K-LLVM: A Complete Semantics of LLVM IR

ANONYMOUS AUTHOR(S)

LLVM is designed for the compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages. In this paper we define **K-LLVM** in $\mathbb{K}$, a reference semantics for LLVM IR. Two contributions of **K-LLVM** are described in the paper. First, to our best knowledge, **K-LLVM** is the most complete formal LLVM IR semantics to date. Second, and more importantly, we propose an execution model and a memory model for describing the non-deterministic behaviors of LLVM IR. The models allow us to describe our formal semantics in terms of simulating a conceptual virtual machine that runs LLVM IR programs. The benefits of the models are that they allow us to bring the out-of-order and speculative execution features into **K-LLVM** and to design our semantics more closely reflecting real-world implementations of compilers and hardwares, as well as to describe the mixed features in an understandable way.

Additional Key Words and Phrases: LLVM, formal semantics, K framework, memory model, execution model

1 INTRODUCTION

The Low Level Virtual Machine (LLVM) is designed for the compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages. The language-agnostic design of LLVM has since spawned a wide variety of front ends, including C, C++, Objective C, Haskell, Java bytecode, Python, C#, and others. LLVM has become a robust, industrial-strength, and open-source compilation platform competing with GCC in terms of compilation speed and performance of the generated code [Lattner and Adve 2004]. As a consequence, it has been widely explored in both industry and academia. 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 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 (or the LLVM assembly language as the LLVM group calls it right now) acts as the central station for transforming high-level languages into target architectures, with a fixed set of language syntax, instructions, library functions and a memory model [llvm.org 2018].

Compiler correctness is one of the users' biggest concerns with an IR language. Previous work [Lopes et al. 2015; Yang et al. 2011] has identified more than 200 LLVM compiler bugs. To verify compiler correctness, we first need to know what is the correct behavior of a compiler, and formalizing a rigorous executable specification for the LLVM IR language is the first step. This paper addresses two issues in defining such a specification. The first issue is the completeness of current LLVM IR specifications. To the best of our knowledge, VeLLVM [Zhao et al. 2012] is the only notable attempt to give LLVM IR a formal semantics, and it only provides a very limited subset of LLVM IR features, which do not include LLVM library functions, a correct memory model and standard C library functions. Our **K-LLVM** semantics defines almost all of the features in LLVM IR that are listed in the LLVM documentation, which has more than 219 pages.

The more important issue is the gap between formal specifications for IR languages or large imperative languages and the real implementations of these languages. On one side, a lot of previous formal executable semantics for these large languages mainly focus on defining individual features, such as VeLLVM [Zhao et al. 2012], CompCert [Blazy and Leroy 2009], K-Java [Bogdănaş and Roşu 2015] and C semantics in $\mathbb{K}$ [Ellison and Rosu 2012]. The execution of these specifications

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

is based on an assumption that program execution always follows the sequential program order. However, a lot of industrial language designers or implementors have already expressed in their speeches [Stroustrup 2014; Ziegler 2015] and language specification documents [llvm.org 2018; Stroustrup 2013] that one should not expect a program will be executed in a strict program order and compilers or processors reorder program instructions and do speculative executions to speed them up. In a lot of cases involving multi-threaded programming, one cannot explore the full range of nondeterministic behaviors by using formal executable semantics that only assume sequential program order. What is needed is a system to show the full range of nondeterministic behaviors for a language. The **K-LLVM** execution and memory models are defined for exploring a full set of nondeterministic semantic behaviors of LLVM IR.

To fully explore the behaviors of LLVM IR, we need a framework that combines its runtime system, out of order and speculative execution features and attached memory model, which is our **K-LLVM**. It is a complete executable LLVM semantics based on LLVM IR 6.0.0, with an execution model that connects the runtime, speculative execution and memory model together. In develop the semantics, we followed a test driven development methodology and used the LLVM test suite with more than 23,000 programs to validate the completeness of our **K-LLVM** specification. In designing the execution and memory model to answer the second issue above, we borrow ideas from real world implementations, especially the Tomasulo algorithm [Tomasulo 1967]. We create a framework that places the necessary components in the Tomasulo algorithm and simulates the execution model that LLVM programs run on. Then we extend the framework with additional devices to capture all of the behaviors of LLVM's memory operators. The final **K-LLVM** form allows the execution of all LLVM operators, intrinsic library functions and important standard C library functions, such as the C dynamic memory allocation functions, C standard I/O functions, pthread library and pthread-mutex library functions. This model also allows the LLVM instructions to be executed out of order, handles speculative execution, and simulates a real world memory environment by taking memory caches into account.

Several benefits come from **K-LLVM**. First, we shorten the gap between the formal executable semantics of languages and the real world implementations of them, where we provide the correct nondeterministic behaviors for users to observe including extra behaviors caused by out-of-order and speculative executions, which lead to strange behaviors such as out-of-thin-air behaviors. Second, we provide a unified framework for people to observe semantic behaviors in a single interface. Transforming programs from a high-level language to a low-level machine code requires a lot of phases, each of which might cause correctness issues. For example, the infamous out-of-thin-air problems can arise in every level of the intermediate AST as a result of a transformation or compiler optimization. It can even appear when some old processors try to execute certain programs [Mckenney 2009]. **K-LLVM** provides a single reference of permissible semantic behaviors for LLVM IR, so that when users define future hardware or new aggressive compiler optimizations, they can make sure the new design respects the LLVM IR semantics by looking at **K-LLVM**. Third, **K-LLVM** utilizes $\mathbb{K}$ features to create conceptual devices and a virtual machine that runs LLVM IR code. This makes **K-LLVM** understandable and interpretable to users. Instead of understanding the axiomatized memory events, users can understand **K-LLVM** by the central processors, threads, memory caches, etc.

2 RELATED WORK

Here we discuss VeLLVM, the other formal executable semantics of LLVM IR, and a few closely related LLVM and C++ memory models. We also review large language specifications influencing the design of **K-LLVM**.

Feature	VL	KL
Modules and layouts	●	●
Function definitions and declarations	●	●
Global Variables and thread local storage model	◐	●
Aliases	○	●
Type system with address space	◐	●
Ensure well-formedness and validity checks	◐	●
Ensure local variables as count values	○	●
Integers, arrays, structs and undefined values	●	●
Floating point values and constant expressions	◐	●
Char list, block addresses and vectors	○	●
Packed struct values, labels and poison values	○	●
Arithmetic, Logical and Floating-point expressions with flags	◐	●
Function calls with musttail and inalloca flags	◐	●
Icmp and fcmp operators	◐	●
getelementptr	◐	●
Br, phi, select, ret and unreachable operators	●	●
Switch and indirectbr operators	○	●
Cast operators	◐	●
Vector operators	○	●
Aggregate type operators	○	●
Exception handlings such as invoke, landingpad and resume, etc	○	●
Basic non-atomic memory operators	●	●
Basic atomic memory operators with different orderings	○	●
fence, cmpxchg and atomicrmw operators	○	●
Va_arg operator	○	●
Intrinsic library functions	○	●
C standard dynamic memory allocation functions	◐	●
C standard I/O functions	○	●
Threading library functions	○	●
Mutex library functions	○	●
Runtime and execution model	○	●
Memory model	◐	●

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

VL represents VeLLVM[Zhao et al. 2012] and KL is our work.

Fig. 1. Completeness comparison of various semantics of LLVM

VeLLVM. VeLLVM was the only attempt to define a complete specification of LLVM prior to **K-LLVM**. It is defined in Coq and covers a limited set of LLVM IR instructions. VeLLVM formalizes a mechanized formal semantics for LLVM IR, its type system, and properties of its SSA form. It also has an interpreter extracted from Coq that runs 145 test programs and passes 134 of them.

Figure 1 provides a comparison between VeLLVM and **K-LLVM** on all features for which they 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. For example, VeLLVM lacks aliases, poison values, vectors and block addresses. Also missing are operators such as `switch`, `indirectbr`, `va_arg`, vector operators, exception handling operators, and some atomic memory operators like `fence` and `atomicrmw`. Furthermore, it does not include any library functions except some dynamic-memory-allocation functions such as `malloc` and `free`. One of the most important differences between VeLLVM and **K-LLVM** is that **K-LLVM** actually splits the semantics into static and dynamic semantics, so that actions that occur at compilation time can be separated from those that occur 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 documentation, where a constant expression cannot contain any local variable, and it must be reduced to a value such as an integer, pointer value, etc., in the compilation time. However, VeLLVM allows constant expressions to contain local variables, so evaluating them becomes an execution time event, which directly contradicts the LLVM documentation.

There are also some missing features for individual LLVM entities in the current implementation of VeLLVM. For example, even though VeLLVM has floating-point values, it does not perform

the check if an input floating-point constant is able to be translated precisely into an IEEE 754 representation, which LLVM requires. Hence, the floating-point constant `1.1` is not a valid input for any floating-point types in LLVM but VeLLVM allows it. In addition, VeLLVM does not check if the input index for a `struct` type in a `getelementptr` is a constant integer value. Finally, it uses the CompCert memory model, which excludes a lot of interesting multi-threaded behaviors.

Runtime Execution Models. The idea to have an execution model that connects the **K-LLVM** runtime, instruction semantics and memory model was influenced by real-world execution models. Mainly, we are enlightened by the Tomasulo algorithm [Tomasulo 1967], and to a lesser extent by some current execution models such as the MPC model by Perache et al. [Pérache et al. 2008], the fractal model by Subramanian et al. [Subramanian et al. 2017], and the copy or discard (CorD) execution model by Tian et al. [Tian et al. 2008]. Tomasulo's algorithm is simple enough to act as a guideline for the **K-LLVM** execution model. The problem with the other models is that they are too complicated, containing too many details about hardware. Even though **K-LLVM** tries to create the abstract view of a conceptual virtual machine to support executions of LLVM instructions, it is still at an abstract level instead of focusing on hardware details. Although the cells in the **K-LLVM** configuration relate to real-world hardware devices, their main functionality is to give an abstract view of the program state pieces that allow the LLVM IR instructions to perform actions.

Memory Models. We are also enlightened by various memory models. Rakamaric and Hu [Rakamarić and Hu 2009] defined a simple single-threaded memory model for lower-level codes. Chakraborty and Vafeiadis [Chakraborty and Vafeiadis 2017] gave a fragment of the LLVM concurrency memory model, but they only defined the LLVM sequential consistency and release-acquire consistency memory orderings for atomic memory operators. In the LLVM document [llvm.org 2018], it specifically says that its memory model is C++ plus a special memory ordering named `unordered` on the operators that capture the weakest happens-before memory consistency, the same as one described by Manson et al. [Manson et al. 2005]. A C++ memory model will be a good reference for **K-LLVM**.

The C++ memory model was designed Boehm and Adve [Boehm and Adve 2008] on the C++ standards committee. Batty et al. then formalized the C++ model and proved soundness of its compilation for x86-TSO [Batty et al. 2013, 2011], and Nienhuis et al. produced an operational semantics based on the C++ memory model [Nienhuis et al. 2016]. However, a number of problems have been found in the C++ model. A number of papers [Boehm and Demsky 2014; Dodds et al. 2013; Norris and Demsky 2013; Vafeiadis and Narayan 2013] unveiled the effects of out-of-thin-air behaviors allowed by the C++ memory model and proposed solutions. Vafeiadis et al. [Vafeiadis et al. 2015] discovered a number of other problems in the memory model, which invalidated some of the source-to-source program transformations that were previously assumed to be valid. In 2016, Batty et al. [Batty et al. 2016] defined a more concise semantics for sequential consistency atomic orderings, but the semantics is stronger than that of the C++ model, so it cannot be compiled efficiently to Power. They also pointed out that sequential consistency fences are too weak to guarantee sequential consistency. A number of papers [Lahav et al. 2016; Lahav and Vafeiadis 2015; Turon et al. 2014; Vafeiadis and Narayan 2013] have studied small fragments of the C++ model, typically the release/acquire fragment. Finally, Pichon-Pharabod and Sewell proposed a concurrency semantics for relaxed atomics in the C++ memory model to avoid out-of-thin-air executions and permit some compiler optimization[Pichon-Pharabod and Sewell 2016].

Our **K-LLVM** memory model only relies on these geniuses' correct description of the C++ memory model and happens-before memory model, and we construct it based on the structures similar to the one in the Tomasulo algorithm. There are several differences between **K-LLVM** memory model and the other models above. First, a lot of them are axiomatic, and talk about

small fragments of the C++ memory model. It is hard to sum them up in a single operational memory model. Even though there are a few operational ones, such as the one done by Nienhuis et al. [Nienhuis et al. 2016], they still talk very little about the interactions between the memory operators and other important instructions such as those for stack access operators, thread library functions and mutexes. Their operational semantics is based on a simple C sequential consistency semantics that does not have the concept of out of order instruction execution and speculative execution. They mainly focus on proving the existence of an operational model that is a refinement of the axiomatic one. What we want is actually a more practical operational memory model that interacts with other LLVM operators and C library functions. So, instead of avoiding harmful behaviors, we want to define abstract devices which will allow users to see the harm. For example, we purposely design a non-atomic memory operator to read/write a byte at a time, and design devices to allow monitoring when races happen. In addition, we want to provide users with abstract views of the different components of our virtual machine executing the LLVM operators so that the users can easily understand what is going on. For example, we purposely do not allow the relaxed atomics (called `monotonic` ordering in LLVM) to have out-of-thin-air executions, but design special devices to allow the LLVM `unordered` atomics to have them (which corresponds with the happens-before memory model). Thus, users can see clearly that in **K-LLVM**, if there is not a special device to provide extra devices so the LLVM `unordered` atomics in a thread will read/write data from other threads from which the program counter is executing operators speculatively; out-of-thin-air executions will never happen.

Other Large Language Specifications. We recognize four real-world language specification forms: a description in English with some mathematical details and examples, such as C standard, which is well written and precise; a compiler/interpreter implementation such as PHP; rigorous mathematical specifications, such as Standard ML [Milner et al. 1997]; and formal and executable specifications. Our semantics of **K-LLVM** is of the fourth kind. Standard ML by Milner, Tofte, Harper, and Macqueen [Milner et al. 1997] is one of the most prominent and mathematical programming language specifications, whose formal and executable specifications were given by Lee, Crary, and Harper [Lee et al. 2007], VanInwegen and Gunter [Inwegen and Gunter 1993], and Maharaj and Gunter [Maharaj and Gunter 1994].

In CompCert, Blazy and Leroy [Blazy and Leroy 2009] intended to verify an optimizing compiler based on CLight, which is a significant portion of C. They used Coq to generate a compiled code that behaved exactly as described by the language specification. Other projects based on CompCert include Appel's, which combined program verification with a verified compilation software tool chain [Appel 2011]. A third such project was CompCertTSO by Sewell [Sevcik et al. 2011], which proposed to verify the x86 weak memory model [Alglave et al. 2010]. Big language specifications have been defined in $\mathbb{K}$ including C [Ellison and Rosu 2012], PHP [Filaretti and Maffeis 2014], JavaScript [Park et al. 2015], and Java [Bogdănaş and Roşu 2015]. They are executable, have been validated by test banks, and, through the addition of some formal analysis tools produced by $\mathbb{K}$, have shown usefulness. We cannot list all of the interesting examples of formalized language specifications in this paper for space reasons. There is a lot of work on formalized specifications in Java and C#: Eisenbach's formal Java semantics [Drossopoulou et al. 1999] and Syme's HOL semantics [Syme 1999] of Drossopoulou; the C# standard by Börger et al. [Börger et al. 2005], which is formally executable and uses abstract state machines [Gurevich 1995]; and the executable Java specification by Farzan et al. [Farzan et al. 2004].

Our mechanized specifications of **K-LLVM** shared many of the difficult challenges faced by the works described above, and involved 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 provide background on the LLVM language and $\mathbb{K}$, and discuss some major challenges we had when developing **K-LLVM**.

3.1 LLVM Language Specification and A Taste of LLVM IR Programs

The LLVM language (LLVM IR) is a statically and strongly typed, assembly-like, Static Single Assignment (SSA) based language. It is used as a target intermediate representation for compiling other high level languages. It has undefined behaviors but the undefinedness is well documented. The official LLVM 6.0.0 language reference manual has more than 219 pages [llvm.org 2018]. 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++'s. 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` functions to provide heap memory access and functions dealing with environment such as `abort`, `exit` and `system`. It also needs functions listed in the `stdio.h` header to provide I/O support, as well as library functions from the pthread and mutex libraries to provide threading and mutual exclusion behaviors. These functions are not strictly part of the LLVM IR listed in the LLVM documentation but we define them anyway.

LLVM IR is a register based IR language. We have had a taste on LLVM IR through the example in Figure 2. LLVM IR distinguishes local variables with global variables. variables starting with the character `%` are local ones, while variables starting with the character `@` are global ones. Global variables can only have a pointer type. The `i32` and `i1` or any number following a character `i` in LLVM IR means that an integer type declaration with bits indicated by the number. while `i32*` refers to a 32-bits integer pointer type declaration. Instruction starting with keyword `icmp` refers to The integer comparison operator. With a keyword `ne`, the instruction `%r4 = icmp ne i32 %r3, 0` means that we compare the value in the variable `%r3` with `0` and stores the comparison result to the variable `%r4`.

Some Important Definitions. Before we go into the details of Challenges on **K-LLVM**, we first define out of order, speculative execution, and an executing block in Definition 3.1, 3.2 and 3.3.

Definition 3.1. **Executing Block.** An **executing block** is a basic block that has been selected by the execution model for execution and assigned a dynamically generated executing block number.

Definition 3.2. **Out of Order Execution.** In a program state, asssume that we have a block number b representing the current executing block. Given an executing block of instructions whose block number is equal to b, **out of order execution** means that we execute the instructions in the basic block in any order as long as the execution has the same effect as the sequential execution of the instructions.

Definition 3.3. **Speculative Execution.** In a program state, assume that we have a block number b representing the current executing block. **Speculative execution** means to pick some instruction in a future executing block with a block number b' ($b < b'$) that might or might not really execute. Executing, selecting or Observing such instruction is called seeing the instruction in **the speculative stage**.

Challenges. The challenges of **K-LLVM** occur in two different levels. The first is the sheer size 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, the `store` operator can be either non-atomic or atomic, and the atomic `store` operator has six different orderings. All of these require different semantic rules. The `getelementptr` operator allows indices to be integer local variables if the input type of the pointer is an `array` pointer, but if the pointer type is a `struct` pointer, LLVM IR requires the indices to be integer constants that can be statically reduced to integer values, and these two types can be mixed together in an LLVM IR pointer. In an input list of the indices for a `getelementptr`, it is very likely that some indices will be required to be integer values, while others will not. The previous LLVM IR formal semantics only defined some of the operators. When they were defined, some of them only had some of their features defined, such as the `getelementptr` operator above. No previous work defined the massive number of intrinsic library functions, which are the key components of LLVM IR. For example, variable argument intrinsics are functions for dealing with the variable input arguments of an LLVM IR function and their definitions rely on having a well-defined stack structure, which requires careful design. Some operators require careful static checks on the input constants, which are ignored by the previous LLVM IR semantics. In LLVM IR, the input decimal representation of a floating-point constant needs to be exact, which means that the value `1.1` cannot be a valid constant for floating-point operators in LLVM IR because the real value `1.1` cannot be precisely represented by finite floating points. This is an error in both Clang (the LLVM compiler), and the VeLLVM semantic framework.

```
Thread 1:                                          Thread 2:

 %r1 = load atomic i32, i32* @x unordered, align 1  %r3 = load atomic i32, i32* @y unordered, align 1
 %r2 = icmp ne i32 %r1, 0                           %r4 = icmp ne i32 %r3, 0
 br i1 %r2, label %bad %exit                        br i1 %r4, label %bad %exit
bad:                                               bad:
 store atomic i32 42, i32* @y unordered, align 1    store atomic i32 42, i32* @x unordered, align 1
exit:                                              exit:
...                                                ...
```

Fig. 2. Example for Speculative Execution Affecting Memory Consistency

The second level of the challenges is the definition of the LLVM runtime. As we said, LLVM IR relies heavily on the C library functions and the C++ memory model to provide multi-threaded behaviors. Defining a runtime environment for the C++ memory model is not a simple value-added quantity like being more careful or giving more variants of essentially the same feature. Instead, it requires comprehensive thought in the design of the LLVM runtime, and also possibly new model designs. In the LLVM documentation, it states that its memory model is based on the memory operators of C++, together with an extra memory ordering `unordered` for compiling Java shared memory operators by using the happens-before memory model and assuming the transformed LLVM programs have enough checks to avoid out-of-thin-air problems. Hence, we can recognize the LLVM memory model as basically a C++ memory model with a little bit from the happens-before memory model. In the clarification of the C++ memory model from Lahav et al. [Lahav et al. 2017], it states that one cannot expect an instruction *L1* happens before another instruction *L2* in a thread only because *L1* is sequenced-before *L2* in the program order of the thread. In fact, if *L1* and *L2* are the only two instructions in a thread and they are both relaxed atomics in C++, committing *L1* first or *L2* first to the main memory are both valid orders. Hence, the system designed for the LLVM runtime must consider the out-of-ordering execution, which is a lot harder than designing a sequential execution model.

The LLVM documentation also indicates that it allows speculative execution [llvm.org 2018]. Taking this into account, however, complicates the story. Figure 2 gives an example of how speculative execution might affect memory consistency. They are an LLVM IR version from the example in Figure 4 of the Java Memory Model paper [Manson et al. 2005]. If we have input content values

such that the global shared memories @x and @y have both content values `0`, the happens-before memory model allows their final values to both be `42`, because the two `store` operators can be executed speculatively before the branches are touched. Since LLVM IR provides a special `unordered` ordering so memory operators can support Java shared memory operators, **K-LLVM** also needs to support this behavior regarding those compilation checks because a formal semantics should be independent from construction of compilers. This means that we need to design a special conceptual device in our runtime system specification to support the fact that speculative execution might affect the values of memory operations.

In **K-LLVM**, we designed the execution model to both support the LLVM runtime and run individual instructions, including out-of-order and speculative execution. We also based our memory model on the C++ memory consistency model to support the LLVM memory consistency model. In the resulting combined semantics, we can see how these two pieces interact with each other. The design was especially challenging and innovative because we cannot find previous work defining a unified specification to describe the execution and memory models with connecting complete instruction semantics. Most previous work focuses on part of the story. As we stated in Section 2, some of them focus on defining individual instructions in strictly sequential program order, others focus on defining axiomatic memory models for CPU assembly languages such as X86 and ARM, as well as memory models to generate compilation schemes. In a sense, **K-LLVM** needs to get small pieces of ideas from all these previous work and combine them together nicely.

3.2 A Taste of the K-LLVM Configuration

$$\left\langle \left\langle \left\langle \langle K \rangle_{\text{k}} \ \left\langle \langle K \rangle_{\text{toExecute}} \ \langle (Set, Set, Set, Set) \rangle_{\text{toCommit}} \ \langle K \rangle_{\text{CPU}} \ \langle List \rangle_{\text{instQueue}} \ \langle Map \rangle_{\text{specTree}} \right\rangle_{\text{tomasulo}} \right\rangle_{\text{thread}*} \right\rangle_{\text{threads}} \right.$$
$$\langle Int \rangle_{\text{memCache-ID}} \ \left\langle \langle Int \rangle_{\text{currBlock}} \ \langle Map \rangle_{\text{registers}} \ \langle Map \rangle_{\text{specRegisters}} \right\rangle_{\text{control}}$$
$$\left\langle \langle Set \rangle_{\text{memoryList}} \left\langle \langle Int \rangle_{\text{cache-ID}} \ \langle List \rangle_{\text{message}} \langle List \rangle_{\text{memOpList}} \langle Map \rangle_{\text{byteMap}} \right.\right.$$
$$\left.\left. \langle \langle \langle (Int, Int) \rangle_{\text{chunkRange}} \langle Map \rangle_{\text{complete}} \langle Set \rangle_{\text{race}} \right\rangle_{\text{object}*} \right\rangle_{\text{objects}} \right\rangle_{\text{memory}*} \left.\right\rangle_{\text{memories}} \left.\right\rangle_{\text{T}}$$
$$\left\langle \langle Map \rangle_{\text{threadStatus}} \ \langle Map \rangle_{\text{waitJoinThreads}} \right\rangle_{\text{globalControl}} \langle Map \rangle_{\text{tempMem}}$$

Fig. 3. Subset of the LLVM Configuration

In $\mathbb{K}$, the configuration of a running program is represented by nested multisets of labeled cells. Figure 3 shows the most important cells used in our semantics. While this figure only shows 31 cells, we use over 110 in the full semantics. The large `T` cell contains the cells used during program evaluation. At the top, the `k` cell contains the current computation itself. A `globalControl` represents the global control unit. It contains the `threadStatus` cell that holds information for the `pthread_join` operator in the pthread library about a thread's returning information, and the `waitJoinThreads` cell that implements the feature that detects deadlock due to cyclic dependences arising from two threads waiting to join each other. For each `thread` cell, we have a `tomasulo` cell that contains all pieces implementing the **K-LLVM** execution model. The name indicates that the execution model originated from the Tomasulo algorithm. We will further introduce these cells and more cells inside the `tomasulo` cell in Section 5.1.

The `control` cell stores some local control unit information about the current computation. Inside the `control` cell, `currBlock` records the currently executing block number. The `control` cell also contains a `registers` cell and `specRegisters` cell, each containing a mapping from local variables to assigned values. The `specRegisters` cell assigns values to local variables if the instruction is in a speculative execution stage (the instruction is a future one that has been moved to execute early). A `tempMem` represents a temporary memory in LLVM IR and it is used to store global shared memory values for LLVM `unordered` atomics that follows the happens-before memory model. A `memories` cell might contain several `memory` cells, each of which represents a memory cache. A `memoryList` cell is also in the `memories` cell and it contains a set of tuples of two integers indicating the range of an allocated memory chunk. The `memory` cells also contain information

in the `memOpList` cell about performing memory operators. The details of these cells are given in Section 5.2.

3.3 The $\mathbb{K}$ Framework

$\mathbb{K}$ [Roşu and Şerbănuţă 2010] is a rewrite-based executable semantic framework in which programming languages, type systems and formal analysis tools can be defined using configurations, computations and rules. For a given syntax and semantics of a language in $\mathbb{K}$, $\mathbb{K}$ can generate an interpreter, as well as some formal analysis tools for the language at no additional cost. By using the interpreter, we can test our language specifications defined in $\mathbb{K}$ immediately, which increases the efficiency of semantics development significantly. In addition, the formal analysis tools facilitate formal reasoning for the language specifications, and they also help both in terms of the applicability of the specifications and in terms of engineering the specifications themselves. For example, the state-space exploration capability of our **K-LLVM** helps the language designer cover all the non-deterministic behaviors of certain constructs and their combinations.

$\mathbb{K}$ allows users to define language syntax by using conventional BNF annotated with semantic attributes, while the semantics based on the language syntax is given as a set of reduction equations and rules over a configuration. The configuration for the language is an algebraic structure of the program states, organized as nested labeled cells, in XML formats, that hold semantic information, including the program itself. Figure 3 provides a subset of the **K-LLVM** configuration. While the order of cells is irrelevant in the configuration, the contextual relations between the cells are relevant and are preserved by users when they are defining rules and by $\mathbb{K}$'s compilation step when it is "completing" them. Leaf cells represent pieces of the program state like computation stacks or continuations (e.g., k), environments (e.g., `registers`), heaps (e.g., `tempMem`), etc. For example, a simplification of a typical rule for reading a variable from a global shared memory would be:

$$\left\langle \frac{X}{E} \cdots \right\rangle_{\mathsf{k}} \ \langle \cdots X \mapsto N \cdots \rangle_{\text{registers}} \ \langle \cdots N \mapsto E \cdots \rangle_{\text{tempMem}}$$

There are three cells in the rule: k, `registers` and `tempMem`. The k cell is a computation sequence waiting to be performed, while the head element in the list is the next item to be computed. The head of the k cell, here X, is the current position of the computation. The `registers` cell contains a map of variables to location numbers, while the cell `tempMem` is a map of location numbers to expression values. The meaning of the rule above is that if the next computation to be executed is a variable lookup expression X, then we locate X in the environment to get its location number N in the location memory, and locate N in the heap to find its expression value E. Then we transform the computation into the value, E. A horizontal line represents a transition. A cell with no horizontal line means that it is read but does not change during the transition. The "⋯" represent portions of cells that are irrelevant.

This unconventional notation is useful in terms of allowing users to write less. The above rule would be written out as a traditional rewrite rule as follows:

$$\langle X \curvearrowright \kappa \rangle_{\mathsf{k}} \ \langle \rho_1, X \mapsto N, \rho_2 \rangle_{\text{registers}} \ \langle \rho_3, N \mapsto E, \rho_4 \rangle_{\text{tempMem}} \Rightarrow \langle E \curvearrowright \kappa \rangle_{\mathsf{k}} \ \langle \rho_1, X \mapsto N, \rho_2 \rangle_{\text{registers}} \ \langle \rho_3, N \mapsto E, \rho_4 \rangle_{\text{tempMem}}$$

Computations in the k cell are separated by "$\curvearrowright$", which is now observable. The κ and ρ_1, ρ_2, ρ_3, ρ_4 fill in the place of the "⋯" above. The most important thing to notice is that the rule is duplicated on the right-hand side. Duplication in a definition can lead to subtle semantic errors if users are not carefully synchronizing their changes to their specifications in multiple places, once changes have been made. In a big language like C, Java or LLVM IR, the configuration structure is very

complicated, and would require actually including additional cells like `threads` and `memories` (Figure 3). These intervening cells are automatically inferred in $\mathbb{K}$, which keeps the rules more modular.

Modularity is one of the most important features of $\mathbb{K}$. In the process of defining specifications, users usually do not need to modify existing rules to add a new feature to the language. $\mathbb{K}$ accomplishes this by structuring the configuration as nested cells and allowing users to design their specification rules by only mentioning the cells that are needed in those rules, and only the needed portions of those cells. For example, the above rule only refers to the `k`, `registers` and `tempMem` cells, while the entire configuration contains many other cells as shown in Figure 3. The modularity of $\mathbb{K}$ not only allows users to create a compact and human readable language specification, but also contributes to speeding up the semantics development process. For example, the above lookup rule will not change when a new cell is added to the configuration to support a new feature.

4 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 creates a parser for parsing any LLVM IR program into a FAST format generated from $\mathbb{K}$.

K-LLVM static semantics means the LLVM IR behaviors happen at compilation time. After the parsing, the static semantics takes as input the FAST representation of an LLVM IR program. Through the translation process in the static semantics including several checks such as type, static single assignment and well-formedness checks. The program in FAST is processed and translated into a representation in the BAST format. Then, the dynamic semantics takes the BAST for execution. Figure 4 describes the phases in the **K-LLVM** static semantics.

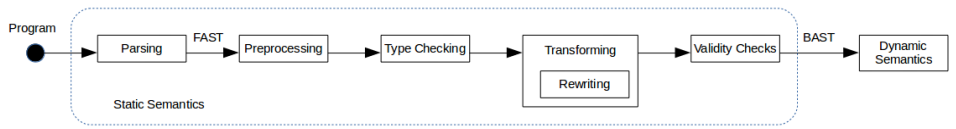


Fig. 4. Static Semantics to Dynamic Semantics in **K-LLVM**

Preprocessing. In this phase, a list of LLVM IR modules in FAST is analyzed and transformed. For each module, some metadata such as the target data layout (a string deciding how memory is laid out) and target triple (a string describing the target host) are analyzed and stored in special cells as the global data of the module. Global variables are analyzed and stored in a form of BAST as a quadruple of a global variable name, type, pointer and set of attributes (including linkages, preemption specifiers and visibilities, etc.). During the preprocessing phase, each non-external global variable is also given the static memory location, and its data is stored there. After dealing with the global variables, aliases are also analyzed and transformed to the BAST format. Aliases rely on the information from the global variables to get pointer address. During the preprocessing of the global variables and aliases, small type checks are performed to guarantee that the values of global variables and the target variables of any alias are coherent. In addition, syntactic sugar resolution is also performed so the possible constant expressions of the aliasee expression in each alias will get its pointer address correct. Finally, for function declarations and function definitions, the preprocessing phase only collects their header information including return types, argument type lists and attributes. The type checking and transformation of the bodies of the function definitions are done in the next few phases.

Type Checking. LLVM IR is a strongly typed language. Except for memory operations, the type system of LLVM IR guarantees that once an instruction is type checked, no program goes wrong because of a type mismatch problem. For memory operations, the LLVM documentation clearly states that its memory fields have no concept of types. Every piece of data is split and stored in blank data fields where each field has the size of a byte. Hence, it is possible that a value of one type is stored in a memory location that is then loaded with another type and becomes an undefined value (`undef` in LLVM IR). For example, suppose we store a struct value `{ i1 1, i64 0}` in the memory field referenced by the local variable *%x*. In LLVM IR, a struct is supposed to have alignment of its elements. Let's assume the user-defined alignment for the struct is eight bytes. Hence, if the element `i1 1` is stored in the first memory location, the element `i64 0` is stored in the ninth location. The second to eighth memory locations are all aligned fields having undefined values. If we load the memory locations of the struct with a 128-bit integer value, the result is an undefined value. The **K-LLVM** type checking process is a complete implementation of the LLVM IR type system listed in the document http://releases.llvm.org/6.0.0/docs/LangRef.html. The input for the **K-LLVM** type checking function is a term and its type, and the function outputs `true` if the term has type checked and it has the input type and it outputs `false` otherwise.

Constant Expression Rewriting. Constant expressions are terms in LLVM IR that can be used to express a complicated term in a constant position. For example, in the `icmp` instruction in Figure 2, one can replace the variable *%r3* or value `0` with a constant expression `add (i32 0, i32 1)` to express more complicated meaning. Constant expression rewriting is a compilation time process in LLVM IR. It is not hard, but can be very confusing in some cases. Other two requirements are needed for rewriting a constant expression besides the compilation time requirement. First, they cannot contain local variables as arguments. Second, Some constant positions of some instructions might have additional requirements, such as requiring all values for a constant expression to be ready in parsing time. For example, two `getelementptr` operations use constant expressions (`inttoptr (i32* @a to i32)` and `add (i32 1, i32 0)`) in index constant positions as follow:

(1) `getelementptr { i32, i32 }, { i32, i32 }* %x, i64 0, i32 inttoptr (i32* @a to i32)`
(2) `getelementptr { i32, i32 }, { i32, i32 }* %x, i64 0, i32 add (i32 1, i32 0)`

In the two instructions, the local variable *%x* is a pointer for a struct and the global variable *@a* is a pointer for a 32-bit integer. The constant expression `inttoptr` in the first instruction converts the pointer *@a* into a 32-bit integer value, while the constant expression `add` in the second instruction adds two integers together. In LLVM IR, the second instruction is valid, while the first one is not. In `getelementptr` instructions, if an index position is for indexing on a struct, the value is based on information discerned from parsing time, but the pointer address for *@a* is known in compilation time. This is why the first `getelementptr` instruction is not valid. When we define a function to rewrite constant expressions, we need not only to input the constant expressions and a map containing information about global variables and their compilation time generated addresses, but also to include a flag indicating if the positions holding the values of the constant expressions are required to be parsing time known or compilation time known. The output of the function is the values rewritten from the constant expressions.

Transformation. The **K-LLVM** transformation phase regularizes LLVM IR programs in FAST and rewrites them in BAST form. Figure 5 depicts most of the transformations performed by **K-LLVM**.

Rows 1 and 2 show how **K-LLVM** translates a single line of instruction into BAST form. Below, we define two operators for BAST to hold single line instructions. The operator `Inst` has three arguments: a term that is either an `Assign` or a `NoAssign` term, a set containing attributes in the instruction and a set containing all of the metadata in it. The operator `Assign` requires the arguments

#	Concrete LLVM	BAST
1	`%x = sub nuw i32 0, 1`	`Inst(Assign(%x, Sub(i32, 0, 1)),` `SetItem(nuw), .Set)`
2	`store i32 0, i32* %y,` `align 4, !range !0`	`Inst(NoAssign(Store(i32, 0, i32*, %y, 4)),` `.Set, SetItem(Meta(!range, !0)))`
3	`entry:` `%1 = sub i32 0, 1`	**rejected**

4	`entry:`
	`%0 = sub i32 0, 1`
	`%x = add i32 1, 1`
	`mul i32 2, 1`
	`ret i32 %1`
		`Block(%entry,`
		`0 -> Inst(Assign(%0, Sub(i32, 0, 1)), .Set, .Set)`
		`1 -> Inst(Assign(%x, Add(i32, 1, 1)), .Set, .Set)`
		`2 -> Inst(Assign(%1, Mul(i32, 2, 1)), .Set, .Set)`
		`3 -> Inst(NoAssign(Ret(i32, %1)), .Set, .Set))`

5	`sub i32 0, 1`
	`%x = add i32 1, 1`
	`mul i32 2, 1`
	`ret i32 %2`
		`Block(%0,`
		`0 -> Inst(Assign(%1, Sub(i32, 0, 1)), .Set, .Set)`
		`1 -> Inst(Assign(%x, Add(i32, 1, 1)), .Set, .Set)`
		`2 -> Inst(Assign(%2, Mul(i32, 2, 1)), .Set, .Set)`
		`3 -> Inst(NoAssign(Ret(i32, %2)), .Set, .Set))`

6	`entry:`
	`add i1 0, 1`
	`br i1 %0, label %1, label %2`
	
	`ret i32 0`
	
	`ret i32 1`
		`FunBody(`
		`%entry -> Block(%entry,`
		`0 -> Inst(Assign(%1,`
		`Add(i1, 0, 1)), .Set, .Set)`
		`1 -> Inst(NoAssign(`
		`Br(i1, %0, label, %1, %2)), .Set, .Set))`
		`%1 -> Block(%1,`
		`0 -> Inst(`
		`NoAssign(Ret(i32, 0)), .Set, .Set))`
		`%2 -> Block(%2,`
		`0 -> Inst(`
		`NoAssign(Ret(i32, 1)), .Set, .Set))`

Fig. 5. Examples of Transformations performed by the static semantics

to be a local variable and the instruction contents. It means that the instructions eventually returns a value assigned to the local variable. The operator `NoAssign` receives an argument from the instruction contents and indicates that the instructions do not return a value.

SYNTAX *KItem* ::=`Inst(` *K* `,` *Set* `,` *Set* `)`
SYNTAX *KItem* ::=`Assign(` *LocalVar* `,` *K* `)` | `NoAssign(` *K* `)`

Each LLVM IR operation has a corresponding BAST format. The BAST format contains only minimal operands extracted from the concrete LLVM IR operation. For example, `sub nuw i32 0, 1` is translated into a BAST form as: `sub(i32, 0, 1)`. The attribute `nuw` is removed here and put into the set of attributes in the `Inst` term. The same thing happens in Row 2: the metadata `! range !0` is removed from the BAST form of `store` and put in the metadata set of the `Inst` term.

Row 3 shows a case where the transformation may fail. **K-LLVM** adopts the LLVM IR variable counting conversion. If an instruction returns a value but has no specified local variable, and if a basic block has no label variable, **K-LLVM** assumes that there is a counter to generate the local variable for it. The counter is set to zero at the beginning of a function body. For each new position that needs a variable generated, **K-LLVM** assigns a new local variable by extracting it from the value of the counter, and then adding one to the counter. If there is an instruction that uses a number as its local variable. **K-LLVM** checks if the local variable has the same value as the counter. If not, the program fails.

Row 4 shows how to translate an LLVM IR basic block into BAST format. The construct `Block` takes a basic block label name and a map as its argument. The map is a mapping from the instruction

position number in the block to the actual BAST instruction term. Row 5 is basically the same as Row 4, except that it generates a new local variable to represent the block label name.

Row 6 shows how a whole function body is transformed into BAST format. The construct `FunBody` is the target transformed BAST format for a function body, and it takes a map as its argument. The map is a mapping from the basic block label name to its contents which is a `Block` term.

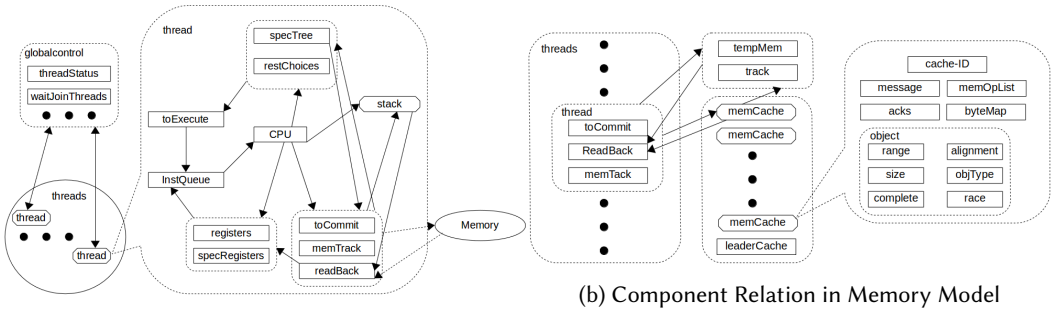
Validity Checks. There are a lot of validity checks (well-formedness checks) that **K-LLVM** needs to perform before a program can be sent to execution. In previous work, Zhao et al. [Zhao et al. 2012] defined the procedure to ensure Static Single Assignment (SSA) form in an LLVM IR program by using Kildall's method [Kildall 1973]. A little procedural modification in **K-LLVM** is that we distinguish between local variables used in a `phi` instruction and other ordinary instructions. The local variables that are arguments of a `phi` instruction only require definitions back on through all paths ending at the end of the block indicated by the label name associated with them. On the other hand, a non-`phi` instruction uses a local variable, then it must either be an input argument or be defined in all paths from the current block back to the entry block.

There are also other small well-formedness checks that LLVM IR needs to perform. They are simple but we need to conquer all of them. Here, we list all of the checks we have to perform in **K-LLVM**. First, for every label name mentioned in a `phi` instruction, we need to make sure that it is a real label name mentioned as block label, because LLVM IR actually allows users to register a new local variable by `select`-ing from two different block label names, and the new local variable should not be valid as a label name in a `phi` instruction. Second, we also need to check that the block mentioned in a `phi` instruction indeed has an edge pointing to the block where the `phi` instruction resides. Third, all `phi` instructions must appear before any other ordinary instructions in a block. Fourth, if there is a `blockaddress` value in a function, then the block label names mentioned in the `blockaddress` are indeed block names in the function, and the `blockaddress` should not be the entry block name. Fifth, if a block has a `landingpad` instruction, it must be the first non-`phi` instruction and the block can only have one `landingpad` instruction. Sixth, all block edges pointing to a block containing a `landingpad` instruction must come from the `unwind` destination block of an `invoke` function call. Seventh, each `resume` instruction must be dominated by an earlier `landingpad` instruction. Eighth, `catchswitch`, `catchret`, `cleanupret`, `catchpad` and `cleanuppad` instructions also have checks similar to those of `landingpad`, `invoke` and `resume` instructions.

After a program has been checked and transformed through **K-LLVM** static semantics, the transformed BAST program is ready for execution by the **K-LLVM** dynamic semantics.

5 DYNAMIC SEMANTICS OF K-LLVM

In this section, we define the dynamic semantics of **K-LLVM**, which describes the actions of running a LLVM IR programs. Figure 6 describes the major components of the conceptual machine that we have previously introduced. It contains two parts. The execution model (Figure 6a) describes how an LLVM IR instruction is selected and executed and how its value is assigned to a target register. Obviously, an LLVM IR instruction can have memory effects. The memory model (Figure 6b), which is the second part, describes these effects. It describes how we use different devices in the conceptual machine to guarantee the LLVM memory model. The directions of the arrows in the figure represent the action directions between the devices in the models. An octagon or round cell means a program state entity that might contain other cell structures as content, while a square cell means a program state entity whose content is values such as integers, lists, sets or maps.



(a) Component Relation in Execution Model

(b) Component Relation in Memory Model

Fig. 6. Relations Among Model Components

5.1 The Execution Model

Figure 6a represents the **K-LLVM** execution model. It contains a group of conceptual devices which together execute LLVM IR programs. It takes an LLVM IR program, outputs a sequence of computations and performs actions affecting threads, the main memory and other devices. For any single thread execution of a program, the execution model guarantees that every sequence of computations generated from it has the same effects as executing the program sequentially. In the top level (left column of the figure), there are a global control unit (the `globalControl` cell) and a thread pool (the `threads` cell). The `globalControl` cell stores global state information about the threads, mutexes and other entities that need to interact globally, such as output/input channels. For example, we have already described the `threadStatus` cell in Section 3.2 that stores return information about a thread, and the `waitJoinThreads` cell detects deadlocks among `pthread_join` operators in different threads.

The thread pool contains threads, each of which can be described by the single thread execution model on the right side of Figure 6a. The execution model mainly guarantees that an execution sequence of a single threaded program does not go wrong, so the interactions of devices inside a single thread need to guarantee this property. We summarize four major tasks for the single thread execution model to perform. First, it selects an instruction that might execute out-of-order or speculatively and guarantees that the selection does not change the meaning of the whole single threaded program. We do not distinguish out-of-order executions from in-order executions because they have no differences in our model. Second, it performs different actions that might affect other devices such as the main memory. Third, it assigns to registers the values returned by executing instructions making a distinction between registers receiving results from instructions in the speculative stage (the `specRegisters` cell) and those receiving results from instructions not in the speculative stage (the `registers` cell). Fourth, it helps decide from among the potentially speculative register values, which ones should be assigned to a local variable, and which memory operations are ready to commit to the memory through the `specTree` cell.

Here, we introduce the devices of the single threaded execution model which finish each of the four tasks named above. The first task is finished mainly by the `toExecute` and `instQueue` cells. In **K-LLVM** dynamic semantics, program evaluation starts at a single thread by executing a function named `main`. Its entry block is selected, assigned a newly generated executing block number, and placed in the `toExecute` cell. The executing block number is unique. By given an executing block, The main task of the `toExecute` cell is to move individual instructions in sequential order from the executing block to the `instQueue` cell. It also updates the local variables in the instructions

with values in the registers (the `registers` and `specRegisters` cells). Since **K-LLVM** allows out-of-order and speculative execution, we might have several different copies of assignments in the registers for a local variable. The `toExecute` cell needs information from the `specTree` cell to help decide which assignment is the right one to use for updating a local variable in an instruction. This is basically the fourth task described above. When all instructions in an executing block have been moved to `instQueue`, the `toExecute` cell speculatively selects a new executing block according to the control flow graph of the input program and returns to the beginning of its process. The new block is a child block of one of the existing executing blocks in the `specTree` cell.

The main task of the `instQueue` cell is to select an instruction to put in the `cpu` cell for execution once the `cpu` is empty. The `instQueue` cell is implemented as a set, and we move one instruction at a time from the set to the `CPU` cell once certain conditions are met. We name the conditions as **availability**. We separate the instructions into two groups. The first group includes function calls, branches and return instructions. One of these instructions is available if and only if it is the oldest one in the `instQueue` cell. The determination of oldest instruction is made by selecting an instruction with the least value of pairs of the instructions' execution block number and instruction number. The pair is named the **instruction pair**. This condition means that we do not want to execute out-of-order or speculatively for such instructions. The second group contains all other instructions. For this group, an instruction is available if and only if all its arguments have no local variables anymore. All arguments have been properly replaced by values in registers (the `registers` or `specRegisters` cells). The availability property can guarantee the single threaded program execution does not go wrong because when an instruction is available, its arguments have been filled with values that were calculated by previous instructions. The dependence of other instructions on this instruction has gone, so this instruction is free to be executed.

The second and third task are basically fulfilled by the `CPU`, `registers` and `specRegisters` cells. The main job of the `CPU` cell is to perform actions for each instruction by interacting with different devices and assigning return values to registers, executing them one at a time. In **K-LLVM**, we define all instruction level semantics based on the assumption that the instruction has been moved to the `CPU` cell. There are two kinds of instructions. One is those that return computed values to existing local variables. The `CPU` cell is responsible for assigning such value to the `specRegisters` or `registers`, depending on if the instruction is in the speculative stage or not. The other kind is those that do not return values. In these cases, the `CPU` cell is responsible for performing the semantic action of the instruction and notifying `InstQueue` that the `CPU` cell is ready to receive a new instruction. The `registers` cell is a map from variables to values, while the `specRegisters` cell is a map from pairs of an executing block number and a local variable to values. The purpose of the two registers is to act as a database for other cells to get the correct values for local variables that can have multiple instances in different speculative stages in the execution model. Two things need to be updated once registers change: the content in the `specTree` cell and the instructions in `instQueue`. We need to update the values for variables in the memory operation prototypes in the `specTree` cell, and we need to update the local variables in each instruction in `instQueue` with the new register values.

The fourth task is finished by the `specTree` `restChoices` cells. The cells contain information to help the model execute instruction speculatively. The `specTree` and `restChoices` cells store information to help the `instQueue` cell decide which instruction is moved to `CPU`, and help the `toCommit` cell decide which memory operation is ready to be committed to the memory. Its information gets updated when the `CPU` cell executes branching operators. We implement the cell by giving a map from an executing block number to a `RunningBlock` construct. A `RunningBlock` construct has five arguments: the executing block's original basic block label name, the parent executing block number, a list of the memory operators that will occur in it, the set of local variables defined

in the block and a set of child executing block numbers. There are three main jobs of the `specTree` cell: first, it is used to track all executing blocks and their parent-child relationships. The **K-LLVM** semantics allows speculative execution. An instruction can be executed even if the current program pointer is not pointing at the executing block where the instruction resides. Hence, we need to track the executing block information and be able to disable the effects of all instructions in it once we discover that it is not the correct speculative execution guess. Second, `specTree` has the task of determining if a memory operator in the `toCommit` cell is ready to be sent to the memory for execution, which will be discussed later. Third, the `specTree` cell has the task of maintaining enough information for **K-LLVM** to recognize which map entry in the `specRegisters` should be used for updating a particular local variable. The `restChoices` cell is used to store the remaining choices for a speculative guess. We need this cell because we do not specify a particular speculative strategy in our model. When our model is facing a branching choice, we might send any block choice to execute of any time; so we need to record our choices.

Besides the four main tasks of the execution model, the model also has other tasks to interact with other devices in the computer such as stacks, thread information cells (global control units) and the main memory. In Figure 6a, we use the `stack` cell to represent the interaction. Its structure is similar to that of the main memory in **K-LLVM**. The `toCommit`, `memTrack` and `readBack` cells are devices communicating with the main memory (the `Memory` cell in Figure 6a). It relies on `specTree` to select a memory operation to send to the main memory. We believe that the interaction between these cells and the main memory guarantees properties in our memory model so we will detail them in Section 5.2.

In this section, we have introduced components of our **K-LLVM** execution model and their relationship. Through the execution model, we see how **K-LLVM** provides a way to describe the execution of LLVM IR program instructions that are run out of order and speculatively, while guaranteeing the correctness of the program execution. We will examine the memory model in the next section.

5.2 The Memory Model

The **K-LLVM** operational memory model allows the execution model access and simulates the nondeterministic behaviors which occur when multiple threads try to reach the main memory at the same time. In particular, LLVM IR has a set of atomic memory operations that perform behaviors in the same way as the C++ memory model and an `unordered` atomic operation that performs the happens-before memory model behavior. **K-LLVM** combines these two different models into one involving three parts: threads, memory caches and a temporary memory store. The interactions among these three through message passing are the essence of our memory model. The inputs are memory operations sent by threads, and the output is a sequence of events that interact with the main memory. The components and relations of the **K-LLVM** memory model are described in Figure 6b. From the figure, we can see that the memory model is not strictly separated from the execution model. Some components in the latter play key roles in the former to guarantee the correctness. The top level of the model is comprised of a set of threads (in the column on the left in Fig 6b) communicating with a set of memory caches (middle column, bottom), and a temporary memory store (the `tempMem` cell). The memory caches and temporary memory store are the main memory. Each thread communicates with one cache, which might be shared with others.

In the top level, each thread sends memory operations to the temporary memory store or memory caches depending on if the memory operations have `unordered` ordering. The store or caches perform memory events based on the requests of the memory operations and feed information back to the threads if the operations are memory reads. All of the components in the figure are cells in the **K-LLVM** configuration that represent program state pieces. The left column contains a set of threads

(the threads cell) trying to communicate with the memory. Each thread follows the single thread execution model defined in the right column of Figure 6a. In terms of interaction with the memory, only three cells join the communication with memory caches and the tempMem cell: the memTrack, toCommit and readBack cells. The middle column represents the main memory. Memory caches are devices designed for implementing the C++ memory model and serving memory operations without unordered ordering. The extra device tempMem is used to implement the happens-before memory model through LLVM IR unordered atomics. The unordered atomics in each thread are allowed to interact with the tempMem cell speculatively. The detailed components of a memory cache are described in the right column of Figure 6b. We will discuss their usage through some sample rules in this section.

K-LLVM assumes that memory operations can be classified into only seven kinds that are transformed from memory type instructions when they are moved to the CPU cells. They are non-atomic write (writeByte), atomic write (atomicWrite), non-atomic read (readByte), atomic read (atomicRead), atomic read write (atomicReadWrite), seq_cst ordering fence (seqFence) or memory free (toClose). In **K-LLVM**, all fences except the seq_cst one have their effects inside a single thread, and we implement them by encoding them into the specTree cell. For simplicity, **K-LLVM** assumes all memory generation operators (e.g., malloc or alloca) happen right at the moment when they are moved to the CPU cell without going through the memory model process defined in this section, while a memory free operator will take part in the model. In **K-LLVM**, memory operations are wrapped by an additional singleMem construct that contains five arguments: the ID of the thread where the toCommit cell resides, the executing block number of the memory operator, the instruction position number (so these two numbers can be merged into an instruction pair), a flag indicating if the memory operation is a heap or stack, and a memory operator.

If we view only the tempMem cell and memory caches of the main memory, there are two things happening: memory operations are reordered and selected inside a single thread, and memory operations are sent to the main memory to perform memory reads or writes and wait for the feedback. We first investigate what happens inside a single thread. Three main tasks are applied in a single thread memory model. First, we have a set of memory operations that are waiting to be sent to the main memory, so the memory model randomly emits one of them. Second, the random selection must not violate the rules for non-atomic and atomic memory operations. In fact, one of the important facts that we observe from our memory model is that single thread behaviors take care of a large portion of the behavioral correctness of the non-atomic and atomic memory operations. Third, a single thread needs to have a device that waits for the feedback values from the read operators and then places them into correct devices (such as the registers and specRegisters cells). As we said above, the memTrack, toCommit and readBack cells are the three most important cells affecting the memory model inside a single thread.

The memTrack cell is related to guaranteeing the correctness of unordered atomics. It will be described in a later section. The readBack cell finishes the third task above. Its main functionality is to wait for messages containing the values of the non-atomic and atomic read operators from the main memory. For a non-atomic operation, a message contains one byte at a time, while for an atomic operator, we assume that a message contains enough bytes for the value. Once the readBack cell has enough bytes for the value of a read, it merges them and then pushes the value to the registers or specRegisters, which affect the execution model eventually.

The toCommit cell is the central device for finishing the first and second tasks above. It needs other cells in the execution model, such as the specTree and memCache-ID cells, to provide information. The former provides information on the program orders of the memory operations, while the latter tells by the ID which memory cache the thread should send operations to. The first task is easy to finish. To finish the second task, we find that all the behavioral correctness of the atomics (except

the seq_cst ones) can be guaranteed through a correct design of the toCommit cell within a single thread. We implement the toCommit cell as a tuple of four sets. The first set *R* stores the memory operations that are **ready** to move to the main memory. Being ready means that an operation has no dependent memory operations before it, and it satisfies the properties of the acquire, release and acq_rel orderings. In **K-LLVM**, we have a function for checking readiness that utilizes the information in specTree. The specTree cell records the program positions, operation types and memory location addresses of the memory operations. The second set *PR* stores operations that are ready but in the speculative stage. Moreover, those with the unordered ordering in *PR* have already been treated by special treatment rules in **K-LLVM**, so they are just waiting for the current executing block number to match their block numbers in order to be discharged from the speculative stage (this will be done through executing branching operators). When this happens, these memory operations will be moved to set *R*. The third set *NPR* stores memory operations that are ready but in the speculative stage. Those with the unordered ordering have not yet been treated by the special treatment rules, so they will be moved to *NPR* one by one or moved to the *R* set if the operations are not in speculative stage anymore. The fourth set *NR* stores memory operations that are either not ready or have not been checked; and they will be moved to the *R* or *NPR* sets once their readiness has been confirmed. The *PR* and *NPR* sets are only used to help memory operations with the unordered ordering in the speculative stage. This will be described later in this section.

The memory model describing the relationship between the threads and main memory is actually very simple. We can view this part as a set of threads and a set of memory caches. Each thread talks to a specific cache and a cache can talk to multiple threads and different caches. The behavior of this part can be better understood by a synchronous message passing communication model without failures, where messages sent from a single thread are received in order by the other party. The message passing between the threads and memory caches is just a send-and-receive message passing model, while the model among the different caches is basically solving the consensus problem. We first introduce the model between the threads and caches. A memory operation is sent to the end of the memOpList cell in a memory cache if it is in the set *R* of a toCommit cell. The determination of which memory cache is by a match of the IDs in the thread's memCache-ID cell and the cache's cache-ID cell. Inside a memory cache, every operation in the memOpList is executed one by one. If the operation is a write, its value is split into bytes and written to the right places in the byteMap cell. If the operation is a read, it reads the bytes in the byteMap cell and sends a message containing the bytes back to the readBack cell in the thread. The readBack cell further manages these bytes, merges them into values and pushes the values to registers. The byteMap cell is a map from a memory byte address to its content. Each byte's content is represented as a list of bits. Certainly, **K-LLVM** memory caches are not just maps from memory locations to values. We create special items called memory objects to control the information for a range of memory caches. For a memory object in a memory cache (Fig 6b), there is a sub-cell named range pointing out the range of memory cache in the byteMap cell that the object is controlling. An object also contains other information. The alignment cell has the memory address alignment information, the size cell contains the size of the memory range, and the objType cell has information telling if the memory range is constant or not. The complete and race cells are used to record the status of the operations occupying the memory range. In LLVM IR, non-atomic memory operations are supposed to access a memory range one byte at a time. At the time when a non-atomic operation is accessing the memory range, if there is another memory operation accessing the range, a race happens, and the result for the memory operation should be undef. The complete and race cells are use to record this status and give undef results if races happen.

We now introduce the consensus model among the memory caches. Its purpose is to simulate the full behavior of the memory operations with the seq_cst ordering. Mostly, we need this model to

show behavioral difference between the `seq_cst` and `acquire/release` orderings. The different behaviors between these two kinds of orderings can be well understood in Batty et al.'s work [Batty et al. 2011]. The consensus protocol is based on selecting a central leader among the caches. We have a cache acting as the leader (the `leaderCache` cell) and no thread in our memory model can send memory operations to it. Its main job is to receive messages from a cache, send messages to all other caches and notify them that there is a change in the `byteMap` cell, wait for acknowledgments from them stating that they have executed the change, and then send a final acknowledgment to the original cache. When a cache receives an atomic operation with the `seq_cst` ordering, it sends a message to the leader and blocks any memory operation execution until it receives an acknowledgment from the leader. The acknowledgment means that all other caches in the system have reached a consensus. Otherwise, a cache sends a message to the leader only if it contains a write operation, and the cache does not wait for the acknowledgment of the message in this case. To implement the consensus protocol, **K-LLVM** has `messages` and `acks` cells for each cache. They are both implemented as lists. Once a cache deals with a memory write operation or `seq_cst` ordering operation, it places a message in the `messages` cell. The `acks` cell is to hold all acknowledgments from the leader cache and the cache will deal with the acknowledgments one by one. In short, the consensus model synchronizes the statuses of all memory caches.

Our **K-LLVM** memory model, not including the `unordered` ordering of the atomic memory operators, is basically a C++ memory model. In evaluating it, we need to target the unavoidable and infamous out-of-thin-air problem. After reading a lot of previous work on the definitions of out-of-thin-air problems [Kang et al. 2017; Lahav et al. 2017; Nienhuis et al. 2016; Pichon-Pharabod and Sewell 2016], we decided to make our own version which deals more directly with trace behaviors and different cells and devices as follows:

Definition 5.1. No Out-Of-Thin-Air Condition. We define observable memory operations to be those that have been sent by a thread and are residing in the `tempMem` cell or in a memory cache. Then, the No Out-Of-Thin-Air Condition means that when executing an LLVM IR program, not a single thread through any possible trace outputs an observable memory operation that is in the speculative stage.

The reason why the above definition satisfies the traditional "no out-of-thin-air condition" is that all **K-LLVM** memory operations except those with the `unordered` ordering interact with memory values in the memory caches. We guarantee not to have memory operations in the speculative stage showing up in the memory cache, so that we do not read or write them until they do happen. The only possible source of memory operation reordering is an out of order execution, but our availability and readiness checks can avoid out-of-thin-air problems in such executions. By this definition, our **K-LLVM** actually satisfies the theorem below.

Theorem 5.2. *When running any program that has no* `unordered` *ordering in its memory operations,* ***K-LLVM*** *satisfy the No Out-Of-Thin-Air Condition.*

Dealing with Unordered Memory Ordering. As we discussed in the example in Figure 2, the `unordered` ordering in memory operations is used to describe the weak happens-before memory model. According to the LLVM documentation, the `unordered` ordering is only used for atomic writes and reads and describes the Java shared-memory model. It is very weak because when a Java program is compiled to LLVM IR, it involves a lot of extra generated codes to guarantee the conservatism of the Java memory model. In defining **K-LLVM**, we cannot assume the way of compilation from Java to LLVM IR, but we do assume that those `unordered` memory operations will be used in any form which future compilers may design. Our **K-LLVM** implementations of the `unordered` memory ordering has a restriction: we do not consider cases in which memory

locations of pointers overlap but are not identical. Throughout our test cases, it is hard to find a counter-example where we need to consider the case of two `unordered` memory operations with pointer values whose memory locations overlap each other. The reason behind this is that LLVM IR is a lower-level language. Non-atomic memory operations are based on the byte operations, while atomic memory operations are only allowed to deal with an integer, floating-point or pointer value. Hence, we believe that our implementation is enough for understanding the behaviors of the `unordered` memory operations. The implementation of **K-LLVM** on memory operations without the `unordered` ordering takes the cases of memory overlaps into account.

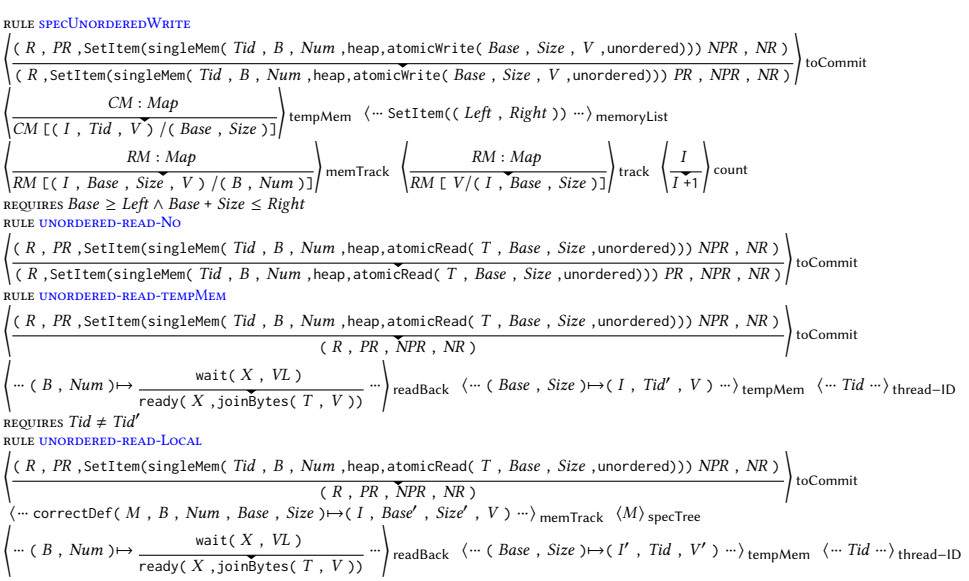


Fig. 7. Selected Unordered Rules

We have seen that the `unordered` and `monotonic` (was called `relexed` in C++) atomics have different behaviors, but what exactly causes the difference? We have seen the No-Out-Of-Thin-Air condition talking about memory operation behaviors without `unordered` atomics above. As a contrast, the `unordered` operations are weak enough to see executions of memory operations in the speculative stage, which has out-of-thin-air behaviors. The general way of materializing the behaviors of `unordered` operations is to use a special cell named `tempMem` in the main memory (Figure 6b). the `tempMem` cell is a special memory piece to store memory values for `unordered` memory operations that may be in the speculative stage. Recall that we have the *NPR* and *PR* sets in the `toCommit` cell storing memory operations in the speculative stage. Once an `unordered` atomic operation is in set *NPR*, its thread can write or read memory information by communicating with the `tempMem` cell. Then, the operation is moved to set *PR* to wait for the thread to move them to the set *R*. This scenario is not enough to guarantee the correctness of the `unordered` atomics. Certainly, the `unordered` atomics contain out-of-thin-air behaviors that are viewed as not correct traditionally. The correctness here means that the behaviors of the atomics respect the happens-before relation. To guarantee this, we need to take care of three problems. First, when we execute an `unordered` operation to the `tempMem` cell,

First, the `tempMem` cell is universal to all threads, so we need to make sure `unordered` read atomics in a thread is not going to read a value from the same thread but in different branching

cases. Hence, the mapping values in the tempMem cell are associated with thread IDs to distinguish threads. Second, we need to a unordered read operation is reading the correct value that is written by a write operation directly sequenced before the read. To guarantee this, we use the memTrack cell .

In designing the semantics for the unordered memory operations, we discovered that if there are no additional devices (temporary memory store), a well defined semantics has no way to output out-of-thin-air behaviors, which is the key behavioral difference between the unordered and monotonic memory operations. The temporary memory store in **K-LLVM** contains two sets and four cells. The sets are the *NPR* and *PR* sets in the toCommit cell, and the cells are the tempMem, memTrack, track and count cells. Sample rules are listed in Figure 7. Rule specUnorderedWrite describes when and how we move the values of an unordered atomic write to the tempMem cell. When an unordered atomic write is in the *NPR* set, which means that the write is in the speculative stage, we assign the value it carries to the tuple of the memory base (variable *Base*) and size (variable *Size*) in tempMem, and then move the write operator to the *PR* set, so eventually the operator will affect a memory cache. In doing the assignment, we also associate the value with a global counter value (variable *I* from the count cell) that allows the assignment to be distinguished from other ones, and the thread ID of the current thread. We also place corresponding assignments in the memTrack and track cells. The memTrack cell is a local control unit in a thread, and allows unordered atomic reads from the same thread to access values, and it also allows retrievals of the assignments from previous unordered atomic writes to be put into the global tempMem cell. When a branching operator happens, some executing blocks in the speculative stage might be discovered to be not token, so we need to throw out everything the blocks did. The memTrack cell helps throw out assignments that have been committed to the global tempMem or track cells. The track cell also affects the retrieval process. When we throw out assignments in the tempMem cell, we need to know what the old value for the key (a *Base* and *Size* pair) of an assignment is. We rely on the track cell, and find the largest *I* with the pair in the keys of the track cell, and access that value as the retrieval value for the tempMem cell.

Rule unordered-read-tempMem describes the situation of an unordered atomic read happening in the *NPR* set; we get the value for the *Base* and *Size* pair from the tempMem cell if the value is not from the same thread as the atomic read. Then we move the value to the readBack cell. The readBack cell is a device in a thread. Once an LLVM IR load instruction (non-atomic or atomic) is in the CPU cell, we put a read operator in the toCommit cell and also place an assignment for the instruction pair to a construct wait containing a pair of the needed assignment variable (variable *X*) and an empty byte list. A non-atomic read reads a byte at a time and places the byte in the byte list for the corresponding instruction pair. An atomic read reads a list of bytes from a target place (like the tempMem cell) and uses the function joinBytes to merge the list into a value directed by the defined type (variable *T*). The LLVM documentation suggests that if the size of type for a read is bigger than the bits or bytes actually read, the output is an undefined value (undef), so the function joinBytes also adjusts the problem of a size of type and read-in bytes mismatch. The rule unordered-read-Local describes the case when an unordered atomic read reads from values in the same thread as the memTrack cell. The function correctDef gets an instruction pair that represents the instruction which is the write operator that defines the value for the atomic read based on specTree. The instruction pair might not exist as a key in the memTrack cell. In fact, only an unordered atomic write is able to modify content in the memTrack cell, so a following unordered atomic read might not find its value in the memTrack cell. In this case, the unordered-read-No rule fires, which move the read operator to the *PR* set. The three rules (unordered-read-tempMem, unordered-read-Local and unordered-read-No) happen non-deterministically. In fact, the rules unordered-read-tempMem and unordered-read-Local have

another versions in which the `unordered` atomic read operators start in the R set, so an `unordered` atomic read can also read data from the temporary memory store when they are not in the speculative stage. The distinction between the UNORDERED-READ-TEMPMEM and UNORDERED-READ-LOCAL rules helps us achieve the property (2) in Theorem 5.3.

We show a theorem about our `unordered` memory operations below based on their design. The proof outline is listed in the supplement.

THEOREM 5.3. *When running any program with* `unordered` *ordering memory operations,* ***K-LLVM*** *satisfies the following:*

(1) *A trace behavior can contain an out-of-thin-air problem.*
(2) *A single threaded program can never directly read a value from a write in the same thread that is in the speculative stage and not a valid definition for the read.*
(3) *Observable memory operations in the speculative stage can be directly read only through* `unordered` *memory operations.*
(4) *All allowed behaviors of the* ***K-LLVM*** *memory model are allowed in happens-before memory model.*

6 EVALUATION AND APPLICATIONS

Evaluating **K-LLVM** took more than half of the development time. We used $\mathbb{K}$ to generate the interpreter for **K-LLVM** and ran LLVM IR programs on it. The testing mainly focused on executing single threaded programs and also a few multi-threaded programs. We mainly used the testing process as a tool to guarantee our semantics, comprised of individual instruction semantics, and our execution and memory models. We also developed a state space searching tool to show how useful **K-LLVM** can be in exploring multi-threaded behaviors.

6.1 Testing Process of K-LLVM

The validation of language semantics is usually accomplished through the use of external test suites [Bodin et al. 2014; Ellison and Rosu 2012; Filaretti and Maffeis 2014], which was also our strategy. We executed a set of 385 unit testing programs, and **K-LLVM** successfully ran them all. We also used the LLVM test suite for LLVM 6.0.0, which contains 23,000 LLVM IR programs as a regression test suite and **K-LLVM** passed 90% of them.

Our methodology for developing **K-LLVM** was based on a strategy from Test Driven Development (TDD), whose basic idea is to develop tests before implementing the actual feature. LLVM IR has an official test suite, but it is hard to break down the test suite into individual pieces. In developing **K-LLVM**, what we needed was a principle to test individual features while coordinating new features with old defined ones. When we define a new feature in **K-LLVM**, we follow the following steps. We first read the details about the feature in the LLVM documentation, and thinking about how to define the static and dynamic semantics of it; then we write out test cases to test the feature in the current LLVM implementation (Clang/Clang++). We make sure that we cover enough corner cases by designing a good set of new unit tests. We then define the feature and test it with the new unit tests, making sure it can pass them all. Third, we add the new feature to all of the defined unit tests to see if it causes any new problems. Finally, we test the whole thing with the regression LLVM test suite and make sure that it passes more test cases than before and does 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 guaranteed for every variable, expression, instruction, function and module. After that, we define the **K-LLVM** execution model according to those that appeared in the traditional literature, like the Tomasulo algorithm [Tomasulo

1967]. Then, we defined those instructions that interact least with other instructions such as arithmetic and conversion instructions. We then defined other instructions that would affect the design of the execution model like branching and exception handling instructions. It took us a lot of time to define the branching instructions due to the design modifications of the execution model. After all these instructions that do not affect the memory had been defined, we started defining the **K-LLVM** memory model and memory instructions. Finally, we came up with the **K-LLVM** semantics as they are today.

We did not spend a lot of time searching for undesirable behaviors in Clang, but we found some in the process of defining the **K-LLVM** semantics. Mainly, we ran test programs, and compared their output with that listed in the LLVM documentation. Undesirable behaviors happened in very diverse circumstances. A large number of them relate to the fact that Clang does not place enough checks to guarantee what the LLVM documentation suggests. For example, LLVM IR claims that if the input of a floating point constant cannot be written precisely in IEEE754 format, this is an incorrect input; but we found that Clang actually allows a `1.1` to be a `double` value, which is problematic because `1.1` can never be expressed by an IEEE754 `double` value precisely. 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 type, and allows a comparison of two floating point types. However, it does not give a precise list for which the floating point type is larger than the other kind. 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.

6.2 State Space Exploration

A trivial utility of **K-LLVM** is state space exploration. $\mathbb{K}$ allows us to generate a **K-LLVM** interpreter and see the trace behaviors of an executing program. Users can use `krun` to execute a program and see the final result. By using the option `--search` in $\mathbb{K}$, users are able to see all possible executions and traces for a multi-threaded program. 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 in a multi-threaded program. For example, in **K-LLVM**, the `waitJoinThreads` cell is used to store the states when a thread is waiting to join its child threads. It should be empty ($\langle .Map \rangle_{\text{waitJoinThreads}}$) eventually. If not, the execution of the program ends up in deadlock. We can use the `--pattern` option with the $\langle .Map \rangle_{\text{waitJoinThreads}}$ pattern, to detect the trace of a program whose execution resulted in such a state. If so, we know that the program ends up in a deadlock. Using the two options shows that the program in Figure 2 has out-of-thin-air behavior, so we show that the first part of Theorem 5.3 is valid by using the state space exploration application.

7 CONCLUSION

In this paper, we proposed **K-LLVM**, which is a formal semantics of LLVM IR in $\mathbb{K}$. The first advantage of **K-LLVM** is its completeness. To our best knowledge, our **K-LLVM** is the most complete formal semantics of LLVM IR. We fully explore the static semantics and dynamic semantics of LLVM IR, as well as semantics of library functions. To validate the completeness, we ran 385 unit test programs and more than 23,000 regression test programs, of which **K-LLVM** successfully executed all 385 unit-test programs and 90% of the regression test programs. **K-LLVM** provide guidance and reference to future compiler developers on exactly what are an permissible behaviors in running an LLVM IR program. More importantly, **K-LLVM** provides a framework to allow other researchers to build their imperative language specifications with nondeterminism. The execution

and memory models are the center of the framework, which simulate a conceptual virtual machine that runs LLVM IR programs. It covers all multi-threaded behaviors of LLVM IR programs in a uniformed framework. It is also easy to be understood by users because its analogy of real computer devices by $\mathbb{K}$ cells. Based on our **K-LLVM** framework model, we provided theorems on the memory operations that distinguish the C++ and happens-before memory models, and we also found more than 20 bugs in the current LLVM implementation, Clang, although finding bugs was not our main focus.

The project is full of further possibilities. For example, we are working on two on-going studies of **K-LLVM**. First, we are trying to axiomatize the **K-LLVM** execution and memory models in Isabelle and relate them to other models. Second, we are defining a formal semantics of Haskell and trying to verify the correctness of the compiler from Haskell to LLVM IR.

REFERENCES

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Proceedings of the 22Nd International Conference on Computer Aided Verification (CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 258–272. https://doi.org/10.1007/978-3-642-14295-6_25

Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software (ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 1–17. http://dl.acm.org/citation.cfm?id=1987211.1987212

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library Abstraction for C/C++ Concurrency. *SIGPLAN Not.* 48, 1 (Jan. 2013), 235–248. https://doi.org/10.1145/2480359.2429099

Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC Atomics in C11 and OpenCL. *SIGPLAN Not.* 51, 1 (Jan. 2016), 634–648. https://doi.org/10.1145/2914770.2837637

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

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

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

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2618128.2618134

Denis Bogdănaş and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, 445–456. https://doi.org/10.1145/2676726.2676982

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

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

Mike Dodds, Mark Batty, and Alexey Gotsman. 2013. C/C++ Causal Cycles Confound Compositionality. *TinyToCS* 2 (2013).

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

Chucky Ellison and Grigore Rosu. 2012. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. ACM, 533–544. https://doi.org/10.1145/2103656.2103719

Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. 2004. *Formal Analysis of Java Programs in JavaFAN.* Springer Berlin Heidelberg, Berlin, Heidelberg, 501–505. https://doi.org/10.1007/978-3-540-27813-9_46

Daniele Filaretti and Sergio Maffeis. 2014. *An Executable Formal Semantics of PHP.* Springer Berlin Heidelberg, Berlin, Heidelberg, 567–592. https://doi.org/10.1007/978-3-662-44202-9_23

Yuri Gurevich. 1995. Specification and Validation Methods. Oxford University Press, Inc., New York, NY, USA, Chapter Evolving Algebras 1993: Lipari Guide, 9–36. http://dl.acm.org/citation.cfm?id=233976.233979

Myra Van Inwegen and Elsa L. Gunter. 1993. HOL-ML. In *Higher Order Logic Theorem Proving and its Applications, 6th International Workshop, HUG '93, Vancouver, BC, Canada, August 11-13, 1993, Proceedings*. 61–74. https://doi.org/10.1007/3-540-57826-9_125

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

Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*. ACM, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

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

Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *Proceedings, Part II, of the 42Nd International Colloquium on Automata, Languages, and Programming - Volume 9135 (ICALP 2015)*. Springer-Verlag, Berlin, Heidelberg, 311–323. https://doi.org/10.1007/978-3-662-47666-6_25

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

Chris Lattner and Vikram Adve. 2004. 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)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

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

llvm.org. 2018. LLVM Language Reference Manual. (2018). http://releases.llvm.org/6.0.0/docs/LangRef.html

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

Savi Maharaj and Elsa Gunter. 1994. *Studying the ML module system in HOL*. Springer Berlin Heidelberg, Berlin, Heidelberg, 346–361. https://doi.org/10.1007/3-540-58450-1_53

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. *SIGPLAN Not.* 40, 1 (Jan. 2005), 378–391. https://doi.org/10.1145/1047659.1040336

Paul E. Mckenney. 2009. Memory Barriers: a Hardware View for Software Hackers. (2009).

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

Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An Operational Semantics for C/C++11 Concurrency. *SIGPLAN Not.* 51, 10 (Oct. 2016), 111–128. https://doi.org/10.1145/3022671.2983997

Brian Norris and Brian Demsky. 2013. CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics. *SIGPLAN Not.* 48, 10 (Oct. 2013), 131–150. https://doi.org/10.1145/2544173.2509514

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

Marc Pérache, Hervé Jourdren, and Raymond Namyst. 2008. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Euro-Par 2008 – Parallel Processing*, Emilio Luque, Tomàs Margalef, and Domingo Benítez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 78–88.

Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. *SIGPLAN Not.* 51, 1 (Jan. 2016), 622–633. https://doi.org/10.1145/2914770.2837616

Zvonimir Rakamarić and Alan J. Hu. 2009. A Scalable Memory Model for Low-Level Code. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*. Springer-Verlag, Berlin, Heidelberg, 290–304. https://doi.org/10.1007/978-3-540-93900-9_24

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

Jaroslav Sevcik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-memory concurrency and verified compilation. In *POPL*. Austin, TX, United States. https://hal.inria.fr/hal-00907801

Bjarne Stroustrup. 2013. *The C++ Programming Language* (4th ed.). Addison-Wesley Professional.

Bjarne Stroustrup. 2014. The Essence of C++. (2014). https://www.youtube.com/watch?v=86xWVb4XIyE

Suvinay Subramanian, Mark C. Jeffrey, Maleen Abeydeera, Hyun Ryong Lee, Victor A. Ying, Joel Emer, and Daniel Sanchez. 2017. Fractal: An Execution Model for Fine-Grain Nested Speculative Parallelism. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 587–599. https://doi.org/10.1145/3140659.3080218

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

Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. 2008. Copy or Discard Execution Model for Speculative Parallelization on Multicores. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 330–341. https://doi.org/10.1109/MICRO.2008.4771802

R. M. Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM J. Res. Dev.* 11, 1 (Jan. 1967), 25–33. https://doi.org/10.1147/rd.111.0025

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. *SIGPLAN Not.* 49, 10 (Oct. 2014), 691–707. https://doi.org/10.1145/2714064.2660243

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. *SIGPLAN Not.* 50, 1 (Jan. 2015), 209–220. https://doi.org/10.1145/2775051.2676995

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A Program Logic for C11 Concurrency. *SIGPLAN Not.* 48, 10 (Oct. 2013), 867–884. https://doi.org/10.1145/2544173.2509532

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

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

Valentin Ziegler. 2015. The C++ Memory Model. (2015). https://www.youtube.com/watch?v=gpsz8sc6mNU