard ML. *SIGPLAN Not.*, 42(1):173–184, January 2007. doi:10.1145/1190215.1190245.

22 Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. Reconciling High-level Optimizations and Low-level Code in LLVM. *Proc. ACM Program. Lang.*, 2(OOPSLA):125:1–125:28, October 2018. doi:10.1145/3276495.

23 Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. Taming Undefined Behavior in LLVM. *SIGPLAN Not.*, 52(6):633–647, June 2017. doi:10.1145/3140587.3062343.

24 Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012. URL: https://hal.inria.fr/hal-00703441.

25 Liyi Li and Elsa Gunter. LLVM Semantics, 2019. URL: https://github.com/liyili2/llvm-semantics-1.

26 Liyi Li and Elsa L Gunter. IsaK: A Complete Semantics of K. Technical Report http://hdl.handle.net/2142/100116, University of Illinois at Urbana-Champaign, June 2018.

27 Liyi Li and Elsa L. Gunter. IsaK-Static: A Complete Static Semantics of K. In Kyungmin Bae and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software*, pages 196–215, Cham, 2018. Springer International Publishing.

28 llvm.org. LLVM Language Reference Manual, 2018. URL: `http://releases.llvm.org/6.0.0/docs/LangRef.html`.

29 Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably Correct Peephole Optimizations with Alive. *SIGPLAN Not.*, 50(6):22–32, June 2015. `doi:10.1145/2813885.2737965`.

30 Savi Maharaj and Elsa L. Gunter. Studying the ML Module System in HOL. In *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 346–361, London, UK, UK, 1994. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=646521.759249`.

31 William Mansky. Specifying and verifying program transformations with PTRANS. Technical Report http://hdl.handle.net/2142/49385, University of Illinois at Urbana-Champaign, May 2014.

32 William Mansky, Elsa L. Gunter, Dennis Griffith, and Michael D. Adams. Specifying and Executing Optimizations for Generalized Control Flow Graphs. *Science of Computer Programming*, 130:2–23, November 2016. `doi:10.1016/j.scico.2016.06.003`.

33 Narciso Martí-Oliet and José Meseguer. Rewriting Logic as a Logical and Semantic Framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

34 Paul E. Mckenney. Memory Barriers: a Hardware View for Software Hackers, 2009.

35 Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.*, 3(POPL):67:1–67:32, January 2019. `doi:10.1145/3290380`.

36 Robin Milner, Mads Tofte, and David MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

37 Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015. `doi:10.1145/2737924.2737991`.

38 Lawrence C. Paulson. Isabelle: The Next 700 Theorem Provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

39 Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL):69:1–69:31, January 2019. `doi:10.1145/3290382`.

40 Grigore Roşu. K Implementation, 2016. URL: `https://github.com/kframework/k`.

41 Grigore Roşu and Traian Florin Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. `doi:10.1016/j.jlap.2010.03.012`.

42 Don Syme. Proving Java Type Soundness. In *Formal Syntax and Semantics of Java*, pages 83–118, London, UK, UK, 1999. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=645580.658814`.

43 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011. `doi:10.1145/1993316.1993532`.

44 Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. *SIGPLAN Not.*, 47(1):427–440, January 2012. `doi:10.1145/2103621.2103709`.