K-LLVM Memory Model and Proofs of Theorems

ANONYMOUS AUTHOR(S)

This document provides extra information about the full LLVM IR concurrency model. This is one of the future work of **K-LLVM**. All theories about the full concurrency model is under construction. The proof about the type system theorem in Section 4 is shown in the document.

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

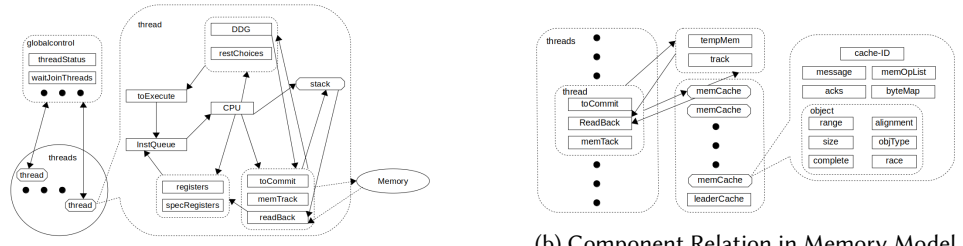
1 DYNAMIC SEMANTICS OF K-LLVM

As we have discussed, to define the complete behavior of the LLVM IR language, we are basically simulating a conceptual virtual machine that runs the LLVM IR codes. In Figure 1, the conceptual machine includes execution machines (models) that select a **K-LLVM** BAST instruction to execute, returning a result that is assigned to a local variable as the target register; and the conceptual machine has a memory machine (model) that interacts with memory operations from different threads represented by the execution machines. The directions of arrows in the figure represent the target force of a cell. In this section, we will describe the formation of the execution model and memory model in detail.

1.1 The Execution Model

The **K-LLVM** execution model in Figure 1a directs how a single operation is selected and executed and how the result is returned. All of the rectangles in Figure 1a relate to a cell in the **K-LLVM** configuration. All these rules are attributed with `transition`, so they can be executed nondeterministically. We start by assuming that an executing block has been selected and put in the `toExecute` cell. When a program begins to be executed, the entry block is selected and put in the `toExecute` cell. The rule that interacts the `toExecute` and `instQueue` cells is listed as the toExecute-out rule in Figure 2. In the `toExecute`, the content is a term with constructor `blockExecution` with two arguments, and the first one is an integer referring to the executing block number and the second one is a list of instructions for the block in the program order. Each new executing block that is injected into the cell is given a unique dynamically generated number that helps distinguish it from all other executing blocks. Each element in the list is a compiled instruction term whose constructor is `instNumInfo` and has three arguments: an integer (*Num*) representing the instruction position number, a term (*In*) representing the content of the actually instruction and a flag (*T*) representing the type of the instruction. Each instruction in the executing block is sent to the `instQueue` cell in the numerical order of *Num* by wrapping with another construct `dynInstInfo` with an additional executing block number *B*. We refer the executing block number *B* an the position number *Num* together as the instruction number which not only allows the instruction to be distinguished from all other instructions in the `instQueue` cell, but also the new input instruction to be identified as the latest one in the cell. `updateVars` is a function to update local variables in *In* with values. In **K-LLVM**, we have two register cells `registers` and `specRegisters`, whose values are mapped by local variables from instructions in a normal stage or speculative stage, respectively. In `specRegisters`, its keys are tuples of an executing block number and a local variable, so a local variable can be assigned differently in different executing blocks. The `updateVars` functions relies on the content *Tr* from the `specTree` to determine which value is

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



(a) Component Relation in Execution Model

(b) Component Relation in Memory Model

Fig. 1. Relations Among Model Components

the correct one to be assigned to a local variable. The condition of the toExecute-out rule shows that there is a limit of maximum instructions `maxNum` that the `instQueue` cell can take at a time.

The main task of the `instQueue` cell is to select an instruction to put in the cpu for execution once the cpu is empty (represented by `.K`). There are two rules (rule instQueue-out and instQueue-heavy-out) to select an instruction: first(instQueue-heavy-out rule), if an instruction is a function call, branching or return instruction (checked by the `isHeavyInst` function), it can be selected if and only if its instruction number is the oldest one in the `instQueue` cell (checked by the `isSmallest` function); second (instQueue-out rule), if an instruction is not one of these three, it can be selected if and only if all its arguments are constant values (no local variables) (checked by the `isAvailable` function). Fulfilling one of these two rules means that an instruction is available. The reason that we want to have the selection rules is that we want to simulate the speculative execution behavior implicit inside the LLVM IR language; and we do not specify a strategy for the speculative execution, but randomly guess a block to execute if we face a branching situation. The first rule says that we do not want to move a function call, branching or return operation ahead for execution, because it is hard to believe that any modern computer moves these instructions ahead for execution. The second rule gives **K-LLVM** the power to randomly select an instruction to execute out of its program order, and it can be out-or-order execution or speculative execution.

As we see in the rule toExecute-out in Figure 2, the job of the `CPU` cell is to push the instruction (*In*) to the `k` cell for evaluation. In Section **??**, we show that every instruction is either an `assign` term or a `noAssign` depending if the instruction returns value. the syntactic definitions of the constructs `assign` and `noAssign` are attributed as `strict` and number to indicate the non-terminal positions that the attribute is applied on. The `strict` means that for a given non-terminal position in a construct, a pair of `heat/cool` rules is created. A `heat` rule means that if the head position of the `k` cell has the target construct term and the subterm in the non-terminal position of the term has no sort *KResult* (a special $\mathbb{K}$ sort indicating the evaluation results), the subterm in the specific position of it is replaced by a $\square$ and the subterm is put on the `k` cell head position. A `cool` rule means that if the head position term in the `k` cell has sort *KResult*, and the second position term has a $\square$ in one of its subterm position, we merge the head position term back to the $\square$. Because of the semantic meaning of the `strict` attribute, one can expect subterms of `assign` and `noAssign` terms are pulled out and evaluated to *KResult* terms by the semantic rule of the instruction and the results will be pushed back to the subterm positions in those terms. Then, CPU-noAssign, CPU-assign and CPU-assign-spec rules take place. The CPU-noAssign rule means that the instruction does not return values so the `CPU` cell just clean up the content and wait for the next instruction. The CPU-assign and CPU-assign-spec rules describe the situation when the instruction returns values. The difference between them is if the instruction is in the

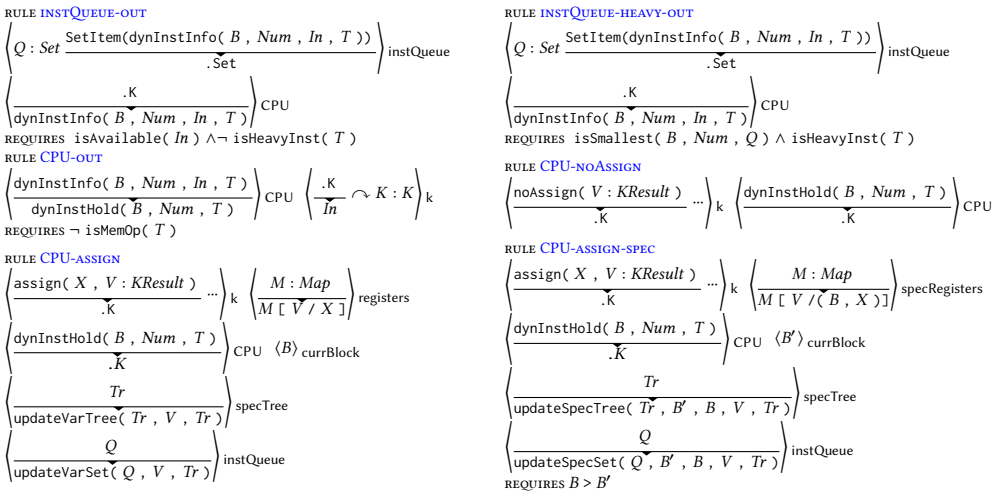


Fig. 2. Selected Rules for **K-LLVM** Execution Model

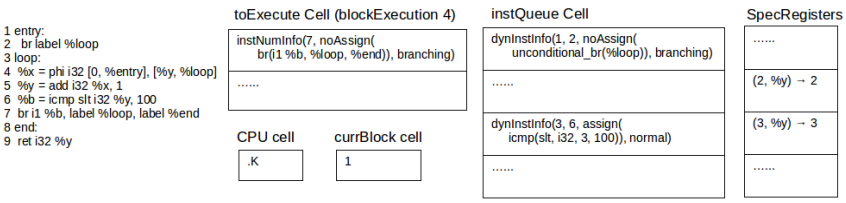


Fig. 3. SpecTree Example

speculative stage. In rule CPU-ASSIGN, if the executing block number (B) is equal to the current block number (B') in the `currBlock` cell, the instruction is not in the speculative stage so that the final result is assigned to X in the `registers` cell; otherwise, if B is greater than B', which means that the current execution is executing an instruction that is not in the current block, the value is assigned to a tuple of variable X and the executing block number B in the cell `specRegisters`, as shown in the rule CPU-ASSIGN-SPEC. Two things need to be updated once registers change: the content in the `specTree` cell and the instructions in `instQueue`. We need to update values for variables in the memory operation prototypes in the `specTree` cell that determines which memory operators become ready for commitment and we need to update local variables in each instruction in `instQueue` with the new arrival value. The functions `updateVarTree`, `updateVarSet`, `updateSpecTree` and `updateSpecSet` are to update the value for the local variable X in a set or a `specTree`.

Different semantic rules for different instructions in the `k` cell can affect the interaction with different components in the execution model. For example, if the instruction is a function call, the `CPU` cell needs to interact with the stack. If the instruction is a heap memory operation, the `CPU` cell sends the instruction to the `toCommit` cell where it is dealt with. Finally, if the instruction is a branching operation, the `CPU` cell also needs to interact with the speculative information cells (the `specTree` and the `restChoices` cells) by updating information in them.

The specTree and restChoices cells are the speculative information cells, which contain information for performing speculative executions. The type of content in the specTree cell is a map from an executing block number to a construct RunningBlock. RunningBlock has five arguments: the executing block's original basic block label name, the parent executing block number (the entry block's parent is labeled as none), 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 tasks 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 lives. 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, the specTree cell also needs to contain enough information for **K-LLVM** to recognize which map entry in the specRegisters should be used for updating a particular local variable. In Figure 3, an example on how specTree can be used to calculate the correct assignment for a local variable. The left in the figure is an LLVM IR program, and executing this program in **K-LLVM** will require 100 creation of executing blocks for the basic block %loop. The right shows a graph on a over simplified **K-LLVM** configuration including the cells toExecute, CPU, instQueue and specRegisters (we only show the list of the blockExecution construct and put the executing block number on the top). We assume that the No.2, No.3 and No.4 executing blocks have block label name %loop. From this program state piece, we can see that **K-LLVM** has not yet executed the branching operation br label %loop, because its compiled **K-LLVM** AST is still in the first position of the instQueue cell. The addition operations for the No.2 and No.3 executing blocks have been speculatively executed, since there are two entries in the specRegisters ((2, %y) ⟶ 2 and (3, %y) ⟶ 3) indicating that. The system is about to move the phi instruction from the No.4 executing block to the instQueue cell, since the compiled **K-LLVM** AST of the phi instruction is in the first position of the toExecute cell. At this point, the arguments of the instruction need to be updated before the instruction can be moved to instQueue, and the local variable %y has two instances in specRegisters. (2, %y) (coming from the No.2 executing block) maps to the value 2, while (3, %y) (coming from the No.3 executing block) maps to value 3. To determine which %y the system should pick to update the one in the phi instruction, we need the information from the set of defined local variables in the RunningBlock of an entry in specTree. Since the LLVM IR program is required to be in SSA form, a local variable can only be defined once it is in an executing block. Hence, there are only two situations in which %y occurs here: (1) either it has been defined inside the executing block, which can be determined by seeing if there is a definition of %y in the block and comparing the instruction position numbers between its use and definition; or (2) the correct instance for the user of %y is that of the latest preceding executing block containing a definition of %y. Third, specTree also has the task determining a memory operator in the toCommit cell ready to be sent to the memory for execution, which will be discussed later.

The restChoices cell is used to store the remaining choices for a speculative guess. When the toExecute cell finishes putting all of the instructions of a block in instQueue, if the final instruction is a branching operator, it will guess a branch and place its executing block in toExecute. There might be remaining branches that are not selected. They will be contained in restChoices. **K-LLVM** allows the toExecute cell to randomly select a speculative guess either from the next possible choice or from one of the remaining branches in the restChoices cell.

The registers include two different cells, the registers and specRegisters cells. The difference between them is the time when the CPU finishes computing the value for an instruction and assigning the value to the defined local variable associated with it. In **K-LLVM**, we have a global metavariable to indicate the current executing block, and only executing a branching operator can

change its value. As computing is finishing, if the instruction's executing block number is not the current block number, then the value will be assigned to a tuple of the executing block number and the defined local variable, and put in the `specRegisters` cell. If the executing block number is the current block number, we create a map entry in the `registers` to map the defined local variable to the computed value.

The `toCommit` and `readBack` cells are used to deal with heap memory operators in **K-LLVM**. Their main functionality is to determine when a memory operator is sent to the main memory to perform an action. `toCommit` receives memory operators from the `CPU`, while `readBack` waits for the main memory to send back values for the load operators, and then pushes them into the registers. Since **K-LLVM** operators may be executed out of order, we cannot expect the heap memory operators to be put into the `toCommit` in program sequence order. It relies on information from `specTree` to determine when to commit an operator to the main memory. We will explain this in detail in Section 1.2. The `stack` cell implements the stack structure of **K-LLVM**. It not only stores the call stack for function call information, but also manages the stack memory operators since LLVM IR specifically states that the `alloca` operators are creating memory pieces in the stack. The structure of the `stack` cell and the semantics of the stack memory operations are similar to the heap ones, except that we define a global variable simulating the fixed stack size. Once a stack is out of this fixed size, **K-LLVM** stalls and gives an error message stating that the stack is out of bound.

In this section, we have introduced components of our **K-LLVM** execution model and the relations between different components. Through the execution model, we see how **K-LLVM** provides a way to describe the semantics of the executing LLVM IR program instructions out of order and speculatively, while guaranteeing the correctness of the program execution. We will examine in the memory model in the next section and provide examples to how an instruction is run based on the execution model and memory model.

1.2 The Memory Model

The **K-LLVM** operational memory model is similar to a message passing model. Its components and relations are described in Figure 1b. All of the components in the figure are cells in the **K-LLVM** configuration that represents some program state pieces. In the graph, a rounded 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.

The left column contains a set of threads trying to communicate with the main memory. Each thread follows the execution model defined previously, and they contain several different cells. In terms of interaction with the main memory, only three cells join the communication with memory channels: `channel-ID`, `toCommit` and `readback`. The middle column represents the main memory. The main memory contains a cell named `memoryList` comprised of the set of memory ranges that have been allocated for use at a time. The main memory also contains a set of memory channels, each of which has a structure similar to the right column of the graph.

Each thread has an assigned channel ID when it is created. The job of the `toCommit` cell of each thread is to manage the ordering of the memory operators sent to the main memory. The `toCommit` cell's content is a tuple of four sets: the first set (*R*) containing all memory messages that are ready to commit to the memory channel; the second set (*PR*) containing all memory messages that are ready but in the speculative stage and have been checked and dealt with the `unordered` ordering memory messages, we can mark these memory messages being partially ready; the third set the fourth set (*NPR*) representing all partially ready memory messages that have not yet been checked if they have `unordered` orderings so that they need special treatment; the fourth set (*NR*) containing all memory messages that are not ready. The *PR* and*NPR* sets are used to help memory opeartors

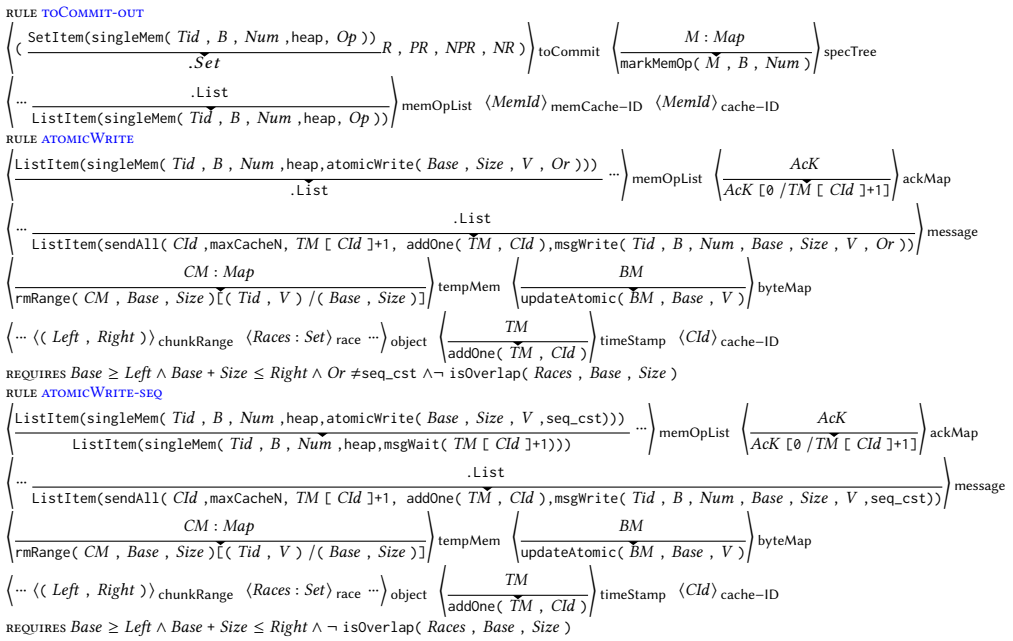


Fig. 4. Selected Rules for **K-LLVM** Memory Model

happening in speculative stage with unordered ordering and it will be described later this section. Each memory message is a singleMem construct that contains five arguments: an identifier for the thread ID of the thread the toCommit cell resides, the executing block number of the memory operator, the instruction position number, a flag indicating if the memory message is a heap or stack one and a memory operator. Memory operators are transformed from LLVM IR memory operators to one of seven: a non-atomic write (writeByte), atomic write (atomicWrite), non-atomic read (readByte), atomic read (atomicRead), atomic read write (atomicReadWrite), seq_cst ordering fence (seqFence) or a memory free (toClose) operator. In **K-LLVM**, all fences except the seq_cst one only has effects inside a single thread, and we implement them by encoding them into specTree. For simplicity, **K-LLVM** assumes all memory generation operators (e.g., malloc or alloca) happen right at the moment when the CPU cells move the operators to the k cell, and they do not go through the memory devices defined in this section, while a memory free operator will take in effects in the memory model.

The TOCOMMIT-OUT rule in Figure 4 describes how we move a memory message to the the memOpList in the memory channel. It relies on the matching between content of the the memChannel-ID and channel-ID cells to help locating the correct memory channel for the thread. The TOCOMMIT-READY and TOCOMMIT-PAR-READY rules talk about how to determine if a memory message can be moved to the *R* or *NPR* sets depending if the memory message is in the speculative stage (by checking the executing block number *B* with the current block number in the currBlock cell). The way to check that is through a special function named isReady that has six arguments: the current block number, the execution block number of the memory message, the instruction position number, the heap /stack flag, the memory ordering and the specTree in the specTree cell. From the previous section, we know that the value of specTree has a list field containing information about all memory operators in an executing block. Each item in the list is implemented as a construct

named memProtoType. It has six arguments, an instruction number, a memory location expression, a field indicating the type of its memory prototype (either a read, a write, a readWrite such as cmpxchg or atomicrmw, or a fence), a field having the memory ordering, a Boolean value indicating if the operator is volatile and a Boolean value indicating if the corresponding memory operator has been committed. We do not show how the isReady function utilizes specTree and memProtoTypes. The main idea of the function is to check if a memory message is ready to be committed to memory channels. It compares the memory pointer address (p) of the message to check (m) with all other memory message prototypes (Φ) that are sequence before this memory message in the memProtoType fields of specTree. There are three different properties to compare. First, we check if there is a memory message m' in Φ, such that p and the memory pointer address p' of m' overlaps. If there is a memory message prototype that is sequenced before m but has not yet known the pointer address, we assume that it overlaps with m. The second property relates to memory orderings. Basically, we are comparing the orderings of m with the orderings in each prototype in Φ to determine if the operator is ready at this point. For example, if m and all prototypes in Φ have unordered, and monotonic (the LLVM IR version of relaxed atomic ordering), m is ready if the first property was satisfied. If m is a write operator, and a memory prototype m' in Φ is a write with release ordering or read with acquire ordering, the operator is not ready. If m is a read operator, and the memory prototype m' in Φ is a write with release ordering, the operator is not ready; however, if every m' in Φ is a read with acquire ordering, m is ready. The third property guarantees that if m is marked with a volatile key word, it is not ready if a prototype m' in Φ is also marked as volatile. By comparing m with all memory prototypes in Φ, we can determine if it is ready to be committed, then we can move it to the R set or NPR set by comparing its executing block number with the current block number.

The idea of memory channels in Figure 1b is basically that we have different memory cashes to manage requests from different cores or threads. A single memory address might have different values in different memory cashes. A thread only talks to one memory channel in its lifetime. The idea of memory channels is a compromise between theoretical concerns and real world usage. Theoretically, we want to be able to observe the difference between memory operators with seq_cst ordering and acquire/release ordering. Consider the two LLVM IR program fragments in Figure 5. We assume the addresses of the pointer variables $\%x$ and $\%y$ have the value zero at first. The difference between these two systems is that the variables $\%a$, $\%b$, $\%c$ and $\%d$ can have values 1, 0, 1 and 0 in the left system, respectively; while the right system never shows this group of results, because seq_cst ordering requires total order across all threads. If **K-LLVM** implemented the main memory with only one channel talking to all threads, we would never be able to see the difference between these two systems, because every memory operator put in the main memory would already be in order. That is why we want to have more than one channel in the main memory. The practical side is that we can simulate a conceptual machine with multiple cores and multiple cashes that executes LLVM IR codes through **K-LLVM**. The memory channels simulate different cashes for different cores. Implementing a memory model with different threads talking to different memory channels and the different channels interacting with each other is a better fit with the multi-cash memory storage environment.

The right column in Figure 1b represents the contents of a single memory channel. The channel-ID cell contains the identifier of the channel. The cell memOpList contains the list of the memory operators coming from threads. The channel performs the events defined by the operators in the order of the list. Each unit of memory location is a byte, so we have a cell named byteMap that maps from the memory location to the byte value in integer bit forms. The Object cell contains information about the specific chunk of memory spaces defined by the pointer value of a memory operator. Inside an Object cell, the range cell contains the range of a memory space. It is

```
Thread 1:
 store atomic i32 1, i32* %x release, align 1

Thread 2:
 store atomic i32 1, i32* %y release, align 1

Thread 3:
 %a = load atomic i32, i32* %x acquire, align 1
 %b = load atomic i32, i32* %y acquire, align 1

Thread 4:
 %c = load atomic i32, i32* %y acquire, align 1
 %d = load atomic i32, i32* %x acquire, align 1
```

(a) Example for acquire/release

```
Thread 1:
 store atomic i32 1, i32* %x seq_cst, align 1

Thread 2:
 store atomic i32 1, i32* %y seq_cst, align 1

Thread 3:
 %a = load atomic i32, i32* %x seq_cst, align 1
 %b = load atomic i32, i32* %y seq_cst, align 1

Thread 4:
 %c = load atomic i32, i32* %y seq_cst, align 1
 %d = load atomic i32, i32* %x seq_cst, align 1
```

(b) Example for seq_cst

Fig. 5. Comparison Between acquire/release and seq_cst Memory Ordering

implemented as two integer numbers in **K-LLVM**. The first one represents the base value of the memory space, while the second represents the bounds of the memory space. The `size` stores the number of bytes the memory location has. The `alignment` cell stores an integer value that represents the number of bytes that serve as alignment packing bytes before the memory chunk. The `objType` can be either `static` or `heap`, indicating if the memory chunk can be modified or not. The `complete` and `race` cells are used to simulate the data race behaviors in LLVM IR. The LLVM IR `read` and `write` operators both have non-atomic and atomic versions. The non-atomic `read` and `write` operators are performed one byte at a time. According to the LLVM document, while performing a non-atomic `read` or `write`, if another `read` or `write` happens in the middle of the process, and a race happens, then the return value for a `read` operator should be an `undef` value. To implement this feature in **K-LLVM**, we need a cell (`race`) to indicate that the non-atomic operator that was working on the memory chunk, if there is one; and the cell (`complete`) that indicates how many bytes the non-atomic operator have finished performing. Hence, if there is another operator coming in from another thread, the memory channel can detect the race immediately.

The `timeStamp`, `channelOps` and `acks` cells in a channel are used to communicate with other channels. The `timeStamp` cell contains the current vector timestamp that is implemented as a map and one entry per memory channel. Once a `write` or `readWrite` memory operation is performed in a channel, the channel will send out messages to notify all other channels of the changes, including the memory location, the new value, and the vector timestamp with updated values for the new memory operator, and channel ID. The message passing is assumed to be synchronous without failure for simplicity. The `channelOps` receives this kind of message from the other channels. Then it compares its own timestamp with that attached to the message; if its own is larger, then the channel ignores the message; if it is smaller, the channel updates the value of the memory location with the message as well as the timestamp. If the two timestamps cannot be compared, the channel will compare the channel ID with the one attached to the message to determine if it will perform the memory update. The difference between a memory operator with `seq_cst` ordering and all other kinds of memory operators is that it will wait for all of the acknowledgements to come back from its change-notification messages sending to different channels. `acks` maps from message IDs to the number of acknowledgements. Once a `seq_cst` ordering operator performs sending out messages to notify all of the other channels. It knows the total number of channels in the system, so it can wait for that number of acknowledgements to come back, then perform the memory operation in its own channel and then move to the next operator.

In **K-LLVM**, we have series of rules to perform behaviors of memory messages in memory channels. In Figure 4, we show two rules (atomicWrite and atomicWrite-seq) for describing committing atomic write operators. Rule atomicWrite describes the behavior when the memory ordering of the message is not `seq_cst`. When an atomic write is in the head of the list in the `memOpList` cell, we can remove the head, and add one for the current memory channel in the `timestamps` cell. The *TM* `[` *CId* `]` gets the value of channel ID *CId* in the map *TM*, which is the timestamps for the channel and adding one to it is the event number of the new message sending out to other channels. Function `addOne` gets a map and a key and returns a map with adding one to the value of the key. We add an entry with the event number being the key with a value `0` in the `ackMap` cell. It is used to indicate the number of acknowledgements getting back from other channels when we send out messages to other channels to notify a change. In the `byteMap` cell, we update a chunk of memory with a list of bytes *V* starting on the position *Base*. The number of bytes changed is defined by *Size*. In the `channelOps` cell, we send out messages to all other channels by using an operator `sendAll`. The message content notifies the other channels that there is a write updating the memory by using the construct `msgWrite`. The condition $Base \geq Left \wedge Base + Size \leq Right$ is used to locate the right memory chunk (a `object` cell) in the configuration. Each `object` cell has a cell `chunkRange` that stores the range of the memory chunk by using two integers (*Left* and *Right*). The condition locates the memory chunk by checking if the *Base* is within the range of the chunk. The main item that involves in an atomic write rule in a memory chunk is the `race` cell. It relies on the `isOverlap` function to check if there is a non-atomic memory operations that are occupying the memory chunk, which cause a race. The rule atomicWrite-seq is dealing with the case when the ordering of the memory message is `seq_cst`. The only difference between the rules atomicWrite-seq and atomicWrite is that the atomicWrite-seq rule puts the `memOpList` cell in a memory channel on hold by using an operator `msgWait` in the cell. The operator will wait for all acknowledgements coming back from other memory channels, then it allows the `memOpList` cell to execute other memory messages. The process of waiting for all acknowledgements is to synchronize the status of all memory channels, because when other channels receive messages `msgWrite`, they update their own `byteMap` cell with the new writes, and then send the acknowledgement back to the sender.

Our **K-LLVM** memory model not including the `unordered` ordering on atomic memory operators is basically a C++ memory model. In evaluating the memory model, we need to target the unavoidable infamous out-of-thin-air problem. A lot of previous work [Kang et al. 2017; Lahav et al. 2017; Nienhuis et al. 2016; Pichon-Pharabod and Sewell 2016] defines the problem as relations on memory events. For example, Lahav et al. defined the out-of-thin-air problem as having a cycle on a graph combining the sequenced-before relation and reads-from relation. **K-LLVM** is an operational semantics that simulates a virtual machine running LLVM IR programs, so we want to make the definition more direct on trace behaviors and different cells as follow:

Definition 1.1. No Out-Of-Thin-Air Condition. We define observable memory operations to be those that have been committed by a thread and are living in the `CDB` cell or in a memory channel. Then, The No Out-Of-Thin-Air Condition means that through all possible traces by executing a LLVM IR program, no a single thread 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 `unordered` ordering interact memory values in the memory channels. If we guarantee not to have memory operations that are in the speculative stage to show up in a memory channel; obviously, we will not read or write those values that may or may not happen in the future. The only possible source of memory operation reordering is the

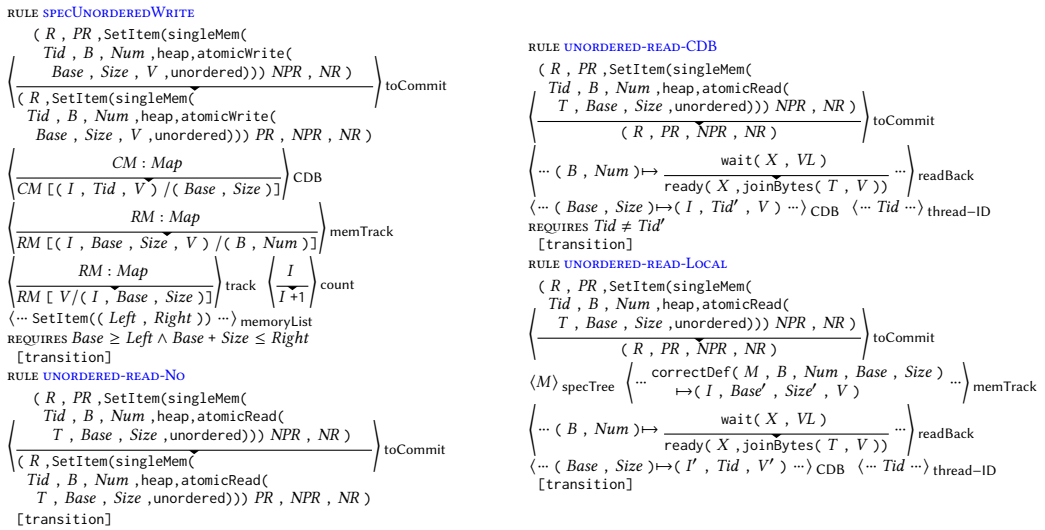


Fig. 6. Selected Unordered Rules

out of order execution. Due to the availability check on selecting instructions from `instQueue` to CPU, we guarantee all instructions become available before it is sent to execute, which also respect the modification order. By this definition, our **K-LLVM** actually satisfy the theorem below.

THEOREM 1.2. *Given any program that has no* `unordered` *ordering on memory operations, when running such program, **K-LLVM** satisfy the No Out-Of-Thin-Air Condition.*

Dealing with Unordered Memory Ordering. As we discussed in the example in Figure **??**, the `unordered` ordering in memory operations is used to describe the weak happens-before memory model. According to the LLVM document, the `unordered` ordering is only used for atomic write and read operators and is to describe 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, and we assume that those `unordered` memory operations are used in any form on which future compilers may design. Our **K-LLVM** implementations of `unordered` memory ordering has a restriction, where we do not consider the cases when memory locations being overlapped but not exactly the same. Through out our test cases, it is hardly to find a counter example where we need to consider the case when two `unordered` memory operations having pointer values whose memory locations are overlapped with each other. The reason behind this is that LLVM IR is a lower-level language. A non-atomic memory operations are based on the byte operations, while an atomic memory operations in LLVM IR can 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 `unordered` memory operations. The implementation of **K-LLVM** on memory operations without `unordered` orderings takes the cases of memory overlapped into account.

In designing the semantics for the `unordered` memory operations, we discovered that if there is no additional jump wires, a well defined semantics has no way to output the out-of-thin-air behaviors, which is the key behavioral difference between the `unordered` and `monotonic` memory

operations. The jump wires in **K-LLVM** are two sets and four cells. The sets are the *NPR* and *PR* sets in the `toCommit` cell, and the cell is the `CDB`, `memTrack`, `track` and `count` cells. The example rules are listed in Figure 6. Rule specUnorderedWrite describes when and how we move the values of a `unordered` atomic write to the `CDB` cell. When a `unordered` atomic write is in the *NPR* set, which means that the write is in the speculative stage, in the `toCommit` cell, we assign the value it carries to the tuple of the memory base (variable *Base*) and size (variable *Size*) in `CDB`, and then move the write operator to the *PR*, so eventually, the operator will have effect on a memory channel. 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 distinguish with 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 local to the thread, which allows `unordered` atomic reads from the same thread to access values, and it also allow retrievals of the assignments putting into the global `CDB` cell from some previous `unordered` atomic writes. When a branching operator happens, some executing blocks being in the speculative stage might be discovered to be not token, so we need to throw out all things the blocks did. The `memTrack` cell helps throw out assignments that have been committed to global `CDB` or `track` cells. The `track` cell is also taking effects in the retrieval process. When we throw out assignments in the `CDB` cell, we need to know what is the old value for the key (a *Base* and *Size* pair) of an assignment. 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 to be the retrieval value for the `CDB` cell.

Rule unordered-read-CDB describes the situation when an `unordered` atomic read happens in the *NPR* set, we get the value from the `CDB` cell for the *Base* and *Size* pair 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 in **K-LLVM**. Once an LLVM IR `load` operator (non-atomic or atomic ones) is in the `CPU` cell, we put a read operator in the `toCommit` cell and also place an assignment for assigning the instruction number (a block number and instruction position number pair) to a construct `wait` containing a pair of the return variable (variable *X*) and an empty byte list. A non-atomic read reads a byte at a time and place the byte in the byte list for the corresponding instruction number. An atomic read reads a list of bytes from a target place (like the `CDB` cell) and use a function `joinBytes` to merge the list into a value directed by the defined type (variable *T*). LLVM document suggests that if size of type for a read is bigger than the bits or bytes that actually read, the output is an `undef` value, so the function `joinBytes` also adjusts the problem of size of type and read-in bytes mismatch. Rule unordered-read-Local describes the case when an `unordered` atomic read reads from values in the same thread from the `memTrack` cell. The function `correctDef` gets an instruction number that represents the instruction which is a write operator that defines the value for the atomic read based on the `specTree`. The instruction number 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-CDB, unordered-read-Local and unordered-read-No) happen nondeterministically. In fact, the rules unordered-read-CDB and unordered-read-Local have another versions where the `unordered` atomic read operators start at the *R* set, so an `unordered` atomic read can also read data from the jump wires when they are not in the speculative stage. The distinction between the unordered-read-CDB and unordered-read-Local rules help us achieve the property (2) in theorem 1.3.

Based on the design of our `unordered` memory operations, we want to prove a theorem about our `unordered` memory operations below.

Theorem 1.3. *Given any program that has* `unordered` *ordering on memory operations, when running such program,* ***K-LLVM*** *satisfy the following:*

(1) *A trace behavior of running such program can contain 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 being in the speculative stage can be transported only through* `unordered` *memory operations.*
(4) *All allowed behaviors of the* ***K-LLVM*** *memory model is allowed in happens-before memory model.*

2 K-LLVM TYPE CHECKING THEOREM

The theorem that we want to prove is the one in Section 4 of the **K-LLVM** paper. This theorem states that any valid LLVM IR program is statically type checked in **K-LLVM**.

Theorem 2.1. *Assuming every load returning a value in a type prescribed in the load instruction, the program is well-typed by the* ***K-LLVM*** *type system, and the program executes, then every register and every return value of the program will be of the type assigned during the type checking.*

We first discuss some terms that will be used in the proof. We assume there is a partial function φ representing the process of the **K-LLVM** static semantics, such that for every term t as a input LLVM IR program in the FAST format, φ either produces $\bot$ meaning that the term t is not a valid type checked or validity checked LLVM IR program, or produces a term t' that is a transformed term from t in the BAST format. There is a transition system κ implementing the **K-LLVM** abstract machine under the byte-wise sequential consistency assumption. It takes a pair (t, Δ) as input, where t is a valid LLVM IR program in the BAST format, and Δ is the program environment. Through κ, a state (t, Δ) can transition to another state (t', Δ') via the formula $(t, \Delta) \rightarrow_\kappa (t', \Delta')$. We assume that the component name `continuation` of the **K-LLVM** abstract machine is overloaded as a function to achieve the history of all instructions happened for a thread. For example, $\texttt{continuation}(tid, \kappa, t, \Delta)$ produces a list recording all executed instructions from the initial state to the current state defined as (t, Δ) for the thread tid. The history only records instructions happening in the function that is on top of the call stack in Δ. Those instructions happening in another function that was executed before the point of the state (t, Δ) are treated as no-op. The component name `registers` of the **K-LLVM** abstract machine is overloaded as a function to achieve the current status of registers for a state. We first prove a lemma to relate `registers` with input arguments of a function call. The component name `stack` of the **K-LLVM** abstract machine is overloaded as a function to achieve the current status of the call stack of a state. The construct $|\texttt{stack}|$ provides the maximum stack size of a stack in the **K-LLVM** abstract machine.

Lemma 2.2. *Assuming that t is a LLVM IR program for a thread tid, and $\varphi(t) = t'$, so that the program t is checked by the function φ and the BAST term t' is produced, and Δ is the initial environment for t'; Thus, $\sigma = \texttt{registers}(t, \Delta)$ is the initial registers in Δ. If it exists a valid state (t'', Δ''), such that $(t', \Delta) \rightarrow_\kappa^+ (t'', \Delta'')$, let $s = \texttt{continuation}(tid, \kappa, t'', \Delta'')$ is the history of all instructions happened before the state (t'', Δ''), and $|\texttt{stack}| = n$, then, (1) for any use of variable x in an instruction of s that is from the input argument of t', the type of $\sigma(x)$ is equal to the type of the use of variable x, and is also equal to the input argument type of x in t'; (2) in addition, any variable x in an instruction of s that is not from the input argument of t', its value does not come from the initial registers σ.*

We prove the lemma below.

Proof. The proof consists of two parts.

Part 1: we assume that there is no function call in an instruction of $s = \texttt{continuation}(tid, \kappa, t'', \Delta'')$, and then we prove by induction of the length on s, and do a case analysis on all possible instruction that an instruction in s can take. Since in **K-LLVM**, φ checks the type of any use of any input argument matching with the type of the argument defined in a function header, (1) is valid. (2) is valid because LLVM IR programs is required to be SSA format and every use of a variable in every instruction is dominated by its definition; thus, one cannot find any register value is used for any local variable x in an instruction without it being defined before the instruction in s.

Part 2: The proof of this part is done by induction on the maximum stack size $|\texttt{stack}|$ (n).

> **Base case:** when the stack size for a state (t'', Δ'') in the transition system κ is zero, in such case, the result from **Part 1** validates the proof.
> **Inductive hypothesis:** assume that the proof is valid when a state (t'', Δ'') never needs a stack size less than or equal to k.
> **The rest of the inductive step:** With the maximum stack size to be k+1. We have three cases. First, if in a state (t'', Δ'') with its instruction history $s = \texttt{continuation}(tid, \kappa, t'', \Delta'')$, the current stack size of the state is $k + 1$, then the rest of the instruction cannot be a function call before a return instruction; otherwise, the stack overflows. If the current stack size is k, and the next instruction is a function call. By the same argument in **Part 1**, after the function call is applied, the statements (1) and (2) are valid. If the current stack size is less than k, and if the next instruction to execute is a function call. By the inductive hypothesis, the statements (1) and (2) are valid. In summary, we validate the proof of **Part 2**.

By the **Part 1** and **Part 2**, we show the Lemma 2.2 is valid.

□

Based on this information above, we first prove the single thread case below. That is, in an execution of a program t, it never triggers a thread create instruction.

Proof. The theorem is proved by case analyzing on the result of $\varphi(t)$ for any term t in the FAST format.

Case 1: $\varphi(t) = \bot$. In this case, the proof is done since the condition that "the program executes" is not fulfilled.

Case 2: $\varphi(t) = t'$ and $t' \neq \bot$. Let Δ be any initial state for a transition system κ and t' is the initial BAST program. If $\kappa(t', \Delta)$ cannot transition to any other state, then the proof is done since the condition that "the program executes" is not fulfilled. Otherwise, let $(t', \Delta) \rightarrow_\kappa^+ (t'', \Delta'')$ for any (t'', Δ'') through some transition steps. Let $s = \texttt{continuation}(\kappa, t'', \Delta'')$. The prove is done by induction on the length of the steps of the transitions $(t', \Delta) \rightarrow_\kappa^+ (t'', \Delta'')$ starting from length equal to one.

> **Base case:** we do a case analysis on all possible instructions in the start position of s, with the initial registers $\texttt{registers}(t', \Delta)$. If the instruction is a function call, the Lemma 2.2 shows that we can guarantee the correctly type checked property in all instructions of the new function. If the instruction is not a function call. According to Lemma 2.2 and the fact that this is the first transition state and the system only executes one instruction, its uses of variables must come from input arguments of the current function, and the type checked property is also guaranteed by Lemma 2.2.
> **Inductive hypothesis:** we assume that when the length of the transitions in $(t', \Delta) \rightarrow_\kappa^+ (t'', \Delta'')$ is k, the proof is correct.
> **The rest of the inductive step:** when the length is $k + 1$, it means that we have $(t'', \Delta'') \rightarrow_\kappa (t''', \Delta''')$, and $s' = \texttt{continuation}(tid, \kappa, t''', \Delta''')$. In this case, there

is a BAST instruction i such that $s@[i] = s'$. We do a case analysis on all possible instructions i can be. If the instruction is a function call, the Lemma 2.2 shows that we can guarantee the correctly type checked property in all instructions of the new function. The rest of the case analyzes the situation when the instruction is not a function call. Now, since the term t' is type checked through the function φ, so the term t' is also in the SSA format and every use of variable x in t' is dominated by a definition of the variable x sequenced-before the line of the use of the variable. We notice that all instructions in s' are all valid executions of instructions in the term t'. So for every use of variable x in i, there is an instruction j in s and the index of j is less than the index of i, such that x_j is the definition of x_i and x_j dominates x_i. For all every use of variable in i, i.e. x_i, we can find a definition of the variable x_j to form a pair as (x_j, x_i). In any case, the types of the two variables in the pair must be equal, otherwise, the function φ caught the error before it generates a BAST term t'.

Hence, we show the single thread case of Theorem 2.1 is valid.

□

After we have the single thread proof for the Theorem 2.1, we are going to prove the multi-threaded cases. We define the depth of thread chain as the longest thread creation sequence in an execution of a program. For example, if thread x_1 has a thread creation instruction, and it creates thread x_2. x_2 also creates another thread x_3, and so on. Finally, the last thread x_n created in this chain is the one does not create another thread. The number n is the depth of the thread chain from x_1 to x_n.

Proof. We prove the multi-threaded version of Theorem 2.1 via induction on the depth n of the longest thread chain in an execution of a BAST term t' with the initial state Δ, where $\varphi(t) = t'$ based on an input LLVM IR program t.

Base case: when n is equal to one, the thread does not contain any thread creation instruction. The Proof 2 is exactly the theorem we need to prove in this case.
Inductive hypothesis: when $n = 1, ..., k$, there are w threads (named tid_w) in the system, and threads executes $t_1 ... t_w$ different BAST terms. t_i is the one with longest thread chain, and the depth for it is k. In this setting, we assume that the $t_1 ... t_w$ never produces errors due to type mismatch.
The rest of the inductive step: when n is equal to $k + 1$, it means that the longest thread chain is $k + 1$. We assume that there are v threads (named tid_v) in the system, and threads executes $t_1 ... t_v$ different BAST terms. For any one thread tid_h in the system, if thread chain number of tid_h as h is less than or equal to k, based on the inductive hypothesis, the statement is valid. If $h = k + 1$, let tid_x be the thread that creates the thread tid_v. The thread chain number of tid_x is k, so the t_x type checks due to the inductive hypothesis. Without losing generality, we assume that at point s', tid_x creates the thread tid_h, and $s' = s@[is]$ and $s = \texttt{continuation}(tid_x, \kappa, t', \Delta')$. By the same argument from Proof 2, s contains all valid executed instructions. Thus, before the state (t', Δ'), the system does not have type mismatch. In the transition $(t', \Delta') \rightarrow_\kappa (t', \Delta')$, the system creates thread tid_h with some input arguments. The input arguments are from $\texttt{registers}(t', \Delta')$, so they are type correct as same as input perimeters of the code in t_h. After the thread tid_h dies, the return value does not type-mismatch with the sub sequential usage of the value, according to the same argument in Proof 2. Hence, the thread tid_x creating t_h does not have type mismatch problem.

687 In thread tid_h, Since its thread chain depth is $k + 1$, so its program t_h cannot contain
688 any more thread creation. In this case, following exactly the proof of Proof 2 validates
689 the proof.

690 Hence, we have shown that the multi-threaded version of the Theorem 2.1 is valid.

691 □

692

693 ## REFERENCES

694 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-
695 memory Concurrency. *SIGPLAN Not.* 52, 1 (Jan. 2017), 175–189. https://doi.org/10.1145/3093333.3009850
696 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in
697 C/C++11. *SIGPLAN Not.* 52, 6 (June 2017), 618–632. https://doi.org/10.1145/3140587.3062352
698 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
699 Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation
700 and Avoids Thin-air Executions. *SIGPLAN Not.* 51, 1 (Jan. 2016), 622–633. https://doi.org/10.1145/2914770.2837616

701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735