A VERIFICATION FRAMEWORK SUITABLE FOR PROVING LARGE LANGUAGE TRANSLATIONS

BY

LIYI LI

DISSERTATION

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

Urbana, Illinois

ABSTRACT

Previously, researchers established some frameworks, such as Morpheus [1], to specify a compiler translation in a small language and prove the semantic preservation property of the translation in the language under the assumption of sequential consistency. Based on the Morpheus specification language, we extend the verification framework to prove the compiler translation semantic preservation property in a large real-world programming language with a real-world weak concurrency model. The framework combines four different pieces. First, we specify a complete semantics of the $\mathbb{K}$ framework and a translation from $\mathbb{K}$ to Isabelle as our basis for defining language specifications and proving properties about the specifications. Second, we define a complete operational semantics of LLVM in $\mathbb{K}$, named **K-LLVM**, including the specifications of all instructions and intrinsic functions in LLVM, as well as the concurrency model of LLVM. Third, to verify the correctness of the **K-LLVM** operational model, we create an axiomatic model, named hybrid axiomatic timed relaxed concurrency model (HATRMM). The creation of HATRMM is to bridge the traditional C++ candidate execution models [2, 3, 4] and the **K-LLVM** operational concurrency model. Finally, to enhance our framework to prove the semantic preservation property in a relaxed memory model, we define a new simulation framework, named Per Location Simulation (PLS). PLS is suitable for proving semantic preservation property in a relaxed memory model.

To my parents, Jinfa Li and Wu Cheng, and my spouse, Jiaing Huang, with love and gratitude

ACKNOWLEDGMENTS

I would like to express my greatest appreciation to Professor Elsa Gunter. I was advised by Elsa since my undergraduate study in the University of Illinois at Urbana-Champaign. At that time, I was a Chinese student who could not speak proper English and lived in anxiety and loneliness. Now, I am a Ph.D. student who is going to graduate with a family. Elsa affected every aspect of my life. For the rest of my life, I will always remember Elsa as if she is my mother. I would like to thank my colleague, Professor William Mansky, coming up with such great platform, Morpheus, which my work is based on. I would like to thank my committee: Professor Grigore Rosu, who taught me the $\mathbb{K}$ framework and guided me to finish the semantics of Lower Level Virtual Machine in K (**K-LLVM**); Professor Vikram Adve, who provided me knowledge on real-world languages and compilers; Professor David Padua, who advised me on different compiler optimizations; Professor Steve Zdancewic, who suggested me pick a particular Ph.D. thesis direction instead of a very large goal that might cost many years. I would like to thank my parents, who have supported me for many years. Finally, I would like to thank my wife, Jiaming Huang, who have followed me through every twist and turn. She listened to my complaints and gave me suggestions. I couldn't have made it this far without all these great people around me.

TABLE OF CONTENTS

Chapter 1: INTRODUCTION

Compiler correctness is one of the most fundamental problems in programming, but the term can actually be understood on two different levels. When someone talks about a compiler, e.g. GCC [5], being "buggy", they are usually referring to a fault or unverified compiler with the potential to transform correct high-level code into low-level code containing errors. The problems appear in the low-level code, which is actually executed in a single-threaded environment. They are easy to observe but not easy to fix. However, there is a second level of compiler bugs, where multi-threaded programs are translated to low-level multi-core architecture and executed. Users might run their multi-threaded programs many times before observing some strange behaviors that the programs are not supposed to produce under the assumptions of the high-level language. These extra behaviors result from differences in the semantics of these two languages, e.g. because they have different concurrency models. For example, some behaviors that are not allowed in an instruction sequence in the high-level language, according to its concurrency model, occur in the compiled program because the low-level language has a more flexible concurrency model. Gaining behaviors is generally not desirable, and they can be classified as bugs. These behaviors can also invalidate the analyses of codes, causing correctly written programs to execute incorrectly; these problems may require considerable effort to detect and resolve.

Previously, Gunter and Rosu designed a formal framework in the VeriF-OPT project that addresses part of the compiler correctness problem on the first level [6]. The framework focused primarily on specifying compiler optimizations in a particular source language and proving them correct. Gunter and Mansky also developed a specification language, Morpheus [1], to specify and prove the correctness of compiler optimizations in Isabelle [7]. There are three points to note about VeriF-OPT. First, it focuses on a domain specific language (Morpheus). Even though the language is good for specifying properties related to control flow graphs, it might not be adequate for specifying all compilation steps in a compiler. Specially, it might not be able to specify all interesting compiler optimizations. For example, the Morpheus language is not able to specify the thread-inlining optimization, i.e. executing two threads' programs in one sequentialized thread. Second, the previous compiler correctness proof [1, 8] is based on a simple well-founded memory model. The proof strategy might not work for a language based on a real-world relaxed memory model such as the C++ memory model [2, 3]. Furthermore, the compiler correctness proof framework in the previous work [1, 8] is based on a simple bi-simulation framework. It neither considers program executions that lead to failure, nor does it include the extra program execution behaviors caused by a relaxed memory model.

In this thesis, we propose a framework, **Morpheus+**, that combines the Morpheus specification language with a system (**IsaK**) for defining concurrent language semantics and a system for proving the per-simulation relation between two language specifications for a particular transformation, in order to verify the compiler correctness for languages under a real-world multi-threaded concurrency model, such as the relaxed concurrency model. The primary goal of the project is to provide a memory-model-aware generalized framework for proving the correctness of transformations, especially compiler optimizations, and will include: the **IsaK** [9, 10] and **TransK** [11] tools for connecting language specifications in $\mathbb{K}$ to theories in Isabelle, a generic memory model that is able to define relaxed concurrency models, such as the LLVM IR concurrency model; and a compiler correctness proof system based on the Per-Location Simulation framework (PLS). We rely on $\mathbb{K}$ to easily define large and real-world language specifications and on Isabelle to prove correctness properties about the compilations between two languages. As an example of **Morpheus+**, we define compiler optimizations based on the complete semantics of LLVM IR with its operational concurrency model. Using PLS, we are able to prove the correctness of these compiler optimizations.

Chapter 2: RELATED WORK

We provide here some related work of the **Morpheus**+ project.

2.1 COMPILER CORRECTNESS RELATED WORK

Compiler verification efforts date back at least to a 1973 paper by Morris [12]. Early efforts include the construction of an interpreter for the Piton language in the Boyer-Moore theorem prover [13, 14] and the verification of a compiler for a subset of the Gypsy language [15]. Also notable is Elsa Gunter's work in formalizing the syntax and semantics of SML, as well as its module system, in HOL-90 (the predecessor to Isabelle) [16, 17]. The proof assistants Coq [18] and Isabelle [19] have been the platform used for various significant formal verification efforts, including several in the area of verified compilation. Early Isabelle projects included an interpreter for a small functional language verified aginst its denotational semantics [20] and a verified lexer [21]. More recent developments have included verification of a code generation algorithm from SSA form [22], a bytecode verifier for the Java Virtual Machine (JVM) [23], and compilers for substantial subsets of C [24, 25] and Java [26]. Gesellensetter *et al.* [27] have also used a translation-validation approach in Isabelle to find bugs in GCC. The Jinja project has formalized a significant subset of Java, including its concurrency model, in Isabelle [28, 29]. Coq projects have included verification of the JavaCard smartcard platform [30], development of a formally specified and executable JVM [31], verification of an algorithm for sequentializing parallel assignments [32], verification of CPS transformations for functional languages [33], and creation of a verified compiler for simply-typed lambda calculus [34].

Perhaps the most impressive achievement in compiler verification to date is the CompCert project, culminating in an end-to-end proved-correct compiler from a subset of C to PowerPC machine code written entirely in Coq [35, 36, 37]. Though complete formal compiler verification was previously considered near-impossible by the formal analysis community (see [38]), the CompCert project has provided a powerful counterexample to this argument. While it does not cover the entire C specification, the compiler does include a range of real-world optimizations, and introduces a general method for proving correctness of optimizations involving dataflow analysis. Further work on the project includes a framework for adding new optimizations [39] and an extension for garbage collection [40], as well as a C-like memory model for verifying pointer-based programs [41]. Ŝevčik *et al.* [42] are in the process of developing an analogous compiler for a language including shared-memory concurrency, and have lifted the proofs of correctness for several of CompCert's optimizations, including constant propagation and common subexpression elimination, to the parallel case. Blazy and Leroy [43] in the CompCert project have verified an optimizing compiler based on CLight, including compilation steps and C-like modular systems. They used Coq to generate a compiled code behaving exactly as described by the specification of the language. Other projects based on CompCert include Appel's, which combined program verification with a verified compilation software tool chain [44].

Building on CompCert, the Verified Software Toolchain (VST) project [45] seeks to link via formal methods the results of static analysis tools on high-level programs to the behavior of compiled code. VST has a particular focus on concurrent programs, and has so far produced several levels of operational semantics for a C-like source language with threads and locks, as well as a versatile program logic called Concurrent Separation Logic [46]. Integration of Morpheus with VST could potentially provide considerable benefits to both frameworks; Morpheus would offer improved generality and modularity and easier links to other source and target languages, while VST would offer an extensive set of tools and analysis approaches for verification of properties on compiler specifications.

The PVS theorem prover has also been used in some compiler verification efforts, such as a verification of peephole optimizations [47] and a framework for translation validation of optimizations [48]. PVS has not generally been a popular target for formal verification of compilers, perhaps due to the relatively large size of its trusted code base, but is potentially a versatile framework for formal analysis of transformations.

The TRANS language that forms the core of our approach to compiler specification is based on work by Lacey *et al.* [49, 50], and the original statement of TRANS is due to Kalvala *et al.* [51]. More recently, Kalvala and Warburton have used the TRANS approach to automatically find and fix bugs in Java programs [52]. Rather than using the formal semantics of TRANS to verify program correctness, this approach intentionally uses TRANS to change the semantics of programs, from potentially buggy programs (according to some user-provided pattern for bugs) to more correct ones. Since this could potentially lead to the introduction of bugs as well as their elimination, potential rewrites are not carried out automatically, but are rather suggested to the user of the system as possible fixes. Thus, in parallel with our work, TRANS is also being used as a software engineering tool for offering bug fixes to Java programmers. The Morpheus language, based on the TRANS language, allows users to specify program transformations and optimizations in a program that may involve parallelism and using proof theories in the previous VeriF-OPT project to prove the program optimizations correct under a semantics involving parallelism [1].

Cobalt [53] and Rhodium [54] are systems for specifying compiler optimizations developed in parallel with TRANS, and also inspired by Lacey's work on optimizations and temporal logic [49]. As in TRANS, optimizations are written as rewrites conditioned on temporal properties of the control-flow graph. Cobalt and Rhodium extend Lacey's simple intermediate language to handle more complex features such as pointers, dynamically allocated memory, and recursive function calls. They also restrict the range of temporal formulae from general CTL to a tightly constrained subset, which in practice has allowed them to prove soundness of optimizations completely automatically. Optimix [55, 56] is another graph-rewriting-based system for expressing compiler optimizations, developed prior to TRANS and with a less compact method of establishing the conditions under which the rewrite should be performed.

In terms of functional programming languages (ML languages) and theorem provers, Milner, Tofte, Harper, and MacQueen [57] formalizes one of the most prominent and mathematical programming language specifications, whose formal and executable specifications were given by Lee, Crary, and Harper [58], also by VanInwegen and Gunter [16], and by Maharaj and Gunter [59]. In contrast to ML, formalizing other real world language specifications is a challenge because they are designed without formalism in mind. There have been a number of formal language specifications given in the HOL [60] and Coq [18]. For example, A small step semantics of C in HOL was specified by Norrish [61], who proved substantial meta-properties, but the specification has not been tested for conformance with implementations. Many other projects were done on Coq, such as the VeLLVM project [62].

2.2 𝕂 RELATED WORK.

In this section, we discuss other work describing 𝕂 semantics, and language specifications defined in 𝕂, as well as order-sorted algebras.

Order-sorted algebras were first introduced systematically by Goguen *et al.* [63]. Many people defined rewriting strategies, unifications and equational rules on top of order-sorted algebras and further extended the operational semantics of order-sorted algebras [64, 65, 66, 67, 68]. Based on order-sorted algebras, Meseguer *et al.* [69, 70] developed rewriting logic. A major contribution of rewriting logic is to contain the operational semantics of order-sorted algebras. The core idea of rewriting logic is that it distinguishes equations from rewriting rules – equations partition terms into equivalence classes while rewriting rules act like traditional transition rules in structural operational semantics. Maude [71] implemented the syntax and semantics of rewriting logic and provided several useful tools and applications [72, 73, 74]. Another implementation of an order-sorted algebra is PROTOS(L) [75], which has an operational semantics based on a polymorphic, order-sorted resolution.

𝕂 has a brief English description of its semantics and provides some examples in the 𝕂 overview document [76]. In addition, there is a compiler implementation in Java to allow users to define their language specifications and show traces of execution programs. The compiler has almost fifty compilation steps, and eventually executes a program in a very small core language that has no English description to describe its grammar or semantics. In this sense, these $\mathbb{K}$ specifications are far from being formal. Matching Logic [77] is a logic system that is built on top of $\mathbb{K}$ for reasoning about structures. The current invention of Matching Logic is Reachability Logic [78, 79]. It is a seven rule proof system and is language independent. It generalizes transitions of operational language specifications defined by users and the Hoare triples of axiomatic semantics [80] to prove properties about programs in the specifications, so that users do not need to define the axiomatic semantics of a specification. There is an ongoing project by Moore [81] that transfers the $\mathbb{K}$ specifications to Coq [82] and plans to prove properties of the programs of the specifications in Coq. The current state is that Moore has managed to define a useful co-induction tool in Coq and prove some properties by defining small language specifications in Coq. Big language specifications have been defined in $\mathbb{K}$ including C [83], PHP [84], JavaScript [85], and Java [86]. 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.

Goguen *et al.* [63] introduced a way of translating solely initial free (having no equations or rules) order-sorted algebras to many-sorted ones. One recent attempt at translating order-sorted algebras into many-sorted ones was made by Meseguer and Skeirik [87]. Their algorithm provided a theoretical framework to translate order-sorted equations and rules to many-sorted ones by generating (potentially exponentially many) new copies of transition rules for each of the sorts subsorting to the ones in the original transition rules. In addition, Li and Gunter [11] provided a new translation method to translate order-sorted algebras into many-sorted ones, which is the theoretical foundation of our project.

The study of many sorted algebras has a long history. Their logic system was explored by Wang [88]. Many well-known programming languages such as C, Java, LLVM, and Python are based on them.

2.3 RUNTIME EXECUTION MODELS AND MEMORY MODELS

The idea to have an execution model that connects the runtime, instruction semantics and memory model was inspired by real-world execution models. Mainly, we were enlightened by the Tomasulo algorithm [89], and to a lesser extent by some current execution models such as the MPC model by Perache *et al.* [90], the fractal model by Subramanian *et al.* [91], and the copy or discard (CorD) execution model by Tian *et al.* [92]. Tomasulo's algorithm is simple enough to act as a guideline for the execution model for our LLVM IR semantics (**K-LLVM**). The other models are too complicated for our purposes, containing too many details about hardware.

We are also helped by various memory models. Lamport probably was the first to define a memory model weaker than sequential consistency for multi-threaded programs [93]. Adve and Hill [94] started defining weak memory orders for memory operations. Focusing just on hardware models: Ahamad *et al.* [95] axiomatized causal memory and proved some important theorems. Higham *et al.* [96] formalized SPARC and a number of simpler memory models in both axiomatic and operational styles. Sevcik *et al.* created a formal verification framework for a small C-like language [42]. The same group [97] later developed the CompCertTSO to verify a compiler from CLight to X86 based on a relaxed memory model. Mansky *et al.* [98] developed an axiomatization of the CompCert sequential consistency memory model and combined the model with a subset of the LLVM language to verify the correctness compiler optimizations.

In the SPARC documentation [99], an axiomatic style similar to the candidate execution model was used. Alglave *et al.* [100] specified in great detail how to use a candidate execution model to define relaxed memory models and provided several verification tools. The C11 memory model was designed by the C++ standards committee based on a paper by Boehm and Adve [101]. Batty *et al.* formalized the C11 model with some improvements and proved the soundness of its compilation to X86-TSO [2]. A number of papers [102, 103, 104, 105] found that Batty *et al.*'s model enabled thin-air behaviors. Vafeiadis *et al.* [106] found many other limitations in Batty *et al.*'s model and proposed ways to handle them. In 2016, Batty *et al.* proposed a more concise model for `sc` atomics [107], but the model is stronger than C11, and their `sc` fences are too weak. Much previous work [103, 108, 109, 110] focused on a fragment of C++ concurrency. From this corpus, we select Lahav *et al.*'s SRA model [110] to show the soundness of our `acq`/`rel` atomics. In 2017, Lahav *et al.* [3] defined a comprehensive C++ model (RC11) based on all previous models, with extra extensions on Batty *et al.*'s model. In Chapter 6, we discussed this model as a part of several topics. The main limitations from our perspective with the model is that its OUT-OF-THIN-AIR condition is too strong and rules out too many good executions. Many previous papers [111, 112, 113] also proposed solutions for defining what are "good" out-of-thin-air behaviors. These models were not in the axiomatic candidate execution fashion, and one of them (the promising memory model [113]), which we have compared in Chapter 6, has been proved to be represented by the IMM model [4]. Chakraborty and Vafeiadis [114] provided a concurrent abstracted memory model for LLVM IR. It provided the semantics for a fragment of LLVM IR memory operations while keeping the model stronger than Lahav *et al.*'s. The IMM model by Podkopaev *et al.* [4], based on RC11 and the promising memory model, defined its OUT-OF-THIN-AIR property with a weaker one than the one in RC11. We have shown in Chapter 6 that it is not suitable in handling many thin-air behaviors, and some of its control dependency is too weak so it enables some thin-air behaviors. The essential difference is that IMM is designed to provide a spiritual sample for people to understand how to compile C++ to hardware code, while HATRMM is designed to be used by a PLS to prove properties about a compiler.

2.4 LANGUAGE SPECIFICATIONS RELATED TO LLVM IR

Besides **K-LLVM**, other formal executable sematics for LLVM-IR include VeLLVM [115] and the previous LLVM semantics in $\mathbb{K}$ [116]. VeLLVM was the first project to define a relatively complete specification for the core of LLVM IR. It was defined in the theorem prover Coq [18] and covered a core set of LLVM IR instructions. VeLLVM formalizes a mechanized semantics for LLVM IR, its type system, and the properties of its SSA form. It also has an interpreter extracted from Coq that ran 145 test programs and passed 134 of them. The memory model of VeLLVM is based on CompCert with newly developed features that are specifically designed by the VeLLVM team for capturing the memory data layout features in LLVM IR. Their model associates metadata to memory byte data fields, so that an execution of a LLVM IR program can utilize the metadata. This feature is similar to the memory data layout in **K-LLVM** (see Sec. 5.3.3). With VeLLVM, users can prove properties about translations defined in LLVM IR. Many papers, such as [115, 117], have been published about compiler correctness, memory models, and verification of compiler schemes using VeLLVM. The LLVM semantics in $\mathbb{K}$ by Ellison and Lazar [116] is a prior work of **K-LLVM**, and it provides many definitions for LLVM IR instructions for single-threaded programs. The definition influences **K-LLVM**, particularly the definition of LLVM's syntax, static semantics, and single-threaded dynamic semantics.

There are other pieces of work that are not meant to directly define the LLVM IR semantics but influence **K-LLVM**. First, Lee *et al.* [115] investigated the LLVM IR undefined behaviors with no concrete semantics for all undefined behaviors. Kang *et al.* [117] provided a model in C to support the `inttoptr`/`ptrtoint` casting operations and enabled the correctness proofs of many LLVM optimizations that rely on certain memory provenance model features that the previous CompCert model is not able to provide. The difference between Kang *et al.*'s work and the handling of `inttoptr`/`ptrtoint` casting operations in **K-LLVM** is that they focus more on the relations between the executions of such operations with respect to the memory and define the relations in terms of logical properties, whereas **K-LLVM** focuses more on the formalization of an abstract machine, and emphasizes different conceptual components finishing different tasks without communicating with each other through "unofficial" channels. Ellison and Rosu [83] defined the full C semantics with a simplified version of the CompCert model. Chakraborty and Vafeiadis [114] provided a concurrent abstracted memory model for LLVM IR that focused on an abstraction of the concurrent LLVM IR memory behaviors. Lee *et al.* [118] proposed a novel LLVM memory model including a data layout and memory pointer provenance model based on a small set of LLVM IR memory related instructions. Specially, they provide an algorithm for preventing address guessing when the total number of possible allocations in a system is bounded. A current challenge is the extension of that memory model to the full set of LLVM-IR memory related instructions. Memarian *et al.* [119] provided two pointer provenance models for C/C++ languages and reconciled the ISO C standard. Similar to Lee *et al.*'s work, Memarian *et al.* focused on creating better pointer provenance models for C/C++.

Other interesting work includes a JavaScript specification by Bodin *et al.* [120], and formalized semantics of OCaml Light by Owens *et al.*, which is built in Ott [121], which provides an easy way to write specifications, and automatically translates them into HOL, Isabelle, and Coq. Isabelle, HOL and Coq, being proof assistants, have a relatively steep learning curve.

A lot of work has been done on formalized specifications in Java and C#: Eisenbach's formal Java semantics [122] and Syme's HOL semantics [123] for Drossopoulou, the C# standard by Börger *et al.* [124], which is formally executable and uses abstract state machines, and the executable Java specification by Farzan *et al.* [125].

Chapter 3: BACKGROUND

Below we discuss the necessary background information for this paper, including the $\mathbb{K}$ style rules that we use in this thesis.

3.1 PREVIOUS APPROACHES AND THEIR PROBLEMS

We first investigate what the available verification approaches for compiler correctness. There are basically two kinds: proofs and validations. In terms of proofs, the TRANS approach is the trend. The basic idea is to define a domain-specific language as the core, and then build a proof infrastructure around it. The TRANS language developed by Kalvala *et al.* [51] is one of the significant pieces of this approach. Mansky and Gunter constructed a framework for the formal verification of compiler optimizations [126], which is an extension of the TRANS work. They also developed a domain specific language, Morpheus [1], based on temporal logic. It is used to express and verify properties over paths in transition systems. It has been used for a long time to verify properties of programs. In contrast to traditional first-order and higher-order logic systems, temporal logic systems have a built-in notion of progress through time, and have "next" and "until" as first-class concepts. The two most commonly used temporal logics are LTL [127], which is used to express properties on single paths, and CTL [128], which is used to express properties on sets of paths and can be thought as trees that branch over time. In the standard interpretations of both LTL and CTL, programs are modeled as finite-state automata (FSA), and formulae are evaluated over paths through the automata. In their Morpheus paper, Mansky and Gunter specified a set of compiler optimizations on LLVM that included redundant store elimination and loop peeling, and utilized the Isabelle semantics to verify the correctness of these optimizations on a subset of LLVM – MiniLLVM. The generalized control flow graphs along with the Morpheus paper allow users to describe transformations that may involve parallelism. In the VeriF-OPT project [6], Mansky and Gunter specified a simulation framework for proving compiler transformation correctness for language semantics involving parallelism[126], and provided the supporting theories. In the paper, they described how to use a Morpheus-like language (PTRANS was being developed into Morpheus) and its associated theories to verify a reordering optimization on parallel programs. Their method relies on an analysis of the use of locks in the target program.

The main trend in validation approaches is translation validation, in which each source program is compared with the corresponding low-level output of the compiler, and checked to ensure that the translation preserved the property φ [38]. This approach treats the compiler as a black box, producing unreliable output that must be verified every time. Individual proofs are considerably more lightweight than any complete compiler verification, and may often be performed automatically. Since its inception, this technique has been successfully applied to optimizing compilers in various instances [38, 129, 130, 131, 132].

The **Morpheus+** project uses the two approaches for different tasks. The problem with the TRANS approach is that a domain specific language is often restricted. For example, the Morpheus language is only good for specifying compiler optimizations within a program that has a specific control flow graph. Some optimizations might not be able to be specified by using Morpheus only, such as thread inlining optimization. Another problem with the VeriF-OPT project is its simulation framework. The simulation framework is a traditional one similar to the CompCert simulation framework. In **Morpheus+**, we specify the per-location simulation framework to verify compiler optimizations based on real-world relaxed memory model (Chapter 6). The **Morpheus+** project also takes the validation trend into account. It is typically hard to prove if a language specification achieves the spirit of the original design. We use extensive program validation to test if the language specifications, especially the source language specifications, meet the design spirit of the language designer in $\mathbb{K}$, as we have done to test **K-LLVM**.

3.2 MORPHEUS SYNTAX

Here we introduce the syntax of Morpheus. More details can be found at the work of Mansky *et al.* [1]. The basic approach of the Morpheus specification language is modeled after the TRANS language of Kalvala *et al.* [133]. Optimizations are specified as conditional compositions of rewrites on a generalized control flow graph (GCFG) containing the program's code. The language is partitioned into three largely independent components: core graph transformations (H below), conditions given in a variant of Computation Tree Logic (CTL) (φ below), and a strategy language (T below) for building complex transformations out of component transformations and conditions.

Intuitively, the rewrite portion of an optimization expresses the local transformation to be made, the condition characterizes the situations in which the optimization should be applied, and the strategy language allows us to build whole-system transformations out of collections of local ones. Morpheus is a special-purpose language for the transformation of GCFGs, and as such is parametrized by aspects of GCFGs, namely node names, node labels (program instructions), and edge labels (marking control flow). Transformation specifications may mention aspects of GCFGs concretely, but more generally, they use pattern variables that will be instantiated with control flow graph components in each specific application. We will use the term "expressions" to refer to patterns built from both concrete entities and metavariables (which will be instantiated with concrete entities when the transformation is applied). We use the term metavariable (a) to refer to the variables in the patterns and expressions in Morpheus transformations, as opposed to the concrete programming variables that will be found in instructions.

$$\begin{aligned}
H \triangleq\ & \texttt{add\_node(}\pi\texttt{,}B\texttt{,(}l_1\texttt{,}\pi_1\texttt{),...,(}l_n\texttt{,}\pi_n\texttt{))} \mid \texttt{add\_node(}\pi\texttt{)} \\
& \mid \texttt{relabel\_node(}\pi\texttt{,}B\texttt{)} \mid \texttt{move\_edge((}\pi\texttt{,}l\texttt{,}\pi_1\texttt{),}\pi_2\texttt{)} \\
\varphi \triangleq\ & \texttt{true} \mid p\texttt{(}\overrightarrow{x}\texttt{)} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists\, a.\ \varphi \mid \mathcal{AX}\ \varphi \mid \mathcal{AY}\ \varphi \mid \mathcal{EX}\ \varphi \mid \mathcal{EY}\ \varphi \\
& \mid \mathcal{A}\ \varphi\ \mathcal{U}\ \varphi \mid \mathcal{E}\ \varphi\ \mathcal{U}\ \varphi \mid \mathcal{A}\ \varphi\ \mathcal{S}\ \varphi \mid \mathcal{E}\ \varphi\ \mathcal{S}\ \varphi \\
T \triangleq\ & H \mid \texttt{SATISFIED\_AT}\ \pi\ \varphi \mid \texttt{NOT}\ T \mid T\backslash T \mid \texttt{EXISTS}\ a.\ T \mid T \texttt{ + } T \mid T\ \texttt{;}\ T \mid T\ \texttt{*}
\end{aligned}$$

Figure 3.1: The Morpheus Language

The syntax of Morpheus consists of actions (H), conditions (φ), and transformations (T) (Fig. 3.1). The atomic actions H begin with `add_node` and `remove_node`, which add and remove nodes that have no incoming edges. In the case of `add_node`, the addition only takes place if the node description is well-formed as a CFG node (i.e., it has the right number and kind of outgoing edges for its instruction label). The `relabel_node` action relabels an existing node with a new instruction, as long as that new instruction is compatible with the existing edge structure. The only action that operates directly on edges (rather than nodes) is `move_edge`, which moves the destination of an edge from one node in the graph to another. The conditions φ of Morpheus are based on First-Order CTL (FOCTL). Starting from a set of atomic predicates p, they include all of the usual propositional and temporal operators. The $\mathcal{S}$ ("since") and $\mathcal{Y}$ ("yesterday") operators are the past-time counterparts to the $\mathcal{U}$ ("until") and $\mathcal{X}$ ("next") operators respectively; for instance, $\mathcal{E}\ \varphi_1\ \mathcal{S}\ \varphi_2$ holds when there exists some path backwards through the graph such that φ_1 holds until a previous point at which φ_2 holds. The existential quantifier $\exists$ is used to quantify over metavariables in a formula: these metavariables may then appear in the atomic predicates of a formula, enhancing the expressive power of the conditions. At the top level, a transformation T combines conditions and rewrites using strategies. Strategies are inherently non-deterministic, as is reflected by their returning a set of possible transformed graphs. The simplest strategy is just to perform an action H. The strategy `SATISFIED_AT` π φ acts as the identity transformation if φ holds of the GCFG at the node π, and returns the empty set if φ fails to hold on π. Thus `SATISFIED_AT` π φ acts as filter, allowing through only those GCFGs and nodes π that satisfy φ. On the other hand, `NOT` T and $T_1\backslash T_2$ allow us to deselect graphs by the ability to perform a transformation. The transformation `NOT` T selects those graphs that T cannot transform, i.e., those not in the domain of T, and deselects those that it can transform. The transformation $T_1 \backslash T_2$ restricts the output of T_1 to those graphs that could not be produced by T_2, i.e., those not in the image of T_2. Note the difference between these two filters: `NOT` T filters based on the complement of the domain of a transformation, while $T_1 \backslash T_2$ filters based on the complement of the range of a transformation. The strategy `EXISTS` $a. T$ binds a in T, limiting its scope to the free occurrences of a in the conditions and actions of T. Finally, the constructs `+` and `;` allow for choice between and sequencing of two transformations respectively, and the iteration operator `*` allows for the repeated application of a transformation any number of times.

For a simple example of a Morpheus transformation, assume we have a language of instructions that supports assign- ments and binary arithmetic expressions. In this setting, if we have a variable assigned the result of applying an arithmetic operation to two constants, we might want to replace the operation with its result. This can be done by the transformation in Fig. 3.2.

```
simple_constant_folding(π) ≜ EXISTS x a b c oper.
SATISFIED_AT π stmt( x = oper( a,b)) ∧ is_const(a) ∧ is_const(b) ∧ is_const(c)
∧eval(oper(a,b),c) ; relabel_node(π,x = c)
```

Figure 3.2: A Morpheus Example

This is an existentially quantified sequence of a condition and an action. (Note that *oper* is a metavariable that will be bound to an arithmetic operator appearing in the program syntax, and `eval(`e`,`c`)` is a predicate asserting that the expression e evaluates to the constant c.) We may apply `simple_constant_folding` to a program with a node labeled `diff = 10 - 2` , and the transformation will match π to (the name of) this node, x to `diff` , a to `10`, b to `2`, *oper* to `( op - )`, and c to `8` (because of the clause is `(`c`,`*oper*`(`a`,`b`))` , and relabel the node to `diff = 8`.

3.3 ISABELLE/HOL

Isabelle/HOL [19] is an interactive proof engine that allows users to prove properties about language specifications. The basis of Isabelle is high-order typed λ-calculus (System F). To prove that a theory in **IsaK** is equivalent to another one that was translated from the theory in **IsaK** by **TransK**, we define a simple Isabelle (based on simple typed λ-calculus) in Isabelle/HOL. A fragment of it is shown in Figure 3.3.

In Isabelle, we use a construct `datatype` to define a datatype. Variables beginning with the symbol "'" are type variables such as the `'tyVar` in the datatype `isaType`. Constructor parameters begin with a capital letter, while type or function names begin with a lower case one.

```
datatype ('tyVar, 'tyConst) isaType =
 TyVar 'tyVar
 | TyConst "'tyConst" "('tyVar, 'tyConst) isaType list"

datatype ('tyVar, 'iVar, 'cVar) isaTerm =
 VarTm 'tyVar 'iVar | Const 'tyVar 'cVar
 | App "('tyVar, 'iVar, 'cVar) isaTerm" "('tyVar, 'iVar, 'cVar) isaTerm"
 | Lambda 'iVar 'tyVar "('tyVar, 'iVar, 'cVar) isaTerm"
...
locale ITheory =
 fixes TypeConsts ::  "'tyConst set"
 and Types ::  "('tyVar, 'tyConst) isaType set"
 and FunType ::  "'tyConst"
...
 assumes funTypeRule :  " ∀  a b .  a ∈ Types ∧  a = TyConst FunType b ⟹ (length b = 2)"
...
```

Figure 3.3: A Simple Isabelle Theory

As in Standard ML, users are able to use the keywords `primrec`, `fun` and `function` to define a function. The difference is that a `primrec` function has a fixed structure so that it does not require termination proof; a `fun` function needs an automatic termination proof from Isabelle; and a `function` labeled function requires a programmer termination proof. Users are also able to use the keyword `inductive` to define transition rules for a transition system. The keywords `lemma` and `theorem` allow users to define theorems about datatypes, functions and inductive relations in Isabelle. The typical way to write theorems is with a combination of first-order logic and high-order Isabelle variables. In Figure 3.3, the assumption after `funTypeRule` provides a way to define a theorem in Isabelle. A `locale` (Fig. 3.3) is a special way to define a parameterized theory in Isabelle. The body of a `locale` is just a set of functions, inductive relations, or theorems. In the header of a `locale`, users are able to use keyword `fixes` to provide the `locale` with a list of different global data, while the keyword `assumes` can allow users to provide a list of assumptions that a system needs to satisfy in order to use the `locale`. In Figure 3.3, there is only one assumption, which means that any term with the constructor `TyConst` in the `locale` named `ITheory` must have two children.

3.4 THE $\mathbb{K}$ FRAMEWORK

We will briefly introduce $\mathbb{K}$ in this subsection. $\mathbb{K}$ [76] 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. Most of the information about $\mathbb{K}$ here is from the work of Li and Gunter [9].

$$\texttt{(a)}\ \textsc{syntax}\quad \mathit{Exp} ::= \mathit{Exp}\ /\ \mathit{Exp}\ \ [\mathit{strict}(1)]\ \ \texttt{(b)}\ \left\langle \begin{matrix} \langle \$\mathit{PGM} : \mathit{KItem}\rangle_{\mathsf{k}}\ \langle \texttt{.Map}\rangle_{\mathsf{env}}\ \langle \texttt{.Map}\rangle_{\mathsf{heap}} \\ \langle\langle\langle \texttt{.K}\rangle_{\mathsf{name}}\ \langle \texttt{.K}\rangle_{\mathsf{body}}\rangle_{\mathsf{class*}}\rangle_{\mathsf{classes}} \end{matrix} \right\rangle_{\mathsf{T}}$$

$$\texttt{(c)}\ \left\langle \frac{X}{E}\ \cdots \right\rangle_{\mathsf{k}}\ \langle \cdots\ X \mapsto N\ \cdots\rangle_{\mathsf{env}}\ \ \langle \cdots\ N \mapsto E\ \cdots\rangle_{\mathsf{heap}}\ \ \texttt{(d)}\ \begin{matrix} \dfrac{X : \mathit{Int}\ /\ Y : \mathit{Int}}{X : \mathit{Int}\ /\texttt{Int}\ Y : \mathit{Int}} \\ \mathit{requires}\ \ Y \neq 0 \end{matrix}$$

$$\texttt{(e)}\ \begin{matrix} \langle B : \mathit{Bag}\ \ \langle X \curvearrowright \kappa\rangle_{\mathsf{k}}\ \langle \rho_1, X \mapsto N, \rho_2\rangle_{\mathsf{env}}\ \langle \rho_3, N \mapsto E, \rho_4\rangle_{\mathsf{heap}}\rangle_{\mathsf{T}} \\ \Rightarrow \langle B : \mathit{Bag}\ \ \langle E \curvearrowright \kappa\rangle_{\mathsf{k}}\ \langle \rho_1, X \mapsto N, \rho_2\rangle_{\mathsf{env}}\ \langle \rho_3, N \mapsto E, \rho_4\rangle_{\mathsf{heap}}\rangle_{\mathsf{T}} \end{matrix}$$

$$\texttt{(f)}\ \texttt{exp(}\ X\ \texttt{)}\ \equiv\ \texttt{aExp(}\ X\ \texttt{)}$$

Figure 3.4: A Simple $\mathbb{K}$ Program

Figure 3.4 `(a)` shows how a syntactic definition is defined in $\mathbb{K}$. It uses the assignment operator (`::=`) to connect a target sort with a list of terminals or non-terminals. After that, $\mathbb{K}$ automatically generates a `kLabel` name (having the sort *KLabel*) representing the constructor and a sort *KList* term representing the argument list of the construct. Inside the bracket, $\mathbb{K}$ allows users to define attributes, some of which have semantic meanings. For example, the `strict(1)` attribute generates a pair of `heat`/`cool` rules for the first non-terminal position of the construct. Two features that $\mathbb{K}$ uses to keep object language specifications succinct are localization and concision. Localization means to allow 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 interpreted over a configuration, mentioning only those components accessed or altered by the rule. Figure 3.4 `(b)` is a very simplified version of the **K-LLVM** and **Kaskell** configuration. The configuration of a language is an algebraic structure of the program states, organized as nested labeled cells, in XML formats, holding semantic information, including the program itself. While the order of cells is irrelevant in a configuration (having a *Bag* sort in $\mathbb{K}$), the contextual relations between cells are relevant and must be preserved by the rules defined by the users and subsequently "completed" by the compilation step in $\mathbb{K}$ according to the configuration. Leaf cells represent pieces of the program state, like a computation stack or continuation (e.g., `k`), environments (e.g., `env`), heaps (e.g., `heap`), etc. A typical rule for reading a variable would be as in Figure 3.4 `(c)`. There are three cells in the rule: `k`, `env` and `heap`. The content of the `k` cell symbolizes a computation sequence waiting to be performed, while the head element in the cell represents the next item to be computed. The `env` cell contains a map of variables to location numbers, while the cell `heap` is a map of location numbers to expression values. The meaning of the above rule 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 that value, E; the horizontal line represents a transition. A cell with no horizontal line means that it is read but not changed during the transition.

Concision in rule Figure 3.4 `(c)` in $\mathbb{K}$ refers to the "$\cdots$" operator, which represents portions of cells that are irrelevant, and it could have different types depending on the context. This unconventional notation including its two features is useful in terms of allowing users to write less. The rule in Figure 3.4 `(c)` would be written out as the traditional rewrite rule (also allowed in $\mathbb{K}$) shown in Figure 3.4 `(e)`, which still relies on the $\mathbb{K}$ configuration, but without localization and concision. We need to add the top cell `T` and a variable B with its sort *Bag* in the rule to indicate irrelevant program state pieces. Computations in the `k` cell are separated by "$\curvearrowright$" (a built-in sort *KItem* list concatenation operator in $\mathbb{K}$), which is now observable. κ, ρ_1, ρ_2, ρ_3, and ρ_4 take the place of "$\cdots$". The most important thing to notice is that many parts of the rule are duplicated on the right-hand side. Duplication in a definition can lead to subtle semantic errors if users are not careful to synchronize changes in their specifications in multiple places. In a big language like C, Java or LLVM IR, the configuration structure is very complicated, and requires including more cells than a typical rule needs to mention. These intervening cells are automatically inferred in $\mathbb{K}$, which keeps the rules more succinct. Figure 3.4 `(d)` shows another way of defining rules in $\mathbb{K}$. A rule that does not mention any cell structure means that it matches the content of the head of the `k` cell. X:*Int* in the rule means that a variable X has sort *Int*. The keyword *requires* is a way of introducing a condition expression in a rule. Figure 3.4 `(f)` shows an equation in $\mathbb{K}$. An equation forms an equivalence relation for two groups of ground terms represented by the parameterized terms on the two sides of the equation. In the example, we equate the ground terms represented by `exp(` X `)` with those represented by `aExp(` X `)`.

Modularity is another important feature of $\mathbb{K}$. Its module system can be classified as a set of separate files whose contents might not be related. In fact, the rules in Figure 3.4 could be put in a single module by adding a module name. In defining specifications, users usually do not need to modify the existing rules to add a new feature to the language. $\mathbb{K}$ maintains this feature by structuring its configuration as nested cells and by allowing users to design their specification rules by only mentioning the cells that are needed in those rules, indeed only the portions of those cells. For example, the above rule only refers to the `k`, `env`, and `heap` cells, while the entire configuration contains other cells (Figure 3.4 `(b)`). The modularity of $\mathbb{K}$ not only allows users to create a compact and humanly readable specification, but also speeds up the semantics development process. For example, the above lookup rule does not change, even though a new cell is added to the configuration to support a new feature. The modular system of 𝕂 also allows users to develop syntax and rules incrementally by defining a syntactic construct in one 𝕂 module with rules containing the construct in different 𝕂 modules.

Another important feature of 𝕂 is the inherent support of non-determinism. 𝕂 is based on rewriting logic [134], so users can easily define, execute, and reason about non-deterministic specifications in 𝕂. One example is the execution of multi-threaded programs in **K-LLVM** [135]. The execution of multi-threaded programs in **K-LLVM** results in two different `thread` cells, which represent two thread processes with their environment information. Typically, when we have two threads, we need to talk about the multi-threaded behaviors that occur when interleaving threads. By using the ***ksearch*** command in 𝕂, one can see the final results of executing the `Thread-1` and `Thread-2` programs with interleaving. By setting a proper step number and flag for the ***ksearch***, one can also get the results of all traces of interleaved threads. In addition, by using the ***krun*** command, 𝕂 fixes an order (the lexicographical order of the rule text) and produces a single trace for executing a **K-LLVM** program. In **K-LLVM**, We can use ***ksearch*** to collect the set of all final results, compare it with the set of expected results defined by the Pthread library, and see if the second one contains the first one.

Chapter 4: THE SEMANTICS OF $\mathbb{K}$ AND ITS TRANSLATION TO ISABELLE

We introduce the semantics of $\mathbb{K}$ (**IsaK**) and a translation procedure (**TransK**) from language specifications defined in **IsaK** to transformed specifications in Isabelle. $\mathbb{K}$ is a domain specific language that takes a language specification as an input and generates an interpreter for the specification, including an execution engine to show trace behaviors of a program in the specification. There is a rich body of published work on $\mathbb{K}$ itself [136], and specifications given in $\mathbb{K}$, such as the Java semantics [86], the Javascript semantics [85], the PHP semantics [84] and the C semantics [83, 137]. Despite the success of $\mathbb{K}$, there are issues. While there have been a number of papers published concerning theorems related to $\mathbb{K}$, there is no source sufficiently complete to define the complete syntax and semantics of $\mathbb{K}$, or allow for rigorous proofs of properties of the languages defined in $\mathbb{K}$. In addition, while $\mathbb{K}$ supports specific tools for analyzing programs in a language defined in $\mathbb{K}$, it provides very little support for formal reasoning about the languages it defines. Finally, the fact that early versions of $\mathbb{K}$ had features that were dropped in intermediate versions, only to be reintroduced in the latest versions, and different versions have displayed different behaviors unveils the fact that researchers in the $\mathbb{K}$ community do not have a consensus on what $\mathbb{K}$ is. As an answer for these problems, we have created a complete formal semantics of $\mathbb{K}$ clarifying different aspects and features of $\mathbb{K}$.

Our contribution, a full, formal language specification of $\mathbb{K}$, called **IsaK**, addresses these concerns and forms the foundation of tools for the maintenance, revision, and expansion of $\mathbb{K}$. We also define a shallow embedding of $\mathbb{K}$ into Isabelle (Sec. 4.5), named **TransK**, and prove that the embedded $\mathbb{K}$ specification in Isabelle bi-simulates it original $\mathbb{K}$ specification in **IsaK** (Sec. 4.6) for any specification defined in $\mathbb{K}$, so we now can define a language specification in $\mathbb{K}$ and prove theorems about the specification in Isabelle. See Sec. 4.2 and 4.3.2 for more details.

Several benefits accure from our work. To the best of our knowledge, **IsaK** is the *first complete semantics* of $\mathbb{K}$. Other than the two simple descriptions of $\mathbb{K}$ [76] and [138], there are no resources talking about its syntax or semantics. Indeed, all $\mathbb{K}$ implementations contain some undesirable behaviors, so it is hard for us to learn the exact meanings of $\mathbb{K}$ operators. In the process of defining $\mathbb{K}$, we needed to constantly interview the $\mathbb{K}$ team to understand the meanings of the $\mathbb{K}$ operators and look at the Java source code of the $\mathbb{K}$ implementation to understand how $\mathbb{K}$ was being defined, which was a time-consuming task.

To the best of our knowledge, **TransK** is the *first translation process* from an order-sorted algebraic system ($\mathbb{K}$) to a many-sorted one (Isabelle). Previously, we have defined the general concepts for translating order-sorted terms into many-sorted ones [11]. **TransK** is the first complete formation of these general concepts on real-world systems ($\mathbb{K}$ and Isabelle). The result of translating specifications defined in $\mathbb{K}$ into Isabelle theories is the sudden marriage of the programming language field and the theory proving field. Now, we are able to define a formal language specification and learn about the formal meaning of the language in **IsaK**, then translate it into an Isabelle theory through **TransK** and prove inductive theories about the whole language in Isabelle. Before **TransK**, no $\mathbb{K}$ tools were able to handle this job. We want to define language specifications in $\mathbb{K}$ because of how much faster it is than defining them in Isabelle directly, and the several complete real-world specifications [83, 84, 85, 86, 137] in $\mathbb{K}$ are the proof. We want to prove theorems about such specifications in the theory translated by **TransK** because it is a lot simpler and clearer than the representations of these specifications in **IsaK**. See Sec. 4.5.2 for one example. All **IsaK** and **TransK** programs and theorems have been formalized and proved in Isabelle in the following link: `https://github.com/liyili2/KtoIsabelle`, and the implantation of **TransK** is defined as a Java and Ocaml combined program.

4.1 A BREIF OVERVIEW OF **ISAK**

We briefly discuss $\mathbb{K}$'s current semantic layout. The formal semantics as it is presented in **IsaK** is divided into two parts: static and dynamic semantics. The static semantics takes as input the frontend-AST (FAST) representation of a user-defined language specification ($\mathbb{K}$/**IsaK** theory) or programs that are allowed by the specification. Through the translation process in the static semantics, which performs computations that can be done statically (referred to as compile-time operations), the $\mathbb{K}$/**IsaK** theory in FAST is processed and translated into a representation in backend-AST format (BAST). Then the type checking step in the static semantics outputs a type-checked BAST, which is passed to the dynamic semantics for execution. We first discuss some features of $\mathbb{K}$ through an example below.

4.1.1 An Example $\mathbb{K}$ Specification (Theory)

Here, we introduce $\mathbb{K}$ as a language specification platform with the ability to automatically generate an interpreter for a specification. The operational behavior of the $\mathbb{K}$ specification language contains four major steps: parsing, language compilation, sort checking, and semantic rewriting. Parsing itself comes in two phases: one to learn the grammar of the object language (the programming language being defined), and the second to incorporate that grammar into the grammar of $\mathbb{K}$ for parsing the definitions of the rules and semantic objects defining the executable behavior of programs in the object language. These parsers translate the concrete syntax for both $\mathbb{K}$ and the object languages defined therein into concrete syntax, eliminating mixfix syntax and other syntactic sugar in the process. The language compilation, sort checking and semantic rewriting are the major steps in **IsaK**. Here, we briefly introduce what a $\mathbb{K}$ specification (theory) looks like.

Syntax

SYNTAX Exp ::= Var | Int | Exp / Exp `[strict]` | ... SYNTAX KResult ::= Int | Bool
SYNTAX BExp ::= Bool | Exp < Exp `[strict]` | BExp && BExp `[strict]` | ...
SYNTAX Stmt ::= Bloc | Var := Exp `[strict(2)]` | `if` (BExp) Bloc Bloc `[strict(1)]` | `while` (BExp) Bloc
| Stmt ; Stmt | Var := `thread`(Stmt)
SYNTAX Bloc ::= `{}` | `{`Stmt`}` SYNTAX Prog ::= `int` Vars ; Stmt SYNTAX Vars ::= `list{Var, ","}`

Configuration

$$\left\langle \langle\langle\langle 0 ::\mathsf{Int}\rangle_{\mathsf{key}}\ \langle \$pgm::\mathsf{Prog}\rangle_{\mathsf{k}}\ \langle \texttt{.Map}\rangle_{\mathsf{env}}\rangle_{\mathsf{thread*}}\rangle_{\mathsf{threads}} \atop \langle 0\rangle_{\mathsf{count}}\ \langle \texttt{.Map}\rangle_{\mathsf{heap}}\ \langle \texttt{SetItem(0)}\rangle_{\mathsf{keys}} \right\rangle_{\mathsf{T}}$$

Rules

(a) $\langle (x::\mathsf{Var} \Rightarrow e) \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ x \mapsto n\ \cdots\rangle_{\mathsf{env}}\ \langle \cdots\ n \mapsto e\ \cdots\rangle_{\mathsf{heap}}$ (b) $x::\mathsf{Int}\ /\ y::\mathsf{Int} \Rightarrow x\ \texttt{/Int}\ y$ `when` $y \neq 0$

(c) $\langle (\texttt{int}\ (x\ ,\ xs \Rightarrow xs)\ ;\ \_) \cdots\rangle_{\mathsf{k}} \langle \cdots\ x \mapsto n\ \cdots\rangle_{\mathsf{env}}$
$\langle M \Rightarrow M[\,0\,/n]\rangle_{\mathsf{heap}} \langle n \Rightarrow n\ \texttt{+Int}\ 1\rangle_{\mathsf{count}}$

(d) $\langle \cdots\ \langle (x := \texttt{thread}(t)\ \Rightarrow\ \texttt{.K}) \cdots\rangle_{\mathsf{k}}\ \langle M \Rightarrow M[n/x]\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{thread}}$
$(\texttt{.Bag} \Rightarrow \langle \cdots\ \langle t\rangle_{\mathsf{k}} \langle M\rangle_{\mathsf{env}} \langle \texttt{fresh}(S,\ 0)\rangle_{\mathsf{key}} \cdots\rangle_{\mathsf{thread}})$
$\langle \cdots\ n \mapsto (\_ \Rightarrow \texttt{fresh}(S,\ 0)) \cdots\rangle_{\mathsf{heap}}\ \ \langle S\rangle_{\mathsf{keys}}$

(e) $(\langle \cdots\ \langle key\rangle_{\mathsf{key}} \langle \texttt{.K}\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{thread}} \Rightarrow \texttt{.Bag})\ \ \langle S \Rightarrow S \backslash \{key\}\rangle_{\mathsf{keys}}$

Function and Function Rules

SYNTAX Int ::= `fresh`(Set , Int) `[function]`
(f) $\texttt{fresh(.Set,}\ n) \Rightarrow n\ \texttt{+Int}\ 1$ (g) $\texttt{fresh(SetItem(}n::\mathsf{Int})\ S,\ m) \Rightarrow \texttt{fresh}(S,\ n)$ `when` m `<Int` n
(i) $\texttt{fresh(SetItem(}n::\mathsf{Int})\ S,\ m) \Rightarrow \texttt{fresh}(S,\ m)$ `when` m `>=Int` n

Heating/Cooling Rule Example

(heat) $x\ /\ y \curvearrowright tl \Rightarrow x \curvearrowright \square\ /\ y \curvearrowright tl$ `when` $\neg\texttt{isKResult}(x)$ (cool) $v::\mathsf{KResult} \curvearrowright \square\ /\ y \curvearrowright tl \Rightarrow v\ /\ y \curvearrowright tl$

Expanded Configuration Rule

(j)
$$\left\langle C_1::\mathsf{Bag}\ \langle\langle\langle x \curvearrowright \kappa\rangle_{\mathsf{k}}\ \langle \rho_1, x \mapsto n, \rho_2\rangle_{\mathsf{env}}\ C_2::\mathsf{Bag}\rangle_{\mathsf{thread}}\rangle_{\mathsf{threads}} \atop \langle \rho_3, n \mapsto e, \rho_4\rangle_{\mathsf{heap}} \right\rangle_{\mathsf{T}}$$
$$\Rightarrow \left\langle C_1::\mathsf{Bag}\ \langle\langle\langle e \curvearrowright \kappa\rangle_{\mathsf{k}}\ \langle \rho_1, x \mapsto n, \rho_2\rangle_{\mathsf{env}}\ C_2::\mathsf{Bag}\rangle_{\mathsf{thread}}\rangle_{\mathsf{threads}} \atop \langle \rho_3, n \mapsto e, \rho_4\rangle_{\mathsf{heap}} \right\rangle_{\mathsf{T}}$$

An Example Execution In IMP

Program: `int` x ; x `:= 1`

Initial Configuration:
$$\left\langle \langle\langle\langle 0\rangle_{\mathsf{key}}\ \langle \texttt{int}\ x\ ;\ x\texttt{:= 1}\rangle_{\mathsf{k}}\ \langle \texttt{.Map}\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}}\rangle_{\mathsf{threads}} \atop \langle 0\rangle_{\mathsf{count}}\ \langle \texttt{.Map}\rangle_{\mathsf{heap}}\ \langle \texttt{SetItem(0)}\rangle_{\mathsf{keys}} \right\rangle_{\mathsf{T}}$$

Configuration After One Step (Rule (c)):
$$\left\langle \langle\langle\langle 0\rangle_{\mathsf{key}}\ \langle \texttt{int .Ids ;}\ x\texttt{:= 1}\rangle_{\mathsf{k}}\ \langle x \mapsto 0\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}}\rangle_{\mathsf{threads}} \atop \langle 1\rangle_{\mathsf{count}}\ \langle 0 \mapsto 0\rangle_{\mathsf{heap}}\ \langle \texttt{SetItem(0)}\rangle_{\mathsf{keys}} \right\rangle_{\mathsf{T}}$$

Figure 4.1: A Summary of $\mathbb{K}$ By IMP

Fig. 4.1 contains a small language specification, named IMP, with most of its syntax and some semantic definitions. In IMP, all program variables are heap ones that can be shared through different threads. In $\mathbb{K}$, the keyword SYNTAX introduces a finite set of syntactic definitions, separated by "|", such as the definition of the sort Exp. Each syntactic definition is a list of names. The names in Sans serif font are non-terminals (sorts), while the names in `tt` font are terminals. A syntactic definition (e.g. Exp ::= Var) introducing only a singleton sort defines a subsort relation subsorting the singleton sort (Var) to the target sort (Exp). The subsort definition that subsorts sorts to KResult defines the evaluation result sorts in a specification. Other kinds of syntactic definitions introduce syntactic definitions as user defined terms used to express rules and programs. Every real syntactic definition (not subsorting) creates a prefix AST format like $\mathcal{KLabel}$(*KList*), where the $\mathcal{KLabel}$ term acts as the constructor automatically generated from the terminals and the structure of the definition, and the $\mathcal{KList}$ term is the argument list generated from the non-terminals of the definition. $\mathbb{K}$ allows users to define attributes (written in brackets e.g. `[strict]` in Fig. 4.1), some of which have semantic meanings. For example, the `strict(2)` attribute (in the definition: Var := Exp `[strict(2)]`) means to generate a pair of heating/cooling rules for the second non-terminal position of the term created by the definition. We show an example pair of heating/cooling rules for the first non-terminal position of the "`/`" operator in Fig. 4.1. The `[strict]` attribute without any numbers indicates there is a pair of heating/cooling rules generated for each non-terminal position in the definition. "$\curvearrowright$" is a list concatenation operator for connecting the computation sequence in a `k` cell, while "$\square$" is a special builtin operation in $\mathbb{K}$ representing the removal of a redex subterm from a term and the creation of a "hole" waiting to be filled. The syntax definitions in a $\mathbb{K}$ theory are compiled by the **IsaK** static semantics (Sec. 4.2) into a sort set, a symbol table, a subsort relation and several heating/cooling rules as inputs for the **IsaK** dynamic semantics in Sec. 4.3.

The initial configuration of a specification is an algebraic structure of the program states, organized as nested, labeled cells, in XML formats that hold semantic information, including the program itself (prefixing by the \$ operator in Fig. 4.1). While the order of cells in a configuration is irrelevant, the contextual relations between cells are relevant and must be preserved by rules defined by users and subsequently "completed" in the compilation step in $\mathbb{K}$ according to the configuration. In a trace evaluation, each step of computation should produce a result state that "matches" the structure of the initial configuration, meaning that the cell names, sorts of cells, and structural relations of the cells are the preserved in the result states and the initial configuration. Leaf cells represent pieces of the program state, like computation stacks or continuations (e.g., `k`), environments (e.g., `env`), heaps (e.g., `heap`), etc. The content in each cell in an initial configuration has dual roles. It is the initial value defined for the computation and also defines the sort of the cell content. For example, the `key` cell in the IMP configuration in Fig. 4.1 is defined as `0` and sort Int; during an evaluation, the cell's initial value is `0`, and in every state of the evaluation, its content has sort Int. The last part of Fig. 4.1 provides an example program in IMP combined with its initial configuration. After evaluating the initial configuration by rule `(c)` in IMP, the contents of several cells are updated, but the structure relations and sorts of the cells do not change during the evaluation.

Fig. 4.1 also contains a set of IMP rules. The simplest form of rules, such as rule `(b)`, describe behaviors that can happen in the first element position in a `k` cell, without mentioning any cells and without mentioning the tail of the computation list of the `k` cell. A little more complicated form of rules, such as rules `(heat)` and `(cool)`, mention the tail of the computation list (connecting by "$\curvearrowright$"). They describe behaviors that can happen in a `k` cell, especially the relationship among different positions in the computation list. In BAST format, these two kinds of rules are compiled to the same form (K rules). The most complicated form of rules, such as rule `(a)`, are typical configuration rules in $\mathbb{K}$ and they describe interactions among different device components in a system. For example, rule `(a)` is reading from a value in the main memory (the `heap` cell) for a variable in the `k` cell through an local stack (`env`). The "$\cdots$" operator in these rules represents portions of cells that are irrelevant. This unconventional notation allows users to write less.

In this thesis, we focus on the dynamic semantics of $\mathbb{K}$. All these unconventional configuration rules are assumed to be compiled to a standard form (BAST form) by the **IsaK** static semantics (Sec. 4.2), and the dynamic semantics is defined based on the compiled format. For example, the rule `(a)` is compiled to rule `(j)`. In translating `(a)` to `(j)`, we would need to add the cells `T`, `thread`, `threads` in rule `(j)` and the variables C_1 and C_2 with its sort Bag, to indicate the irrelevant program state pieces. Computations in the `k` cell are separated by "$\curvearrowright$", which is now observable in `(j)`. The κ and ρ_1, ρ_2, ρ_3, and ρ_4 fill in the place corresponding to the "$\cdots$" in rule `(a)`.

In $\mathbb{K}$, configuration rules are also powerful enough to manipulate language device resources. For example, rules `(d)` and `(e)` create or finish a thread by adding or deleting a `thread` cell. These are handled by rewriting an empty Bag place (`.Bag`) to a new cell `thread` (rule `(d)`), or rewriting a `thread` cell to an empty place (rule `(e)`). In $\mathbb{K}$, this is allowed only if the specific cell in the initial configuration (e.g. the configuration in Fig. 4.1) is marked as "$*$".

$\mathbb{K}$ also allows users to write equational rules, named function rules. The format is like the `fresh` definition in Fig. 4.1. Its syntactic definition (e.g. `fresh`) is labeled by an attribute `function`, and then the rules whose left-hand top-most constructor is the same as the $\mathcal{KLabel}$ term syntactic definition are recognized by $\mathbb{K}$ to be the function rules under the function definition. The left-hand-side of a valid function rule has argument sorts subsorting to the argument sorts defined in the function definition, and the target sort of the right-hand-side subsorts to the target sort of the definition. All these rules are compiled to standard BAST forms (described in Sec. 4.3.1) and stored in a rule set as the input for **IsaK** dynamic semantics.

4.1.2 Challenges of Defining $\mathbb{K}$ Specification

Several formidable challenges are faced by the **IsaK** project. First, other than the two simple descriptions of $\mathbb{K}$ [76] and [138], there are no resources talking about its syntax and semantics, as we have mentioned in the beginning of the Chapter.

Second, the 𝕂 implementations usually contain a front-end language and a back-end language that perform different tasks. The allowed syntactic definitions for users in the 𝕂 front-end is strictly larger than the allowed syntactic definitions in the 𝕂 back-end. More precisely, there are some constructs and semantic rules in 𝕂 that users think they can define but are in fact not supported by 𝕂. 𝕂 implementations sometimes produce no error messages or warnings about these limits, so users have no way to figure out if there is something wrong in their specifications or there are some problems in 𝕂. For example, in the configuration [86], we can see that the `class` cell is associated with the key word *, and in the `class` cell, the `methodDec` cell is also labeled with the key word *. This means that it can have multiple `class` cells and `methodDec` cells when we define a Java rule or evaluate a Java program by using this rule in 𝕂. When we interpret Java programs in 𝕂, we find that a rule mentioning two nested cells both having the key word * is actually not valid in 𝕂, even if someone can define two such cells nested together in a rule. If users define this kind of rule in their specifications and use ***krun*** to run the testing programs, once the program triggers the rule, ***krun*** crashes immediately without giving any valid error messages. More surprisingly, if users write a Java rule to add a method definition to a specific class (a cell with *), and run their testing programs, when ***krun*** triggers the rule the first time, it works, but it fails the second time in the current 𝕂 implementation. Users will have no clue what is going on here. The problem is a poor design decision made by the 𝕂 team. In their early 𝕂 implementation in Maude, the nested cell feature was supported. When they implemented 𝕂 in Java, they decided not to support this feature because it would slow down the generated interpreter for a language specification in 𝕂. However, since some big languages such as Java have used this feature, they decided to partially support it in their Java implementation, but gave no information on the boundaries of what is and is not allowed for it.

Third, the path compiling from the front-end language in 𝕂 to the back-end one is not so clear. In the implementation of 𝕂 4.0 (in Java), there are 48 compilation steps to compile the front-end language to the back-end one. These 48 steps have different tasks. To understand the different tasks, and combine all of them in **IsaK** is a tough job to do.

Fourth, one of the best features of 𝕂 is the modularity system, but it is also one of the hardest to understand in 𝕂. Resolving the modularity in each rule is a compilation step in the 𝕂 implementation. The basic idea of the compilation step is to take the configuration in a language specification, compare it with a given rule, and fill the missing pieces in the rule to make the rule "complete". The problem is that adding the missing pieces is not so trivial. For example, there are two ways to define a rule for removing the existence of all holding locks in a thread based on the Java configuration [86] in Fig. 4.2.

At first glance, people might think that the left-hand side rule (Rule 1) and the right-hand

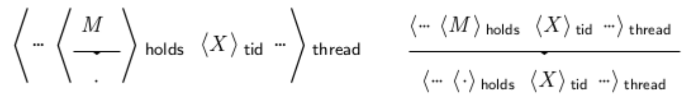


Figure 4.2: An Example Configuration From The Java Semantics [86]

side rule (Rule 2) are the same in Fig. 4.2, but they are not. Rule 1 means that in a given thread with id X, we remove all its holds. Rule 2 means that for a given thread with id X, except the `tid` cell, we discard all the program states in the thread and initialize them with the ones in the initial configuration, such as the one in Figure 4.1. Specifically, we remove all locks in the `holds` cell. The main problem here is that the "⋯" is not a simple syntactic sugar when it is associated with $\mathbb{K}$ cells. The compilation of the "⋯" in the $\mathbb{K}$ `cells` level desires a well-defined algorithm to accomplish this problem properly.

Nevertheless, even if these challenges are many and hard, **IsaK** is defined without compromise and includes every feature of $\mathbb{K}$.

4.2 THE STATIC SEMANTICS OF **ISAK**

We briefly introduce the static semantics of $\mathbb{K}$ in this section and will discuss some design issues of $\mathbb{K}$, especially $\mathbb{K}$ modularity, localization and concision features in the next section. The static semantics of $\mathbb{K}$ describes how we compile away the localization, concision and modularity features of $\mathbb{K}$ to a uniformed backend AST. It contains several phases, as listed in Figure 4.3. Each phase digs deeper into the syntactic structure of $\mathbb{K}$ and either performs a set of transformations over the user-defined $\mathbb{K}$ specifications or applies some checks on the input FAST of the specifications.

We assume that there is an external parser that parses user-input $\mathbb{K}$ object language specifications and object level programs to a FAST format. The parser is divided into two phases. In the first phase, it uses ocamllex and ocamlyacc (variants of lex and yacc for Ocaml) to read all syntactic definitions in a given specification, and then generates a symbol table based on the syntactic definitions. In the second phase, it uses the symbol table to generate lexers and parsers in the formats of ocamllex and Dypgen (a general LR parser) to parse rewrite rules and programs for the specification. The two-phase parser is a direct copy of the $\mathbb{K}$ parser (SDF-to-K adapter [139]) and is intended to be suitable for the OCaml-based $\mathbb{K}$ implementation extracted directly from **IsaK** in Isabelle.

After the parsing, the static semantics takes as input the FAST representation of a user-defined language specification or programs that are allowed in the specification. Through the translation process in the static semantics, which performs computations that can be done statically (referred to as compile-time operations), the specification in FAST is processed and translated into a representation in BAST. Then the sort adjustment step in the static semantics outputs a sort-adjusted BAST, which is passed to the dynamic semantics for execution.

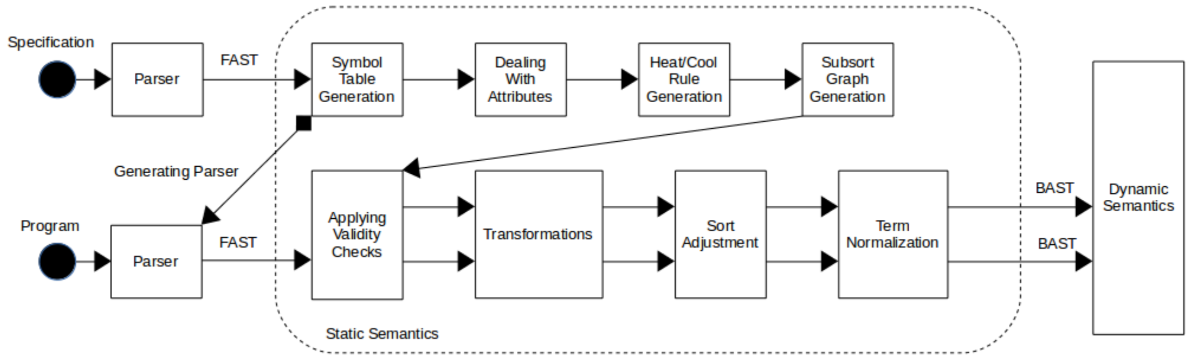


Figure 4.3: The structure of **IsaK**

Symbol Table Generation. In this phase, a symbol table is acquired from the syntactic definitions of a object specification, a database is formed for later phases to use, and a program parser is generated to parse object level programs. This symbol table is accumulated across all modules.

Dealing with Attributes and Heat/Cool Rules Generation. $\mathbb{K}$ provides syntactic and semantic attributes to allow for more succinct specifications, which are part of the concision feature and allow users to write less. For example, if we define an division operator as Figure 4.1 `(a)`, the `strict` attribute associated with the above syntactic definition means that a pair of `heat` and `cool` rules for the first non-terminal argument position is generated in Fig. 4.4.

$$\frac{X : Exp \ / \ Y : Exp}{X : Exp \curvearrowright (\square \ / \ Y : Exp)} \qquad \frac{X : KResult \curvearrowright (\square \ / \ Y : Exp)}{X : KResult \ / \ Y : Exp}$$

$$requires \neg \ \texttt{isKResult(} \ X \ \texttt{)}$$

Figure 4.4: Heat/Cool Rule Examples

The rule on the left (Fig. 4.4) takes a term $X{:}\mathit{Exp}$ `/` $Y{:}\mathit{Exp}$, and splits it into a redex $X{:}\mathit{Exp}$ and a context $\square + Y{:}\mathit{Exp}$, provided that the term $X{:}\mathit{Exp}$ is not a subsort of *KResult*. The rule on the right merges a *KResult* redex term $X{:}\mathit{KResult}$ and a context $\square + Y{:}\mathit{Exp}$ back to a term $X{:}\mathit{KResult} + Y{:}\mathit{Exp}$. As shown in the example in the previous paragraph, specifying a `strict` attribute in $\mathbb{K}$ is the same as generating a pair of `heat/cool` rules. The `heat/cool` rules in $\mathbb{K}$ work by separating context and redex in the evaluation context framework. A `heat` rule splits a term into a redex and a context with a hole ($\square$) in the specified position, and moves the redex to the front of the `kCell`, provided that the redex does not have a sort that is a subsort of *KResult*. A `cool` rule moves a *KResult* redex back into the $\square$ in the context and merges them into a term without $\square$. In the later section, we will see that the meaning of a `heat/cool` rule pair might change due to the extention of *KResult* sort in different modules of a $\mathbb{K}$ specification.

In this phase, we also take care of other syntactic attributes (associated with syntactic definitions) and semantic attributes (associated with rewrite rules) by grouping and collecting the parsed input language specification pieces. For example, if a syntactic definition is labeled with a `function` attribute, which is called a `function` operator (its `kLabel` name is called a `function kLabel`), we collect all rules across all modules whose left-hand side top-most `kLabel` name is the same as this definition, store all these rules in a set and label them as belonging to the function operator.

Subsort Graph Generation. In this phase, we collect all subsort information defined in a specification and form it into a graph. In $\mathbb{K}$, the only way to define a subsort relation is to use a syntactic definition as Fig. 4.5.

$$\text{SYNTAX} \quad \mathit{Exp} ::= \mathit{Int}$$

Figure 4.5: A Subsort Definition Example

In the example in Fig. 4.5, sort *Int* is defined to be a subsort of sort *Exp*. $\mathbb{K}$ has a very special subsort structure. First, the subsort relation in a $\mathbb{K}$ specification is antisymmetric and transitive. Second, the sorts *K*, *KLabel*, *KList*, *Set*, *List*, *Map* and *Bag* are $\mathbb{K}$ built-in sorts representing the respective $\mathbb{K}$ built-in terms. Users cannot define subsort relations involving these built-in sorts; otherwise, the specification is not well-formed. Third, the built-in sort *KItem*, representing the elements in a sort *K* (sort *K* means a list of *KItem*), is a subsort of the sort *K* implicitly, while all other user-defined sorts in a specification are subsorts of

KItem implicitly. Finally, users can define subsorts of the built-in sort *KResult* representing the evaluation result terms of an execution. However, those user-defined sorts that have not been defined to be subsorts of *KResult* through transitivity are implicit supersorts of *KResult*. Indeed, *KResult* is also a subsort of *KItem* implicitly.

We need to generate the subsort graph by combining the user-defined subsort relations and the implicit subsort relations above. In addition, we also need to apply checks, such as the well-formedness check above, to reject ill-formed specifications. Another important check is to see if there are cycles in the subsort graph, and reject those specifications as well.

Applying Validity Checks. In this phase, several checks are applied to a specification to rule out ill-formed specifications and programs. A lot of these checks are related to the improvement of $\mathbb{K}$ due to design issues in modularity, localization and concision features in Section 4.4.1. The first important check is to ensure that the `kLabel` names are used in a uniformly consistent manner. Second, users cannot define any new syntactic constructs or subsorts to the built-in sorts K, *KLabel*, *KList*, *Set*, *List*, *Map* and *Bag*, except with a `function` attribute. Third, at most one of the `function`, `strict` and
textttseqstrict attributes is allowed to appear in a syntactic definition. Without this restriction, a syntactic definition could be both a function and generate `heat/cool` rules, which is not sound in $\mathbb{K}$. Also, the specified natural number for a `strict` attribute cannot exceed the total number of non-terminal positions in the syntactic definition it is associated with. Finally, any two user-defined lists cannot have the same target sort. For example, the two syntactic definitions in Fig. 4.6 cannot appear in the same specification because they are adding two user defined lists with two different element sorts to the same target sort *Exp*.

SYNTAX *Exp* ::=`List{` *Int* `,","}`   SYNTAX *Exp* ::=`List{` *Bool* `,","}`

Figure 4.6: Forbidden Example List Definition in $\mathbb{K}$

Transformation. We will select some important transformation steps to discuss here. Interesting readers can refer to our technical report [10] to get more information. There are some rewrite rules and programs that users are able to write down in FAST, but they have no meanings in $\mathbb{K}$ because the semantics of $\mathbb{K}$ terms are only defined if the terms can be written in BAST and the transformation from FAST to BAST is a partial function. The process of transforming FAST to BAST is both a transformation and a narrowing process.

If a FAST term is transformable, we have a corresponding BAST term; if not, we reject the input specification or program.

$$X\ \texttt{+}\ \texttt{(}\ Y \Rightarrow \texttt{(}\ \texttt{-}\ \ Y\ \texttt{))} \Rightarrow \texttt{.K}$$

(a)

$$\left\langle \begin{array}{c} \langle \texttt{1 + 2 + 3} \rangle_{\mathsf{k}}\ \langle \texttt{.Bag} \rangle_{\mathsf{classes}} \\ \langle \texttt{.Map} \rangle_{\mathsf{env}}\ \langle \texttt{.Map} \rangle_{\mathsf{heap}} \end{array} \right\rangle_{\mathsf{T}}$$

(b)

Figure 4.7: Transformation Examples

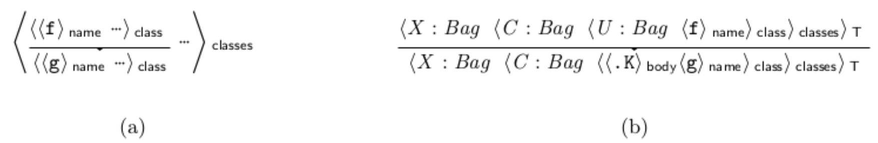
The first step in the transformation process is to rule out some terms permissible in FAST but invalid in BAST. For example, Figure 4.7 `(a)` shows a rule that will be rejected in the transformation process because its LHS contains nested $\mathbb{K}$ rewrite operators ($\Rightarrow$). Usually, a rule will be translated into a five tuple format that BAST adopts. In BAST, the five tuple includes the rule label field indicating the type of the rule for execution, the pattern representing the LHS of the rule, the expression representing the RHS of the rule, the condition expression representing the condition of the rule, and a boolean flag indicating if the rule is a transition rule, which determines if the rule is part of multi-threaded behaviors in an object specification. For example, if we have a rule in Figure 4.1 `(d)`, what we get in the BAST form is a tuple in Fig. 4.8.

$$\texttt{(KNormal,}\ \ X : \mathit{Int}\ \texttt{/}\ Y : \mathit{Int}\ \texttt{,}\ \ X : \mathit{Int}\ \texttt{/Int}\ Y : \mathit{Int}\ \texttt{,}\ \ Y : \mathit{Int} \neq \texttt{0, true)}$$

Figure 4.8: BAST FORM of A Symbol Table Entry

Recall in Section 3.4, we says that the configuration of an object language specification represents the program state that is necessary to describe the behaviors of the specification. Executing a program of a specification in $\mathbb{K}$ actually means that we apply semantic rules to a program state and generate a sequence of computations (or a set of sequences if we want to see multi-threaded behaviors). The generation of the initial program state is just to combine an object program with the configuration together. For example, if we define the configuration in Figure 4.1 `(b)`, and our program is `1 + 2 + 3`, the initial state is generated as Figure 4.7 `(b)`. The cell `class` is replaced with a term `.Bag` as prescribed by the configuration because it is attributed with * keyword in the configuration which indicates that it should start with zero occurrences.

Before we transform a semantic rule in FAST to the tuple format in BAST as we described in Fig. 4.8, if the rule contains sort *Bag* terms (representing cells or the configuration), we need to perform a step called configuration concretization. It is the most difficult part of the transformation, because it mixes the localization and concision features together. The solution requires a combined process of splitting, "completing" and translating terms of sort *Bag*. The goal of the configuration concretization step is to regularize each rule involving sort *Bag* terms to a form where it only involves one rewrite operator, and the rule is spitted into a LHS and a RHS which have no rewrite operators inside them, and both the LHS and RHS are completed by filling those missing cells according to the configuration.



$\left\langle \dfrac{\langle\langle \mathtt{f}\rangle_{\mathsf{name}} \cdots\rangle_{\mathsf{class}}}{\langle\langle \mathtt{g}\rangle_{\mathsf{name}} \cdots\rangle_{\mathsf{class}}} \cdots \right\rangle_{\mathsf{classes}}$
$\dfrac{\langle X:Bag\ \langle C:Bag\ \langle U:Bag\ \langle \mathtt{f}\rangle_{\mathsf{name}}\rangle_{\mathsf{class}}\rangle_{\mathsf{classes}}\rangle_{\mathsf{T}}}{\langle X:Bag\ \langle C:Bag\ \langle\langle .\mathtt{K}\rangle_{\mathsf{body}}\langle \mathtt{g}\rangle_{\mathsf{name}}\rangle_{\mathsf{class}}\rangle_{\mathsf{classes}}\rangle_{\mathsf{T}}}$
(a)
(b)

Figure 4.9: Configuration Concretization Example

As we have seen in Section 3.4, the key difficulty is that the combination of a "⋯" operator in a sort *Bag* term and a rewrite operator cannot be understood as simple syntactic sugar for writing less cell information and term rewriting from left to right. The detail of the transformation algorithm is listed in the technical report [10]. There are two main tasks. First, the algorithm needs to compare a *Bag* rule with the configuration, find cells containing "⋯" operators in the *Bag* rule and replace those operators with correct values based on the configuration. Second, we need to split a *Bag* rule to have clear left-hand and right-hand sides. Figure 4.9 shows an example displaying the most important features of the transformation. We first locate all the rewrite operators ($\Rightarrow$) and their subterms in a rule, and take the left-hand sides of these ($\Rightarrow$) terms as patterns and right-hand sides as expressions. If there is a "⋯" in a pattern, we replace it with a variable with the correct sort indicated by the configuration, like the U:*Bag* variable in Figure 4.9 `(b)`. For each "⋯" operator in an expression, we replace it with the correct terms according to the configuration, like the <`.K`> in Figure 4.9 `(b)`. After finishing filling patterns and expressions, we put patterns in the pattern side and expressions in the expression side of the generated rule, and compare both sides and the remaining pieces with the configuration, and fill the gaps with variables or corresponding cells, like the cells `classes`, `T`, C:*Bag* and X:*Bag* in Figure 4.9 `(b)`.

Type Checking. The type checking step should probably be named "sort checking" since it checks sort correctness for terms. There is a small difference between a $\mathbb{K}$ theory and an **IsaK** theory. There are five different kinds of rules that users are allowed to define in a BAST-formed $\mathbb{K}$ theory: function, `K`, configuration, anywhere, and macro rules. With these rules in a theory, the type system in $\mathbb{K}$ does not guarantee the type preservation property. To make a type-safe system, we disallow the anywhere and macro rules in an **IsaK** theory. By dropping these two kinds of rules, the type system for **IsaK** becomes very simple: a type system appearing in simple-type λ-calculus with subtyping. We show type preservation property for the type system (see Sec. 4.3.2).

Term Normalization. Normalization is a process that happens after sort adjustment, to apply idempotent and functional equational rules to any subterms of rules and program states having the sort *Set* and *Map*. The process is to get rid of redundant child elements of *Set* or *Map* subterms and make sure every *Map* term is functional. Normalization also happens after sort checking when doing a pattern matching in the dynamic semantics.

4.3 THE DYNAMIC SEMANTICS OF **ISAK**

Here, we introduce the **IsaK** dynamic semantics based on the BAST term transformed from the static semantics in Sec. 4.2. Given an **IsaK** theory and a program belonging to the theory, the **IsaK** dynamic semantics produces a trace of the program execution according to the semantic rules in the theory. In the rest of the paper, we will name an **IsaK** theory to mean an **IsaK** theory in BAST.

To facilitate the presentation, some useful relations derived from a relation (R) include the reflexive ($R^?$), transitive (R^+), and reflexive-transitive (R^*) closures. $[A]$ is the identity relation for a set A. $A \times B$ is the cross product of sets A and B, and $\times$ binds tighter than $\cup$. Some notation conventions in the paper are provided in Fig. 4.10 and 4.12. For example, s ranges over sorts and c ranges over $\mathscr{KLabel}$, etc. In these figures, every name in $\mathscr{Chancery}$ font is a type in Isabelle we defined for an **IsaK** component; every name in $\mathsf{Sans\text{-}serif}$ font is a sort or configuration (of type $\mathscr{CName}$) in a **IsaK** theory; everything in `tt` font is a construct programmers can use in **IsaK**, including constructors and terms, and everything in *Italics* is a variable representing a term in **IsaK**.

4.3.1 **IsaK** Sorts and BAST Syntax

We first introduce the syntactic formulation of a given **IsaK** theory in BAST before we introduce the semantics of evaluating a program for a theory. Syntactically, every **IsaK** theory is expressed as a tuple of $(\Psi, \sqsubseteq, \Upsilon, \Delta)$, where Ψ is a set of sort names and $(\Psi, \sqsubseteq)$ is a poset, Υ is a symbol table and Δ is a set of $\mathscr{Rule}$ terms, which will be introduced later in the section. $\sqsubseteq$ is a subsort relation built on pairs of sorts in Ψ. We have restrictions on Ψ and $\sqsubseteq$ in Fig. 4.10

Every sort is disjointly either a user-defined sort ($\mathscr{UsrSort}$) or a built-in sort ($\mathscr{BuiltinSort}$). Each sort in $\mathscr{ResultSort}$ is either Bool, or a user-defined sort that can be the sort of the result

Sorts

$\mathcal{SystemSort} \triangleq \{\mathsf{K}, \mathsf{KItem}, \mathsf{KList}, \mathsf{List}, \mathsf{Set}, \mathsf{Map}, \mathsf{Bag}\}$

$\mathcal{BuiltinSort} \triangleq \mathcal{SystemSort} \cup \{\mathsf{Bool}\}$

$\mathcal{RName} \subseteq \mathcal{UsrSort} \qquad \mathcal{BuiltinSort} \cap \mathcal{UsrSort} = \emptyset$

$\mathcal{ResultSort} \triangleq \mathcal{RName} \cup \{\mathsf{Bool}\}$

$s \in \Psi \triangleq \mathcal{UsrSort} \cup \mathcal{BuiltinSort}$

Sort/Subsort Sat Properties

$(\Psi, \sqsubseteq)$ is a poset

$\sqsubseteq\ \supseteq (\mathcal{UsrSort} \times \{\mathsf{KItem}\}) \cup \{(\mathsf{KItem}, \mathsf{K})\}$

$\forall s_1\ s_2.\ s_1 \in \{\mathsf{KList}, \mathsf{List}, \mathsf{Set}, \mathsf{Map}, \mathsf{Bag}\} \wedge s_1 \sqsubseteq s_2 \Rightarrow s_1 = s_2$

Figure 4.10: **IsaK** Sorts and Subsorts

of a computation, like Int (see Fig. 4.1). There are several restrictions on $\sqsubseteq$. For example, sort K is an upper bound of $\mathcal{UsrSort}$, while KItem is the supremum of the same set. The elements in $\{\mathsf{KList}, \mathsf{List}, \mathsf{Set}, \mathsf{Map}, \mathsf{Bag}\}$ are incomparable under $\sqsubseteq$; and $\mathcal{SystemSorts}$ are not result sorts.

Additionally, in the original $\mathbb{K}$, when a result sort is declared, the sort is subsorted to a special sort KResult. This formalization causes a problem in the type (sort) system soundness in $\mathbb{K}$: a term with a result sort can be rewritten to another result-sorted term, but the position holding the term is defined to hold one of the sorts but not the other one. For example, assume that x has value `true` in the heap, and we want to compute x/`1`. By applying rules `(heat)` and `(a)` in Fig. 4.1, the result is the term: `true` $\curvearrowright$ ($\square$/`1`). Since `true` is a KResult term, we can use rule `(cool)` in Fig. 4.1 to rewrite the term to `true/1`. This term is clearly ill-typed. In an evaluation in a $\mathbb{K}$ theory, this feature makes some rule applications result in type-errors that cannot make any further evaluations, but the $\mathbb{K}$ type (sort) system cannot detect this error in the theory. In **IsaK**, we discard the KResult sort and view the sorts subsorting to KResult as defining a set of result sorts. We use the predicate `isKResult`, whose meaning is the membership of $\mathcal{ResultSort}$. We replace every place in a $\mathbb{K}$ theory that describes a term subsorting to KResult with an `isKResult` predicate on the term. Thus, the subsort relation of KResult in a $\mathbb{K}$ theory is replaced in **IsaK** by the checking of a property on terms by `isKResult`. For example, the sort enforcement of KResult in rule `(cool)` becomes the one in Fig. 4.11.

$$v{:}\mathsf{Exp} \curvearrowright \square\ /\ y \curvearrowright tl \Rightarrow v\ /\ y \curvearrowright tl\ \texttt{when isKResult}(v)$$

Figure 4.11: A Correct Cool Rule with Property KResult Enforcement Predicate

We now describe **IsaK** terms through the symbol table Υ. Any term in **IsaK** satisfies the grammar defined in Isabelle in Fig. 4.12. The symbol table (Υ) is a translated product of the **IsaK** static semantics in Sec. 3, and each of its entries describes a syntactic definition for a specific constructor. This is represented as a tuple of a target sort (s), a list of argument sorts (sl), a set of symbol names (CS) representing a set of constructors that is either a singleton set of a *KLabel* term or set of many *KLabel* terms (generated from user defined tokens, like variable names and integers), and a Boolean value (b) indicating if the constructor is a function constructor. For a given **IsaK** theory, the constructors (having the type *KLabel*) appearing in the AST tree of a term must be a constructor name (*Symbol*) in an entry of Υ. For a given symbol table entry (s, sl, CS, b), the sort information for the constructor $c \in CS$ is $sl \rightarrow s$. A valid term in an **IsaK** theory satisfies the following definition.

Definition 4.1. Given a symbol table Υ, a term is a valid **IsaK** term iff every subterm (having the form $c(c_1, ..., c_n)\texttt{::}s$) appearing in the AST of the term satisfies the following:

- If $c_1, ..., c_n$ is an empty list, then s is a supersort of the sort of c in Υ.
- If $c_1, ..., c_n$ is not empty, let $s'_1, ..., s'_n$ be the argument sorts of c in Υ. Then, for every term c_i in $c_1, ..., c_n$, its sort s_i is a subsort of s'_i, and the sort s is a supersort of the sort of c in Υ.

$\mathbb{K}$ is a language that allows users to define a specification by giving a set of terms containing meta-variables, and it also allows them to define a specific ground term (without any meta-variable) as a "program" that produces a trace of the states when executed with the rules defined in the specification. An **IsaK** theory represents the specification, doing so as a tuple of $(\Psi, \sqsubseteq, \Upsilon, \Delta)$. Δ is a finite set of rules, each of which is a *Rule* term (possibly with meta-variables) defined in Fig. 4.12. A "program" for the theory is a *Bag* ground term.

In Fig. 4.12, we now describe briefly the grammars that define the Υ set in an **IsaK** theory. The variables appearing on the left (before $\in$) range over the sets on the right. We assume that the name sets (*UsrSort*, *LName*, and *BName*) are all disjointly unioned with each other. Any term in *KItem* is a user defined one allowed in a computation, with the fixed format of a constructor (*KLabel*) applied to a list of arguments (*KList*). The operation ($\texttt{::}$) represents a type enforcement by giving a sort. It can appear in any term in **IsaK**, and sometimes we omit such information in examples.

The $\square$ symbol in the *KItem* definition in Fig. 4.12 represents a family of symbols, one for each sort, each which represents a "hole" in the context term created when we split a term into context and redex terms, such as the $\square$ in Fig. 4.1 `(heat)`. These symbols are mainly used for $\mathbb{K}$ heating/cooling rules. With the $\square$ symbols, the evaluations of a term using heating/cooling rules are the same as for other rules. In **IsaK**, there are built-in lists (*List*),

Domains and Terms

$v \in \mathcal{CName} \triangleq \mathcal{BName} \cup \{\texttt{k}\}$	Config Names
$c \in \mathcal{KLabel} \triangleq \mathcal{LName} \cup \{\texttt{klabel}, \texttt{isKResult}, \wedge, \neg, \texttt{=}\}$ $\setminus\{\texttt{lConstr}, \texttt{sConstr}, \texttt{mConstr}, \texttt{bConstr}\}$	KLabels (Constructors)
$k \in \mathcal{KItem} \triangleq \mathcal{KLabel}\,(\mathcal{KList})::s \mid \square::s$	KItem Terms
$k \in \mathcal{K} \triangleq \mathcal{KItem}\ \mathit{list}$	Associative and Identitive KItem Sequences
$kl \in \mathcal{KList} \triangleq \mathcal{K}\ \mathit{list}$	Associative and Identitive K Sequences
$\mathcal{ListItem} \triangleq \texttt{lConstr}(\mathcal{K})$	Singleton List Terms
$l \in \mathcal{ListItem}' \triangleq \mathcal{ListItem} \mid \mathcal{KLabel}\,(\mathcal{KList})::\mathsf{List}$	List Terms With Funs
$l \in \mathcal{List} \triangleq \mathcal{ListItem}'\ \mathit{list}$	Associative and Identitive List Terms
$\mathcal{SetItem} \triangleq \texttt{sConstr}(\mathcal{K})$	Singleton Set Terms
$S \in \mathcal{SetItem}' \triangleq \mathcal{SetItem} \mid \mathcal{KLabel}\,(\mathcal{KList})::\mathsf{Set}$	Set Terms With Funs
$S \in \mathcal{Set} \triangleq \mathcal{SetItem}'\ \mathit{list}$	Idempotent Set Terms
$\mathcal{MapItem} \triangleq \texttt{mConstr}(\mathcal{K}, \mathcal{K})$	Singleton Map Terms
$M \in \mathcal{MapItem}' \triangleq \mathcal{Map} \mid \mathcal{KLabel}\,(\mathcal{KList})::\mathsf{Map}$	Map Terms With Funs
$M \in \mathcal{Map} \triangleq \mathcal{MapItem}'\ \mathit{list}$	Idempotent, and Functional Map Terms
$\mathcal{BagItem} \triangleq \texttt{bConstr}(\mathcal{CName}\texttt{,}\mathcal{Term})$	Singleton Configurations
$C \in \mathcal{Bag} \triangleq \mathcal{BagItem}\ \mathit{list}$	Associative, Commutative, and Identitive Configuration Terms
$t \in \mathcal{Term} \triangleq \mathcal{KItem} \cup \mathcal{K} \cup \mathcal{List} \cup \mathcal{Set} \cup \mathcal{Map} \cup \mathcal{Bag}$	Allowed Terms
$\mathcal{Pat} \triangleq \mathcal{Term}$	Patterns
$\mathcal{Exp} \triangleq \mathcal{Term}$	Expressions

Transition Rule Syntax

$rl \in \mathcal{Rule} \triangleq$

$\mathcal{KLabel}\,(\mathcal{KList})::s \Rightarrow \mathcal{KLabel}\,(\mathcal{KList})::s_1\,\texttt{when}\ \mathcal{KLabel}\,(\mathcal{KList})::s_2$	(* KItem Function Rules($s_1 \sqsubseteq s \sqsubseteq \mathsf{KItem} \wedge s_2 \sqsubseteq \texttt{Bool}$) *)
$\mid \mathcal{KLabel}\,(\mathcal{KList})::\mathsf{K} \Rightarrow (\mathcal{K} \mid \mathcal{KLabel}\,(\mathcal{KList})::\mathsf{K})\texttt{when}\ \mathcal{KLabel}\,(\mathcal{KList})::s_2$	(* K Function Rules($s_2 \sqsubseteq \mathsf{Bool}$) *)
$\mid \mathcal{KLabel}\,(\mathcal{KList})::\mathsf{List} \Rightarrow (\mathcal{List} \mid \mathcal{KLabel}\,(\mathcal{KList})::\mathsf{List})$ $\texttt{when}\ \mathcal{KLabel}\,(\mathcal{KList})::s_2$	(* List Function Rules($s_2 \sqsubseteq \mathsf{Bool}$) *)
$\mid \mathcal{KLabel}\,(\mathcal{KList})::\mathsf{Set} \Rightarrow (\mathcal{Set} \mid \mathcal{KLabel}\,(\mathcal{KList})::\mathsf{Set})$ $\texttt{when}\ \mathcal{KLabel}\,(\mathcal{KList})::s_2$	(* Set Function Rules($s_2 \sqsubseteq \mathsf{Bool}$) *)
$\mid \mathcal{KLabel}\,(\mathcal{KList})::\mathsf{Map} \Rightarrow (\mathcal{Map} \mid \mathcal{KLabel}\,(\mathcal{KList})::\mathsf{Map})$ $\texttt{when}\ \mathcal{KLabel}\,(\mathcal{KList})::s_2$	(* Map Function Rules($s_2 \sqsubseteq \mathsf{Bool}$) *)
$\mid \mathcal{K} \Rightarrow \mathcal{K}\ \texttt{when}\ \mathcal{KLabel}\,(\mathcal{KList})::s_2$	(* K Transition Rules($s_2 \sqsubseteq \mathsf{Bool}$) *)
$\mid \mathcal{Bag} \Rightarrow \mathcal{Bag}\ \texttt{when}\ \mathcal{KLabel}\,(\mathcal{KList})::s_2$	(* Configuration Transition Rules($s_2 \sqsubseteq \mathsf{Bool}$) *)

IsaK Theory Input

$\mathcal{Symbol} \triangleq \mathcal{KLabel} \mid \mathcal{KLabel} \rightarrow \mathcal{Bool}$	Symbols
$\Upsilon \subseteq \Psi \times \Psi\ \mathit{list} \times \mathcal{Symbol} \times \mathcal{Bool}$	Symbol Table
Rule Set: $\Delta \subseteq \mathcal{Rule}$	Subsort Relation: $\sqsubseteq$

Figure 4.12: **IsaK** Syntax in Isabelle (No Meta-Variables)

sets ($\mathcal{Set}$), and maps ($\mathcal{Map}$) for users with different built-in equational properties (listed in Fig. 4.12). Type $\mathcal{Bag}$ contains lists (with associative, commutative, and identity equational properties) of basic program state pieces, named configuration pieces or cells and having the type $\mathcal{BagItem}$. Each cell contains a cell name ($\mathcal{CName}$) and sub-configuration components, an example of which the IMP configuration is shown in Fig. 4.1. In Sec. 3, we introduced how a configuration works. Users need to define an initial configuration along with their language specification. For every step of the evaluation of an input program, the result program state obeys the sort and position relations among the different cells in the initial configuration. One feature in **IsaK** that is useful for configuration translation (Sec. 4.5.2) is that Υ contains entries for all cell names appearing in the initial configuration. In an **IsaK** theory, the name of each cell has an entry in Υ that contains a target sort the same as the sort of the element in the cell in the initial configuration, an empty argument sort list, a singleton set of the cell name, and a `false` Boolean value. For example, cell `env` has a content of sort Map (Fig. 4.1), so it has the entry: $(\mathsf{Map}, [], \{\texttt{env}\}, \texttt{false})$.

Besides the above syntactic definitions and restrictions, an **IsaK** theory also has other syntactic restrictions that appear in the static translation process from FAST to BAST (introduced in Sec. 4.2). For example, all rules ($\mathcal{Rule}$) have the format: $\mathcal{Pat} \Rightarrow \mathcal{Exp}$ `when` $\mathcal{Exp}$, where the left hand side of $\Rightarrow$ is the pattern ($\mathcal{Pat}$) to match with, and the right hand side of $\Rightarrow$ is the target expression ($\mathcal{Exp}$) to rewrite to, provided that the condition expression ($\mathcal{Exp}$) after the keyword `when` is satisfied. Terms in both types $\mathcal{Pat}$ and $\mathcal{Exp}$ are **IsaK** terms ($\mathcal{Term}$), but they have different syntactic restrictions checked by the **IsaK** static semantics (Sec. 4.2). For example, no term in $\mathcal{Pat}$ can have proper sub-terms possessing function constructors. For a given $\mathcal{Rule}$ term, meta-variables can only represent a term ($\mathcal{Term}$ffr in Fig. 4.12).

4.3.2 The **IsaK** Dynamic Semantics

Here we introduce the **IsaK** evaluation semantics. From the static process in Sec. 3, we derive an IsaK theory in BAST format as a tuple of a set of sorts (Ψ), a subsort relation ($\sqsubseteq$), a symbol table (Υ), and a set of rules ($\Delta \subseteq \mathcal{Rule}$), which are briefly presented in Fig. 4.12. A program in Θ is represented as a ground term configuration (C_0) that has the type $\mathcal{Bag}$. It is a term whose syntax is specified by the syntactic definitions in Θ, and is translated into a $\mathcal{Bag}$ configuration through the static process in Sec. 4.2. The evaluation of a program (C_0) in Θ produces a set of traces, each of which contains a sequence of configurations, where the $(i+1)$-th configuration (C_{i+1}) is the result of applying a rule from Δ to the i-th configuration (C_i). We first introduce procedures that are common to every evaluation step and then we introduce evaluation semantics particular to different rules.

Common Evaluation Procedures. There are three procedures that every evaluation step in **IsaK** needs. We introduce them separately. The first one is the pattern matching procedure (`match`). The pattern matching algorithm in **IsaK** is a normal top-most pattern matching process. Given a rule rl and term t, the procedure $\texttt{match}(rl, t)$ pattern-matches the left-hand side of rl (the pattern side) with a ground term t, and generates a map from the meta-variables on the left side of rl to subterms in t or $\bot$ if there is no match. Pattern-matching here means that for a pattern of the form (p, t) with $p = c(p_1, ..., p_n)$ and ground term $t = c'(t_1, ..., t_m)$, we have $c = c'$, $n = m$, and σ_i is the result of matching p_i with t_i, then the result is $\bigcup_i \sigma_i$ so long as for all meta-variables $x \in (\texttt{dom}(\sigma_i) \cap \texttt{dom}(\sigma_j))$ we have $\sigma_i(x) = \sigma_j(x)$, and for a pattern that is a meta-variable x, the result of the match is $\{x \mapsto t\}$.

For simplicity, we define $\texttt{match}(\Theta, t)$ to find the rule rl in the rule set of Θ whose left-hand side matches term t and which generates a mapping.

The second common procedure is the substitution procedure ($\texttt{subs}$). Given a term p with meta-variables $x_1, ..., x_n$ and map m from the meta-variables to ground terms, $\texttt{subs}(m, p)$ substitutes the ground term $m(x_i)$ for the occurrences of every meta-variable x_i ($1 \leq i \leq n$) in p. The third procedure is the normalization procedure ($\texttt{norm}$) that we have described a little in Sec. 4.2. Normalization only applies to the whole ground term configuration C, and $\texttt{norm}(C)$ searches every subterm in C and rewrites it to a canonical form. Mainly, it rewrites two identical $\mathscr{Set}$ and $\mathscr{Map}$ subterms to be one (the idempotent property), and checks if there is a $\mathscr{Map}$ term that is not functional; i.e. $\texttt{mConstr}(x, 1)$ and $\texttt{mConstr}(x, 2)$ appear in the same $\mathscr{Map}$. In this case, $\texttt{norm}(C)$ returns the global error state $\texttt{Err}$.

Semantics for Different Rules. We first introduce the concept of configuration context. To do so, we modify the syntax in Fig. 4.12 by inserting a new $\square$ term in each type $\mathscr{ListItem}'$, $\mathscr{SetItem}'$, and $\mathscr{MapItem}'$ in Fig. 4.13.

$$\mathscr{ListItem}' \triangleq ... \mid \square\text{::List} \qquad \mathscr{MapItem}' \triangleq ... \mid \square\text{::Map} \qquad \mathscr{SetItem}' \triangleq ... \mid \square\text{::Set}$$

Figure 4.13: Additional Syntactic Insertions for **IsaK** Semantics

We then define the configuration context $C[]^s_f$ to be a $\mathscr{Bag}$ term with exactly one of the $\square$ subterms (Fig. **??**) or the $\square$::s in the $\mathscr{KItem}$ definition in Fig. 4.12 ($s \in \mathscr{UsrSort} \cup \{\text{KItem}, \text{K}, \text{List}, \text{Map}, \text{Set}\}$), and the configuration redex to be a term $c(kl)$::s. We then define a valid combination of context and redex ($C[c(kl)::s]^s_f$) to be a $\mathscr{Bag}$ term C derived by replacing the $\square$ subterm with the redex $c(kl)$::s, where the sort for the $\square$ matches the sort s. We also insert a new $\square$ term in the $\mathscr{BagItem}$ definition in Fig. 4.12, and define another context-redex pair as $C[]_{\mathsf{k}}$ and a $\mathscr{K}$ term k, such that $C[]_{\mathsf{k}}$ has a unique $\square$ term as a $\mathscr{BagItem}$ subterm and $C[k]_{\mathsf{k}}$ replaces the $\square$ subterm with the $\mathscr{BagItem}$ term ($\texttt{bConstr}(\mathsf{k}, k)$) whose cell name ($\mathscr{CName}$) is k and $\mathscr{K}$ type subterm is k.

Any **IsaK** evaluation ($\longrightarrow^{\Theta}$) can be viewed as an application of one of three different rules: function rule applications ($\longrightarrow^{\Theta}_{\mathtt{B},f}$; (1) and (2) in Fig. 4.14), K rule applications ($\longrightarrow^{\Theta}_{\mathsf{k}}$; (3) in Fig. 4.14), and configuration ($\mathscr{Bag}$) rule applications ($\longrightarrow^{\Theta}_{\mathtt{B}}$; (4) in Fig. 4.14). In these rules, $\texttt{right}$ is a function to get the right-hand side (expression side) of an **IsaK** rule, while $\texttt{cond}$ is to get the condition expression of an **IsaK** rule. The term $\texttt{true}$ is a built-in Boolean term in **IsaK** representing the true value. The Kleene star ($*$) in Fig. 4.14 represents applying the arrow-rule inside the parentheses multiple times until a final result (like `true`) shows up or there are no more such arrow-rules to apply. The basic evaluation strategy of these rule applications is to split the current configuration C into a context and a redex, apply a rule to rewrite the redex, and insert the new redex back into the context.

$$\texttt{(1)}\quad \frac{\begin{array}{l} rl \in \Theta \qquad m = \texttt{match}(rl, c(kl)) \\ t = \texttt{subs}(m, \texttt{cond}(rl)) \qquad t\ (\longrightarrow_f^{\Theta})^*\ \texttt{true} \end{array}}{c(kl) \longrightarrow_f^{\Theta} \texttt{subs}(m, \texttt{right}(rl))}$$

$$\texttt{(2)}\quad \frac{C = C[c(kl)\texttt{::}s]_f^s \qquad c(kl)\ (\longrightarrow_f^{\Theta})^*\ t}{C \longrightarrow_{\texttt{B},f}^{\Theta} \texttt{norm}(C[t]_f^s)}$$

$$\texttt{(3)}\quad \frac{\begin{array}{c} C = C[k]_{\mathsf{k}} \qquad rl \in \Theta \qquad m = \texttt{match}(rl, k) \\ t = \texttt{subs}(m, \texttt{cond}(rl)) \qquad t\ (\longrightarrow_f^{\Theta})^*\ \texttt{true} \end{array}}{C \longrightarrow_{\mathsf{k}}^{\Theta} \texttt{norm}(C[\texttt{subs}(m, \texttt{right}(rl))]_{\mathsf{k}})}$$

$$\texttt{(4)}\quad \frac{\begin{array}{l} rl \in \Theta \qquad m = \texttt{match}(rl, C) \\ t = \texttt{subs}(m, \texttt{cond}(rl)) \qquad t\ (\longrightarrow_f^{\Theta})^*\ \texttt{true} \end{array}}{C \longrightarrow_{\texttt{B}}^{\Theta} \texttt{norm}(\texttt{subs}(m, \texttt{right}(rl)))}$$

$$\texttt{(group)}\quad \Longrightarrow^{\Theta} \triangleq (\longrightarrow_{\texttt{B},f}^{\Theta})^*(\longrightarrow_{\mathsf{k}}^{\Theta} \mid \longrightarrow_{\texttt{B}}^{\Theta})$$

Figure 4.14: **IsaK** Semantics for Different Rules

The `(group)` definition in Fig. 4.14 represents a typical combination of rule applications in forming different $\mathbb{K}$ tools, mainly, the ***krun*** and ***ksearch*** tools. The ***krun*** tool is defined as $(\Longrightarrow^{\Theta})^*$ or $(\Longrightarrow^{\Theta})^n$ if users specify the number of trace steps n they want to see. The ***ksearch*** tool is defined by a transition from a singleton set of a configuration to a set of configurations in the form $(\{C\} \Rrightarrow^{\Theta} Cl)^*$, where the set configuration transition has the property in Fig. 4.15.

$$Cl \Rrightarrow^{\Theta} Cl' \triangleq (\forall C \in Cl.\ C \Longrightarrow^{\Theta} C' \Rightarrow C' \in Cl') \wedge (\forall C' \in Cl',\ \exists C \in Cl.\ C \Longrightarrow^{\Theta} C')$$

Figure 4.15: The Transition Property of ***ksearch***

One way **IsaK** is different from $\mathbb{K}$ is that $\mathbb{K}$ has two additional kinds of rule applications: anywhere and macro rule applications. We found out that a specification with the two kinds is not type safe. We find that in the current big language specifications in K (LLVM [140], C [83], etc), all of the instances of anywhere and macro rules can be replaced by the three different rules stated here. For any **IsaK** theory with only function, K, and configuration rules, we have the following type preservation property. Note that the **IsaK** type system does not satisfy the type progress property, since $\mathbb{K}$ allows users to define language semantics incrementally. It is fine in $\mathbb{K}$ to define a language syntax without defining its semantics.

Theorem 4.1. For an **IsaK** theory (Θ) containing only function, K, and configuration rules, with any initial type correct configuration C, evaluating C in the $\mathbb{K}$ theory never results in a type-error.

Proof. Given an **IsaK** theory (Θ) and an initial configuration C that are type-checked, the proof is based on structure induction on different rules applied to configuration C. There are three possible rule applications:

- If the rule application is a function rule one, for any given configuration C that is split into a context $C[]_f^s$ and redex $c(kl)\mathbf{::}s$, a function rule application transit the term $c(kl)\mathbf{::}s$ into a possible new term t with sort s', and s' must subsort to the sort s based on the function rule type correctness (the target sort of the right-hand-side of a rule must be a subsort of the target sort of the left-hand-side). Thus, if we plug the term t back into the context $C[]_f^s$, the configuration becomes $C[t\mathbf{::}s']_f^s$. Since s' subsorts to s, the new configuration $C[t\mathbf{::}s']_f^s$ is type-correct.

- If the rule application is a K rule, for any given configuration C that is split into a context $C[]_{\mathsf{k}}$ and redex k that is a $\mathcal{K}$ term, a K rule only rewrites the term k to a new term k' with the same type $\mathcal{K}$. All $\mathcal{K}$ terms have sort K. Thus, the final result configuration $C[k']_{\mathsf{k}}$ is type-correct.

- If the rule application is a configuration rule, for any given configuration C, applying a configuration rule rewrites the term C to a new configuration C' that is type-correct.

Thus, evaluating a configuration C in the type-checked theory Θ never results in a type error. QED.

4.4 $\mathbb{K}$ DESIGN ISSUES AND THE EVALUATION OF **ISAK**

4.4.1 The Design Issues and Improvement

In Section 4.2, we briefly describe how $\mathbb{K}$ compiles away its modularity, localization and concision features in the translation from FAST to a BAST format that is normalized for execution. We did not specify how to compile module systems in that section because the compilation of module systems in $\mathbb{K}$ is as easy as putting all files together. For a specification, different modules are not independent, so $\mathbb{K}$ has some unexpected behaviors. In this section, we mainly focus on the design issues of $\mathbb{K}$, especially the design issues of module systems in $\mathbb{K}$ and suggest ways to handle some of these problems.

Too Flexible FAST Semantic Rules. The LHS and the RHS of a $\mathbb{K}$ semantic rule have actually different tolerance on what can be correctly accepted. $\mathbb{K}$ views the LHS of a rule as the pattern and the RHS of a rule as the expression, and provides a different set of syntax for them in BAST. Their FAST formats are the same, and this can lead to confusion. For example, if we have the syntactic declaration and rule in Figure 4.16 `(a)` and 4.16 `(b)`, the construct `test` takes an argument of sort *Set*. The rule tries to rewrite an element of value `1` to `2` in an element of a set. This rule is invalid in $\mathbb{K}$. As a pattern, $\mathbb{K}$ only allows one variable to represent elements of built-in term with sort K, *KList*, *Set*, *List*, *Map* or *Bag*. So, if we cut off the variable B in the rule, the rule becomes permissible. This design can allow $\mathbb{K}$ developers to design the $\mathbb{K}$ pattern matching algorithm simply, and avoid having exponential search steps in the algorithm that arise once we allow more than one variable for the elements of these built-in terms. The current $\mathbb{K}$ pattern matching algorithm is especially efficient when built-in terms having implicit equational rules.

SYNTAX $\mathit{Exp} ::= \texttt{test(}\ \mathit{Set}\ \texttt{)}$

(a)

$$\frac{\texttt{test(}\ A : \mathit{Set}\ \ \texttt{SetItem(1)}\ \ B : \mathit{Set}\ \texttt{)}}{\texttt{test(}\ A : \mathit{Set}\ \ \texttt{SetItem(2)}\ \ B : \mathit{Set}\ \texttt{)}}$$

(b)

$$\frac{\texttt{test(.Set)}}{\texttt{test(.K)}}$$

(c)

Figure 4.16: Pattern and Expression Being Different Example

Additionally, being overly flexible can be problematic in $\mathbb{K}$. For example, we found a bug in $\mathbb{K}$ which can allow the rule in Figure 4.16 `(c)` to compile. This rule basically allows $\mathbb{K}$ to rewrite a sort *Set* term to a sort K term. We disallow this rewrite in **IsaK** by designing different types in Isabelle to represent different built-in terms of $\mathbb{K}$ and sort-checking each input rule.

Module Systems and Configurations. As we know in Section 3.4, each $\mathbb{K}$ object language specification has a unique configuration that provides users the localization property of $\mathbb{K}$. Users need to use a key word `configuration` in $\mathbb{K}$ to define it, and $\mathbb{K}$ does not restrict users where to place the configuration. Once a user declares `configuration` in a module A, it means that other modules depend on module A to direct how to interpret rules defined in them. This problem itself can be classified as a feature of $\mathbb{K}$ and only shows that $\mathbb{K}$ module system is not independent. However, if we define two configurations in two different modules in an object language specification, current $\mathbb{K}$ implementations can actually allow the specification to compile and execute programs. It will choose one of the configurations as the oracle. Which configuration will be pick is not clear. In **IsaK**, we place a check to disallow two different configurations in a specification in the stage of **Applying Validity Checks** (in Section 4.2).

Module Systems and Extended Syntactic Definitions and Subsorts. The modularity of $\mathbb{K}$ allows users to define new syntactic constructs, new subsort relations or new semantic rules where they need. For an object language specification, users are able to define a new module with a new set of syntactic constructs and rules regarding the old modules. This brings problems. In the stage of **Heat/Cool Rules Generation** (Section 4.2), we see that a pair of `heat`/`cool` rules relying heavily on a built-in sort *KResult*. Defining new subsort relations in a new module with the sort *KResult* (which is a typical things to do in using $\mathbb{K}$) is in some sense changing the meaning of these `heat`/`cool` rules. It is completely possible that pattern matching a `heat` rule on a term might suddenly becomes invalid because we add a new module with a new subsort relation subsorting the target sort of a subterm in the term to *KResult*.

Moreover, extending a syntactic definition might invalidate an object language specification. In the **Applying Validity Checks** of the previous section, we have seen two such examples. If the two list syntactic definitions mentioned there are in two different modules, or if users define a new construct that has the same `kLabel` name as one in other modules, they will cause the object language specification to be invalid. In addition, if a rule is attributed with `function` in a new module, the rule might refer to be a part of semantics for a function application previous defined in other modules. The new rule might bring additional non-determinism for $\mathbb{K}$ functions in an object language specification. To solve the problem, our static semantics perform several checks in **Applying Validity Checks** to check if subsort relations are acyclic and anti-symmetric, if users define two syntactic definitions with the same `kLabel` name, and if two list syntactic definitions result in the conflict described in **Applying Validity Checks** (Section 4.2). If any of these is true, we recognize the input object language specification is not well-formed. For $\mathbb{K}$ functions, we only collect them and check if they follow certain formating such as having no more than one rule attributed with `owise` for a function construct, but we do not check if a function has non-determinism behaviors.

Failure in Generating Nested Cells. We have seen how $\mathbb{K}$ complies localization and concision features by "completing" a rule with configuration information in the **Transformation** stage in Section 4.2. Actually, current $\mathbb{K}$ implementations ($\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0) implicitly (without any error message or mention in any document) prevent people from defining in a configuration more than two levels of nested cells with the key word *, meaning that these cells can have zero or more copies through executions. What is more, there are some undesirable behaviors that happen when the nested * key word cell has only two levels. In K-Java [86], the method invocation rule connects an operator with a specific method body (some cells in the `methodDec` cell) in a specific class (the `class` cell). The `methodDec` and `class` cells are both attributed with *. The method invocation rule in K-Java is valid but only by chance. If the author had changed the Java configuration by adding one more cell with the * key word inside the `thread` cell (labeled with * key word as well), an application of the method invocation rule would have crashed. This is not being picky because a lot of users might actually want to use the K-Java semantics to do further research. For example, when researchers want to enhance K-Java by making a better memory model, one thing they do is to change the stack structure. The current stack is implemented as a *List* data structure in $\mathbb{K}$, but it is only used to store function information. Users might want to implement a real stack structure with stack range, types and map from byte location to value. We can model the stack structure in Fig. 4.17 `(a)`.

`(a)` $\langle\langle\langle StackType\rangle_{\mathsf{stackType}}\ \langle Map\rangle_{\mathsf{byteMap}}\ \langle(Int, Int)\rangle_{\mathsf{stackRange}}\rangle_{\mathsf{stackObject*}}\rangle_{\mathsf{stack}}$

$$\left\langle \frac{\texttt{getStackType}(X : Int)}{T : StackType} \cdots \right\rangle_{\mathsf{k}}\ \langle (L : Int, R : Int)\rangle_{\mathsf{stackRange}}\ \langle T : StackType\rangle_{\mathsf{stackType}}$$

`(b)` $requires\ L : Int \leq X : Int \leq R : Int$

Figure 4.17: The Stack Structure and Example Rules

The `stackType` stores the information about the types of the values stored in the stack piece; the `byteMap` cell stores the values for each byte location associated with the stack piece, and the `stackRange` cell determines the stack locations in the machine. By replacing the old `stack` cell with new stack structure in the K-Java configuration, we create two-level nested * cells in the configuration. The top level * cell is `thread`, and the inner level cell is `stackObject`. Suppose we define the semantics of an operator `getStackType` to lookup the type of a stack as Fig. 4.17 `(b)`.

Once a program state requires this rule, the whole execution in the $\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0 crashes, because the special cell `k` representing the program computation sequence is inside a cell `thread` marked with the keyword *, and some variable inside the execution cell `k` is trying to match with some content inside another cell (`stackObject`) marked as *, which is inside the `thread` cell containing the execution cell `k`. Apparently, 𝕂 3.6 and 𝕂 4.0 do not allow this. If one is not a 𝕂 developer and is trying to define some language semantics with complicated stack or thread data structures, it is almost certain that they will need the special cell `k` inside a * keyword cell and define other cells in the * keyword cell with another * keyword. Nevertheless, determining there has been a crash, testing and finding the problem takes a 𝕂 starter a great deal of effort and needless trouble because there are no error messages and the only way to locate it is to test each rule separately. Our **IsaK** solve the problem by eliminating the restriction of level of nested cells with * keyword that users can write.

These are some design issues of 𝕂 and our improvement, and we believe that this is one of the key advantages **IsaK** is bringing to the 𝕂 community.

4.4.2 **IsaK** Evaluation

Evaluating **IsaK** took more than half of the development time. In testing it, we extracted OCaml code from **IsaK** directly in Isabelle, and tested the 𝕂 specifications and programs based on the extracted OCaml 𝕂 interpreter. The extracted interpreter is using the dynamic semantics that we defined for 𝕂 in the technical report [10]. The Ocaml interpreter is also a trivial utility of **IsaK**, which is extracted directly from the Isabelle source code and users can use the `krun` function to execute a program of the specification and see a single trace of the program. In the following paragraphs, we describe our evaluation, especially the testing, which resulted in the first thorough set of bug reports for 𝕂.

Testing process of IsaK. The validation of language semantics is usually accomplished through the use of external test suites [83, 84, 120], which was also our strategy. A set of 13 specifications with 356 programs, which we call the 𝕂 standard test suite, was the basis of our testing. It was used by the 𝕂 team to test the 𝕂 implementations.

Our methodology for developing **IsaK** was through a strategy of combining Test Driven Development (TDD) with questioning the 𝕂 team. We first talked to the 𝕂 team in depth. In the first several months of our 𝕂 semantics project, we only did multiple cycles of (1) discussing existing documents and materials with the 𝕂 team, (2) implementing critical experiments of some small language specifications and running them in the 𝕂 implementations, and (3) discussing more materials with them. After that, we developed our semantics largely by following the TDD process. The reason for employing this design methodology was because 𝕂 had no semantics in print, so we needed to understand exactly what the 𝕂 team was thinking. In addition, 𝕂 is complicated enough that its design should be driven by tests. Our TDD design process required us to design our features carefully. When developing a new feature, we first tried to cover all corner cases of the feature under test in isolation, and then define it in the simplest way possible so as to pass all tests. The test suite also covered test cases when features overlapped, so we could make sure that the combinations of features in 𝕂 were implemented correctly. This is extremely important in cases dealing with overlapped features.

We first used our design methodology to test our semantics with respect to the dynamic execution engine of 𝕂. We ran the 𝕂 standard test suite, and our results showed that our 𝕂 interpreter passed 338 of the programs. Among the test cases, we had no single specification that we could not handle. Our ***kompile*** function compiled all test specifications, but there were test programs that we could not handle with ***krun*** or ***ksearch***. All of them related to the standard input channel. 𝕂 allows users to define a cell as an input/output channel so that they can type in inputs to the cell from a keyboard, just as I/O operators in C and Java do. The behaviors of reading I/O input (the input channel) is hard to implement in Isabelle, and it is best to just define it with the 𝕂 interpreter. We have not yet finished the job in the interpreter, but we believe that it will be an easy fix.

In the process of testing, we also questioned the behaviors of the current 𝕂 implementations (𝕂 3.6 and 𝕂 4.0). If we implemented a feature according to a 𝕂 document and descriptions from the 𝕂 team of the correct behaviors for it, and then found that test results for the feature were not what the 𝕂 implementations did, we would extend the specifications or programs to include new aspects to see what the problems were. Thus, we found possible undesirable behaviors in the 𝕂 implementations. Eventually, we located the bugs and made a new small 𝕂 "program" (a small language specification and a single input program for the specification ”k”) to test against the bugs; we also added them to the test suite for later tests in the development process of **IsaK**. In developing **IsaK**, we identified 25 kinds of undesirable behavior in the 𝕂 implementations. Each can have many different versions, and we specified a small 𝕂 "program" for each of them in our test files.

These undesirable behaviors happen in very diverse circumstances. In fact, we have already seen one such failure in Section 4.4.1. Some implementations in 𝕂 might have design problems. For example, rules labeled with a `macro` attribute (`macro` rules) are harmful and useless. There is no proper 𝕂 documents suggesting the use of `macro` rules. When we test the rules, we find that applying such rules on a user defined program is error-prone. The only few cases when the `macro` rules can be applied successfully without any undesirable behaviors are those listed in the 𝕂 test suites or in some previous defined language specifications in $\mathbb{K}$ [83, 84, 85, 86]. In these cases, users always wanted to define a syntactic sugar and used a `macro` rule to rewrite the syntactic sugar to another term once in the beginning of a evaluation of input programs, which can be easily replaced by using `function` rules in $\mathbb{K}$. Hence, `macro` rules are unnecessary in $\mathbb{K}$.

Other undesirable behaviors are the implementation bugs in $\mathbb{K}$. For example, some are related to sort checking/adjustment. The current $\mathbb{K}$ implementations allow users to write down rules rewriting a sort K term to a sort *List* or *Set* term, which are bugs because they do not allow users to write down rules rewriting a sort sort *List* or *Set* term to a sort K term. In addition, some undesirable behaviors are related to the pattern matching algorithm in $\mathbb{K}$ (the atomic step). The current $\mathbb{K}$ implementations allow some implicit associative and identity equational rules for user-defined list operators in a language specification. However, there are some cases where the associative rewriting does not work, which is why we decided not to allow implicit associative and identity equational rules for user-defined list operators. Moreover, the implementation of the implicit commutative equational rule also fails in some cases. There are many of these undesirable behaviors, we will not list all of them here. Interested users can read the technical report [10] or go to `https://github.com/liyili2/k-semantics` to see these undesirable behaviors.

4.5 **TRANSK**: TRANSLATION FROM $\mathbb{K}$ TO ISABELLE

Here, we introduce **TransK**, the translation from a $\mathbb{K}$ theory to an Isabelle one. The input of the translation is an **IsaK** specification (theory) (Sec. 4.3.2), where a specification contains a symbol table (Υ), a subsort relation ($\sqsubseteq$), and a set of transition rules (Δ), possibly including function rules, K rules, and/or configuration rules. The subsort relation is assumed to have no mention of the $\mathsf{KResult}$ sort to fulfill the type-correct $\mathbb{K}$ specification requirement in Theorem 4.1. The output of translating a specification by **TransK** is an Isabelle theory containing a list of Isabelle datatypes, a list of quotient types with proofs, and a list of Isabelle rules translated from rules in the input **IsaK** specification.

4.5.1 Translating Datatypes

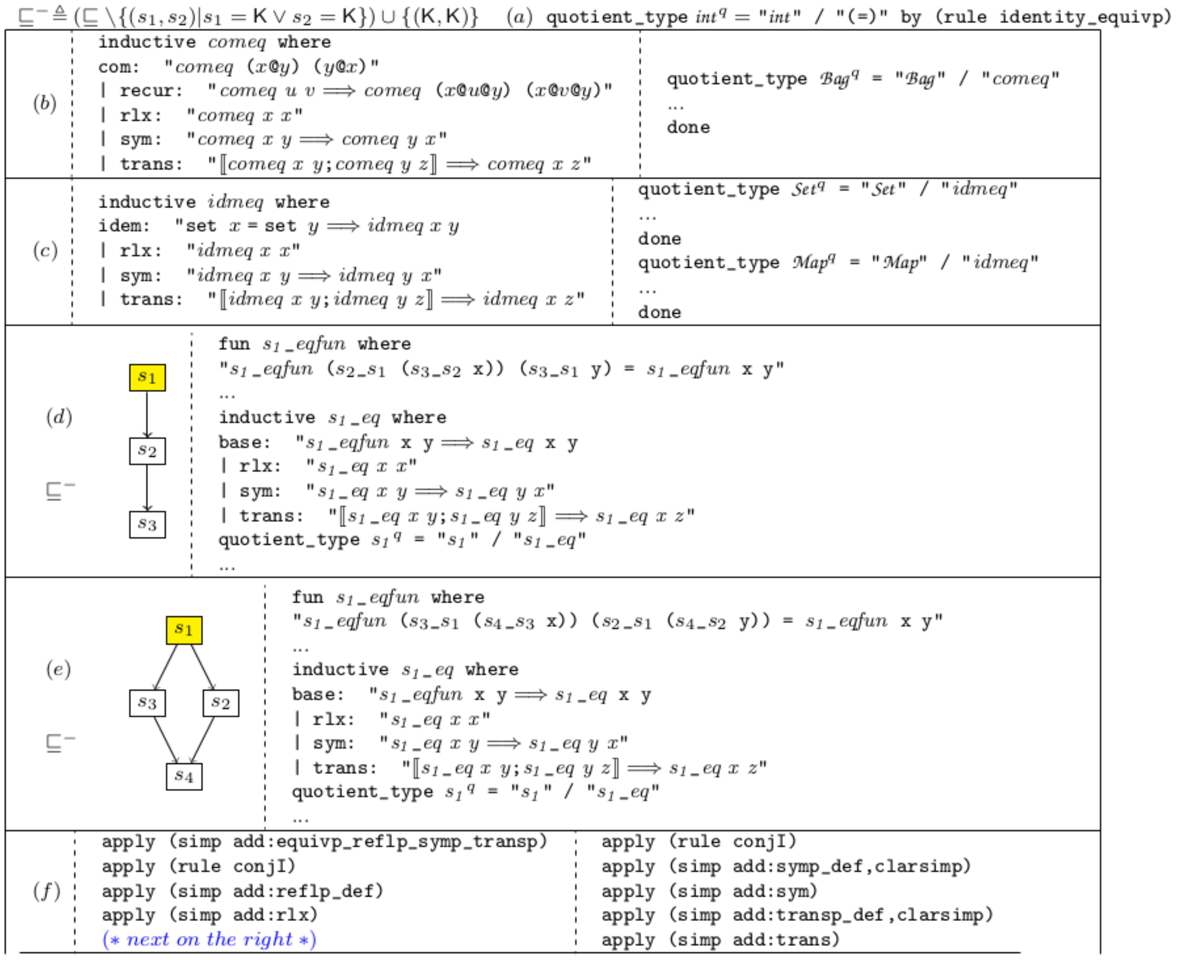
For a given **IsaK** theory $\Theta = (\Psi, \sqsubseteq, \Upsilon, \Delta)$, we first translate the tuple $(\Psi, \sqsubseteq, \Upsilon)$ to a pair of a finite quotient type set and a finite set of Isabelle proofs (Ω^q, Π) in the translated Isabelle theory (Ξ), such that all relations in $\sqsubseteq$ are invisible in Ξ, but their functionality is merged in Ω^q. The way to achieve this is to utilize Isabelle quotient types: we first translate the **IsaK** datatype tuples $(\Psi, \sqsubseteq, \Upsilon)$ to a finite Isabelle datatype set Ω by explicitly coercing every pair in $\sqsubseteq$, and then translate Ω to a quotient type set Ω^q with a finite set of proofs (Π), one for each target sort in Ω^q, to show that each quotient type in Ω^q defines an equivalence relation over all of the syntax defined in Ω. We describe the two processes below.

Builtins
datatype $\mathcal{KItem}$ = s_1`_KItem` s_1 |...| s_n`_KItem` s_n $\quad s_1, ..., s_n \in \mathcal{UsrSort}$
type_synonym $\mathcal{K}$ = $\mathcal{KItem}$ *list*
datatype $\mathcal{SetItem}$ = SConstr $\mathcal{K}$ type_synonym $\mathcal{Set}$ = $\mathcal{SetItem}$ *list*
datatype $\mathcal{ListItem}$ = LConstr $\mathcal{K}$ type_synonym $\mathcal{List}$ = $\mathcal{ListItem}$ *list*
datatype $\mathcal{BagItem}$ = BagC $\mathcal{CName}$ $\mathcal{Bag}$ | MapC $\mathcal{CName}$ $\mathcal{Map}$ | SetC $\mathcal{CName}$ $\mathcal{Set}$
| ListC $\mathcal{CName}$ $\mathcal{List}$ | KC $\mathcal{CName}$ $\mathcal{K}$
type_synonym $\mathcal{Bag}$ = $\mathcal{BagItem}$ *list* datatype $\mathcal{MapItem}$ = MConstr $\mathcal{K}$ $\mathcal{K}$
type_synonym $\mathcal{Map}$ = $\mathcal{MapItem}$ *list*
Translated Syntax
datatype $\mathcal{Exp}$ = Exp_Hole | Var_Exp $\mathcal{Var}$ | Int_Exp *Int* | Div $\mathcal{Exp}$ $\mathcal{Exp}$ | ...
datatype $\mathcal{BExp}$ = BExp_Hole | Bool_Exp $\mathcal{Bool}$ | Less $\mathcal{Exp}$ $\mathcal{Exp}$ | And $\mathcal{Exp}$ $\mathcal{Exp}$ | ...
datatype $\mathcal{Stmt}$ = Stmt_Hole | Bloc_Stmt $\mathcal{Bloc}$ | Assign $\mathcal{Var}$ $\mathcal{Exp}$
| If $\mathcal{BExp}$ $\mathcal{Bloc}$ $\mathcal{Bloc}$ | While $\mathcal{BExp}$ $\mathcal{Bloc}$ | Seq $\mathcal{Stmt}$ $\mathcal{Stmt}$ | Thread $\mathcal{Var}$ $\mathcal{Stmt}$
datatype $\mathcal{Bloc}$ = Empty | Single $\mathcal{Stmt}$ datatype $\mathcal{Prog}$ = Prog $\mathcal{Vars}$ $\mathcal{Stmt}$
datatype$\mathcal{Vars}$ = VarUnit | VarCons $\mathcal{Var}$ $\mathcal{Vars}$

Figure 4.18: Example of Datatype Translation (IMP)

The Translation from $\mathbb{K}$ Datatypes to Isabelle Datatypes. The translation step from the tuple $(\Psi, \sqsubseteq, \Upsilon)$ to an Isabelle datatype set Ω has two parts: adding builtin datatypes (corresponding to terms in $\mathcal{BuiltinSort}$ in Sec. 4.3.1) and translating user defined datatypes (corresponding to terms in $\mathcal{UsrSort}$ in Sec. 4.3.1). The two parts for translated result of the **IsaK** theory (IMP) syntax in Fig. 4.1.1 are shown Fig. 4.18. The builtin datatypes that are additionally generated in Fig. 4.18 are in a one-to-one correspondence with the datatypes in **IsaK** in Fig. 4.12, except the datatypes $\mathcal{KLabel}/\mathcal{KList}$, which represent constructors and their arguments in $\mathbb{K}$ and are absorbed into different datatypes in Isabelle. We implement the builtin $\mathcal{K}$, $\mathcal{List}$, $\mathcal{Set}$, $\mathcal{Map}$, and $\mathcal{Bag}$ datatypes as type synonyms for Isabelle builtin lists of corresponding singleton item datatypes, e.g. $\mathcal{KItem}$ *list* for $\mathcal{K}$. The reason to translate these builtin datatypes to Isabelle builtin list structures is to capture the aspect that some builtin datatypes have implicit equational properties associated with them (listed in Fig. 4.12). By representing these datatypes as Isabelle list structures and representing a connection operation in **IsaK** (e.g. the set concatenation operation in $\mathbb{K}$) as an Isabelle list concatenation operation (@), we are able to capture the implicit associative and identity equational properties on the these datatypes without extra cares. The other implicit equational properties are dealt with when translating datatypes to quotient types.

The translation of user defined datatypes in $\mathbb{K}$ to Isabelle is done by adding explicit coercions for all subsort relation pairs in $\sqsubseteq$, e.g. the constructor `Var_Exp` coerces a term in $\mathcal{Var}$ to $\mathcal{Exp}$, except that all function constructs (e.g. `fresh` in Fig. 4.1), which are translated directly into inductive relations without having datatype definitions in Isabelle (Sec. 4.5.2). Additionally, Since every user defined sort (s) is a subsort of KItem, we implement $\mathcal{KItem}$ as the union of all coercions of user defined sorts by adding a constructor for each one (s) of them as: `s_KItem`. We also add an extra constructor (like `Exp_Hole`) for each sort that contains some syntactic definitions with `[strict]` attributes to represent the $\square$ term in **IsaK** (Sec. 4.3.2). In IMP (Fig. 4.18), for example, we generate extra "hole" constructs for the types $\mathcal{Exp}$, $\mathcal{BExp}$, and $\mathcal{Stmt}$, but other user defined sorts have no such construct because they do not have a definition with a `[strict]` attribute (Fig. 4.1).



$\sqsubseteq^{-} \triangleq (\sqsubseteq \setminus \{(s_1,s_2) | s_1 = \mathsf{K} \vee s_2 = \mathsf{K}\}) \cup \{(\mathsf{K},\mathsf{K})\}$ (a) `quotient_type int^q = "int" / "(=)" by (rule identity_equivp)`

(b)
```
inductive comeq where
com:  "comeq (x@y) (y@x)"
| recur:  "comeq u v ⟹ comeq (x@u@y) (x@v@y)"
| rlx:  "comeq x x"
| sym:  "comeq x y ⟹ comeq y x"
| trans:  "⟦comeq x y;comeq y z⟧ ⟹ comeq x z"
```
```
quotient_type Bag^q = "Bag" / "comeq"
...
done
```

(c)
```
inductive idmeq where
idem:  "set x = set y ⟹ idmeq x y
| rlx:  "idmeq x x"
| sym:  "idmeq x y ⟹ idmeq y x"
| trans:  "⟦idmeq x y;idmeq y z⟧ ⟹ idmeq x z"
```
```
quotient_type Set^q = "Set" / "idmeq"
...
done
quotient_type Map^q = "Map" / "idmeq"
...
done
```

(d) $\sqsubseteq^{-}$

```
fun s1_eqfun where
"s1_eqfun (s2_s1 (s3_s2 x)) (s3_s1 y) = s1_eqfun x y"
...
inductive s1_eq where
base:  "s1_eqfun x y ⟹ s1_eq x y
| rlx:  "s1_eq x x"
| sym:  "s1_eq x y ⟹ s1_eq y x"
| trans:  "⟦s1_eq x y;s1_eq y z⟧ ⟹ s1_eq x z"
quotient_type s1^q = "s1" / "s1_eq"
...
```

(e) $\sqsubseteq^{-}$

```
fun s1_eqfun where
"s1_eqfun (s3_s1 (s4_s3 x)) (s2_s1 (s4_s2 y)) = s1_eqfun x y"
...
inductive s1_eq where
base:  "s1_eqfun x y ⟹ s1_eq x y
| rlx:  "s1_eq x x"
| sym:  "s1_eq x y ⟹ s1_eq y x"
| trans:  "⟦s1_eq x y;s1_eq y z⟧ ⟹ s1_eq x z"
quotient_type s1^q = "s1" / "s1_eq"
...
```

(f)
```
apply (simp add:equivp_reflp_symp_transp)
apply (rule conjI)
apply (simp add:reflp_def)
apply (simp add:rlx)
(* next on the right *)
```
```
apply (rule conjI)
apply (simp add:symp_def,clarsimp)
apply (simp add:sym)
apply (simp add:transp_def,clarsimp)
apply (simp add:trans)
```

Figure 4.19: Example of Translation to Quotient Types

From Datatypes to Quotient Types. Here we translate the Isabelle datatype set Ω to the quotient type set Ω^q with a set of proofs Π. A quotient type represents a set of terms, with a fixed target sort, whose elements are equivalence classes that are partitioned the whole term domain by a given set of equations. Some datatypes in Ω are only translated to "trivial" quotient types, meaning a quotient type with the Isabelle's builtin `=` operation as its equivalence relation. The translation of *Int* to a quotient type in (*a*) (Fig. 4.1 and Fig. 4.18) is one example, and we just need the one-line proof "`rule identity_equivp`" for such a case.

For any datatype subset of Ω indexed by a specific target sort, there are four cases necessarily needing non-trivial quotient type translations. The general strategy for translating non-trivial cases is to define inductive relations to capture equivalence relations defined for the quotient types, and to prove that these really are such relations. Among these inductive equivalence relation definitions for translating non-trivial cases, we define the `rlx`, `sym`, and `trans` rules to ensure that the definitions are equivalence relations, such as the ones in (*b*) and (*c*) in Fig. 4.19, which capture the implicit equational properties (only the communicative and idempotent properties) hiding in the builtin terms *Bag*, *Map* and *Set*. Case (*b*) deals with the communicative equational property in *Bag* terms. The `com` and `recur` rules capture the communicative relations among the elements in a *BagItem list* (which is a *Bag* term) precisely. In the case, the right figure shows how a quotient type with a proof is defined in Isabelle, and the proof content is in (*f*) in Fig. 4.19. In fact, (*f*) is the generalized proof for every non-trivial case quotient type proof in the translation. Case (*c*) provides an inductive relation capturing the idempotent equational property in the *Set* and *Map* terms. The rule `idem` defines the core equivalence property of two lists of *SetItem* or *MapItem* elements: two lists are equivalent if the set translations of the two lists are the same. As we stated in Fig. 4.12, *Map* terms must also be functional to be valid in a configuration. We incorporate the functional property as a transition rule in Sec. 4.5.2. Cases (*d*) and (*e*) in Fig. 4.19 capture all necessary non-trivial cases translating from datatypes to quotient types for user defined sorts/datatypes, whose general concept has been described in [11]. The process is to identify the possibly equivalent terms due to the removal of the subsort relation and explicit coercions when translating an order-sorted algebra to a many-sorted one. We first manipulate the input subsort relation $\sqsubseteq$ to be the definition of $\sqsubseteq^{-}$ in Fig. 4.19 by eliminating all subsorts related to sort K. Here is the reason. The only immediate subsort of sort K in $\sqsubseteq$ is the sort KItem, and users are not allowed to subsort other sorts to K. The only equivalent terms caused by explicit coercing KItem to K are those recognizing a KItem term as a singleton K term, which will be translated properly at the stage of translating terms and rules (Sec. 4.5.2). Cases (*d*) (and (*e*)) in Fig. 4.19 describe how we generate quotient types when a "line" (and a "diamond") structure is presented in the subsort relation $\sqsubseteq^{-}$. For the terms in the sort marked as yellow in cases (*d*) and (*e*), we generate a function, a relation, and a proof to capture capture the equivalence relations among these terms. Case (d) describes a possible "line" structure in $\sqsubseteq^{-}$, where three different sorts s_1, s_2, and s_3 have subsort relations in a line, like the left graph in (d). In this case, if the terms in s_1 have a combined explicit coercion `(`s_2`_`s_1 `(`s_3`_`s_2 `x))`, it is equivalent to a term being directly coerced from s_3, as `(`s_3`_`s_1 `y)`, provided that the terms `x` and `y` are also equivalent. In Isabelle, we generate a function s_1_eqfun to capture the above description. The ... part contains other trivial cases for two terms in s_1. Sometimes, if a target sort s_1 contains other possible subsort relations fitting patterns in (d) and (e), we also need to take care of those situations in s_1_eqfun. The inductive relation s_1_eq is a trivial equivalence relation wrapper for s_1_eqfun. We can then build the quotient type ${s_1}^q$ based on s_1_eq with the same proof as (f). Case (e) describes a possible "diamond" structure in $\sqsubseteq^{-}$. In four different sorts s_1, s_2, s_3, and s_4, the sorts s_1, s_2, and s_4 have subsort relations, while the sorts s_1, s_3, and s_4 also have subsort relations. In this case, the coercions using different paths from s_4 to s_1 are all equivalent. We implement case (e) by the function s_1_eqfun, relation s_1_eq and quotient type s_1^q, in the same manner for case (d).

We have described how to translate datatypes in $\mathbb{K}$ to quotient types in Isabelle. Next, we will introduce the translation of terms and rules.

4.5.2 Translating $\mathbb{K}$ Terms and Rules

Here we translate the **IsaK** terms and rules described in Sec. 4.3.2. For an **IsaK** theory $(\Psi, \sqsubseteq, \Upsilon, \Delta)$, the translation algorithm for user defined terms simply walks down the ASTs of the terms by adding explicit coercions according to the syntactic translations in Sec. 4.5.1. The only tricky aspect is that the terms translated in Isabelle have no $\mathcal{KLabel}$ or $\mathcal{KList}$ subterms. The translation of builtin terms can be summarized as the translation of **IsaK** configurations to Isabelle ones. The translation algorithm is straightforward with keeping an eye on the symbol table Υ to determine the sort for every cell in the configuration.

Fig. 4.20 (a) is a translated term from the initial configuration in Fig. 4.1; its datatype definition is in Fig. 4.18. To determine the constructor for cell `T` (`BagC` or `MapC`, etc), we look at Υ for the target sort of `T`. Since it has sort `Bag`, we add the constructor `BagC` for cell `T`. If the target sort of a cell (e.g. `key`) subsorts to `K`, we give the cell a constructor `KC`, and turn the cell content to a singleton sort `K` term, such as the translated term `[Int_KItem 0]` in the cell `key`. One of the benefits of using **TransK** instead of **IsaK** in Isabelle is its significantly shorter representations of terms. In fact, the initial configuration in (a) (Fig. 4.20) is one-third the length of what it would be if written in **IsaK**.

Next, we introduce the translation of rules, which is to translate the rule set Δ to a

```
(a) BagC T [BagC threads [BagC thread [KC key [Int_KItem 0],
    KC k [Prog (VarCons x VarUnit) (Assign x (Int_Exp 1))],MapC env []]], KC count [Int_KItem 1],
    MapC heap [MConstr [Int_KItem 0] [Int_KItem 0]], SetC keys [SConstr [Int_KItem 0]]]
```

(b) $c(kl)$::$s \Rightarrow c_1(kl_1)$::s_1 when $c_2(kl_2)$::s_2	$\varphi_1 \wedge ... \wedge \varphi_n \Longrightarrow$ c_ind t_1 t_2
fresh(SetItem(n::Int) S, m) $\Rightarrow$ fresh(S, n) when m <Int n ...	inductive fresh_ind where $[\![m < n$; fresh_ind (S, n) $x_1]\!]$ $\Longrightarrow$ fresh_ind ([SConstr [Int_KItem n]@S, m) x_1 ... definition fresh where "fresh e = (SOME x . fresh_ind e x)"

(c) $t_1 \Rightarrow t_2$ when $c(kl)$::s	$\varphi_1 \wedge ... \wedge \varphi_n \Longrightarrow \tau$_rule t_3 t_4
v:Exp $\curvearrowright \square$ / $y \curvearrowright tl \Rightarrow v$ / $y \curvearrowright tl$ when isKResult(v)	inductive k_rule where ... $[\![t$ = abs_K((Exp_KItem v)#((Div x Exp_Hole)#tl)); isKResult((Exp_KItems v)); t' = abs_K((Div x v)#tl)$]\!]$ $\Longrightarrow$ k_rule t t' ...

```
(d) inductive bag_rule where
  ⟦locate (abs_Bag C) = (C′, t); k_rule (abs_K t) t′⟧ ⟹ bag_rule C (abs_Bag C′[rep_K t′])
  | ⟦C = abs_Bag (BagC T ([BagC threads [BagC thread [KC key [key],KC k []]@xs]@ys, SetC keys S]@zs));
   C′ = abs_Bag (BagC T ([BagC threads ys, SetC keys (SetCut(S, [SetConstr [Int_KItem key]]))]@zs))⟧
   ⟹ bag_rule C C′
  ...
```

```
(e) inductive top where
     ⟦is_map_fun C₁; C = abs_Bag C₁; bag_rule C C′ ⟧ ⟹ top C (Some C′)
   | ⟦¬is_map_fun C₁; C = abs_Bag C₁⟧ ⟹ top C None
```

Figure 4.20: Examples of the Translation of Terms and Rules

set of rules Δ^i, whose elements are all represented as inductive relations in Isabelle. The translated relations are all quantifier-free with all meta-variables represented as universally quantified meta-variables in Isabelle. In Sec. 4.3.2, we introduced the **IsaK** rewriting system by dividing rules into three kinds: function, K, and configuration rules. The rule translation deals with these rules differently. The functional checking step in the common evaluation procedures (Sec. 4.3.2) is disregarded from the rule translation here and will be represented as a specific inductive rule to check that every $\mathcal{Map}$ term in a configuration is functional in the latter part of the section.

Translating Function Rules. We first investigate the translation of function rules. Each rule translation is divided into two parts: a translated inductive relation in Isabelle that captures the meaning of the function rule, and a definition using Hilbert's choice operator to produce the output of the relation. In $\mathbb{K}$, a function is defined as a syntactic definition with several function rewrite rules, whose format is as (b) in Fig. 4.20. Each function is translated to a single inductive relation and a definition using Hilbert's choice operator. Given a subset of the symbol table Υ (as Υ_f) containing only function constructs, and a subset of Δ (as Δ_f) containing only function rules, we produce a set of inductive relations in Isabelle as Δ_f^i containing the translated results of Δ_f. In $\mathbb{K}$, applying a function rule rl_f on a given term t

results in two possibilities: either it terminates and returns the resultant term t', or it never terminates due to endlessly rewriting the condition expression of rl_f. Moreover, in all $\mathbb{K}$ tools, function rule applications are implemented as a transition step in a big-step semantic format, where the function application step produces either an infinite sequence of function rule applications or the result of a finite sequence of function rules being applied to the input term. In **TransK**, we keep this strategy and translate function rules as inductive relations using a big-step semantic format.

Fig. 4.20 (b) describes the translation of a function rule to an Isabelle inductive relation with an example (based on the `fresh` function in Fig. 4.1). For each function label c, we select all function rules belonging to it in Δ_f (having the rule pattern of the top-most constructor being c). We generate an inductive relation header (c`_ind`) in Isabelle (e.g. the `fresh_ind` header in Fig. 4.20 (b)) for the group of rules belonging to c. For a single rule rl_c for the function label c, its translation results in an inductive relation case in the relation c`_ind`, where the term t_1 is the translated term describing the input pattern arguments of rl_c (the kl part). An example of such a pattern is the `(SetItem(`n`:Int)` S`,` m`)` part of the `fresh` function rule; it is translated to `([SConstr [Int_KItem` n `]@`S `,` m `)` in (b). The term t_2 is the translation of the target expression of rl_c. Sometimes we need to call rl_c or other functions recursively, so we might need to use a generated variable (x_1 in Fig. 4.20), and generate an equality in the condition of the inductive rule. The conditions $\varphi_1, ..., \varphi_n$ contains not only the translation of the condition expression of rl_c (the m `<Int` n part), but also the equities to access the recursive or other function calls. The handling of x_1 above is one example. If the rule expression (the t_2 and $c_1(kl_1)::s_1$ parts in (b)) contains other mutually recursive function calls, they also need to be translated into variable terms in t_2, with equities as some conditions in $\varphi_1, ..., \varphi_n$. After we construct the inductive relation for a $\mathbb{K}$ function (or inductive relations for a set of mutually recursive functions), we create a definition with the Hilbert's choice operator `SOME` to force the inductive relation to output terms with the type matching the target sort of the $\mathbb{K}$ function as the Isabelle definition in (b), so that we can use the name of the function in a configuration or other rule expressions directly.

Translating $\mathbb{K}$ and Configuration Rules. The general strategies for translating a K rule or a configuration rule are very similar, as described in Fig. 4.20 (c). They are almost the same as translating a function rule, except that t_3 and t_4 are mostly translated from the terms t_1 and t_2 that appear in the $\mathbb{K}$ or configuration rule. The τ in (c) is either `k` or `bag`. In (c), we show an example of translating the cooling rule example in Fig. 4.1 (actually, the revised of the example in Sec. 4.3.1) to an inductive relation case in the inductive relation `k_rule`. In the translation, the translated terms t and t' are quotient type terms of sort $\mathcal{K}$. This is why we use t and t' as variables and then place equities in the conditions in the case to enforce terms t and t' to be quotient typed terms.

Translating configuration rules is similar to translating K rules, except that configuration rules are applied to the whole program state (configuration). We translate all configuration rules in Δ to cases in an inductive relation named `bag_rule`, with an additional rule case capturing the applications of K rules. Fig. 4.20 (d) shows example rule cases in `bag_rule`. Given an input configuration C, the first case is to apply a K rule to a `k` cell in C. We pre-define a function `locate` in the case, along with the translated Isabelle theory, to locate a `k` cell in C and split C into a context $C[]_{\mathsf{k}}$ and a redex t (the content of the `k` cell). Then we apply the corresponding inductive relation `k_rule` to the quotient type term of t: (`abs_K` t), and merge the context and new redex as $C[\texttt{rep\_K}\ t']_{\mathsf{k}}$ as the new configuration. `abs_K` and `rep_K` are Isabelle functions to get an actual term from a quotient type term in sort K and vice versa. The second rule case in (d) is a translation from the `(j)` example rule in Fig. 4.1. Similar to the situation in translating K rules, the input and output configurations C and C' are quotient types, and their contents are wrapped in the coercion `abs_Bag`. Case (e) in Fig. 4.20 is the `top` inductive relation for a translated Isabelle theory. It implements the functional checking for every $\mathcal{Map}$ term in an input configuration C by a pre-defined function `is_map_fun`; and if the check is valid, then the `bag_rule` relation is allowed to apply to C and to observe one step transition; otherwise, the system enters an error state (`None`).

We introduced different components of **TransK** here. Next we investigate the relation between **IsaK** and **TransK**.

4.6 SOUNDNESS AND COMPLETENESS BETWEEN **ISAK** AND **TRANSK**

Here we construct the relationship between **IsaK** and **TransK**. Fig. 4.21 describes the general soundness and completeness proof diagrams between them. We first look at the underlying system the proofs are based on. **IsaK** is defined in Isabelle, we also implemented **TransK** in Isabelle as well as a small Isabelle system (**Isab**) in Isabelle so that we could capture the semantics of the Isabelle theory translated from an **IsaK** one. If we assume that the generated quotient type proofs are always valid in Isabelle, which is obvious, an Isabelle theory translated by **TransK** only requires the soundness and completeness proofs involving Isabelle datatypes/quotient types and inductive relations. The translated inductive relations have the form as Fig. 4.20 (d), where every relation is a binary one rewriting from a term t_3 to a term t_4 with a list of conditions $\varphi_1, ..., \varphi_n$, such that they are all quantifier-free. In addition, the definitions of functions require the support of Hilbert's choice operator. Thus, the rewriting semantics of the Isabelle system (**Isab**) supporting these features is just a simple typed λ-μ calculus with Hilbert's choice operator and quotient types. **Isab** is based on the λ-μ calculus developed by Matache *et al.* [141], and extended to support the rewriting of the inductive relations described above and definitions using Hilbert's choice operator.

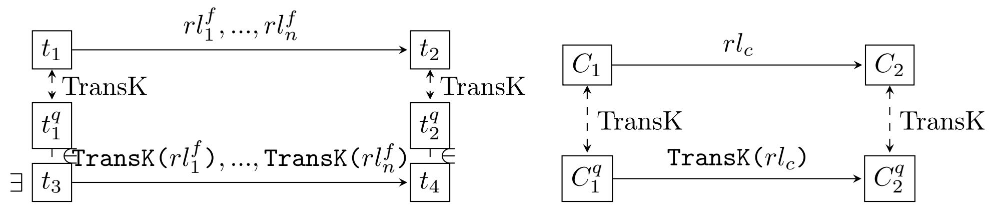

$$\begin{tikzcd}
t_1 \arrow[rrr, "rl^f_1, ..., rl^f_n"] \arrow[d, leftrightarrow, "\text{TransK}"] & & & t_2 \arrow[d, leftrightarrow, "\text{TransK}"] \\
t^q_1 \arrow[d, phantom, "\in"] & & & t^q_2 \arrow[d, phantom, "\in"] \\
\exists\ t_3 \arrow[rrr, "\texttt{TransK}(rl^f_1), ..., \texttt{TransK}(rl^f_n)"] & & & t_4
\end{tikzcd}
\qquad
\begin{tikzcd}
C_1 \arrow[r, "rl_c"] \arrow[d, dashed, leftrightarrow, "\text{TransK}"] & C_2 \arrow[d, dashed, leftrightarrow, "\text{TransK}"] \\
C^q_1 \arrow[r, "\texttt{TransK}(rl_c)"] & C^q_2
\end{tikzcd}$$

Figure 4.21: Soundness and Completeness of **IsaK** and **TransK**

To prove the soundness/completeness between an IsaK theory and its **TransK** translation into **Isab**, we have to prove the soundness and completeness of the functions rules separately from the K/configuration rules (Fig. 4.21). The problem is that every rule in **IsaK** has a conditional expression (having type $\mathscr{Bool}$), and the rewrites of the $\mathscr{Bool}$ term can be infinite. Additionally, the function rules are translated to a definition of inductive relations in the big-step format, and it can be infifnite, too. The soundness and completeness for function rules have to assume that every rewrite of the function rule application on the conditional expression terminates in a finite sequence whose length is n. In addition, a function rule application in Isabelle deals with terms that are the translated datatypes not quotient types. Thus, a function rule is applied to a representative term in a given equivalence class, which is transitioned to another term as a representative in the resulting equivalence class, as described by the existential operation in the first diagram of Fig. 4.21. The term t_3 is a representative of the class t^q_1 which is translated from the $\mathbb{K}$ term t_1. The soundness and completeness of function rule applications are described below.

Theorem 4.2. (Soundness) In **IsaK**, assume that a sequence of function rules $rl^f_1, ..., rl^f_n$ applied to a term t_1 terminates in n steps and results in term t_2, and t^q_1 and t^q_2 are quotient type terms in **Isab** translated by **TransK**, there exists a term t_3 in t^q_1 transitioning through sequence of corresponding rule applications $\texttt{TransK}(rl^f_1), ..., \texttt{TransK}(rl^f_n)$ to term t_4, such that t_4 is in the quotient type class t^q_2, which is a translation from t_2.

(Completeness) If there exist quotient type terms t^q_1 and t^q_2, such that a representative t_3 of t^q_1 is transitioned to t_4 in t^q_2 through a sequence of function rule applications $rl^{f'}_1, ..., rl^{f'}_n$, and $t^q_1 = \texttt{TransK}(t_1)$, $t^q_2 = \texttt{TransK}(t_2)$, and $rl^{f'}_1 = \texttt{TransK}(rl^f_1)$,..., $rl^{f'}_n = \texttt{TransK}(rl^f_n)$, then t_1 is transitioned to t_2 through the sequence of function rule applications $rl^f_1, ..., rl^f_n$.

To show the theorem, we first need to show the following lemma:

Lemma 4.1. For any two transformed terms t and t' in Isabelle, for any k and k', such that $t =$ `TransK(`k`)` and $t' =$ `TransK(`k'`)`, k and k' in **IsaK** are equivalent under the equational properties described in Fig. 4.12.

The proof of the lemma is based on an important assumption about $\mathbb{K}$/**IsaK**. One cannot define two terms in $\mathbb{K}$/**IsaK** with the same constructor. Even if someone makes two identical syntactic definitions in $\mathbb{K}$, the current $\mathbb{K}$ implementation helps map these two definitions with two distinct constructors. A consequence of this assumption is that every term has a least sort in $\mathbb{K}$/**IsaK**.

Proof. The proof is based on induction on the depth of the AST tree of a term t. The base step is simple, and we ignore it here. The inductive step is divisible into three cases:

- If the two terms t and t' are the same term, then the original terms k and k' are also the same.
- If the two terms t and t' are quotient-equivalent involving only $\mathcal{Set}$ and $\mathcal{Map}$ (like case (c) in Fig. 4.19), then k and k' are equivalent under the $\mathcal{Set}$ and $\mathcal{Map}$ equational properties listed in Fig. 4.12.
- If the two terms t and t' are quotient-equivalent through cases (d) and (e) in Fig. 4.19 (the line and diamond structures), then k and k' are the same term. This is because the constructors serving explicit coercions are created in the `TransK` function. The k and k' terms involve only subsort relations, without needing coercions. Once the coercions are removed, the terms k and k' are the same and have the same least sort, since every constructor in $\mathbb{K}$/**IsaK** is unique and has a unique least sort.

QED.

Next, we need to change the semantic rule in Fig. 4.14 a little to the rules in Fig. 4.22. Specifically, we need to change case `(1)` to by adding a number n representing the number of steps an application of the rule can take. We need the step number to avoid the termination proof of a given **IsaK** theory.

With the above rule changes and Lemma 4.1, we can now prove Theorem 4.2.

Proof. We first show its soundness. The proof is rephrased as "for any number n, if applying a rule rl to a term t terminates in n steps, and produces a result of `true` or `false`, then applying `TransK(`rl`)` to the term `TransK(`t`)` also terminates in n step with a result of `True` or `False` in Isabelle."

$$\texttt{(1-a)}\ \frac{}{\texttt{true} \longrightarrow^{\Theta}_{0,f} \texttt{true}} \qquad \texttt{(1-b)}\ \frac{}{\texttt{false} \longrightarrow^{\Theta}_{0,f} \texttt{false}}$$

$$\texttt{(1-c)}\ \frac{c(kl) \neq \texttt{true} \qquad c(kl) \neq \texttt{false}}{c(kl) \longrightarrow^{\Theta}_{0,f} \texttt{error}}$$

$$\texttt{(1-d)}\ \frac{\begin{array}{c} rl \in \Theta \qquad m = \texttt{match}(rl, c(kl)) \qquad n \neq 0 \\ t = \texttt{subs}(m, \texttt{cond}(rl)) \qquad t\ (\longrightarrow^{\Theta}_{(n\texttt{-1}),f})^{*}\ \texttt{true} \end{array}}{c(kl) \longrightarrow^{\Theta}_{n,f} \texttt{subs}(m, \texttt{right}(rl))}$$

Figure 4.22: Changed Rules in **IsaK** For Theorem Proving Purposes

We abbreviate the transition that happens in Isabelle as $\longrightarrow^{tr,\Theta}_{n,f}$.

We induct on the number n. The base step is when the term t is either `true` or `false` (otherwise, it is an `error`). Trivially, when translating the two terms in Isabelle, they are `True` or `False` values in Isabelle.

In the inductive step, for the n+1 step application, we have the assumption that the rule-application case `(1-d)` above is a valid one for term t. It means that a mapping m results from `match(`rl, t`)`. We also have the translation of the rule `TransK`(rl) and the term `TransK`(t). By a case analysis of the different quotient translation cases in Fig. 4.19, we can conclude that there is a term t' which is in the same equivalence class as `TransK`(t), such that we can find a mapping m' for `match(TransK`(rl)`, TransK`(t)`)`; and every mapped term t_i of meta-variable x_i in the mapping m' is in the same equivalence class as the term `TransK`$(m(x_i))$. In addition, translating the condition expression to `true` (through the transitions $(\longrightarrow^{\Theta}_{(n\texttt{+1 -1}),f})^{*}$) is valid because of the inductive hypothesis (n+1 -1 $\leq n$). Thus, applying `TransK`(rl) to the term `TransK`(t) produces a valid term $\overline{t'}$ that is in the equivalence class of t', such that $t \longrightarrow^{\Theta}_{(n\texttt{+1}),f} t'$ via the rule rl.

We then show its completeness. The proof is rephrased as "for any number n, if we have a rule rl and a term t, and their translations `TransK`(rl) and `TransK`(t), then if an application of `TransK`(rl) on the term `TransK`(t) terminates in n steps with a result of `True` or `False` in Isabelle, then applying a rule rl to a term t terminates in n steps, and produces a result of `true` or `false`."

We also induct on the number n. The base step is when the term `TransK`(t) is either `True` or `False` in Isabelle (otherwise, it is an `error`). Trivially, there is a term `true` or `false` in **IsaK** to match with the result term.

In the inductive step, for the n+1 step application, we have the assumption that the rule application case `(1-d)` above is a valid one for the term `TransK`(t), and the rule being applied is $\texttt{TransK}(rl)$. This means that there is a mapping m as a result of $\texttt{match(TransK}(rl)\texttt{,TransK}(t)\texttt{)}$. We need to show that there is also a mapping m' as a result of $\texttt{match(}rl, t\texttt{)}$. We show that there is a term t' which is in the same equivalence class as t in **IsaK** by Lemma 4.1. By the lemma, there is a mapping $\texttt{match(}rl, t'\texttt{)}$, such that every entry of m is the $\texttt{TransK}$ translation result of the entry with the same meta-variable in m'. Furthermore, the terms t and t' are the same term because of the term normalization in **IsaK** introduced in Sections 4.2 and 4.3.2. Every input term for a rule application is assumed to be term-normalized. If two terms are in the same equivalence class, the normalized term for all terms in the equivalence class is the same and it represents the canonical form of the equivalence class. Thus, there is a step transition $t \longrightarrow^{\Theta}_{(n+1),f} t''$ via rule rl, and the translation of term t'' is in the same equivalence class as the term $\overline{t''}$ in the transition $\texttt{TransK}(t) \longrightarrow^{tr,\Theta}_{(n+1),f} \overline{t''}$ via rule $\texttt{match(TransK}(rl)$.

QED.

The proofs of soundness and completeness for K/configuration rule applications are described in the right-hand diagram in Fig. 4.21. A transition step in such cases is a one-step application of a K rule or configuration rule. The soundness and completeness of K/configuration rule applications are described below.

Theorem 4.3. (Soundness) In **IsaK**, assume that a configuration C_1 is transitioned to C_2 through a $\mathbb{K}$ (or configuration) rule rl_c, and C_1^q is a quotient type term translated from C_1, then C_2^q is translated from C_2 by rule $\texttt{TransK(}rl_c\texttt{)}$.

(Completeness) If there exist quotient type configurations C_1^q and C_2^q, such that C_1^q transitions to C_2^q through a rule rl'_c, $C_1^q = \texttt{TransK(}C_1\texttt{)}$, and $rl'_c = \texttt{TransK(}rl_c\texttt{)}$, then C_2 is transitioned from C_1 by rule rl_c and $C_2^q = \texttt{TransK(}C_2\texttt{)}$.

Proof. We first show its soundness. Given an **IsaK** theory Θ and configuration C, we do a structural induction on different kinds of rules in Θ.

- If the application rule rl is a function rule, the configuration is split into a context $C[]^s_f$ and a redex $t{::}s$, such that $t \longrightarrow^{\Theta}_{f} t'$, and the final configuration is $C[t']^s_f$. The final configuration is well-typed because of Theorem 4.1. We then need to show that a translation of rl as $\texttt{TransK}(rl)$ can also be applied to term $\texttt{TransK}(t)$, and it is transitioned to a term $\overline{t'}$, which is in the same equivalence class of the translation result of t'. This step of the proof is similar to the one in proving Theorem 4.2, with the assumption that the rewrites of the condition expression of the rl rule with term t produces a Boolean result of `true` or `false` in n steps. We then need to show that the final configuration from applying function rule $\texttt{TransK}(rl)$ is in the same equivalence class as the term $\texttt{TransK}(C[t']^s_f)$. To accomplish this, we have a lemma stating that the application of an Isabelle `definition` to a term $\texttt{TransK}(C)$ (as in the case (b) translation result in Fig. 4.20) produces a term with the context $C'[]^s_f$ and redex $\overline{t'}\texttt{::}s$, where t' and the hole in $C'[]^s_f$ also have the type s, $\overline{t'}$ is in the same equivalence class as $\texttt{TransK}(t')$, and $C'[\overline{t'}]^s_f$ is in the same equivalence class as the term $\texttt{TransK}(C[t']^s_f)$.

- If the application rule rl is a K rule, the configuration is split into a context $C[]_{\mathsf{k}}$ and a redex k having sort K, such that applying rl to k results in k', and the final configuration is $C[k']_{\mathsf{k}}$. The final configuration is well-typed because of Theorem 4.1. We then need to show that the translation of rl as $\texttt{TransK}(rl)$ can be applied to the term $\texttt{TransK}(k)$ (like the `k_rule` translation in Fig. 4.20 (c)), and transition the term to a term $\overline{k'}$ that is in the equivalence class of term $\texttt{TransK}(k')$. This step of the proof is basically the same as in Theorem 4.2. Finally, we need to show that the transition rule in Isabelle (like the first line of Fig. 4.20 (d)) can split the configuration $\texttt{TransK}(C)$ into a context $C'[]_{\mathsf{k}}$ and redex $\overline{k}$, such that $\overline{k}$ is in the same equivalence class as $\texttt{TransK}(k)$, and $C'[k]_{\mathsf{k}}$ is in the same equivalence class as $\texttt{TransK}(C)$; and we need to show that the application of $\texttt{TransK}(rl)$ to $\overline{k}$ results in $\overline{k'}$, and $C'[\overline{k'}]_{\mathsf{k}}$ is in the same equivalence class as $\texttt{TransK}(C[k']_{\mathsf{k}})$. The proof of this step is done by a structural induction on any Isabelle term with the application of a special inductive rule like the one in the first line of Fig. 4.20 (d).

- If the application rule rl is a configuration rule, then we only need to show that when the configuration is transitioned to C' via the rule, the translation of C to $\texttt{TransK}(C)$ is also transitioned to $\overline{C'}$ via the rule $\texttt{TransK}(rl)$, such that $\overline{C'}$ is in the same equivalence class as $\texttt{TransK}(C')$. This is done by the same strategy as the proof of Theorem 4.2.

We then show its completeness. The proof is to do a rule induction on the `top` inductive relation listed in Fig. 4.20 (e), and then establish a sub-theorem by doing an induction on the rules in Fig. 4.20 (d). There are three important cases here:

- If the rule being applied is not the first line of Fig. 4.20 (d), then the rule is the translation of a configuration rule rl in the given **IsaK** theory Θ. By a similar proof strategy as that for proving Theorem 4.2, we can then show that for any configuration $\overline{C} = \texttt{TransK}(C)$, the application of $\texttt{TransK}(rl)$ results in a new configuration $\overline{C'}$; the configuration C is also transitioned to a new configuration C' via rule rl; and $\texttt{TransK}(C')$ is in the same equivalence class as $\overline{C'}$.

- If the rule being applied is the first line of Fig. 4.20 (d), then the rule is the translation of K rule rl in the given **IsaK** theory Θ. Then, the given configuration $\overline{C} = \texttt{TransK}(C)$

is split into a context $\overline{C}[]_{\mathsf{k}}$ and redex $\overline{k}$. The rule application of $\texttt{TransK}(rl)$ to $\overline{k}$ results in a term $\overline{k'}$. We need to show that if $\overline{k} = \texttt{TransK}(k)$, then k transitions to k' via the K term transition part of rule rl, such that $\overline{k'}$ is in the same equivalence class as term $\texttt{TransK}(k')$. The proof of the existence of the term uses the same strategy as the one in the completeness proof of Theorem 4.2. Finally, we need to show that C is transitioned to $C[k']_{\mathsf{k}}$ via rule rl, such that $\overline{C}[\overline{k'}]_{\mathsf{k}}$ is in the same equivalence class as $C[k']_{\mathsf{k}}$. This proof is done by a similar strategy in proving the existence of the context-redex combination in the soundness proof of an K rule above.

- If a given configuration $\overline{C}$ contains an Isabelle definition t (like Fig. 4.20 (b)), then the definition must be translated from a function constructor appearing in the **IsaK** theory Θ. In this case, configuration $\overline{C}$ is moved through several transitions to a final step $\overline{C'}$ such that the definition is replaced by the ground term t' without any Isabelle definitions. In this case, $\overline{C}$ can be viewed as the split of a context $\overline{C}[]$ and a redex t, such that $\overline{C}[t] = \overline{C}$ and $\overline{C}[t'] = \overline{C'}$. We have shown inductively the following. For n steps of applications of the inductive rule associated with the definition $\overline{t}$, it is continuously translated to $\overline{t_1}, ..., \overline{t_n}$. For any rewrite from $\overline{t_i}$ to $\overline{t_{i+1}}$, the transition is through the inductive rule $\texttt{TransK}(rl)$, which is translated from a function rule rl in Θ. Let $\overline{t_i} = \texttt{TransK}(t_i)$, then t_i transitions to t_{i+1} via the application of rule rl, and $\texttt{TransK}(t_{i+1})$ is in the same equivalence class as $\overline{t_{i+1}}$. Thus, for any n step computations of inductive rule applications for an Isabelle definition, there are n step computations of function rule applications, and the final term t_n computed in the **IsaK** system is translated into the same equivalence class as the final result term $\overline{t_n}$ computed in the Isabelle system. Finally, let $\overline{t'} = \overline{t_n}$. We need to show that if $\overline{C} = \texttt{TransK}(C)$, $\overline{t} = \texttt{TransK}(t)$, and $\overline{t'} = \overline{t_n} = \texttt{TransK}(t_n)$, then $\texttt{TransK}(C[t_n])$ is in the same equivalence class as $\overline{C}[t']$. The proof of this step is the same process as we have already performed multiple times.

QED.

Chapter 5: KLLVM: THE SEMANTICS OF LLVM IR

In this section, we define the semantics of **K-LLVM**. It is divided into two parts: the **K-LLVM** static semantics (Sec. 5.2) and the **K-LLVM** dynamic semantics (Sec. 5.3).

The Low Level Virtual Machine (LLVM) is designed for the compile-time, link-time and run-time optimizations of programs written in unspecified programming languages. An LLVM-based compiler, such as Clang, relies on a translation from a high-level source language to an intermediate representation (LLVM IR) that hides details about the specific target execution platform and acts as an interface for LLVM. Then users are able to use the LLVM tools to perform program optimizations, transformations, and static analyses based on LLVM IR, which can also be translated into target architectures such as x86, PowerPC, and ARM. Hence, LLVM IR acts as a "central station" for translating high-level languages to target architectures, with a fixed set of language syntax, instructions, library functions, and memory model [142].

The full details of our semantics can be found in the following link: `https://github.com/liyili2/llvm-semantics-1`. This chapter highlights an interesting portion of **K-LLVM** to show how one can possibly find a balance between abstractions and real world programming to provide a better, clearer, and more useful language semantics. First, we introduce some benefits, features, and a limitation of **K-LLVM**. In this chapter, the **K-LLVM** memory model is based on byte-wise sequential consistency. LLVM IR specifies a range of behaviors for memory operations with different orderings and for `volatile` memory accesses, while **K-LLVM** does not support the full range. In **K-LLVM**, every memory location is mapped to a single byte datum; there is only one memory cache to deal with all memory operation requests from the different threads. Single thread instruction execution is in the program order. Based on this model with the **K-LLVM** abstract machine, we provide an observation in Section 5.3.4. We implemented the full LLVM IR concurrency model in $\mathbb{K}$ with all of the memory ordering behaviors of the atomic memory operations in Chapter 7.

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

A Unified and Rigorously Mathematical Framework. We provide a unified and rigorously mathematical framework where people can observe the semantic behaviors in a single interface and also prove properties of compilers, with a focus on LLVM IR and LLVM IR compilers. Transforming programs from a high-level language to a low-level machine code requires a lot of phases, each of which might cause correctness concerns. For example, the infamous out-of-thin-air problems can arise at every level of intermediate AST as a result of a transformation or compiler optimization. They can even appear when some old processors try to execute certain programs [143]. **K-LLVM** provides a way for users to reason about the behaviors of these translations based on the rigorous executable semantics of LLVM IR.

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

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

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

Undefined Behavior. We classify three different types of undefinedness in LLVM IR. The first one is `undef`, which represents an unspecified value for a program position; the program should proceed no matter what the value is. In some cases, `undef` also means that the program has ill-defined behavior, such as representing a race in the memory. There are two ways to deal with `undef` in **K-LLVM**: ***krun*** can be used to execute a program with `undef` and get a fixed deterministic behavior by assuming one path, or ***ksearch*** can be used to search for all different behaviors by executing the program non-deterministically. Sometimes, the non-deterministic search space caused by `undef` values in LLVM IR is too large. In such cases, the symbolic execution engine in ***ksearch*** with the $\mathbb{K}$ equivalence checker can be used to determine if two programs return the same results. Additional discussion can be found in Section 5.4. The second kind of undefinedness is an undefined behavior represented by a poison value, because LLVM IR does not have a defined symbol for it. Its meaning is similar to `undef`, but it has certain undefined behaviors associated with it. **K-LLVM** will carry the poison value and continue computation until a non-deterministic point is reached, then give an error message saying that there is a poison value in the program, and stop the continuation of the computation. If no non-deterministic point is found, **K-LLVM** can finish the computation successfully. The third kind results from underspecification in the LLVM IR documentation. We named this as **unspecified behaviors** in this chapter. When facing the third kind, **K-LLVM** immediately labels the computation an error state, saying there is an unspecified behavior in the system. More information can be found in Section 5.4.

Independent of $\mathbb{K}$. The implementation in $\mathbb{K}$ gives **K-LLVM** the power to have an interpreter automatically, and have tools for state space searching and symbolic executions. Essentially, $\mathbb{K}$ [76] is an executable semantic framework based a rewriting logic [134]. Once a language semantics is defined in $\mathbb{K}$, it automatically turns it into a logical form by turning each semantic rule into an axiomatic rule with pre and post-conditions; thus, it creates an axiom set for the language. Additionally, there are many tools available in $\mathbb{K}$. For example, ***kompile*** can be used to see if the semantics has static type problems and to generate an interpreter, so that ***krun*** can be used with the interpreter to test their semantics by actual concrete programs. ***ksearch*** allows searches of traces of multi-threaded programs based on the interpreter. The symbolic engine in ***ksearch*** and the program equivalence checker in $\mathbb{K}$ can allow for two sets of traces to be compared by symbolically executing two different multi-threaded programs and seeing if the two sets produce the same output. Even though we have defined **K-LLVM** in $\mathbb{K}$, the semantics is independent of its implementations in $\mathbb{K}$. In fact, we have defined the **K-LLVM** abstract machine in Isabelle [19] for manually proving theorems about **K-LLVM**. Additional discussion is presented in Section 5.4.

5.1 **K-LLVM** BACKGROUND AND CHALLENGES

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

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

The LLVM language (LLVM IR) is a statically and strongly typed, assembly-like, Static Single Assignment (SSA) based language. It has undefined behaviors but the undefinedness is well documented. The LLVM language itself does not have operations or libraries to support multi-threaded behaviors, but LLVM IR's structure is highly related to the C/C++ library. LLVM IR basically assumes a runtime environment of C++. LLVM IR also contains a set of functions comprising an intrinsic library, in which part of the standard C library is included.

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

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

Figure 5.1: LLVM IR Example Programs

It also relies on other functions in the `stdlib.h` header. For example, it needs dynamic memory management functions such as `malloc`, `realloc` and `free` to provide heap memory access, as well as functions dealing with the environment such as `abort`, `exit` and `system`. Furthermore, it needs functions listed in the `stdio.h` header to provide I/O support, as well as library functions from the Pthread and Pthread-mutex libraries to provide threading and mutual exclusion behaviors. These functions are not strictly part of the LLVM IR listed in the documentation but we define them anyway.

The current LLVM IR can be viewed as "C- -". Except function bodies, most features in C can be found in LLVM IR, such as global variables, `struct` datatypes, function headers and different flags for global variables or functions, etc. The main difference between LLVM IR programs and C programs are the function bodies, a.k.a. expressions. The LLVM IR expressions are register-based, SSA based and assembly-like. These features eliminate the undefinedness of the evaluation order in an LLVM IR program. We show some examples of LLVM IR expressions in Figure 5.1 to provide a taste of LLVM IR. These expressions are used throughout the whole chapter. We believe that these expressions are enough to show the key features of LLVM IR and the construction of LLVM IR programs based on these expressions and other components (function headers, global variables and modules, etc) can be easily found in the LLVM documentation. This is also the reason we refer to these expressions as "programs" in the rest of the chapter.

LLVM IR distinguishes local variables from global variables. Variables starting with the character % are local ones, while those starting with the character @ are global. Global variables can only have a pointer type. Any number following the character `i` in LLVM IR, such as `i32` or `i1`, means an integer type declaration with the size of the bits. `i32*` refers to a 32-bit integer pointer type declaration. Instructions starting with the keyword `icmp` are the integer comparison operators. With the keyword `eq`, the instruction *%r8* `= icmp eq i64` *%r7* `, 47244640267` tests whether the value in the variable *%r7* and `47244640267` are the same and stores the result to the variable *%r8*. The "`;`" operation allows users to put comments after a line of code.

`Program-A` does several pointer arithmetic operations and memory operations. Several key observations about LLVM IR are made here. First, `getelementptr` is a memory address calculation operation and has an `inbounds` flag. The definition of `inbounds` is hard because it not only affects the final result but also affects every intermediate result of computing the memory address. For example, in `Program-A`, *%u1* (line 5) is a poison value because we have `inbounds` in the `getelementptr`, and the second index is `i64 -1`, which makes the intermediate result out-of-bounds. Even though the final result is in bounds because we add back numbers, the `inbounds` still makes the final result a poison value. We talk about our definition of the `getelementptr` operation in Section 5.3.4. Second, LLVM IR views the main memory as having no type. We can store an array `[11, 11]` (line 13) and magically get back the `i64` value 47244640267 (line 16). This has effects on defining the **K-LLVM** type system, which will be explained in Section 5.2. Finally, executing `Program-A` in **K-LLVM** stops at the line 22. It is an unspecified behavior in LLVM IR to read data from a memory location pointed to by a pointer that was not properly created. The combined effects of casting, pointer arithmetic, and memory operations are based on the definition of the provenance model in **K-LLVM**. More details are in Section 5.3.1 and 5.3.4. `Program-B` and `Program-C` distinguish between a non-atomic and atomic memory operation. Thanks to our **K-LLVM** virtual machine definition, we are able to produce the race caused by two non-atomic operations in two different threads. Additional details are in Section 5.2 and 5.3. While maintaining sequential consistency, the execution of `Program-D` could result in a race on $@x$ because of the special instruction execution order of LLVM, which the **K-LLVM** abstract machine models. More details are in Section 5.3.2. `Program-E` is an example for showing the usage of the $\mathbb{K}$ symbolic execution engine in Section 5.4.

After reading the programs in Figure 5.1, questions about the memory locations and memory alignments may come to mind. Memory implementation is very complicated in real world programming languages. LLVM IR does not actually fix a special implementation of memory addresses. For simplicity, we assume in this chapter that there is a one-to-one mapping from natural numbers to memory addresses, and a memory chunk is always in a range that can be defined between a left and a right integer bound. The memory addresses refer to conceptual memory byte data. Conceptual memory bytes are not actual byte data – details are in Section 5.3.3. LLVM IR also allows setting up alignments for different types, memory endianness and address space information by using `target datalayout`. Although we have implemented these features in **K-LLVM**, for simplicity, we assume in this chapter that alignments, paddings for `structs` and address spaces never cause a problem in calculating memory addresses or type checking, and we assume little-endian byte-order. Finally, we assume that the heap size is infinite while the stack for each function is finite and has a maximum bound, and if a stack overflows in a thread, the whole system reaches an error state.

5.1.2 K-LLVM Challenges

Here are some challenges that we face when we define the **K-LLVM** semantics.

Sheer Size of LLVM IR. The first challenge is the sheer size and precision of LLVM IR. With respect to instructions, LLVM IR has more than 60 operators and 100 intrinsic library functions. Some operators have complex rules or different requirements according to the input. For example, `store` operators can be either non-atomic or atomic, and atomic `store` operators have six different orderings. All of these require different semantic rules. The previous work only defined some of the operators, or some of their features. No previous work has defined the massive number of intrinsic library functions. **K-LLVM** defines all the LLVM operations and intrinsic functions. We handle this challenge through a special heavily testing strategy to define **K-LLVM** described in Sec. 5.4.

Subtlety of Well-formedness. In LLVM IR, the subtlety of various instructions and the well-formedness of instructions are often directly connected with the semantics of the instructions in a particular place in a given program. The syntactic nature of even a single instruction is determined by the semantic context. For example, the `getelementptr` operator allows indices to be integer local variables if the pointer input is an `array` pointer. However, if it is a `struct` pointer, LLVM IR requires the indices to be integer constants that can be statically reduced to integer values. These two types can be mixed together in a single usage of `getelementptr` in an LLVM IR program. Another example is that the input containing a decimal representation of a floating-point constant needs to be exact. This means that the value `1.1` cannot be a valid constant for floating-point operators in LLVM IR because it cannot be precisely represented by a finite floating point number, and LLVM IR requires the compilers to LLVM IR to round the float to a hexadecimal format. This is an error in Clang (the LLVM compiler).

Detailed Low-Level Features. As we mentioned in the beginning of the chapter, it is not feasible to to gloss over the details of LLVM IR's low-level features, such as how to represent integers, floats and pointers. The effects are easily felt when we combine casting operations with memory operations. It is a common source of confusion among LLVM IR users, and thus, a common source of bugs. We also need to admit the fact that memory locations are highly related to integer behaviors; so converting pointers to integers, doing certain arithmetic on them, and converting them back to pointers are valid program exercises within a memory chunk created by a `malloc` operation. This brings us a big challenge. For example, in `Program-A` (Fig. 5.1), we cannot use pointer *%r9* to store data to the main memory (line 22), even if it is accidentally at the right range of a memory chunk, because *%r9* is not a valid pointer according to the LLVM IR pointer-aliasing rule. Defining a data structure to capture the behaviors covering all corner cases is one of the key contributions of **K-LLVM**. In addition, it is important to admit that the low-level structure of LLVM IR is based on bits and bytes; as well as the integer, float and pointer calculations are based on two's compliments, IEEE 754, and integer pointer calculations.

Instructions Having Side-effects on Subsequent Instructions. Some instructions may cause side-effects on subsequent instructions depending on their behaviors. For example, in `Program-A` (Fig. 5.1), one can use the pointer `%r4` to access memory because it was a subsequent computation result of the pointer `%r1` from a `malloc` function, while `%r9` cannot be used to access memory data because it is from an integer constant. Defining these complicated side-effects requires new ideas. In addition, LLVM IR instructions can have very different requirements for different computer components. This complicates the design of different components of the **K-LLVM** abstract machine.

5.2 THE STATIC SEMANTICS OF ISAK

When giving the semantics of LLVM IR, **K-LLVM** uses two different ASTs, a front-end AST (FAST) and a back-end AST (BAST). The syntax of LLVM IR 6.0.0, which is documented in the website `http://releases.llvm.org/6.0.0/docs/LangRef.html`, is directly parsed into the FAST. We have formally defined the LLVM IR 6.0 syntax in $\mathbb{K}$, and it parses any LLVM IR program into the FAST format.

K-LLVM static semantics refers to the LLVM IR behaviors that happen at compilation time. For an LLVM IR program, parsing is not enough to rule out unqualified programs. After parsing, a series of checks are needed to perform on an LLVM IR program including well-typedness, static single assignment and well-formedness. The **K-LLVM** static semantics is to apply these checks and rule out unqualified programs, and it also translates a FAST program into a representation in the BAST format; then, hands to the dynamic semantics for execution. Figure 5.2 depicts the phases in the **K-LLVM** static semantics.

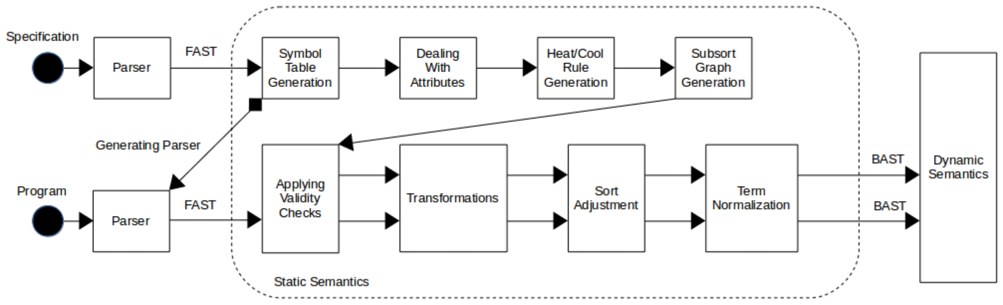


Figure 5.2: Static Semantics to Dynamic Semantics in **K-LLVM**

Preprocessing. The preprocessing step mainly deals with LLVM IR programs outside function bodies. It takes care of LLVM IR compilation jobs during the linkage time, such as discovering module dependency and recording them, analyzing and dealing with LLVM IR metadata, target triples, global/static variables and aliases, etc. 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/static 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/static variable is also given a static memory location, and its data is stored there. After dealing with the global/static variables, aliases are also analyzed and transformed to the BAST format. Aliases rely on the information from the global variables to get the pointer address of the object aliases. During the preprocessing of the global/static variables and aliases, small type checks are performed to guarantee that the values of the global/static and target variables of any alias are coherent. In addition, syntactic sugar resolution is performed so that the possible constant expressions of the aliasee expression in each alias will be correctly evaluated to their pointer values. Finally, for function declarations and definitions, the preprocessing phase only collects their header information including return types, argument types and attributes. The type checking and transformation of the bodies of the function definitions are done in the next few phases.

Type Checking. This step emulates the behaviors of LLVM IR type checking for functions in LLVM IR modules. LLVM IR is a strongly typed language even though its type system is very straight-forward and simple. The **K-LLVM** type checking process is a complete implementation of the LLVM IR type system listed in the its documentation. The input for the **K-LLVM** type checking function is a term and its type, and the function outputs `true` if the term has been type checked and has the input type, and it outputs `false` otherwise. The "strongly typed" here means that LLVM IR guarantees that a typed value produced from a typed LLVM IR expression is compatible with the size of the value in runtime, and any later usage of the value will not result in a type error or size error. However, the program still has a chance to go wrong due to other problems such as division by zero. in `Program-A` in Figure 5.1, every line of code except `store` and `br` instructions assigns a value to a variable. After type checking, each variable has a type. *%r1* has type `i8*` (line 1) and *%r2* has type `[3 x i32]*` (line 2). If we eliminate line 2 and replace the variable *%r2* in line 3 with *%r1*, the line results in a type error.

There are some tricky cases of the type system. In Figure 5.3, we show a `getelementptr` instruction on a `struct` type. For a `struct`, the value of the index for the `getelementptr` affects the type result of the final value of an instruction, because every position in a `struct` can have different type. Type checking a `getelementptr` relies on executing part of the semantics of the `getelementptr` arguments. That is why some index values of `getelementptr` that are associated to `struct` type positions are required to be input-time calculable. This means that such positions can contain neither local nor global/static variables, even if a constant expression (no variables inside) is allowed. For other non-`struct` index positions, variables are allowed, such as the x `getelementptr` in Figure 5.3.

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

Figure 5.3: Type Definitions and Examples

LLVM IR takes the view that values stored in the memory have no types, and that memory instructions will always produce values of the prescribed types. In fact, LLVM IR does not have a clear idea of main memory. It does not even have a built-in memory allocation instruction, instead, it relies on Standard-C library to provide such instructions. It basically assumes that the memory machine as a black box, and every memory request is **valid** as long as the size of the requested data matches the size of its type, the memory pointer is not out-of-range, and there is no race. In addition, one can have a correctly typed program where the result value produced by the program does not make sense. For example, loading items from the main memory can result in poison values or undefined values (`undef` in LLVM IR) in some cases. In such cases, the LLVM IR type system is of limited values. To support the type system, we need to create two extra constructs in **K-LLVM** (in Fig. 5.3), one for poison values and the other for undefined values, with each carrying additional type information. As a result, poison values and undefined values are group of values with one for each type in **K-LLVM**. Combining all these features of LLVM IR type system, we have shown the following type preservation theorem:

Theorem 5.1. 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 LLVM IR program (as term t) in the FAST format, φ either produces $\bot$, meaning that t is not a valid type checked or validity checked program, or term t', a transformation of t in the BAST format. A transition system κ exists to implement 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 for achieving the history of all instructions produced for a thread. For example, $\texttt{continuation}(tid, \kappa, t, \Delta)$ produces a record of 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 state (t, Δ) are treated as no-op. The component name `registers` of the **K-LLVM** abstract machine is overloaded as a function for achieving the current status of registers for a state. We first prove a lemma to relate `registers` to input arguments of a function call. The component name `stack` of the **K-LLVM** abstract machine is overloaded as a function for achieving the current status of the call stack of a state. The construct $|\texttt{stack}|$ provides the maximum size of a stack in the **K-LLVM** abstract machine.

Lemma 5.1. Assuming that t is an LLVM IR program for a thread *tid*, and $\varphi(t) = t'$, so that program t is checked by function φ, and the BAST term t' is produced, and Δ is the initial environment for t'; Then, $\sigma = \texttt{registers}(t, \Delta)$ are the initial registers in Δ. If a valid state (t'', Δ'') exists, such that $(t', \Delta) \rightarrow^{+}_{\kappa} (t'', \Delta'')$, and $s = \texttt{continuation}(tid, \kappa, t'', \Delta'')$ is the history of all instructions occurring 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) for 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.

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 the length of s, and do a case analysis of all possible instructions for s. Since, in **K-LLVM**, φ checks the type of any use for any input argument matching the type of the argument defined in a function header, (1) is valid. (2) is valid because LLVM IR programs are 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 used for any local variable x in an instruction if it has not been 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, 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: let the maximum stack size be $k + 1$. We have three cases. First, if in a state (t'', Δ'') with instruction history $s = \texttt{continuation}(tid, \kappa, t'', \Delta'')$ and the current stack size 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), statements (1) and (2) are valid. If the current stack size is less than k, and the next instruction to execute is a function call, by the inductive hypothesis, statements (1) and (2) are valid. In sum, we have validated the proof of **Part 2**.

By the **Parts 1** and **2**, we have shown that the Lemma 5.1 is valid.

QED.

Based on the information above, we first prove the single thread case. That is, in the execution of a program t, a thread creation instruction is never triggered.

Proof. The theorem is proved by case analyzing 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' be 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 $(t', \Delta) \to_{\kappa}^{+} (t'', \Delta'')$ for any (t'', Δ'') through some transition steps. Let $s = \texttt{continuation}(\kappa, t'', \Delta'')$. The proof is done by induction on the length of the steps of the transitions $(t', \Delta) \to_{\kappa}^{+} (t'', \Delta'')$ starting from length equal to one.

Base case: we do a case analysis of all possible instructions in the start position of s, with the initial registers $\texttt{registers}(t', \Delta)$. If the instruction is a function call, Lemma 5.1 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 5.1, the fact that this is the first transition state, and the system only executes one instruction, its uses of variables must come from the input arguments of the current function, and the type checked property is also guaranteed by Lemma 5.1.

> **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 of all possible instructions that i can be. If the instruction is a function call, Lemma 5.1 guarantees the correctly type checked property in all instructions of the new function. The rest of the case analyzes the situations when the instruction is not a function call. Now, since the term t' is type checked through the function φ, 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 its use. We notice that all instructions in s' are valid executions of instructions in 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 every use of a 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 generated the BAST term t'.

Hence, we have shown that the single thread case of Theorem 5.1 is valid.

QED.

After we have the single thread proof for Theorem 5.1, we can prove the multi-threaded cases. We define the depth of the thread chain as the longest thread creation sequence in the execution of a program. For example, if thread x_1 has a thread creation instruction, it creates thread x_2; x_2 creates another thread x_3, and so on. Finally, the last thread x_n created in this chain is the one that does not create another thread. The number n is the depth of the thread chain.

Proof. We prove the multi-threaded version of Theorem 5.1 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 $\varphi(t) = t'$ is 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. Proof 5.2 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 the threads execute $t_1...t_w$ different BAST terms. t_i is the one with the longest thread chain, whose depth is k. In this setting, we assume that $t_1...t_w$ never produces errors due to type mismatches.

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, which execute $t_1...t_v$ different BAST terms. For any one thread tid_h, if 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 t_x is type checked due to the inductive hypothesis. Without losing generality, we assume at point s' that tid_x creates the thread tid_h, $s' = s@[is]$, and $s = \texttt{continuation}(tid_x, \kappa, t', \Delta')$. By the argument from Proof 5.2, s contains all valid executed instructions. Thus, before the state (t', Δ'), the system does not have a type mismatch. In the transition $(t', \Delta') \rightarrow_\kappa (t', \Delta')$, the system creates thread tid_h with some input arguments from $\texttt{registers}(t', \Delta')$, which are type correct, as are the input perimeters of the code in t_h. After the thread tid_h dies, the return value does not type-mismatch with the subsequent use of the value, according to Proof 5.2. Hence, the thread tid_x creating t_h does not have a type mismatch problem.

In thread tid_h, since its thread chain depth is $k+1$, its program t_h cannot contain any more thread creations. In this case, following the proof of Proof 5.2 exactly validates the proof.

Hence, we have shown that the multi-threaded version of Theorem 5.1 is valid.

QED.

Constant Expression Rewriting. This step rewrites constant expressions in an LLVM IR program with values or variables. Constant expressions are terms in LLVM IR that can be used to express a complicated term in a constant position. For example, in the `getelementptr` operators below, the term `i32 inttoptr (i32*` $@a$ `to i32)` is a constant expression. Constant expression rewriting is a compilation time process in LLVM IR. It is not hard, but can be very confusing in some cases. Two other 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 in a constant expression to be ready in parsing time. For example, two `getelementptr` instructions use constant expressions (`inttoptr (i32*` $@a$ `to i32)` and `add (i32 1, i32 0)`) in index constant positions as follows:

(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/static 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 in the early compilation time when static field addresses are not known, but the pointer address for $@a$ is known in the late compilation time. This is why the first `getelementptr` instruction is not valid. When we define a function for rewriting constant expressions, we need not only to input the constant expressions and a map containing information about global/static 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 a constant or not. The output of the function is the values rewritten from the constant expressions.

Transformation. The K-LLVM transformation phase regularizes function bodies in FAST and rewrites them in BAST. In LLVM IR, a function body contains several basic blocks. Each has a list of instructions and might or might not have a label name. We transform each basic block into a construct with two arguments. The first one is the label name for the block. If a basic block does not have a label name, we generate one for it as LLVM IR does. The second argument is a map from instruction position numbers to instructions. Each instruction is associated with a unique position number in a basic block. There are two kinds of instructions in LLVM IR. The first computes values and assigns them to local variables in registers, and the second does not return values. Each instruction in this phase is transformed from the position number into a construct named `instNumInfo`. The construct has three arguments: the position number, the compiled instruction in BAST, and the type of the instruction. We type LLVM IR instructions based on their functionality. For example, branching, function call, and return instructions are classified as different types, and they contain different arithmetic, comparison, and array instructions. The different types of instructions are useful in doing analysis of LLVM IR programs such as the use of the Morpheus tool in $\mathbb{K}$ (in Section 5.4).

Validity Checks. Other than the type checking and some checks introduced above, 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.* [62] defined the procedure to ensure Static Single Assignment (SSA) form in an LLVM IR program by using Kildall's method [144]. A little procedural modification in **K-LLVM** is that the local variables used in a `phi` are set apart. They only require variables being defined through all paths backwards from the end of the block indicated by their label name to the entry block. On the other hand, if a non-`phi` instruction uses a local variable, 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 a 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 other instructions in a block. Fourth, if there is a `blockaddress` value in a function, 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 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, the instructions `catchswitch`, `catchret`, `cleanupret`, `catchpad` and `cleanuppad` also have checks similar to those of `landingpad`, `invoke` and `resume`.

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

5.3 THE **K-LLVM** DYNAMIC SEMANTICS

The dynamic semantics of **K-LLVM** executed BAST programs in an abstract machine style. We first discuss the pointer provenance model and the abstract machine model of the **K-LLVM** dynamic semantics.

5.3.1 The **K-LLVM** Pointer Provenance Model

In a language, a pointer provenance model defines the behaviors concerning how pointers are generated, how memory locations are calculated, how pointer address offsets are computed and validated, and how memory data are stored. The combined effects of these items seriously influence the language's consistency and the compilation correctness proofs from a high-level language to machine-level code. The pointer provenance model in **K-LLVM** was developed based on the de facto Standards by Memarian *et al.* [119, 145] and consideration of other current projects on developing pointer provenance models for the "fat pointer" approach [117, 118]. The de facto provenance model is not based on the C standards printed in the book. Instead, it is based on an understanding of the needs of programmers for some useful but unconventional usage of pointer arithmetic. They did a survey [146] querying many programmers and came up with the de facto Standards.

The de facto provenance model developed by Memarian *et al.* has two branches. The PVI branch is a semantics that tracks provenance via integer computation, associating a provenance with all integer values (not just pointer values), preserving provenance through integer/pointer casts, and making some particular choices for the provenance results of integer and pointer + / - integer operations. The PNVI branch is a semantics that does not track provenance via integers; instead, at integer-to-pointer cast points, it checks whether the given address points are within a live object and, if so, recreates the corresponding provenance.

There are three major problems in the de facto model. First, there is actually no way to unify the PVI and PNVI models in a framework. The latter model is based on the assumption that there is a set S of abstract memory addresses. Any actual memory address has the form (s, i) where $s \in S$ and i is an offset integer. However, in the current PNVI settings, there is no way to instantiate an arbitrary address pair (s, i) to an integer memory address, like the integer addresses in the PVI model, which is the reason why we cannot unify the two models. A side-effect is that the PNVI model might not be practical in real-world implementations. Most real-world implementations are based on pointers being integers, no matter whether they are 32-bits, 64-bits, or 256-bits. The best effort to make a real-world implementation of the PNVI model is to split an n-bit integer pointer address to a model as $a + b$ address, such that $a + b = n$; using the first a bits to represent the abstract address s, and the last b bits to represent the offset i. However, an assumption in the PNVI model is that no two pairs (s_1, i_1) and (s_2, i_2) can be identical if s_1 and s_2 are different. This is impossible to achieve if s_1 and s_2 are both integers. In **K-LLVM**, we develop an abstract pointer provenance model based on the PNVI model, and it can be instantiated to support a pointer model whose memory addresses are represented by integers.

The second problem with the de facto model is that although it does a fantastic job dealing with heap pointers, there are other kinds of pointers in an imperative language like LLVM, such as stack pointers, static pointers, and function pointers, which the model does not propose clear solutions to. These pointers are stored in different places in the memory, and they have different restrictions and properties. To model all of them, we need to have a model that distinguishes these different pointers. In the **K-LLVM** provenance model, we develop the concept of different memory domains, each of them having a set of abstract addresses. We assume that there are no overlap of these abstract addresses in these different domains. This concept models the fact that these different pointers are stored in different parts of the memory. In addition, we have defined different properties for these pointers.

Third, the de factor model assumes that the memory has infinite space. In reality, the assumption causes some problems. One of the problems proposed by LEE *et al.* [118] is that some `malloc` calls happen when the memory space is nearly full causes the security problem of exposing memory addresses. They proposed an algorithmic solution to the security problem. The solution is based on allocating two chunks of memory addresses when a `malloc` call happens: one for actual usage, the other for a reservation. In the end, there are large chunks of memory addresses being reserved, so that users cannot guess which memory addresses are left for allocations. In **K-LLVM**, we generalize the solution as a set of unknown abstract addresses for reservations during the memory allocation transition process, and achieve the same guarantees as LEE *et al.* without using a specific algorithm. Then, in the instantiation of the **K-LLVM** abstract provenance model, which is the integer pointer model used in the **K-LLVM** abstract machine, we can utilize the solution proposed by LEE *et al.* by showing that it satisfies the guarantees we made in the abstract provenance model.

In the formalization of the the **K-LLVM** abstract pointer provenance model, we have four abstract memory address domains. Their record structures are shown in Fig. 5.4.

$$\mathcal{MType} \triangleq \texttt{fun\_ptr} \mid \texttt{static\_ptr} \mid \texttt{stack\_ptr} \mid \texttt{heap\_ptr}$$
$$\mathcal{Rec} \triangleq \{\texttt{mtype} = \mathcal{MType}; \texttt{addrs} = \mathcal{Addr}\ \mathcal{Set}; \texttt{max} = \mathcal{Nat}; \texttt{used} = \mathcal{Addr}\ \mathcal{Set}; \texttt{res} = \mathcal{Addr}\ \mathcal{Set}; \texttt{range} = (\mathcal{Addr} \to \mathcal{Nat})\}$$

Figure 5.4: The Record Structures of **K-LLVM** Pointer Provenance Model

A pointer domain record contains five fields: a type for the domain, a set of abstract addresses, a natural number representing the maximum number of pointers the domain can generate, the set of addresses that have been used, a set of reserved addresses for protecting the field being exposed, and a map from the used addresses to the maximum number of values (bytes) in the region that they represent. Each domain is either a function pointer, a static pointer, a stack pointer, or a heap pointer domain. A transition state in our provenance model is $\mathcal{Tr} \triangleq (\mathcal{Rec} \times \mathcal{Rec} \times \mathcal{Rec} \times \mathcal{Rec} \times \mathcal{Prog} \times \mathcal{H} \times \mathcal{Reg} \times \texttt{'a})$, where the first $\mathcal{Rec}$ type is a record for function pointers, the second one is a record for static pointers, the third one is a record for stack pointers, the fourth $\mathcal{Rec}$ is a record for heap pointers, $\mathcal{Prog}$ is a program, $\mathcal{H}$ is a partial map from the memory address (type: $\mathcal{Addr} \times \mathbb{N}$) to values that are typically bytes, $\mathcal{Reg}$ is the registers in the transition system mapping from variables to values including pointer values, and the polymorphic variable `'a` represents the additional transition entities in other usage, which will be explained in Sec. 5.3.2. A transition system is $\mathcal{Tr} \longrightarrow \mathcal{Tr}$. For simplicity, we assume the programs are LLVM IR programs. We also assume that we have a list of functions: $\texttt{fst}, ..., \texttt{seven}, \texttt{eight}$ to get the specific numbered field in a transition state. We can see clearly that we have the predicates for any transition state in Fig. 5.5.

$$\forall T.\ \texttt{mtype}(\texttt{fst}(T)) = \texttt{fun\_ptr} \qquad \forall T.\ \texttt{mtype}(\texttt{snd}(T)) = \texttt{static\_ptr}$$

$$\forall T.\ \texttt{mtype}(\texttt{trd}(T)) = \texttt{stack\_ptr} \quad \forall T.\ \texttt{mtype}(\texttt{four}(T)) = \texttt{heap\_ptr}$$

$$\texttt{(a)} \quad \forall f\ T.\ (f = \texttt{fst} \vee f = \texttt{snd} \vee f = \texttt{trd} \vee f = \texttt{four}) \Rightarrow \texttt{used}((f(T))) \subseteq \texttt{addrs}((f(T))) \wedge \texttt{res}((f(T))) \subseteq \texttt{addrs}((f(T)))$$

$$\texttt{(b)} \quad \forall T\ T'\ f.(f = \texttt{fst} \vee f = \texttt{snd} \vee f = \texttt{trd} \vee f = \texttt{four}) \wedge (T \longrightarrow T') \Rightarrow |\texttt{used}((f(T')))| \leq \texttt{max}((f(T')))$$

$$\texttt{(c)} \quad \forall T\ T'\ f.(f = \texttt{fst} \vee f = \texttt{snd} \vee f = \texttt{trd} \vee f = \texttt{four}) \wedge (T \longrightarrow T') \Rightarrow |\texttt{res}((f(T')))| + (\texttt{max} - |\texttt{used}((f(T')))|) \geq \mathcal{N}$$

Figure 5.5: Selected Predicates for Transition States

Statement `(b)` in Fig. 5.5 restricts the memory address allocation number to be less than a threshold. In order not to expose pointer address in the final domain of pointers, we assume that any transition step requires there to be a number $\mathcal{N}$ of memory locations in the system at anytime, which is enough so that the pointer address is not exposed due to a lack of unallocated memory locations as described by predicate `(c)`.

As we mentioned above, any address used in the memory has the type: $\mathcal{Addr} \times \mathbb{N}$. It is an abstract address plus a natural number offset. These are the addresses in the memory, but they are not the pointers created by the system. Any pointer created by an `malloc` function or a stack allocation call is a polymorphic type object (`'point`) in the abstract provenance system. We provide two getter functions to access a pointer's abstract address field and the offset field as: `base` and `offset`, while the function `maxoff` accesses a pointer's maximum offset. For any pointer v, the range $[\texttt{base}(v), \texttt{base}(v) + \texttt{maxoff}(v)]$ contains all of the valid memory addresses for a given pointer. In addition, we provide a getter function (`mtype`) to access a pointer's `mtype` and a function (`type`) to access its type, such as integer or array type. For all of these pointers, there is a partial order $\sqsubseteq$. We have the assumptions for the partial order in different memory domains in Fig. 5.6.

$$\begin{array}{l}
\forall a\ n\ m.\ n < m \Rightarrow a + n \sqsubseteq a + m \\
\forall T\ a\ b.\ \mathtt{base}(a) \in \mathtt{addrs}(\mathtt{fst}(T)) \wedge \mathtt{base}(b) \in \mathtt{addrs}(\mathtt{snd}(T)) \Rightarrow a \sqsubseteq b \\
\forall T\ a\ b.\ \mathtt{base}(a) \in \mathtt{addrs}(\mathtt{snd}(T)) \wedge \mathtt{base}(b) \in \mathtt{addrs}(\mathtt{trd}(T)) \Rightarrow a \sqsubseteq b \\
\forall T\ a\ b.\ \mathtt{base}(a) \in \mathtt{addrs}(\mathtt{trd}(T)) \wedge \mathtt{base}(b) \in \mathtt{addrs}(\mathtt{four}(T)) \Rightarrow a \sqsubseteq b
\end{array}$$

Figure 5.6: Selected Predicates for Memory Domains

The first statement in Fig. 5.6 says that $\sqsubseteq$ must follow the natural number order if the bases are the same. The addition operation $(+)$ is an overloaded notation. $a + n$ means an arbitrary pointer whose abstract address is a and `offset` value is n. The other predicates in the figure provide some ways that programmers compare pointers in different domains. A common example is to compare the address of a stack pointer and a heap pointer to determine which one is created by a `malloc` function and which one is the stack pointer. With the transition system described above, we have the properties defined for pointers in Fig. 5.7.

$$\begin{array}{ll}
\texttt{(1)} & \forall T\ T'\ H\ f.(T \longrightarrow T') \Rightarrow \mathtt{fst}(T') \cap \mathtt{snd}(T') = \emptyset \wedge \mathtt{fst}(T') \cap \mathtt{trd}(T') = \emptyset \wedge \mathtt{fst}(T') \cap \mathtt{four}(T') = \emptyset \\
& \quad \wedge \mathtt{fst}(T') \cap \mathtt{four}(T') = \emptyset \wedge \mathtt{snd}(T') \cap \mathtt{four}(T') = \emptyset \wedge \mathtt{trd}(T') \cap \mathtt{four}(T') = \emptyset \\
\texttt{(2)} & \forall T\ T'\ H\ f.(T \longrightarrow T') \wedge \mathtt{dom}(\mathtt{six}(T)) \neq \mathtt{dom}(\mathtt{six}(T')) \wedge H = \mathtt{new}(\mathtt{six}(T'), \mathtt{six}(T)) \\
& \quad \wedge (f = \mathtt{fst} \vee f = \mathtt{snd} \vee f = \mathtt{trd} \vee f = \mathtt{four}) \Rightarrow (\forall (a, n) \in \mathtt{dom}(H).\ a \notin \mathtt{used}(f(T))) \\
\texttt{(3)} & \forall T\ T'.(T \longrightarrow T') \wedge \mathtt{dom}(\mathtt{six}(T)) \neq \mathtt{dom}(\mathtt{six}(T')) \wedge H = \mathtt{new}(\mathtt{six}(T'), \mathtt{six}(T)) \\
& \quad \Rightarrow (\forall (a, n) \in \mathtt{dom}(H).\ (\exists f.\ (f = \mathtt{fst} \vee f = \mathtt{snd} \vee f = \mathtt{trd} \vee f = \mathtt{four}) \wedge a \in \mathtt{used}(f(T'))))
\end{array}$$

Figure 5.7: Selected Predicates for Pointers

The group of predicates in Fig. 5.7 define the relations among different types of memory addresses. The `dom` function is to get the domain of a map. The `new` function compares two memory pieces, and computes the difference between the first and second map as $T' - T$. The map difference $(-)$ is based on elements having the same memory locations. This function returns a new map containing all the map entries that are newly generated in T'. Predicate `(1)` states that no any two types of memory addresses overlap each other. Predicates `(2)` and `(3)` say that any newly generated pointers have new abstract memory addresses. We give another group of predicates in Fig. 5.8 defining what happens when specific operations are executed.

The `inst` function gets the specific instruction that is executed during a transition $(T \longrightarrow T')$. The `is_def` function checks if a instruction is a dereference instruction. The `ptr`

$$\begin{aligned}&\forall T\ T'\ H.(T \longrightarrow T') \wedge \texttt{is\_def}(\texttt{inst}(\texttt{five}(T')))\\&\quad\Rightarrow \texttt{addr}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) \sqsubseteq \texttt{addr}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) + \texttt{offset}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T'))))\\&\qquad\sqsubseteq \texttt{addr}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T'))))\\&\qquad\qquad+\texttt{offset}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) + \texttt{sizeof}(\texttt{type}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))))\\&\qquad\quad\sqsubseteq \texttt{addr}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) + \texttt{maxoff}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T'))))\end{aligned}\tag{4}$$

$$\begin{aligned}&\forall T\ T'\ H.(T \longrightarrow T') \wedge \texttt{is\_def}(\texttt{inst}(\texttt{five}(T'))) \wedge \texttt{mtype}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) = \texttt{fun\_ptr}\\&\quad\Rightarrow \texttt{addr}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) \in \texttt{used}(\texttt{fst}(T')) \wedge \texttt{offset}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) = \texttt{0}\end{aligned}\tag{5}$$

$$\begin{aligned}&\forall T\ T'\ H.(T \longrightarrow T') \wedge \texttt{is\_def}(\texttt{inst}(\texttt{five}(T'))) \wedge \texttt{mtype}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) = \texttt{static\_ptr}\\&\quad\Rightarrow \texttt{addr}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) \in \texttt{used}(\texttt{snd}(T'))\end{aligned}\tag{6}$$

$$\begin{aligned}&\forall T\ T'\ H.(T \longrightarrow T') \wedge \texttt{is\_def}(\texttt{inst}(\texttt{five}(T'))) \wedge \texttt{mtype}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) = \texttt{stack\_ptr}\\&\quad\Rightarrow \texttt{addr}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) \in \texttt{used}(\texttt{trd}(T'))\end{aligned}\tag{7}$$

$$\begin{aligned}&\forall T\ T'\ H.(T \longrightarrow T') \wedge \texttt{is\_def}(\texttt{inst}(\texttt{five}(T'))) \wedge \texttt{mtype}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) = \texttt{heap\_ptr}\\&\quad\Rightarrow \texttt{addr}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) \in \texttt{used}(\texttt{four}(T'))\end{aligned}\tag{8}$$

Figure 5.8: Selected Predicates for Operation Executions

function accesses the pointer of the dereference instruction. Function `sizeof` gets a natural number representing the number of bytes for a type. Statement `(4)` says that at the time of dereferencing a pointer, the actual location pointed to by the pointer must be within the address range that the pointer allows reference to. Statements `(5)` to `(8)` say that if we dereference a pointer, then the pointer's `mtype` must match with the `mtype` domain, specifically the address region of the domain, the pointer is supposed to be in. For example, users are free to cast a function pointer to a heap pointer. However, the users are not allowed to dereference the pointer, because its base address is in the function pointer domain. For function pointers, the restrictions are stronger. We additionally require the function pointer's offset value to be zero. This means that we don't allow users to use a function pointer to get to the middle of a program text. For a specific `malloc` function, we have the predicates in Fig. 5.9.

$$\begin{aligned}&\forall T\ T'\ H.(T \longrightarrow T') \wedge \texttt{is\_malloc}(\texttt{inst}(\texttt{five}(T'))) \Rightarrow \texttt{offset}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) = \texttt{0}\\&\quad\wedge(\forall(a,n) \in \texttt{dom}(\texttt{six}(T')).a \neq \texttt{addr}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))) \Rightarrow\\&\qquad\neg\texttt{overlap}((a,(\texttt{range}(\texttt{four}(T')))),(\texttt{addr}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))),\texttt{maxoff}(\texttt{ptr}(\texttt{inst}(\texttt{five}(T')))))))\end{aligned}\tag{9}$$

Figure 5.9: Selected Predicates for Malloc

The `is_malloc` function checks if an instruction is a `malloc` instruction. The `overlap` function checks if two ranges of addresses overlap. (a, n) and (b, m) overlapping refers to

$a \sqsubseteq b \sqsubseteq a+n \sqsubseteq b+m$ or $b \sqsubseteq a \sqsubseteq b+m \sqsubseteq a+n$. The predicate in Fig. 5.9 states what happens when a `malloc` function is executed. It creates a new memory chunk in the memory and a pointer whose `offset` value is `0`. The newly generated memory chunk does not overlap with any of the existing memory chunks whose abstract addresses are in the `used` set in the heap domain. Stack pointer allocations are done in the same manner as `malloc` allocations except that all the stack pointers are generated when a function call happens. When that happens, the transition system grabs all of the stack variable allocations in the function body and generates all the stack pointer allocations at once.

Finally, there are also optional properties defined for some usage of the **K-LLVM** abstract pointer provenance model in some sub-domains. For example, we define a `line` number function to access the line numbers of a program text. For any two instructions (in_1 and in_2) in a function body, if they both contain stack pointers, we then say that $\texttt{line}(in_1) \leq \texttt{line}(in_2) \Rightarrow \texttt{ptr}(in_1) \sqsubseteq \texttt{ptr}(in_2)$. In addition, if two functions f_1 and f_2 are in line order, such that $\texttt{line}(f_1) \leq \texttt{line}(f_2)$, then we say that the function pointers of these two functions have the following relation: $\texttt{fun\_ptr}(f_1) \sqsubseteq \texttt{fun\_ptr}(f_2)$.

In this section, we have briefly introduced the **K-LLVM** abstract pointer provenance model. The actual pointer provenance model **K-LLVM** used for executing programs is based on integer pointers and is proved to be an instance of the **K-LLVM** abstract pointer provenance model. More details are in Sec. 5.3.3.

5.3.2 An Example Sequential Consistency **K-LLVM** Abstract Machine

As we mentioned in the beginning of the chapter, the semantics of the execution of the LLVM IR programs in **K-LLVM** described is via an abstract machine. There are three reasons for this. First, it is a concise way to define all features and aspects of the LLVM IR semantics. LLVM IR is a programming language that connects different computer resources through many different instructions. The best way to model these different features is to design a computer-like mathematical entity which simulates them. Second, the abstract machine is designed to emulate real world computer components. Often, mathematical abstract machines are complicated and confusing. The **K-LLVM** abstract machine execution is easy for users to follow since they can relate it to real world computer components. Third, our abstract machine is modular; as a consequence, it is also extendable. In previous language semantics, designers either only define straight-line single-threaded instruction behaviors or only define a subset of all instructions with complete concurrent behaviors. Once concurrent behaviors are introduced, a single instruction's semantics can affect the whole semantic universe forcing designers to update all existing instruction semantics to handle any side-effects.

The design of the **K-LLVM** abstract machine allows us to focus on designing one feature at a time in isolation. Additionally, because of the modular design, the abstract machine can be easily updated to support progressive concurrent features. For example, we update the byte-wise sequential consistency model in this chapter to a model containing the full LLVM IR concurrency features in Chapter 7, without modifying a *single* instruction semantic rule, and only changing transition rules for describing how to maintain the execution order in the `continuation` and `toCommit` component of each thread.

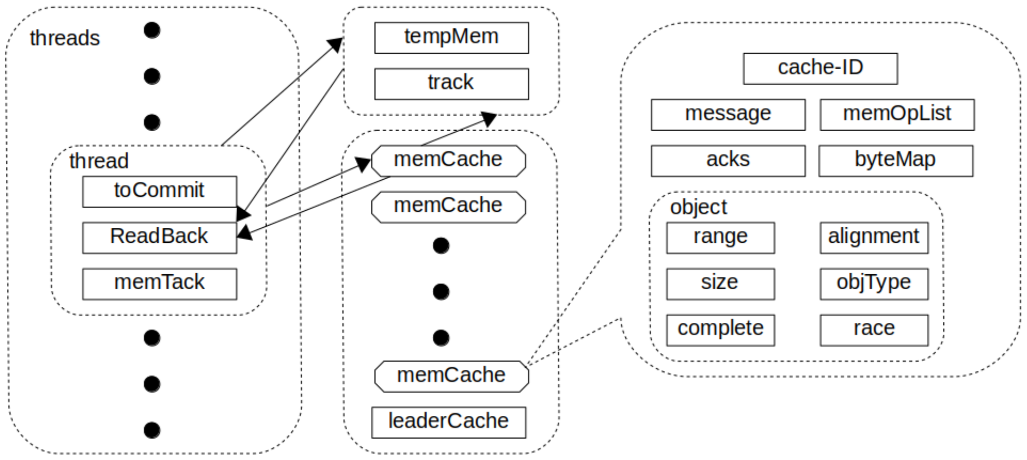


Figure 5.10: Component Relations in the **K-LLVM** Memory Layout Structure

Figure 5.10 describes the overall structure of the **K-LLVM** abstract machine and the interactions of different components. The arrows show the direction of messages passing between the main components. A rounded dashed component means a program state entity that might contain other component structures, while a square component means a program state entity whose content is an integer, list, set, map, etc. The **K-LLVM** abstract machine is independent of the platform in which we implements the machine. At the top level, the abstract machine can be thought of as a set of threads communicating with a set of memory caches, and a global control unit provides global information for threads. As a simplifying assumption to achieve byte-wise sequential consistency, we assume the memory cache set is a singleton set, so we will refer to this cache as the memory cache in the rest of the chapter. The `globalControl` component represents the global control unit containing several sub-components storing information about threads, such as thread identifier calculation, thread final states and mutex lock information. We will see an example of using this information in Section 5.4. There are several components in each thread as shown in the left side of Figure 5.10. In Section 5.2, we said that a LLVM IR program is compiled to a list of BAST control flow graphs (CFGs) for execution. The `continuation` component represents a dynamically executing CFG; it contains a sequence of dynamic basic blocks of instructions to be executed. A thread executes one instruction at a time, i.e., the first instruction in the first block. Thread execution is modeled by consuming instructions as they are executed and possibly inserting a new basic block after the current block during loop execution.

For each thread, the `control` component includes `registers`, a `stack`, `flags`, and `currInst` components. The `registers` component is a map from local variables to values. We introduce how we represent values in the next section. The `stack` component records function call stack frames for context switching in LLVM IR based on `call` and `return` instructions. Each stack also contains fields for local memory allocations in a function directed by the `alloca` instruction. In **K-LLVM**, each function stack has a maximum allowed allocation space, and stack overflow leads to an error state. LLVM IR allows users to declare a set flags for a function, which suggests to the LLVM compiler that more aggressive compilation optimizations are possible. The `flags` component contains the set of function header flags describing the function that is currently executing. For example, the `readonly` flag tells the LLVM compiler that the function will never produce memory write operations, and this need to be reflected in the execution semantics; see Section 5.3.4 for a complete semantics. The `currInst` component contains a dynamic block number and instruction number pair representing the unique identifier for the currently executing instruction. Dynamic block numbers are basically timestamps and uniquely identify each execution of a basic block; when a new basic block of instructions is put into the `continuation` component, a new such number is associated with the block. Instruction numbers can be assigned statically, e.g., using textual order in the LLVM IR file. For example, the numbers on the left side of `Program-A` (Fig. 5.1) are a possible instruction numbering.

The `currInst` pair allows us to modularly add new concurrent behaviors to the **K-LLVM** abstract machine. Even though our model assumes byte-wise sequential consistency in this chapter, the machine has potential for additional concurrent behaviors. When dispatching a memory instruction, a thread need not wait for the instruction commit before proceeding. For example, a thread will not wait for a `load` instruction to write values to `registers`. Instead, it moves on and marks the specific register as unavailable. If the next instruction needs the register value, the thread component blocks. Otherwise, the thread continues to execute instructions. The `currInst` pair identifies a specific instruction and corresponding register during write back. The example `Program-D` (Fig. 5.1) shows how this feature can affect program execution in practice. Without this mechanism, the `load` instruction in `Thread-1` always happens before the `store` in `Thread-2`. With this mechanism, an observer can observe the value `1` or even a race on $@x$. This example is the motivation of having the abstract machine in **K-LLVM** even though its memory model assumes byte-wise sequential consistency in this chapter. **K-LLVM** is mainly used to verify LLVM compiler steps, and verifying programs containing library functions is a key verification component. The *pthread_create* function in `Program-D` is a library function and its functionality should contain fences to prevent the behavior of executing `Program-D` described above. The abstract machine mechanism in **K-LLVM** allows to prove that a particular implementation of *pthread_create* does not have harmful behaviors like the one above; whereas otherwise we do not have a mechanism to verify such library function usage in a program.

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

The memory cache has a fixed structure in **K-LLVM**, which is listed on the right side of Figure 5.10. The `memOpList` component stores the memory operations from different threads, in order to allow the interleaving of memory operations from different threads. The `byteMap` component is a function that maps a memory location to a byte of data. A memory write operation in **K-LLVM** stores an array of bytes in the `byteMap` component. While `byteMap` represents the entire memory cache, a memory chunk refers to a continuous memory region in `byteMap` and is allocated by a global memory initialization or local memory allocation. An `object` component stores metadata for a specific memory chunk. Each `object` contains a `range` component indicating the range of the chunk in the whole memory domain (as keys of `byteMap`), an `alignment` component with alignment information, a `size` component with the size of the chunk in bytes, and an `objType` component indicating if the memory chunk is static or not. The `complete` and `race` components are used to record the status of the operations accessing the memory chunk. According to the LLVM IR documentation, non-atomic memory operations should access a memory range one byte at a time. When a non-atomic memory operation is accessing a memory chunk at the same time as another memory write operation, a race occurs, and the result is `undef`. The `complete` and `race` components are used to record this status and give the result. The implementations of the `byteMap` and `object` components are used to represent the low-level memory layout structures in LLVM IR. We believe that storing metadata on a per-chunk basis is the best way to implement the LLVM IR memory layout model to maximize the concurrent memory access behaviors allowed by LLVM IR.

We have briefly described the different components of the **K-LLVM** abstract machine above. The details of the implementations of each component can be found in the link: `https://github.com/liyili2/llvm-semantics-1`. In the following sections, we will introduce some detailed implementation aspects related to memory accesses. The full LLVM IR concurrency model can also be found in the link : `https://github.com/liyili2/llvm-semantics-1`.

5.3.3 **K-LLVM** Data Layout

In this and the next sections, we introduce a portion of the **K-LLVM** abstract machine in depth, especially, the components and rules related to executing memory related instructions. The manner in which data layout and memory layout are implemented in **K-LLVM** facilitates the precise semantics of many language features of LLVM IR while maintaining a concise abstract machine for the execution semantics. In this section, we introduce the implementation of register and memory location values in **K-LLVM** and example rules using these values. The structure of the memory location values is a part of the pointer provenance model that is instantiated from the **K-LLVM** abstract pointer provenance model in Sec. 5.3.1. The need for two different kinds of values arises as from the fact that memory only sees values as a sequence of bytes, while instructions see registers as holding compound data. We describe these two kinds in Figure 5.11, and we also show some example rules using these data. In Figure 5.11, rules connected by a $\Rightarrow$ operator mean that the transition from the left hand side to the right hand side happens in the beginning of a `continuation` component. There is an implicit rule saying that every transition happening in the beginning of a `continuation` also happens globally. More complex transition rules are introduced in Fig. 5.13. The `add` and `icmp eq` are instructions in LLVM IR appearing here in the concrete syntax.

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

```
Bit ::= 1 | 0 | undef    Range ::= range(Nat , Nat)    'a State ::= Error | 'a Option
                 Byte ::= byte(Bit List , Range Option , Range State)
                  Loc ::= loc(Bit List , Type , Range Option , Range State)
              Int ::= intLoc(Bit List , Type , Range Option , Range State)
              Float ::= floatLoc(Bit List , Type , Range Option , Range State)
(a)  add T intLoc(X,A1,B1,C1), intLoc(Y,A1,B2,C2)
          ⇒  intLoc(bitAdd(T,X,Y), A1, judge(B1,B2), judge(C1,C2))
(b)  icmp eq T loc(X,A1,B1,C1), loc(Y,A1,B2,C2)  ⇒  intLoc([X = Y], i1,none,none )
```

Figure 5.11: Memory Data Structure

For register values, we only introduce integer, float and pointer values here. The description of other register values can be found in the **K-LLVM** semantics in the link: `https://github.com/liyili2/llvm-semantics-1`. Any of the integer (*Int*), float (*Float*) or pointer (*Loc*) data contains a *Bit* list, a *Type* field representing the type of the datum, a range attribute and a flag attribute. The *Bit* list represents the binary format of the value for the datum being either an integer, float (in the IEEE 754 format) or memory address. The size of the list is equal to the size of the integer/float/pointer type defined for the data (the pointer size is parameterized in **K-LLVM**). We assume that all integer, float and pointer arithmetic is based on the computation of binary representations, even though we might show decimal representations in some examples in this chapter for presentation purpose. The range and flag attributes have meaning that is closely related to the ones in a *Byte* datum, as we will explain below.

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

The two rules `(a)` and `(b)` in Fig. 5.11 give an example describing how an arithmetic instruction is executed in **K-LLVM** based on the data structure described above. In evaluating an LLVM IR `add` instruction (rule `(a)`), the value computation happens between the *Bit* lists of two data (`bitAdd` adding two binary numbers together). The function `judge` merges two range or flag attributes from possibly two different data that possibly come from two pointer sources. The `judge` details are in the actual **K-LLVM** semantics in the link: `https://github.com/liyili2/llvm-semantics-1`. Here, we give some interesting examples. If a pointer is cast to a integer constant (with the range attribute `[`L, R`]`) and added to another integer constant (with the range attribute `none`), the `judge` produces a memory range from the pointer in the range attribute of the result datum. If the two range attributes of two `intLocs` have two different memory ranges (like `[`L, R`]` $\neq$ `[`S, T`]`), the judged result is `none`. If two flag attributes of two data have two different memory ranges, in this case, `judge` produces an error state in the flag attribute of the result datum; and if the result datum is further turned into a pointer, and is used to read memory data, the program results in unspecified behavior. Rule `(b)` gives an example of a comparison instruction that discards the pointer information and produces a pure 1-bit integer constant. Depending on the instruction, including the nature of its arguments, pointer information might or might not be transmitted along with the result of the calculation.

5.3.4 Sample Instruction Semantics

In this section, we introduce semantic rules supporting memory related instructions in **K-LLVM**. The set of memory related instructions we select to describe here contains LLVM IR casting, address calculation (`getelementptr`) and memory instructions, as well as memory related flags on the function headers. **K-LLVM** is the first formal semantics to cover all behavioral aspects (under byte-wise sequential consistency) of this set, including the side-effects due to interactions between different instructions inside or outside of the set. Under the byte-wise sequential consistency assumption, the behaviors of different orderings in an atomic memory operation collapses to the behavior of the sequentially consistent (`seqcst`) ordering. It is worth noting that there are cases when an instruction can go to an unspecified behavior or other error states. We will not list all of those rules here, although we have defined them in **K-LLVM**. Interested readers may get more details from the **K-LLVM** semantics [135].

Casting Instructions. Here we describe the semantics of `inttoptr` and `bitcast` as the highlights of the **K-LLVM** semantics of casting instructions in Figure 5.12. The other casting instructions are implemented in a similar manner. Before **K-LLVM**, no complete interpretation for the LLVM IR casting operations existed, especially one supporting casting between integers or floats and pointers. These casting instructions are hard to define because the resulting values can vary depending on the program context for the values of the instructions.

```
(a)  inttoptr(intLoc(X,T1,B,C),T2) ⇒ loc(trunc(X,sizeof(T1) - sizeof(T2)),T2,B,C)
       if sizeof(T1) ≥ sizeof(T2)
(b)  inttoptr(intLoc(X,T1,B,C),T2) ⇒ loc(addZero(sizeof(T2) - sizeof(T1))@X,T2,B,C)
       if sizeof(T1) < sizeof(T2)
(c)  bitcast(Label(X,T1,B,C),T2) ⇒ rebuild(X,T2,B,C)
       if ¬isPointerType(T1) ∧ Label ∈ {intLoc, floatLoc}
(d)  bitcast(loc(X,T1,B,C),T2) ⇒ loc(X,T2,B,C)
```

Figure 5.12: Casting Rules

In Figure 5.12, rules `(a)` and `(b)` describe the semantics of `inttoptr`. The main idea is to replace the type attribute of the source `intLoc` with the target type. If the target type size is smaller than (or equal to) the source one, the semantics truncates (using the `trunc` function) the bits (represented by X as a list) by the difference of the sizes of the two types starting from the most significant bit. Otherwise, we create a list of `0` bits, whose size is the difference between the two type sizes, by using the `addZero` function. We place the bits in front of the source bit list (variable X). For example, in `Program-A` (Fig. 5.1), we assume that the code is running in a 32-bit machine and variable *%r5* has the value represented by `intLoc(`*X,* `i64,` *B, C*`)` in line 15. The code tries to convert the *%r5* value to a pointer. The final result pointer can be represented by `loc(`*X',* `i64*,` *B, C*`)` by taking the right-most 32-bits from X and changing the constructor from `intLoc` to `loc`.

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

The Semantics of `getelementptr`. A `getelementptr` instruction is a memory address calculation whose main idea is to calculate a memory address value based on a sequence of indices. Section 5.1.1 touches on one of the special cases of `getelementptr` semantics. The main idea of `getelementptr` is similar to the one in Zhao's work [62]. It uses a sequence of indices of different types to walk incrementally into a data structure layout to calculate a pointer to the sub-component found at the end of the path the indices describe. Here, we focus on one particularly important feature of the instruction, the keyword `inbounds`, which is a flag applied on the computation results of a `getelementptr` instruction. For this flag, LLVM IR requires all the intermediate and final computation results on the address of the input pointer are within a valid range of the allocated object pointed to by the address. In **K-LLVM**, we implement this with the address computation function `calGEP`. The function calculates an new address value by adding multiplication results of the index and type size to the input address, one adding at a time. In each step, before the calculation, the function first checks if the input address is within the range indicated by the range attribute of the input pointer. After we compute the final address result, we also check if the memory chunk pointed to by the input pointer still exists. For example, line 4 of `Program-A` (Fig. 5.1) is a `getelementptr` instruction, and it is executed successfully in **K-LLVM**. However, if a memory-`free` for the input pointer *%r2* is added before the `getelementptr`, the `inbounds` flag makes the instruction result in a poison value, because the memory chunk pointed to by *%r2* does not exist anymore. As another example of an `inbounds` flag, executing line 5 of `Program-A` highlights how a poison value can be produced from a `getelementptr`. The index `i64 -1` makes an intermediate computation result out-of-bound, so variable *%u1* gets a poison value. Another example is to execute line 21 of `Program-A`. The execution of this `getelementptr` fails the `inbounds` check because its input pointer has range attribute `none`, so variable *%r10* results in a poison value. There is also an `inrange` flag in a `getelementptr` instruction. This flag has subsequent effects on memory instructions after the `getelementptr`. The flag information is carried as the flag attribute in the pointer derived from the `getelementptr` so that the succeeding memory instructions can use it. We will introduce its semantics in the next section.

The `store` Semantics. We only introduce the **K-LLVM** `store` memory instructions here; the other memory instructions are implemented in a similar manner. **K-LLVM** fully implements the semantics of `stores` under the byte-wise sequential consistency assumption. Specifically, **K-LLVM** distinguishes the non-atomic and atomic `store` instructions by breaking the execution of an memory instruction into three different stages, as shown in Figure 5.13. As we mentioned, we do not list negative rules, such as configurations going to an error state when a `store` is performing a `write` operation in the memory cache, when the memory chunk has already been freed by another thread. The rules in Figure 5.13 are simplified versions of the actual **K-LLVM** rules. The information and handling about address spaces and memory alignments is not mentioned here. In fact, the construct `write` has several fields than one shown in the figure. On the other hand, these rules are non-trivial, and they have enough functionality to show manner in which the **K-LLVM** abstract machine distinguishes between the behaviors of atomic and non-atomic `store` instructions.

In Figure 5.13, the *Exp* type represents an instruction that involves in the computation in a `continuation` component (Ψ in Fig. 5.13). We uses `store` and `atomicStore` constructs in Figure 5.13 that are different from the LLVM IR concrete syntax. They are BAST format transformed from an LLVM IR `stores` instruction in their simplified form here. Each of them has three fields. The first represents the type of the value; the second is the value to store in the memory cache and the third is the memory pointer. The `write` construct represents the memory operation that a thread uses to communicate with the memory cache and the memory cache uses to perform memory events. When a `store` is executed in the `continuation` component (Ψ), a list of `writes` are generated in the `toCommit` component

$$\mathit{Key} ::= (\mathit{Nat},\mathit{Nat},\mathit{Nat})$$
$$\mathit{Exp} ::= \texttt{store}(\mathit{Type},\mathit{Exp},\mathit{Loc}) \mid \texttt{atomicStore}(\mathit{Type},\mathit{Exp},\mathit{Loc}) \mid \texttt{write}(\mathit{Key},\mathit{Nat},\mathit{Nat},\mathit{Byte\ List})$$

(a)
$$\frac{[X, X + \texttt{sizeof}(T)] \subseteq [L, R] \wedge [X, X + \texttt{sizeof}(T)] \subseteq [L1, R1] \wedge \texttt{readonly} \notin \Theta}{\begin{array}{l}\Big(\mathit{TID}, (\mathit{BID},\mathit{IID}), (\texttt{store}(T, V, \texttt{loc}(X, B, \texttt{range}(L,R), \texttt{range}(L1,R1)))::\Psi), \Delta, \Theta\Big) \\ \Rightarrow \Big(\mathit{TID}, (\mathit{BID},\mathit{IID}), \Psi, \\ \qquad \Delta @ \texttt{genWrites}(\texttt{toBytes}(V, \texttt{sizeof}(T)), (\mathit{TID}, \mathit{BID}, \mathit{IID}), X, \texttt{sizeof}(T)), \Theta\Big)\end{array}}$$

(b)
$$\frac{[X, X + \texttt{sizeof}(T)] \subseteq [L, R] \wedge [X, X + \texttt{sizeof}(T)] \subseteq [L1, R1] \wedge \texttt{readonly} \notin \Theta}{\begin{array}{l}\Big(\mathit{TID}, (\mathit{BID},\mathit{IID}), (\texttt{atomicStore}(T, V, \texttt{loc}(X, B, \texttt{range}(L,R), \texttt{range}(L1,R1)))::\Psi), \Delta, \Theta\Big) \\ \Rightarrow \Big(\mathit{TID}, (\mathit{BID},\mathit{IID}), \Psi, \Delta @ [\texttt{write}((\mathit{TID}, \mathit{BID}, \mathit{IID}), X, 1, \texttt{toBytes}(V, \texttt{sizeof}(T)))], \Theta\Big)\end{array}}$$

(c)
$$\begin{array}{l}\Big(\big\{(\mathit{TID}, \mathit{CurrInst}, \Psi, E::\Delta, \Theta) \cup \mathit{Threads}\big\}, (\kappa, \mathit{Rest})\Big) \\ \Rightarrow \Big(\big\{(\mathit{TID}, \mathit{CurrInst}, \Psi, \Delta, \Theta) \cup \mathit{Threads}\big\}, (\kappa @ [E], \mathit{Rest})\Big)\end{array}$$

(d)
$$\frac{\mathit{Addr} \in [L,R] \wedge \neg\texttt{isRace}(\mathit{Key},\alpha)}{\begin{array}{l}\Big(\texttt{write}(\mathit{Key},\mathit{Addr},1,V)::\kappa, \Gamma, \big\{([L,R], \alpha, \mathit{Rest})\big\} \cup \Omega\Big) \\ \Rightarrow \Big(\kappa, \texttt{updateMap}(\Gamma,\ \mathit{Addr},\ V), \big\{([L,R], \alpha, \mathit{Rest})\big\} \cup \Omega\Big)\end{array}}$$

(e)
$$\frac{\mathit{Size} > 1 \wedge \beta(\mathit{Key}) = \texttt{none} \wedge \mathit{Addr} \in [L,R] \wedge \neg\texttt{isRace}(\mathit{Key},\alpha)}{\begin{array}{l}\Big(\texttt{write}(\mathit{Key},\mathit{Addr},\mathit{Size},V)::\kappa, \Gamma, \big\{([L,R], \alpha, \beta, \mathit{Rest})\big\} \cup \Omega\Big) \\ \Rightarrow \Big(\kappa, \texttt{updateMap}(\Gamma,\ \mathit{Addr},\ V), \big\{([L,R], \alpha \cup \{\mathit{Key}\}, \beta[\mathit{Key} \mapsto 1]\ ,\ \mathit{Rest})\big\} \cup \Omega\Big)\end{array}}$$

(f)
$$\frac{\mathit{Size} > 1 \wedge \beta(\mathit{Key}) = \mathit{Size} - 1 \wedge \mathit{Addr} \in [L,R] \wedge \neg\texttt{isRace}(\mathit{Key},\alpha)}{\begin{array}{l}\Big(\texttt{write}(\mathit{Key},\mathit{Addr},\mathit{Size},V)::\kappa, \Gamma, \big\{([L,R], \alpha, \beta, \mathit{Rest})\big\} \cup \Omega\Big) \\ \Rightarrow \Big(\kappa, \texttt{updateMap}(\Gamma,\ \mathit{Addr},\ V), \big\{([L,R], \alpha \backslash \{\mathit{Key}\}, \beta[\mathit{Key} \mapsto \texttt{none}]\ ,\ \mathit{Rest})\big\} \cup \Omega\Big)\end{array}}$$

Figure 5.13: Memory Store Rules

(Δ in Fig. 5.13). They have the same group ID represented as a *Key* type that is a triple of the thread ID, dynamic block number, and instruction number of the `store`. A `write` also has other fields: a natural number representing the memory address value, another natural number representing the total size of `writes` from the same *Key*, and a list of bytes to write to the memory. The total size is the same for different `write` operations with the same *Key*. It is both the size of the list of `writes` generated by a non-atomic `store` and the size of bytes of the value to write to the memory cache. An `atomicStore` generates a singleton `write`.

Before we describe the rules in Figure 5.13, some conventions are worth noting. Without special greek letter illustrations on different components, a name of a component with its first character capitalized is the variable representing the component in all rules (e.g. *CurrInst* for the `currInst` component, and *Threads* for the `threads` component). The variable *Rest* appearing in some rules in Figure 5.13 (and Fig. 5.14) represents the rest of components in a `thread` or `object` component, which do not involve in the computation of the rules. As we have said in Section 5.3.2, the **K-LLVM** abstract machine is for a set of threads communicating with a single memory cache. The `globalControl` component is omitted in the computation here, since we do not need it. Based on these assumptions, we define a transition state to be a pair of a set of threads and a memory cache: $(\mathit{Threads}, Memory)$. A single thread contains five components related to memory instructions: `thread-ID`, `currInst`, `continuation` (Ψ), `toCommit` (Δ) and `flag`. For simplicity, we assume that a thread only contains these five components in this section; Also, we assume that the memory cache only contains three components: the `memOpList` (κ), `byteMap` (Γ) and `objects` (Ω) components. The `objects` component (Ω) contains a set of `object`. Only three sub-components (`range`, `race` (α) and `complete` (β)) in the `object` are related in defining the semantics of `store`. The math inclusive range [*A,B*] represents a set of natural number sequencing from the number A to the number B inclusively. Finally, there are implicit rules omitted in Figure 5.13, suggesting that transitions happening in a thread or the memory cache also happens globally.

Rules `(a)` and `(b)` in Figure 5.13 describe how an atomic `store` and non-atomic `store` generate a list of `write` operations that are pushed to the `toCommit` component (Δ) whose job is to convey memory operations to the memory cache. The basic idea is to create a list of `writes` at the end of `toCommit` (Δ) when we have a `store` in the head of `continuation` (Ψ). The two rules describe the cases when an `inrange` flag is present in the flag attribute of the input pointer. In such cases, to execute a `store` not only requires for the address value to be within the range indicated in the pointer but also for it to be in the range carried by the `inrange` flag; otherwise, the whole system results in an unspecified behavior state. In **K-LLVM**, there are rules similar to rules `(a)` and `(b)` dealing with pointers without `inrange` flags derived by removing the checks for the `inrange` edges from rules `(a)` and `(b)`. Since we will use these rules in an example, we call them rules `(ax)` and `(bx)` to distinguish them from rules `(a)` and `(b)`. The function `toBytes` splits a value into an list of bytes (Fig. 5.11). The list size is defined by its natural number argument. Its functionality is similar to the `rebuild` function to turn a value as a list of elements. The only difference is that `toBytes` creates an list of bytes instead of values in the case of `rebuild`. The function `genWrites` takes a list of bytes, a *Key* datum, a memory address, and the size of the byte list, then generates a list of `writes` by distributing a byte at a time from the byte list, and associates each byte with a memory address and other attributes. The address value is selected in sequence from the address range between the address and the address plus the size. Rule `(b)` is for dealing with atomic `stores`. The key difference is that it only generates a singleton `write` containing the full value to be stored instead of a list. Rule `(c)` allows the head element in the `toCommit` component (Δ) of a thread to move to the tail position of the `memOpList` (κ) in the memory cache.

Rules `(d)`, `(e)` and `(f)` deal with different situations of correctly committing a `write` to the `byteMap` (Γ). The `complete` component (β) in `(e)` and `(f)` is a map from a *Key* to a natural number indicating how many `writes` have been committed to `byteMap` (Γ) since the first `write` with the *Key*. The *Key* marks a single instruction and `complete` allows tracking the process of the writes entailed by the instruction. To detect races, the `race` component (α) contains *Keys* indicating every *Key* occupying the memory chunk (`object`) represented by the `range` of the `object` component. The variable *Size* represents the number of `writes` from the same *Key*, i.e. the same `store` instruction. All rules `(d)`, `(e)` and `(f)` need to satisfy two side conditions. The first one is the condition $Addr \in$ `[`L`,`R`]` to locate a specific `object` in the `objects` component (Ω) by comparing *Addr* with the range of the `object` (L and R). In **K-LLVM**, an `object` is created by a memory allocation; thus, the ranges of `objects` (Ω) are always disjoint. Any address (e.g. *Addr*) within a range (e.g. `[`L`,`R`]`) can be a key to locate the range, which in turn locates an `object`. The second condition is to check if a *Key* is in race with other *Keys* in `race` (α) through the function `isRace`. The function `isRace` checks if the `race` component (α) for the `object` pointed to by the memory address value (*Addr*) has been occupied by another *Key*. If *Size* is `1` (rule `(d)`), the `write` represents an atomic memory `store`, and writes a list of bytes (V) to `byteMap` (Γ) using the function `updateMap`. The function updates a range of bytes to corresponding range of addresses in `byteMap` (Γ). Rule `(e)` is executed if two other conditions are satisfied: the *Size* is not `1` and no `write` for this *Key* has yet completed. In such case, rule `(e)` writes a list of bytes to `byteMap` (Γ) and updates the information in the `race` (α), and initializes the *Key* in the `complete` component (β). Rule `(f)` represents the finish of the execution of a non-atomic `store` in the memory cache. In such cases, we remove the appearance of the entities represented by variable *Key* in the `race` (α) and `complete` (β) components. We also need to update `byteMap` (Γ) with the final `write` term. Besides rule `(e)` and `(f)`, another rule not listed here deals with the case when $\beta(Key)$ does exist and is less than *Size* `- 1`. In this rule, we continue to write a byte to the `byteMap` component (Γ) without touching the `race` component (α) and incrementing the `complete` component (β) for *Key*.

As an example of applying the `store` rules, we focus on the `store` instruction in line 13 of `Program-A` (Fig. 5.1). Group `(s)` in Figure 5.14 represents the computations for executing the first few steps of the `store` instruction. In these diagrams, we show the computations as

$$
\texttt{(s)}\ \left(\begin{array}{l}
\Big\{\big(\varphi,\texttt{(1,13)},(\texttt{store([2 x i32],[11,11], loc(100,[2 x i32]*,}\\
\qquad\qquad \texttt{range(96, 108), none))::}\ \Psi),\texttt{[]},\emptyset\big)\Big\}\cup\Xi,\ \big(\texttt{[]},\Sigma,\big\{\big(\texttt{[96,108]},\emptyset,\emptyset,\Upsilon\big)\big\}\cup\Omega\big)
\end{array}\right)
$$

$$
\Rightarrow\left(\begin{array}{l}
\Big\{\big(\varphi,\texttt{(1,13)},\Psi,(\texttt{write((}\varphi\texttt{,1,13),100,8,byte(}B\texttt{,none,none))::}\Delta\texttt{)},\emptyset\big)\Big\}\cup\Xi,\\
\qquad\qquad \big(\texttt{[]},\Sigma,\big\{\big(\texttt{[96,108]},\emptyset,\emptyset,\Upsilon\big)\big\}\cup\Omega\big)
\end{array}\right)
$$

$$
\Rightarrow\left(\begin{array}{l}
\Big\{\big(\varphi,\texttt{(1,13)},\Psi,\Delta,\emptyset\big)\Big\}\cup\Xi,\\
\quad \big(\texttt{[write((}\varphi\texttt{,1,13),100,8,byte(}B\texttt{,none,none)))]},\Sigma,\big\{\big(\texttt{[96,108]},\emptyset,\emptyset,\Upsilon\big)\big\}\cup\Omega\big)
\end{array}\right)
$$

$$
\Rightarrow\left(\begin{array}{l}
\Big\{\big(\varphi,\texttt{(1,13)},\Psi,\Delta,\emptyset\big)\Big\}\cup\Xi,\\
\quad \big(\texttt{[]},\Sigma[\texttt{100}\mapsto \texttt{byte(}B\texttt{,none,none)}],\big\{\big(\texttt{[96,108]},\{(\varphi\texttt{,1,13)}\},\{(\varphi\texttt{,1,13)}\mapsto\texttt{1}\},\Upsilon\big)\big\}\cup\Omega\big)
\end{array}\right)
$$

$$\Rightarrow\cdots$$

$$
\texttt{(t)}\ \Big(\Big\{\Big(\big(\varphi,\texttt{(1,22)},(\texttt{store(i32, 42, loc(100, i32*, none, none))::}\ \Psi),\texttt{[]},\emptyset\big)\Big)\Big\}\cup\Xi,\mathit{Rest}\Big)
$$

$$
\quad\Rightarrow\big(\ \texttt{error unspecifiedBehavior}\ \big)
$$

Figure 5.14: Memory Store Example Configuration Transitions

transitions from one state to another. Each transition state is a tuple of a thread set and the memory cache. In threads, Ξ represents all threads that do not involve in the computation. The thread we care about has a thread ID φ. We assume that the (1,13) in the first state after the label (s) represents the currInst pair. In the continuation component, we have the store instruction of line 13 (Fig. 5.1) on the top of the computation, and Ψ represents the rest of the computations in continuation. For simplicity, we assume that the toCommit and flags components are empty for thread φ, so they have the values [] and $\emptyset$, respectively. In the memory cache, for simplicity, we assume that memOpList is empty, byteMap is represented by the variable Σ. The memory cache contains some objects. The Ω in Fig. 5.14 represents objects not related to this store computation, and there is an object with range value [96,108] that matters in this computation (Let's assume that [96,108] is the memory range created previously). We also assume that the current race and complete components are both empty (an empty set and empty map). Υ represents the rest of the components in the object that does not involve in the computation. The to-store data for the store operation is an array of type [2 x i32] and value [11,11]. Here, we show these data in decimal formats. In the real **K-LLVM** abstract machine, they should be in the binary format. In this example, we assume that the memory pointer address is a natural number 100, and the range of the memory chunk pointed to by the pointer is in the range [96,108].

By applying rule (ax) above, we get a new transition state after the first "$\Rightarrow$" (Fig. 5.1).

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

Another example is group `(t)` in Figure 5.14. It represents the computations of the `store` instruction at line 22 of `Program-A` (Fig. 5.1). In the initial state, the pointer has the range attribute `none`, so the state is transitioned to an error state with the `unspecifiedBehavior` indicator.

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

Observation 5.2. Assume that a trace of memory operations is generated by observing the order of memory operations committed to the `byteMap` in the memory cache. For a valid LLVM IR program, no matter which rule the **K-LLVM** abstract machine chooses to apply in a transition state if such rule correctly pattern matches the state, the memory trace generated by executing the program is byte-wise sequentially consistent.

The `readonly` Function Flag. LLVM IR allows users to set flags on the function headers that suggest that the function has certain features over memory instructions. The `readonly` flag is a representative. It means that the execution of the function with the flag should not use any memory write operations, e.g. a `store` instruction. If executing a function does use a write operation, it is an unspecified behavior. In Figure 5.10, there is a `flags` component in the `control` component of a thread. During the static semantics step in Section 5.2, all functions from a LLVM IR program are compiled to BAST format and stored in a database, including function header flag information. During executing in the **K-LLVM** abstract machine, when a function is called, **K-LLVM** context switches the `control` component for the function, including the flag information called from the database and stored in the `flags` component. When **K-LLVM** is executing a `store` operation, according to the `store` rules (Fig. 5.13), **K-LLVM** checks if the `flags` contains a `readonly` flag. If not, the `store` operation can proceed; otherwise, the whole transition state is transitioned to an error state of `unspecifiedBehavior`.

We have given here a general idea of how **K-LLVM** implements different semantic aspects of LLVM IR here. Next, we do a little evaluation on **K-LLVM**.

5.4 **K-LLVM** EVALUATION AND APPLICATIONS

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

Testing Process of K-LLVM. The validation of language semantics is usually accomplished through the use of external test suites [83, 84, 120], which was also part of our strategy. **K-LLVM** successfully ran a set of 1,385 unit testing programs, which covers all individual single thread LLVM IR features. A C test suite including the GCC-torture test suite is compiled to LLVM IR programs to test **K-LLVM**, which passed all test cases. The test suite contained 2,156 LLVM IR programs and covers all LLVM IR single thread C features. For single-threaded programs, we relied on ***krun*** to fix the execution order and show the final result. The test cases and Clang bugs have been documented in the link: `https://github.com/liyili2/llvm-semantics-1`, and the bugs have been reported to the LLVM community.

The methodology for developing **K-LLVM** was based on a strategy named Test Driven Development (TDD), whose basic idea is to develop tests before implementing the actual features. LLVM IR has an official test suite, but it is hard to break it down into individual pieces. In developing **K-LLVM**, the test principle is to test individual features while coordinating new features with old defined ones. When we defined a new feature in **K-LLVM**, we followed four steps. First, we read the details about the feature in the LLVM IR documentation, and thought about how to define the static and dynamic semantics of it. Next, we wrote out unit test cases to test the feature in the current LLVM IR implementation (Clang/Clang++). We made sure that we covered enough corner cases by designing a good set of new unit tests. We then defined the feature and tested it with the new unit tests, making sure it could pass them all. Third, we added the new feature to all of the defined unit tests to see if it caused any new problems. Finally, we tested the whole semantics with the regression 2,156 test suite (the GCC-torture test suite) and made sure that it passed more test cases than before and did not introduce new problems. When we developed **K-LLVM**, we started by defining the static semantics for each individual feature in LLVM IR, and made sure that all static features were validated for every variable, expression, instruction, function and module. After that, we defined the **K-LLVM** memory model and validated the correctness of the model. Following the definition of the model, we incrementally defined the semantics of the instructions, working from those that interacted least with other instructions and the memory such as the arithmetic and conversion instructions, through to the branching instructions and finally those that affected the memory. Lastly, we defined different memory operations. The distinction between the atomic and non-atomic memory operations is particularly complicated due to the fact that we define the non-atomic memory system to be based on reading/writing one byte at a time.

While searching for undesirable behaviors in Clang was not an objective of this project, we found some in the process of defining the **K-LLVM** semantics. Mainly, we ran test programs, and compared their outputs with those listed in the LLVM documentation. Undesirable behaviors happened in very diverse circumstances. A large number of them related to the fact that Clang does not place enough checks to validate what the LLVM IR documentation suggests. In other cases, Clang has missing features. For example, one cannot cast an `fp128` constant to a `ppc_fp128` constant, which should be allowed. In some cases, the description of the LLVM documentation is not clear. For example, in describing the `fptrunc` and `fpext` instructions, LLVM IR uses the idea of large floating point types, and allows a comparison of two of them. However, it does not give a precise description of how to make this comparison. In fact, we found that the two types `fp128` and `ppc_fp128` are not comparable, so there is no way in LLVM IR to cast from one to the other, contrary to the documentation.

Finally, among the 2,156 LLVM IR programs, we used 26 multi-threaded programs to test the **K-LLVM** thread library with ***ksearch***. **K-LLVM** produced a set of behaviors that are all expected according with respect to our thread and byte-wise sequentially consistent memory model. There are other multi-threaded programs used for testing the full memory concurrency behaviors.

Morpheus on K-LLVM. We built the Morpheus tool [1] on top of **K-LLVM** to support correct transformations of compiler optimizations of LLVM IR programs. The Morpheus core language is a domain-specific one for formal specifications of program transformations. It describes program transformations as rewrites on control flow graphs with temporal logic (CTL) side conditions. Morpheus allows users to specify comprehensible program optimizations including those in data flow analysis and data dependence graph analysis. Its executable semantics allows these specifications to act as prototypes for the optimizations themselves, so that candidate optimizations can be tested and refined before including them in a compiler. We built Morpheus on top of **K-LLVM** in $\mathbb{K}$, so that users are able to specify program optimizations in LLVM IR, and test the optimizations by using $\mathbb{K}$ tools for LLVM IR programs. Through the **IsaK** and **TransK** tools (Chapter 4), we translate **K-LLVM** into a transition system in Isabelle, and merge it with the Morpheus tool in Isabelle. With this system, we are able to prove the correctness of the optimizations in Isabelle under the assumption that programs are executed in the **K-LLVM** abstract machine and a choice of memory model. As an instance, we are able to define redundant store elimination properties on LLVM IR programs in Isabelle under sequential consistency. With the **K-LLVM** abstract machine, we have a framework for proving the correctness of the optimization for all programs in LLVM IR in Isabelle. The finalization of the proof will be an interesting future work of **K-LLVM**. The detailed semantics of Morpheus, and its union with a transition semantics for a fragment of LLVM for use in proving properties of program transformations is in the later chapter.

Detecting Undefined Behaviors. When an undefined behavior happens, **K-LLVM** outputs an error state. This is particularly useful for programmers in revealing unexpected behaviors to programmers, especially memory access errors. For example, in `Program-A` (Fig. 5.1), the execution of the program results in a transition state with an `error` component containing an `unspecifiedBehavior` construct (Fig. 5.14). This is because pointer *%r9* comes from a non-valid source. By using ***krun***, we can see an error message for the `Program-A` execution in Fig. 5.15.

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

Figure 5.15: The Error Message for `Program-A`

For some undefined behaviors in LLVM IR, the ***ksearch*** space exploration method cannot list all outputs. `Program-E` (Fig. 5.1) is such an example. The program is to create a memory field, get a memory pointer, then turn the pointer to an integer and print it. The output is a non-deterministic value with infinite many possible values. When using ***krun*** (the single-thread execution engine in $\mathbb{K}$) to execute the program, it prints out a random integer value depending on the runtime memory address allocation in **K-LLVM**. A better way to analyze the program is to use the $\mathbb{K}$ symbolic execution engine. One can use ***ksearch*** with the `-symbolic` flag to execute this program, and the final result is a variable representing a integer value. One can also use the $\mathbb{K}$ symbolic equivalence checker to check if the executions of two similar programs printing out variables representing the same range of integers. The equivalence checker relies on the Z3 SMT solver to calculate if two variables representing the same range of values.

State Space Exploration. A trivial utility of **K-LLVM** is state space exploration through the ***ksearch*** tool. Users can use ***ksearch*** (actual command: `krun -search`) to see all possible final results and traces of multi-threaded programs based on the automatically generated interpreter for **K-LLVM** in $\mathbb{K}$. This can be useful for detecting out-of-thin-air behaviors. For example, by assuming sequential consistency, if we execute `program-B` (Fig. 5.1) with the initial values of zero in both memory fields for pointers $@x$ and $@y$, the final results of $\%a$ and $\%b$ can never both be zero. We can also detect undefined values of a race. According to the documentation of LLVM IR, when a non-atomic `store` happens, and another memory operation from another thread is trying to access the same field, a race happens, and the two memory operations both get `undef`. By using ***ksearch*** to execute `program-C` (Fig. 5.1), we can see `undef` for variables $\%a$ and $\%b$ in some final results.

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

Chapter 6: HYBRID AXIOMATIC TIMED RELAXED CONCURRENCY MODEL (HATRMM)

A concurrency model describes the semantics of concurrency in a programming language, particularly one with imperative features, delimiting the allowed execution sequences, particularly regarding the values passed among different portions of program executions via variables whose values are stored in the memory. Weak concurrency models for imperative programming languages (C, C++, Java) have been studied broadly [2, 3, 4, 100, 101, 102, 103, 104, 105, 107, 108, 109, 110, 111, 112, 113].

One of the major tasks of using these axiomatic models is to provide correct mappings from primitives in concurrent imperative programming languages to instructions in mainstream architectures, in particular, the architectures x86-TSO [147], POWER [148], and ARMv8 [149], since the semantics of imperative programming languages are implemented in a modern computer as mappings instead of their mathematical meanings. The correctness of such mappings means that for every high-level imperative program P, the set of concurrent behaviors allowed by the target architecture for the translated machine-level program P' is contained in the set of behaviors allowed by the high-level model for P. Building such a proof is an active research topic in the compiler correctness field. However, mapping correctness is difficult to establish. Previous works either provided incorrect claims and proofs or very limited results for a handful of compilation processes. For example, Batty *et al.* [150] built the mapping from C/C++ to POWER and ARMv7, whose correctness claims were found to be incorrect [3]. On the other hand, imperative languages usually have similar primitive behaviors. All these facts led to the creation of intermediate languages (IR), like LLVM [142], to act as bridges connecting the imperative-language and machine-level instructions. Once one set of correct mappings is established on an IR language, then it can be used to provide mappings for many different imperative languages.

6.1 HATRMM BEYOND OTHER PREVIOUS MODELS

Previous researchers [4, 114] tried to provide concurrency models for these IRs and verified the mapping correctness from imperative languages to the IRs, and from the IRs to machine-level instructions. However, they either mimicked the model of a high-level imperative language, like the work of Chakraborty and Vafeiadis [114], who used the C++ concurrency model for LLVM; or they intentionally weakened their models to connect high-level languages and low-level machine instructions, like the IMM model [4]. These IR languages usually have the features of both the high-level and machine-level languages. Modeling them in terms of a

C-like concurrency model is not enough. Moreover, there need to be strategic methodologies for designing a specific IR model so that it can not only act as a bridge connecting the high-level and machine-level languages, but also facilitate correctness proofs of the mappings from the high-level languages to machine-level instructions. Simply weakening a high-level model is inadequate. For example, IMM was purposely weakened to allow the establishment of broader compilation correctness proofs. There are two main problems with the model. First, the semantics of some operations, while violating the user's intuition, make it tough to design a large programming language based on the model. For example, the control dependency in IMM is too weak to prevent out-of-thin-air behaviors (OOTA, Sec. 6.3.2). Second, the properties guaranteed by IMM are blurry. IMM certainly does not satisfy DRF-SC. Even if one can prove a compilation correctness mapping from the model to the machine-level, it does not mean that a high-level language that satisfies DRF-SC can also establish such a mapping.

In this paper, we first investigate three aspects in which an IR concurrency model is different from the high-level and machine-level models. Then we propose three methodologies targeting these aspects. Finally, we propose an IR concurrency model, the hybrid axiomatic timed relaxed concurrency model (HATRMM) in the style of the axiomatic candidate-execution model [100], which combines the three methodologies into one model.

Duality and Hybridization. Generally speaking, the first way that an IR model is different from the previous models is *duality*. A compiler usually contains a long sequence of translation phases. Compiling a program in an IR platform typically involves translating it into a simpler format, with both formats using the same IR syntax. However, the concurrent semantics, i.e., the meanings of an operation, might be different before and after translation. In this case, a correct translation guarantees that the new meaning does not cause harmful behaviors.

For example, according to the `NO-THIN-AIR` constraint in RC11 [3], the reads in the (anti) and (anti-if) example programs in Fig. 6.1 both observe the value `1`, but the behavior is allowed in the machine-level instructions (ARMv8 [149]). This means that one of the compiler steps (e.g. Clang [142]) must slightly weaken the concurrency guarantee in the IR program (obviously, the translation itself must guarantee that no weakened behaviors can happen).

Previously, the IMM model [4] tried to bridge the proof of such translations by weakening its `NO-THIN-AIR` constraint and control-flow dependency (`ctrl`). We have seen the discussion of the model's problem in Fig. 6.1. To overcome the duality, we bring in the methodology, named *hybridization*, which means that we mix the concurrency semantics of high-level and machine-level languages together. In HATRMM, we built a subset of operations that

```
/* initially, x = 0 and y = 0 */

(anti)
a:=rlx y//1   ||  b:=rlx x//1
x:=rlx 1      ||  y:=rlx 1

(anti-if)
a:=rlx y//1   ||  b:=rlx x//1
if(a=1)       ||  if(b=1)
  x:=rlx 1    ||    y:=rlx 1

(anti-eq)
a:=rlx y//1   ||  b:=rlx x//1
if(a=a)       ||  if(b=b)
  x:=rlx 1    ||    y:=rlx 1

(anti-b)
a:=rlx y//1   ||  b:=rlx x//1
if(a=a)       ||  y:=rlx 1
  x:=rlx 1    ||
```

Figure 6.1: Example Programs For Concurrency

satisfy the DRF-SC property. This subset represents the concurrency behaviors of high-level languages. We also have another subset of operations that has weaker concurrency behaviors than the DRF-SC subset to act as the translation target. A translation for changing the concurrent semantics is just a translation of the operations in one subset to ones that fit the other. The hybridization concept seems simple, but the actual formulation is complicated. It involves not only the combination of two subsets of operations, but also the consideration of non-atomics and atomics. See Sec. 6.3.3.

One of the side-effects of hybridization is the recognition of out-of-thin-air behaviors (OOTA). Identifying out-of-thin-air behaviors (OOTA) is a hard problem because they are context-sensitive. For example, RC11 [3] disallows all two-reads-both-reading-1 executions in all of the programs in Fig. 6.1 to achieve DRF-SC, while IMM purposely weakens its model by allowing both reads to load 1 in both (anti-eq) and (anti-if), which is definitely a "bad" OOTA behavior. In a sense, it is impossible to have one unique way for distinguishing OOTA behaviors while satisfying different needs in different circumstances. In HATRMM, distinguishing "good" OOTA behaviors from "bad" ones depends on the layer. Clearly, the HATRMM subset that guarantees DRF-SC distinguishes OOTA behaviors differently than the HATRMM subset that does not satisfy DRF-SC. On top of this, we have a third layer; that is, we rely on a simulation proof through the per-location simulation [151] to build a form of equivalence among all executions exhibiting "the greatest allowance" for OOTA behaviors. For example, the first layer disallows two reads to both load 1 in all the examples in Fig. 6.1. The second layer of HATRMM allows the two reads to both load 1 in (anti) and (anti-b), but disallow the behavior in (anti-if) and (anti-eq). By building equivalence through the per-location simulation, we prove the equivalence of all executions of (anti) and (anti-eq) under the assumption of the compilation of a simple code motion optimization (SCM), but disprove equivalence between (anti) and (anti-if) under that assumption.

Versatility and Fencizing. IR language structures are usually messy and complicated, involving a lot of special flags and operations for controlling devices on different levels, such as OS and machine-level devices. Many of them affect the languages' concurrent behaviors. Defining all of them separately creates a burden that swells the size of the semantics. For example, K-LLVM [140] has more than 50,000 semantic rules. Here, we provide a uniform mechanism for supporting the definitions and extensions of these different operations. Like the unified field theory in physics, we want to use a unified structure to explain each of these burdens and define them in a nice and clean way. To do so, we "fencize" every operations that affects the concurrent behaviors by viewing it as a fence structure. For example, control dependency is defined as a control fence in HATRMM. We also define call fences for capturing the behaviors of function calls. We utilize the call fences to show the correctness of the function inlining optimization in restricted contexts (not all cases; see Sec. 6.3.2).

Linearization. One methodology that we use to couple all these different pieces into a model is to give each execution in the system a sequence, named a time point sequence. Introducing the time points significantly reduces the complexity of defining and combining different constructs, such as non-atomics and atomics, as well as simplifies the correctness proof between HATRMM and an operational model. According to Nienhuis *et al.* [152], no extant operational model can be proved to be equivalent to an axiomatic model (based on the C++ model [2]), if the operational model admits out-of-order/speculative executions. Out-of-order/speculative executions are executions of single-threaded instructions that might not follow the sequenced-before relation (`sb`). The previous models [2, 3, 4] all assumed that each single-threaded execution in a multi-threaded program followed the `sb` order strictly. When an equivalence proof between one of these models and an operational one was conducted, `sb` order was the natural choice for the inductive factor. If the operational model had admitted out-of-order executions, then for every such execution, they would have needed to unfold the program text and attempt to make the new `sb` order match the order of the execution; but it is impossible to know all of the unfoldings of a program before induction on the `sb` order. The previous equivalence proofs were done by having an intermediate in-order operational model that followed the `sb` order exactly, then showing that the axiomatic one was equivalent to the in-order one, and finally showing that the in-order operational model was threadwise bisimilar to the out-of-order one. For every execution, HATRMM gives a sequence (linearization) that does not assume `sb` order. This linearization represents the set of memory events happening at one time, and the construction of the equivalence proof is based on induction on the linearization (see Sec. 6.3.1).

6.2 PRELIMINARY: MEMORY ACTIONS, EVENTS, EXECUTIONS, RELATIONS, AND EXAMPLE PROGRAMMING LANGUAGE

Here we introduce the key syntactic aspects of HATRMM with examples, following the standard axiomatic approach for defining memory consistency models [100]. Some useful relations based on a relation (R) in this paper are the reflexive ($R^?$), transitive (R^+) and reflexive-transitive closures (R^*). We denote by $R_1; R_2$ the left composition of two relations R_1 and R_2, and assume that ; binds tighter than $\cup$ and $\setminus$. $[A]$ is the identity relation on a set A. $A \subset_{\mathtt{fin}} B$ means that A is a finite subset of B. $A \uplus B$ is a property showing that A and B are disjoint. $\sqcap T$ gets the maximum of T. In Fig. 6.2, every name in $\mathcal{Chancery}$ font is a type defined for a language component; everything in `tt` font is a constructor or terminal in the language; and everything in *Italics* is a variable representing a term. The figure also introduces ranging conventions that are employed throughout the paper.

Domains

Values: $v \in \mathcal{Val}$ Memory Locations: $x, y, z \in Loc \subset_{\mathtt{fin}} \mathcal{Loc} \triangleq \mathbb{N}$

Action IDs: $d \in \mathcal{Aid}$ Thread IDs: $tid \in Tid \subset_{\mathtt{fin}} \mathcal{Tid} \triangleq \mathcal{TName}$

Time Points: $s, t \in T \subseteq \mathcal{Times} \triangleq \mathbb{N}$ Call Fence Labels: $u \in \mathcal{LName}$

Function Names: $g \in \mathcal{FName}$ Mutexes: $k \in Key \subseteq \mathcal{KName}$

Action Counters: $n, m \in \mathcal{CName} \triangleq \mathbb{N}$

Orderings

Read Orderings: $\mathcal{O}_r \ni o_r \triangleq \mathtt{rlx} \mid \mathtt{acq} \mid \mathtt{sc}$

Fence Orderings: $\mathcal{O}_f \ni o_f \triangleq \mathtt{rlx} \mid \mathtt{acq} \mid \mathtt{acqrel} \mid \mathtt{sc}$

Write Orderings: $\mathcal{O}_w \ni o_w \triangleq \mathtt{rlx} \mid \mathtt{rel} \mid \mathtt{sc}$

RMW Orderings: $\mathcal{O}_{rw} \ni o_{rw} \triangleq \mathtt{rlx} \mid \mathtt{acq} \mid \mathtt{rel} \mid \mathtt{acqrel} \mid \mathtt{sc}$

$\sqsubseteq \triangleq \{(\mathtt{na},\mathtt{rlx}),\ (\mathtt{rlx},\mathtt{acq}),\ (\mathtt{rlx},\mathtt{rel}),$
$\quad (\mathtt{acq},\ \mathtt{acqrel}),\ (\mathtt{rel},\mathtt{acqrel}),\ (\mathtt{acqrel},\ \mathtt{sc})\}^*$

Memory Actions, Events, & Executions

$\mathcal{Act} \ni ac \triangleq \mathtt{ARead}\ \mathcal{Bool}\ x\ o_r\ [\mathtt{R}^{o_r}_{(x,v)}] \mid \mathtt{AWrite}\ \mathcal{Bool}\ v\ x\ o_w\ [\mathtt{W}^{o_w}_{(x,v)}]$
$\quad \mid \mathtt{NRead}\ \mathcal{Bool}\ x\ d\ n\ m\ [\mathtt{NR}^{d}_{(x,v)}] \mid \mathtt{NWrite}\ \mathcal{Bool}\ v\ x\ d\ n\ m\ [\mathtt{NW}^{d}_{(x,v)}]$
$\quad \mid \mathtt{CallFence}\ g\ d\ [\mathtt{CAF}^{g}_{d}] \mid \mathtt{RMW}\ \mathcal{Bool}\ v\ x\ o_{rw}\ [\mathtt{RW}^{o_{rw}}_{(x,v)}] \mid \mathtt{Fence}\ o_f\ [\mathtt{F}^{o_f}]$
$\quad \mid \mathtt{ControlFence}\ [\mathtt{CF}] \mid \mathtt{Lock}\ k\ [\mathtt{L}_k] \mid \mathtt{UnLock}\ k\ [\mathtt{UL}_k]$

Memory Events: $\mathcal{Ev} \ni ev \triangleq (tid, d, ac)$ Reads From Relations: $\mathtt{rf} \subseteq T \times T$

Time Point Mapings: $\rho \subseteq T \cup \{\mathtt{0}\} \rightarrow \mathcal{Ev} \cup \{\bot\}$

Sequenced-Before Relations: $\mathtt{sb} \subseteq T \times T$

Sequenced-Before Families: $\mathtt{sbs} \subseteq Tid \rightarrow (T \times T)$

Data Dependence Relations: $\mathtt{dd} \subseteq T \times T$

Data Dependence Families: $\mathtt{dds} \subseteq Tid \rightarrow (T \times T)$

Memory Executions: $\mathcal{Z} \ni \zeta \triangleq (Tid, Loc, Key, T, \rho, \mathtt{rf}, \mathtt{sbs}, \mathtt{dds})$

Element Properties

$\mathcal{RName} \uplus \mathcal{TName} \uplus \mathcal{FName} \uplus \mathcal{KName} \uplus \mathcal{LName} \uplus \mathcal{Times} \uplus \mathcal{Aid} \uplus \mathcal{CName}$

$\biguplus_{tid \in Tid} (\mathtt{sbs}(tid)) \qquad \biguplus_{tid \in Tid} (\mathtt{dds}(tid))$

$\forall T.\ \mathtt{0} \notin T \qquad \forall tid \in Tid.\ \mathtt{dds}(tid) \subseteq \mathtt{sbs}(tid)$

$\forall \rho.\ \rho(\mathtt{0}) = \bot \qquad \forall s\ t \in T.\ (s,t) \in (\bigcup_{tid \in Tid} (\mathtt{dds}(tid)) \cup \mathtt{rf}) \Rightarrow s < t$

Figure 6.2: Memory Execution Elements

We assume that there is an underlining programming language and its operational semantics that formulates programs. A program is a mapping from threads (Tid) to single-threaded program pieces. We assume that the program semantics is a labeled transition system. A sequence of transitions, in terms of pairs of labels and states, can be derived by executing an input program. We name these sequences **program executions** (an example is in the Sec. 6.4). From the programming language perspective, HATRMM is defined on top of **memory executions**, which are sequences of memory events generalized from program executions. Given a labeled transition system executing a program, each memory event represents a memory operation "compiled" from a running thread in the system to communicate with other threads and memory.

HATRMM is independent of a particular programming language and it provides the generalization for communication between the threads and memory in a system, through defining constraints on the set of input relations. We introduce memory events and executions described by several input relations in HATRMM in Fig. 6.2. Its basic elements include a set of values representing the data points in the heap, a finite natural number set of memory locations (Loc), a finite set of thread-IDs (Tid), a set of Action-IDs uniquely identifying a memory action in a single-thread execution, a set of function names ($\mathcal{FName}$), a set of function labels ($\mathcal{LName}$) for identifying an executing function call in an execution, a downward closed natural number set of time points (T) excluding `0` and acting as the sequence in a memory execution, a set of mutex locks (Key), and a natural number set of action counters ($\mathcal{CName}$) for identifying non-atomic memory action in an execution. Since time points are natural numbers, there is a natural number less-than operator ($<$) for every time point set. We assume that some of these sets are disjoint unioned (Fig. 6.2), and `0` is never an element in a T set. The `0` time point represents the initial state of the memory before any memory event is committed.

A memory execution represents a sequence of **memory events** that is either an initial event ($\perp$) at time point `0` or a triple ($\mathcal{Ev}$ in T) of a thread-ID, an action-ID and a memory action. A **memory action** represents an operation in a language suited for communication between the threads and memory (defined in Fig. 6.2). It contains atomic reads (`ARead`, abbr. `R`) and writes (`AWrite` abbr. `W`), non-atomic reads (`NRead` abbr. `NR`) and writes (`NWrite` abbr. `NW`), read-modify-writes (`RMW`), ordered fences (`Fence` abbr. `F`), call fences (`CallFence` abbr. `CAF`), control fences (`ControlFence` abbr. `CF`), and mutex operations (`Lock`/`UnLock`). Sec. 6.3 describes some of these. On the right of each action definition in Fig. 6.2, the bracket contains the abbreviation for the action that is used in the execution diagrams throughout this paper. The action-IDs d appeared in non-atomics (`NW` and `NR`) are group-IDs uniquely distinguishing groups of non-atomics. HATRMM allows mixed-size values for non-atomics, so an non-atomic instruction in a program (not memory execution!)

might "compile" to many non-atomic events in an execution. The group-IDs are keys to identify the events from the same group. The details are in Sec. 6.3.3. Throughout this paper, we refer to atomic/non-atomic reads/writes, and read-modify-writes as the **memory operations** that really perform data communications between the threads and memory, while the other actions are **memory fences**, which fix the way of communications. In a memory operation, the $\mathcal{Bool}$ field represents if the operation admits the volatile memory model (LLVM volatile model [142]). In some memory operations and fences, the ordering fields (o_r, o_w, o_{rw}, and o_f) represent C++ like memory orderings. The order relation $\sqsubseteq$ is applied to pairs of memory orderings, which also includes `na` for non-atomics.

A **memory execution** (Fig. 6.2) is defined as $\zeta = (Tid, Loc, Key, T, \rho, \texttt{rf}, \texttt{sbs}, \texttt{dds})$, where ρ $(T \cup \{\texttt{0}\} \to \mathcal{Ev} \cup \{\perp\})$ is a bijective function from time points to memory events (T is bijective to $\mathcal{Ev}$), and `rf` is a write-read relation set defining the source write event that a read event loads from in the execution. Even though the example executions (diagrams) in the paper do not indicate the following, any read in an execution must appear in `rf`, and if a read does not load a value from a write, it is assumed to load from a "conceptual write value" happening in the `0` time point. `sbs` is a family of sequenced-before relations (`sb` for each thread) in different single-threaded programs, while `dds` is a family of data dependency relations in different single-threaded programs (`dd` for each thread), such as traditional data dependency, address dependency, output dependency, and pointer aliasing dependency, etc. Given a multi-threaded program, for each single-threaded control flow graph (CFG, Let's assume that each single-threaded program piece is represented as a CFG) in the program, there are simple and well-known algorithms to compute the `sb` and `dd` relations for the thread, which we will omit here. There are assumptions on a memory execution (Fig. 6.2). For example, different relations (`sb` or `dd`) for different threads in `sbs` and `dds` are disjoint unioned; every relation (`dd`) in `dds` is a subset of the one (`sb`) in `sbs` for the same thread; every `0` time point maps to the initial event ($\perp$) in ρ; and the union of `rf`, `dds` for each thread, and the natural number inequality ($<$) is irreflexive. There is a syntactic sugar that $\overline{\texttt{dds}} \triangleq \bigcup_{tid \in Tid} (\texttt{dds}(tid))$.

Throughout the paper, example programs like (anti) in Fig. 6.3 are all assumed to have every location initialized as the value `0`. a, b and c ranges over register variables, while x, y and z ranges over memory locations. Thus, a:=$_{\texttt{rlx}}x$ is an atomic read instruction, while x:=$_{\texttt{rlx}}a$ is an atomic write, and :=$_{\texttt{na}}$ refers to non-atomic instructions. The colored values appearing in some reads in the program are the reading values thought to appear in some executions of the program. Executions of programs are portrayed as the middle and right diagrams in Fig. 6.3. In these diagrams, the memory events are represented as

(anti)

$$\begin{array}{l||l} a :=_{\texttt{rlx}} y \,//1 & b :=_{\texttt{rlx}} x \,//1 \\ x :=_{\texttt{rlx}} 1 & y :=_{\texttt{rlx}} 1 \end{array}$$

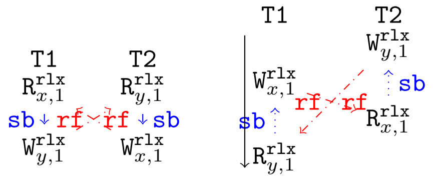


Figure 6.3: The Execution and Execution Diagram of `anti`

abbreviations of the memory actions described in Fig. 6.2. The middle diagram without a downward arrow represents an execution that appeared in previous models [3, 4], while the right diagram with a downward arrow indicating the flow of time (the time point order) is an execution in HATRMM. We have briefly introduced the memory execution syntax and some definitions. Next we will introduce an example language syntax that is used in the chapter.

6.2.1 Example Programming Language Syntax

Domains

Basic Block Numbers: $\pi \in \mathcal{Bn} \triangleq \mathbb{N}$ Dynamic Block Numbers: $\overline{\pi} \in \mathcal{Dn} \triangleq \mathcal{Bn} \times \mathbb{N}$ Instruction Position Numbers: $i \in \mathcal{I} \triangleq \mathbb{N}$

Registers: $a, b \in \mathcal{Reg} \triangleq \mathcal{RName}$ Instantiated Value: $v \in \mathcal{Val} \triangleq (\mathcal{Loc}|\mathbb{Z}|[\mathbb{N} \times \mathcal{Val}])$ Instantiated Action-IDs: $d \in \mathcal{Aid} \triangleq \mathcal{Dn} \times \mathcal{I}$

Instructions

$\mathcal{Type} \ni ty \triangleq \texttt{int} \mid ty\,\texttt{*}\mid [\mathbb{N} \times ty]$ $\mathcal{Exp} \ni e \triangleq v \mid a \mid e \texttt{+} e \mid e \texttt{*} e \mid e \texttt{=} e \mid e \texttt{<} e \mid \ldots$

$\mathcal{S} \ni in \triangleq \texttt{skip}\mid a \texttt{ := } ty\, e \mid a \texttt{ := phi } ((a, \pi) \text{ list}) \mid a \texttt{ := \&}x \mid a \texttt{ := set } ty\, e\, ty\, e \mid a \texttt{ := get } ty\, e\, (\mathbb{N}\ \mathcal{List}) \mid a \texttt{ :=}_{o_r} (\texttt{vol})^{?}\, x \mid \texttt{lock } k$
$\quad \mid x \texttt{ :=}_{o_w} (\texttt{vol})^{?}\, a \mid a \texttt{ :=}_{\texttt{n}} (\texttt{vol})^{?}\, x \mid x \texttt{ :=}_{\texttt{n}} (\texttt{vol})^{?}\, a \mid a \texttt{ :=}_{o_{rw}} (\texttt{vol})^{?}\, \texttt{fadd}(x, a)$
$\quad \mid \texttt{fence}_{o_f} \mid a \texttt{ := } g\, (\texttt{inline})^{?}\, (a \text{ list}) \mid \texttt{unlock } k$

$\mathcal{C} \ni in \texttt{ ::= if } a \texttt{ then } \pi_1 \texttt{ else } \pi_2 \mid \texttt{br } \pi \mid \texttt{return } a \mid \texttt{exit}$ $\mathcal{L} \ni cl \texttt{ ::= seq} \mid \texttt{yes} \mid \texttt{no}$

Programs

$N \subset_{\texttt{fin}} \mathcal{Bn}$ $\pi_0 \in N$ $E \subseteq N \times \mathcal{L} \times N$

Basic Blocks: $\mathcal{BB} \triangleq \mathcal{S}\ \mathcal{List} \times \mathcal{C}$ CFG Label Funs: $\lambda \subseteq N \to \mathcal{BB}$ Control Flow Graphs (CFG): $\mathcal{CFG} \ni G \triangleq (N, \pi_0, \lambda, E)$

Programs: $\mathsf{Prog} \ni \mu \triangleq Tid \to \mathcal{CFG}$ Functions: $\mathcal{SProg} \ni p \triangleq (r\ \mathcal{List}, G)$ Function Database: $\mho \triangleq \mathcal{FName} \to \mathcal{SProg}$

Figure 6.4: Example Language Syntax

HATRMM is independent of a particular programming language. However, as the structure in Chapter. 8, a proof of semantic preservation on top of HATRMM requires an operational semantics for a programming language building on top of HATRMM. In Fig. 6.4, we provide an example CFG-based imperative language. The operational semantics of such language establishes an execution system that can be used for proving semantic preservation of compiler optimizations on the language. In the semantics, the communications between threads and memory is derived by HATRMM. In Sec. 6.4, we discuss the operational semantics of the language and the derivation between its operational semantics and HATRMM.

The language (Fig. 6.4) is LLVM-like, and any program of it is assumed to be in static single assignment format (SSA). We assume that any program is type-checked and well-formed and the execution of the program does not cause well-formed problem. The execution of a program can be analogized as a set of threads, each of which executes a single-threaded CFG in the program, and each thread execution is executing a basic block at a time. Other than the atomic elements in Fig. 6.2, the language (Fig. 6.4) is given a set of basic block numbers ($\mathcal{Bn}$) labeling each basic block in a control flow graph (CFG), a set of dynamic basic block numbers ($\mathcal{Dn}$) labeling each executing basic block and including a basic block number and a counter number, a set of instruction numbers labeling each instruction in a basic block statically, a set of registers. We also instantiate two sets in Fig. 6.2 to more concrete instances. The value set ($\mathcal{Val}$) is instantiated as location values ($\mathcal{Loc}$), integers ($\mathbb{Z}$), and array values ($\texttt{[}\mathbb{N} \times \mathcal{Val}\texttt{]}$). An action-ID ($\mathcal{Aid}$) is instantiated to a triple of a basic block number, a dynamic basic block counter number, and an instruction number. It can uniquely identify an executing instruction in a thread of a program execution, so a pair of a thread-ID and action-ID can uniquely identify an executing instruction in a program execution.

Expressions (e) are constructed from registers (local variables) and integers, and represent values and locations. For simplicity, we assume that there is only one evaluation order in evaluating an expression. There are two kinds of instructions. Normal instructions ($\mathcal{S}$) include assignments, address-of, function calls with a possible `inline` flag, array insertion (`set`) and extraction (`get`) instructions, and memory instructions. Termination instructions ($\mathcal{C}$) only appear at the end of a basic block and include function return, exit, unconditional (`br`) and binary conditional branching. $r \texttt{ := } \ ty\ e$ is the assignment instruction, and an execution first evaluates the expression e and then casts it to the type ty and stores the value in the register r. $r \texttt{:=}_{o_{rmw}} (\texttt{vol})^{?}\ \texttt{fadd}(x, e)$ atomically increments the value in location x by the value of e and loads the old value in register r. The optional `vol` flag forces the given memory instruction to respect the volatile memory access model, which here refers to the LLVM volatile model [142]. For simplicity, read/write instructions are divided into atomic ones ($\texttt{:=}_{o_w}/\texttt{:=}_{o_r}$) and non-atomic ones ($\texttt{:=}_{\texttt{n}}/\texttt{:=}_{\texttt{n}}$).

A **basic block** is a list of $\mathcal{S}$ instructions following by a termination instruction ($\mathcal{C}$). In a basic block, we assign each instruction a **instruction position number** ($i \in I$), where the sequential instructions are assigned their position in the list (starting from 0), and the termination is assigned the length of the list of sequential instructions. Thus, its position number is one greater than that of the last instruction in the list. In a CFG, the node set N contains the basic block numbers of the CFG λ is a labeling of each node, with a basic block comprising the list of sequential instructions terminated by a termination, and $E \subseteq N \times \mathcal{L} \times N$ is a set of edges labeled `seq`, `yes` or `no`, such that, if $\texttt{snd}(\lambda(n)) = \texttt{br}\ \pi$, then there is a unique out-edge of n, labeled `seq`; if $\mathsf{snd}(\lambda(n)) =$ `exit` or `return` e, then n has no out-edges; otherwise, there are exactly two out-edges, one labeled `yes` and one labeled `no`. For every variable in a basic block π that has more than one source from two different incoming blocks of π, there is a `phi` instruction in π to merge the different sources and the `phi` instruction mentions all incoming edges of the block exactly once. A function (p) is defined as a pair of an argument list (registers), and a CFG. A program is defined as a mapping from a set of thread-IDs to CFGs. In this chapter, we assume Boolean values `true` and `false` are just syntactic sugars for the integers `1` and `0`. Notice that we have a `skip` instruction, so `if` e $\{e_1\}$ in all examples in this chapter just means that `if` e `then` $\{e_1\}$ `else skip`.

We have briefly introduced the memory execution syntax and an example language syntax. We will introduce HATRMM and the utility of it with the operational semantics of the language.

6.3 HATRMM AND THREE METHODOLOGIES

Here we introduce several aspects of HATRMM in terms of its implementation and combination of the three methodologies introduced in the beginning of the chapter. Recall that a memory execution in HATRMM has the form: $\zeta = (Tid, Loc, Key, T, \rho, \mathtt{rf}, \mathtt{sbs}, \mathtt{dds})$ (Sec. 6.2). The way to define valid executions is to define constraints that specify which ones are valid among a set of candidate executions having the form ζ. Throughout this section, we use the predicate `is_something` to test if a memory event has an action defined as `something`. For example `is_read` means that an event has a read action (or `RMW`), `is_acq` means that an event is an `acq` atomics, and `is_mem_op` means that the action of an event is a memory operation. The predicate `same_property` means that two input memory events have the same `property`. For example, `same_loc` means that the two input memory events access the same memory location. `dom` is for getting the domain of a function while `ran` is for getting the co-domain. Definitions are provided in Fig. 6.5 for some `is_something` and `same_property` predicates.

6.3.1 Linearization: Benefits of Time Points

Here we introduce the linearization methodology, which also serves as the basis of the other two methodologies. The implementation of linearization in HATRMM is the time point concept. As we have described in Sec. 6.2, the time point set T is a downward closed natural number set in the candidate execution ζ. It is associated with the ρ, a mapping from $T \cup \mathtt{0}$ to memory events. All relations and constraints appearing in previous

$$
\begin{array}{l}
\texttt{get\_loc}(tid, d, \texttt{NW}^{d'}_{(x,n,m)}) \triangleq x \\
\texttt{get\_loc}(tid, d, \texttt{NR}^{d'}_{(x,n,m)}) \triangleq x \\
\texttt{get\_loc}(tid, d, \texttt{W}_{(x,v,\ldots)}) \triangleq x \\
\texttt{get\_loc}(tid, d, \texttt{R}_{(x,v,\ldots)}) \triangleq x \\
\texttt{get\_loc}(tid, d, \texttt{RW}_{(x,v,\ldots)}) \triangleq x \\
\texttt{get\_loc}(tid, d, ac) \triangleq \bot \ \texttt{[owise]}
\end{array}
\quad
\begin{array}{l}
\texttt{get\_i}(tid, d, \texttt{NW}^{d'}_{(x,n,m)}) \triangleq m \\
\texttt{get\_i}(tid, d, \texttt{NR}^{d'}_{(x,n,m)}) \triangleq m \\
\texttt{get\_i}(tid, d, ac) \triangleq \bot \ \texttt{[owise]}
\end{array}
\quad
\begin{array}{l}
\texttt{get\_gid}(tid, d, \texttt{NW}^{d}_{(x,n,m)}) \triangleq d \\
\texttt{get\_gid}(tid, d, \texttt{NR}^{d}_{(x,n,m)}) \triangleq d \\
\texttt{get\_gid}(tid, d, ac) \triangleq \bot \ \texttt{[owise]}
\end{array}
$$

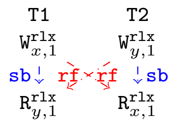
$$
\begin{array}{l}
\texttt{get\_tid}(tid, d, ac) \triangleq tid \qquad \texttt{get\_aid}(tid, d, ac) \triangleq d \quad \texttt{get\_ac}(tid, d, ac) \triangleq ac \\
\texttt{same\_tid}(ev, ev') \triangleq \texttt{get\_tid}(ev) = \texttt{get\_tid}(ev') \\
\texttt{same\_gid}(ev, ev') \triangleq \texttt{get\_gid}(ev) = \texttt{get\_gid}(ev') \wedge \texttt{get\_gid}(ev) \neq \bot \\
\texttt{same\_loc}(ev, ev') \triangleq \texttt{get\_loc}(ev) = \texttt{get\_loc}(ev')
\end{array}
$$

$$
\begin{array}{l}
\texttt{get\_pos}(tid, d, \texttt{NW}^{d'}_{(x,n,m)}) \triangleq n \\
\texttt{get\_pos}(tid, d, \texttt{NR}^{d'}_{(x,n,m)}) \triangleq n \\
\texttt{get\_pos}(tid, d, ac) \triangleq \bot \ \texttt{[owise]}
\end{array}
\quad
\begin{array}{l}
\texttt{get\_num}(tid, d, \texttt{NW}^{d'}_{(x,n,m)}) \triangleq n \\
\texttt{get\_num}(tid, d, \texttt{NR}^{d'}_{(x,n,m)}) \triangleq n \\
\texttt{get\_num}(tid, d, ac) \triangleq \bot \ \texttt{[owise]}
\end{array}
$$

Figure 6.5: Some Useful Predicate Definitions

models are defined over events, while HATRMM's relations and constraints are defined over time points. Essentially, the time points are given a linearization for the execution with no assumption on the relationship between the linearization and the `sb` of the execution. Obviously, such linearization needs to respect each single-threaded `dd` (recall the assumption that $< \cup\ \texttt{rf} \cup \overline{\texttt{dds}}$ is irreflexive). The most important reason to include time points is to solve that there does not exist a direct equivalent proof between a relaxed axiomatic concurrency model and an operational model that admits out-of-order/speculative executions [152].

(flow) $\begin{array}{l} x \texttt{:=}_{\texttt{rlx}}\ 1 \\ a \texttt{:=}_{\texttt{rlx}} y \texttt{//1} \end{array} \Big\| \begin{array}{l} y \texttt{:=}_{\texttt{rlx}}\ 1 \\ b \texttt{:=}_{\texttt{rlx}} x \texttt{//1} \end{array}$


T1
T2
$\texttt{W}^{\texttt{rlx}}_{x,1}$
$\texttt{W}^{\texttt{rlx}}_{y,1}$
sb
rf
rf
sb
$\texttt{R}^{\texttt{rlx}}_{y,1}$
$\texttt{R}^{\texttt{rlx}}_{x,1}$

(flow-a) $\begin{array}{l} x \texttt{:=}_{\texttt{rlx}}\ 1 \\ a \texttt{:=}_{\texttt{rlx}} y \texttt{//0} \end{array} \Big\| \begin{array}{l} b \texttt{:=}_{\texttt{rlx}} x \texttt{//0} \\ y \texttt{:=}_{\texttt{rlx}}\ 1 \end{array}$

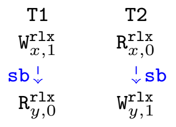

T1
T2
$\texttt{W}^{\texttt{rlx}}_{x,1}$
$\texttt{R}^{\texttt{rlx}}_{x,0}$
sb
sb
$\texttt{R}^{\texttt{rlx}}_{y,0}$
$\texttt{W}^{\texttt{rlx}}_{y,1}$

Figure 6.6: Two Example Memory Executions for Two Similar Programs

When the researchers [152] proved the equivalence property, they inducted on the `sb` relation, and showed for every possible `sb`, the executions generated have an equivalent entity generated from the operational model. However, some possible executions generated from an out-of-order operational machine might violate the `sb` relation. For example, the execution of program piece (flow) in Fig. 6.6 follows the `sb` of (flow), but the execution of (flow-a) is also a valid execution for (flow), if we consider the possibility of executing the

T2 thread in (flow) out-of-order; then, the new sb relation makes the program (flow) looks more like (flow-a). In this case, the proof needs to unfold the single-thread control flow graph (CFG) to match the sb relation due to the out-of-order execution, such as unfold the program (flow) to (flow-a). The unfolding needs to be done before the induction on the sb relation, which is impossible because one cannot guess all possible out-of-order execution sequences before running the operational machine to actually see the sequence. What they did [152] was to have an intermediate in-order operational machine to execute programs exactly follow sb, and show that the equivalence between the in-order machine and the axiomatic model; and then showed that the in-order and out-of-order machines are threadwise bisimilar.

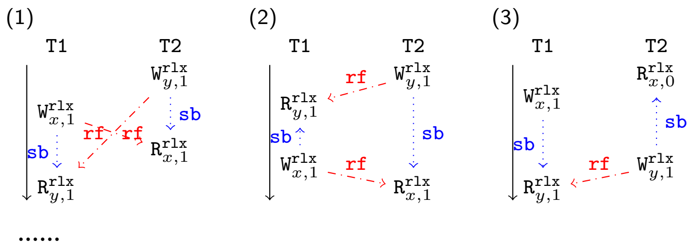


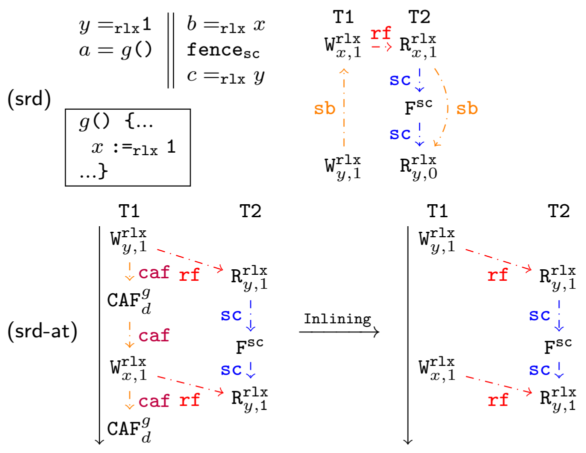
Figure 6.7: HATRMM Execution Diagrams for flow

With the time points in HATRMM, the executions of (flow) can be simply expressed as the diagrams ((1), (2), and (3) and more) in Fig. 6.7. Events are free to execute in positions that violate the sb relations, such as the upward sb edges in the examples (2) and (3). The linearizations of the executions are represented as the time point flow downward arrows. At each time point, an execution can only have one memory event (represented by the y-axis of the execution diagrams). By time points, we can express all out-of-order/speculative executions for a program without unfolding the program in HATRMM. In the Sec. 6.4, we introduce an LLVM operational model that is proved to be equivalent to HATRMM. In fact, the HATRMM model can be extended to include more elements as to instantiate as an operational model directly (see Sec. 6.4).

Obviously, the execution linearizations are not completely random. There are constraints in HATRMM to restrict the selection of the linearizations. We will introduce them in the later sections. Additionally, with time points, we are able to divide the HATRMM model into a single-threaded execution model capturing all single-threaded dependencies for all memory events happened in a thread, and a multi-threaded memory operation scheduling model capturing the interactions between threads and memory. We are also able to define more precise single-threaded dependencies (program order po) to substitute for the sb relation. More details are in Sec. 6.3.3.

6.3.2 Fencizing: Adding `CallFence` and `ControlFence`

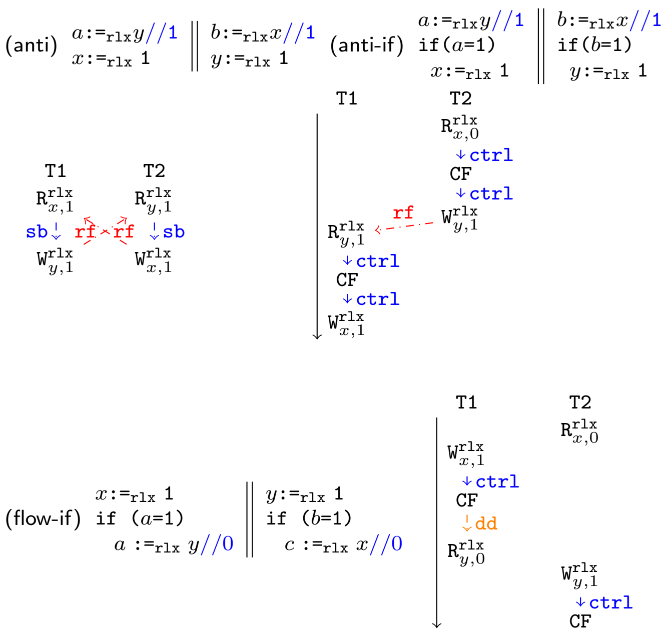
The purpose of "fencizing" is to create a uniform mechanism to define the concurrent behaviors of different operations that might affect the concurrency (other than the non-atomics, atomics and traditional fences). The concept is migrated from the memory fences. We will see two example new fences: `ControlFences` (`CF`) and `CallFences` (`CAF`).



(srd)
$y =_{\texttt{rlx}} 1$
$a = g()$
$b =_{\texttt{rlx}} x$
$\texttt{fence}_{\texttt{sc}}$
$c =_{\texttt{rlx}} y$
$g()$ {...
$x :=_{\texttt{rlx}} 1$
...}
T1
T2
$\texttt{W}^{\texttt{rlx}}_{x,1}$
rf
$\texttt{R}^{\texttt{rlx}}_{x,1}$
sb
sc
$\texttt{F}^{\texttt{sc}}$
sc
sb
$\texttt{W}^{\texttt{rlx}}_{y,1}$
$\texttt{R}^{\texttt{rlx}}_{y,0}$
(srd-at)
T1
T2
$\texttt{W}^{\texttt{rlx}}_{y,1}$
caf
rf
$\texttt{R}^{\texttt{rlx}}_{y,1}$
$\texttt{CAF}^{g}_{d}$
sc
caf
$\texttt{F}^{\texttt{sc}}$
$\texttt{W}^{\texttt{rlx}}_{x,1}$
sc
caf
rf
$\texttt{R}^{\texttt{rlx}}_{y,1}$
$\texttt{CAF}^{g}_{d}$
Inlining
T1
T2
$\texttt{W}^{\texttt{rlx}}_{y,1}$
rf
$\texttt{R}^{\texttt{rlx}}_{y,1}$
sc
$\texttt{F}^{\texttt{sc}}$
$\texttt{W}^{\texttt{rlx}}_{x,1}$
sc
rf
$\texttt{R}^{\texttt{rlx}}_{y,1}$

Figure 6.8: `CAF` Examples

First we discuss the `CAF` fences, whose motivation example is given in Fig. 6.8. Let's assume that the function g in (srd) only has one memory operation (a write to location x). The `sc` fence will be introduced in Sec. 6.3.3. We now assume its functionality is to force the `sb` execution order of the two reads in thread `T2`. The previous models [3, 4] allow one memory execution, as in the diagram shown in (srd), where one can observe the read from y to have the value `0` because the write to x in `T2` can actually be executed before the write to y. However, this does not match with the case of a real-world compiler. In compiling a program to machine-level code, a lot of fences are generated to prevent CPU reordering instructions. In all current compiler implementations (C, C++, LLVM, etc), without specific flags (e.g. enabling function inlining optimization in some specific cases), function calls are always surrounded by fences to prevent later instructions from being executed earlier than the content in the function calls. HATRMM has `CAFs` to prevent exactly this execution behavior in (srd). For each function call in a program, the generated memory execution has a pair of `CAFs` surrounding the execution of the function body as shown in the first execution diagram in (srd-at). Each element of such a pair contains the function name ($\mathcal{FName}$ in Fig. 6.2) that it surrounds, and an action-ID (d) generated to identify the pair. The arguments of `CAFs` allow us to perform function call optimizations in HATRMM executions. For example, if we perform a function inline expansion optimization on the function call g in program (srd), an execution of the program results in an execution diagram like the second one in (srd-at), where the two CAFs have been removed. Generally speaking, inlining expansions are not thread-safe optimizations. However, we like to show that it is valid to apply such optimization under a certain condition, i.e. the function being surrounded has no memory operations in it, because compilers actually perform such optimizations in some cases.



```
(anti)     a:=rlx y//1   ||  b:=rlx x//1
           x:=rlx 1      ||  y:=rlx 1

(anti-if)  a:=rlx y//1   ||  b:=rlx x//1
           if(a=1)       ||  if(b=1)
             x:=rlx 1    ||    y:=rlx 1

(flow-if)  x:=rlx 1      ||  y:=rlx 1
           if (a=1)      ||  if (b=1)
             a :=rlx y//0 ||   c :=rlx x//0
```

Figure 6.9: A Program Having CF Fences and Its Execution Diagrams

A ControlFence (CF) represents the control dependency in an execution for a program. Previously, RC11 [3] did not include control dependency (ctrl), so the execution listed beside (anti-if) in Fig. 6.9 is not valid for both the program (anti) and (anti-if) in RC11. IMM [4] tried to validate the execution for the (anti) case by defining ctrl, but its ctrl relation is too weak, so the execution is also valid for (anti-if) in their model. This is because the ctrl relation in IMM is defined to be a data-dependent relation between the uses of variables in the Boolean guards of a program and the sequenced-after writes of the variables. Clearly, the uses of variables a and b do not bound the writes to x and y in (anti-if). In HATRMM, we define an extra fence CF representing a control dependency in a program. For example, a possible execution for (anti-if) is listed as the right execution in Fig. 6.9. The CFs generally have dependency with all other memory events sequenced-before or after them in the same thread. In these cases, one cannot observe both reads in (anti-if) to have 1.

Moreover, HATRMM allows speculative reads by removing the `ctrl` edges between the `CFs` and their sequenced-after reads. One example is observable in the (flow-if) in Fig. 6.9: the two reads do not have any `ctrl` edges on them because they are sequenced-after the `CFs`. However, thread `T1` still has a data dependency from the `CF` to the read from a, because the Boolean guard in thread `T1` contains the use of variable a, and an assignment of a is sequenced-after the use.

In HATRMM, the behaviors of `CAFs` and `CFs` are defined by two predicates `call_dep` and `control_dep` in Fig. 6.10, provided with time points T, mapping ρ, and `sb` for each thread in Tid in an execution ζ.

$$\begin{array}{l}
\texttt{call\_dep}(T,\rho,\texttt{sb}) \triangleq \\
\quad \{(s,t) | (s,t) \in \texttt{sb} \\
\quad \wedge(\texttt{is\_call\_fence}(\rho(s)) \\
\quad \vee \texttt{is\_call\_fence}(\rho(t)))\}
\end{array}
\qquad
\begin{array}{l}
\texttt{control\_dep}(T,\rho,\texttt{sb}) \triangleq \\
\quad \{(s,t) | (s,t) \in \texttt{sb} \\
\quad \wedge((\texttt{is\_control\_fence}(\rho(s)) \\
\qquad\quad \wedge\neg\texttt{is\_read}(\rho(t))) \\
\qquad \vee(\texttt{is\_control\_fence}(\rho(t))))\}
\end{array}$$

Figure 6.10: The HATRMM Predicates for `CAF` and `CF`

The `CallFence` and `ControlFence` are examples of fencizing in HATRMM. We believe that other operations and flags can also define in the same manner as these fences. For example, to define the `readonly` flag for a function, we only need to add one more condition in `control_dep` in Fig. 6.10, such that there is no writes in between the two call fences surrounding the function.

6.3.3 Hybridization

Hybridization is to properly hybridize HATRMM, which can be split into a subset that satisfies DRF-SC and represents concurrency in high-level language primitives, and the other set that is weaker than the first set and represents concurrency in machine-level instructions. The Hybridization of a concurrency model is not as simple as merging two complete different systems together, but also taking care of the difference between atomics and non-atomics, as well as different memory orderings. With the HATRMM linearization, these differences become possible to implement in a model.

Addressing Non-Atomic Memory Operations. We first discuss how to merge non-atomic operations in HATRMM with duality (see the beginning of the chapter). We first revise the previous definition about the non-atomics without mixed-size values, i.e., the values of atomics and non-atomics having the same size. Then, we develop the model for handling the non-atomics with mixed-size values. For the first aspect, the previous modeling of non-atomic operations [2, 106] was a pandemic, a series of straightened definitions based on a design mistake in the original C++ memory model. The original design in C++ about the non-atomics is based on the happens-before relation (`hb`). The paper [153] discussed several possible definitions for modeling non-atomics in C11, but they pointed out that all of the solutions had problems.

```
      a:=sc x      ║ b:=sc y
(arr) if(a=1)      ║ if(b=1)
        y:=na 1    ║   x:=na 1
```

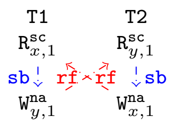


Figure 6.11: A Motivated Program Execution with Non-Atomics and `sc` Atomics

For example, if the behavioral definition of non-atomics is that they does not violate the `hb` relation, then the execution in Fig. 6.11 observing both `1` in the two reads is valid for the program (arr), because there are no `hb` edges between the reads and writes according to the definition in [153]. However, the execution of (arr) is obvious problematic and is a typical OOTA behavior. According to previous hardware concurrency models [149, 154], there is no difference between the concurrent constraints for non-atomics and `rlx` atomics if they deal with the values having the same size (e.g. same 32-bit integer values). The problem of implementing non-atomics solely to have the same constraints as `rlx` atomics is that the final model does not satisfy the DRF-SC property. In HATRMM, non-atomics are implemented hybridly. We have a set of non-atomics having the equivalent constraint as `rlx` atomics as a compilation target for machine-level instructions, and a set of non-atomics having the equivalent constraint as `acq`/`rel` atomics as high-level language primitives. We will introduce the relationship between these two sets and the DRF-SC property in Sec. 6.3.4. Here we refer "the lowest atomic constraint" to be the constraint of `rlx` atomics for machine-level instructions and the `acq`/`rel` atomics for high-level language primitives.

In this sense, the solution in HATRMM to deal with non-atomics is to equate the non-atomics constraints to the lowest atomic constraint, i.e., having no special constraints but requiring single-threaded data dependencies and multi-threaded memory operation scheduling constraints. The constraint details are in Sec. 6.3.3. With the constraints, the (arr) execution in Fig. 6.11 is impossible to happen in HATRMM where the two reads never load the value `1` at the same time.

The second important invention for dealing with non-atomics handles the mixed-size structures. The main usage of non-atomics in a language like C++ is `memcpy`, i.e. to read or write a large chunk of data. Usually, the non-mixed-size non-atomic read/write in C++ is treated the same as an `rlx` atomic read/write. In this case, an execution never causes any race. The main source of races is from the conflict between a large non-atomic read/write chunk and other operations. Flur *et al.* [155] introduces a solution for modeling large chunk non-atomic operations. They view a large chunk non-atomic operation as a list of atomic operations with the lowest atomic constraint. Each atomic operation in the list has an `rf` edge with other operations (not in the list). The race definition pertains a conflict analysis of different `rf` edges on the list. This works fine if we keep the non-atomics and atomics separated (as mentioned in Sec. 1); but, once we combine the two, things get odd.

The execution of (pass) in Fig. 6.12 shows the oddness. The different group-IDs d and d' represent different groups of non-atomics originated from different non-atomic memory instructions in the program. For example, the two non-atomic writes $\texttt{NW}^d_{x,1}$ are in the same group and originated from the instruction $x\texttt{:=}_{\texttt{na}}\ \texttt{[1,1]}$.

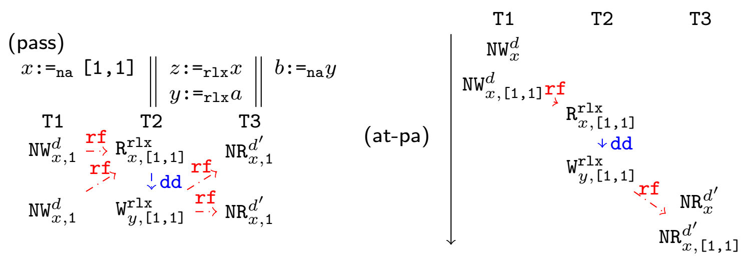

(pass)
$x\texttt{:=}_{\texttt{na}}\ \texttt{[1,1]}$
$z\texttt{:=}_{\texttt{rlx}}x$
$y\texttt{:=}_{\texttt{rlx}}a$
$b\texttt{:=}_{\texttt{na}}y$
T1
T2
T3
$\texttt{NW}^d_{x,1}$
$\texttt{R}^{\texttt{rlx}}_{x,\texttt{[1,1]}}$
$\texttt{NR}^{d'}_{x,1}$
rf
rf
rf
rf
dd
$\texttt{NW}^d_{x,1}$
$\texttt{W}^{\texttt{rlx}}_{y,\texttt{[1,1]}}$
$\texttt{NR}^{d'}_{x,1}$
(at-pa)
T1
T2
T3
$\texttt{NW}^d_x$
$\texttt{NW}^d_{x,\texttt{[1,1]}}$
rf
$\texttt{R}^{\texttt{rlx}}_{x,\texttt{[1,1]}}$
dd
$\texttt{W}^{\texttt{rlx}}_{y,\texttt{[1,1]}}$
rf
$\texttt{NR}^{d'}_x$
$\texttt{NR}^{d'}_{x,\texttt{[1,1]}}$

Figure 6.12: A Complicated Program Execution with Non-Atomics

As shown in the (pass) execution in Fig. 6.12, $\texttt{rf}^{-1}$ is not functional anymore. The data being read in each `rf` edge becomes unclear. Our solution is also to have a group of memory operations representing a non-atomic instruction, but we require that only one operation in the group can appear in an `rf` relation. This operation is always the latest one in time (greater than other operations in the same group) in an execution. The execution with mixed-size non-atomics is like the (at-pa) execution, which is an execution in HATRMM for (pass). In (at-pa), only the latest operation produced by the instruction ($\texttt{NW}^d_{x,\texttt{[1,1]}}$ for $x\texttt{:=}_{\texttt{na}}\ \texttt{[1,1]}$ and $\texttt{NR}^{d'}_{x,\texttt{[1,1]}}$ for $b\texttt{:=}_{\texttt{na}}y$) has an `rf` edge. In HATRMM, we define the behavior with two assumptions on non-atomics. One obvious assumption about our `rf` relations is that $\texttt{rf}^{-1}$ is functional. All source events in `rf` are writes while all target events are reads. In addition, for all pair $(s,t) \in \texttt{rf}$, $s < t$. Finally, we also have the assumption in Fig. 6.13 about the `NRead` and `NWrite` actions.

Remember that there are two natural numbers (n and m) fields in the syntax of the

$$\begin{aligned}\texttt{non\_atomic\_well\_order}\ (\texttt{Tids}, \texttt{sbs}, \rho) \triangleq\ & \forall t \in \texttt{Tids}\ .\forall a\ b \in (\texttt{sbs}\ (t)).\\ & \texttt{is\_non\_atomic}\ (\rho(a))\\ & \wedge \texttt{is\_non\_atomic}\ (\rho(b))\\ & \wedge \texttt{same\_gid}\ (\rho(a), \rho(b))\\ & \Rightarrow \texttt{get\_i}\ (\rho(a)) < \texttt{get\_i}\ (\rho(b))\end{aligned}$$

Figure 6.13: Well-Ordered Assumption for Non-Atomics

actions (Fig. 6.2). The second natural number n represents the total number of non-atomic actions for a particular group-ID (`gid`), while the first m shows that a non-atomic action is the m-th piece of the non-atomic action. In the implementation of HATRMM in Isabelle, for simplicity, we assume that all of the non-atomic actions have an m range from 1 to n. The m-th non-atomic action for a `gid` happens at `gid`'s m-th rank in a candidate execution (defined by the `non_atomic_well_order` predicate in Fig. 6.13). This assumption only talks about the non-atomics in one particular thread, because we also have the assumption that non-atomic actions with the same `gid` only happen in a thread, as an observation on the real-world situation that there is no use for two different threads dealing with a single memory operation. The assumption eases our proofs and allows us to abstract away the mechanism for distributing correct memory actions into correct memory locations. For now, we have a lemma about non-atomic actions to suggest that if an action happens at a time and its m and n values are the same, then the non-atomic memory event is the latest one in the group of non-atomic memory events in terms of time points; otherwise, at a given time, the non-atomic memory operation is still waiting for more actions to finish. Only the latest memory event in a group of memory events can join an edge in `rf`. We establish that if $(s, t) \in \texttt{rf}$ and s or t is a non-atomic event, then its m value is the same as its n value.

The reason for the implementation is that people usually care about the `rf` edges in an execution if there is no race so that they want to analyze `rf` edges to produce the correct value for each read. On the other hand, if there is a race, people usually care about how to detect it instead of analyzing `rf` edges to get the value for a racy read since such read has an undefined value and there is no "correct" value.

The definition of race detection is in Def. 6.1. This definition does not depend on the analysis of `rf` edges. It only finds if there is an outlier operation in the time point range of a group of non-atomic operations in an execution.

Definition 6.1. An execution $(Tid, Loc, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$ has a **race**, iff there is a group d of non-atomics accessing a location x, the earliest time point for d is s, and the latest time point for d is t in the time point range $[s, t]$, and the following happens:

- An atomic write accessing x happens between s and t.
- An atomic read accessing x happens between s and t, and d is a group of non-atomic writes.
- A non-atomic write accessing x happens between s and t, and its group-ID is not d.
- A non-atomic read accessing x happens between s and t, its group-ID is not d, and d is a group of non-atomic writes.

A program is **race-free** if any valid execution of the program in HATRMM has no race.

Here we mainly discussed the difference between the HATRMM modeling of non-atomics and that of previous models. A formal definition of HATRMM non-atomics which also includes the implementation of mutex locks is found in Fig. 6.14.

$$\begin{array}{l}
\texttt{non\_ac\_asm}(T,\rho,\texttt{rf}) \triangleq \ \forall (s,t) \in \texttt{rf}.\ ((\rho(s) = (tid, d, \texttt{NW}_{(x,n,m)}) \Rightarrow n = m) \wedge (\rho(t) = (tid, d, \texttt{NR}_{(x,n,m)}) \Rightarrow n = m)) \\
\texttt{non\_ac\_asm1}(T,\rho) \triangleq \ \forall \{s,t\} \subseteq T.\ s < t \wedge \texttt{is\_non\_atomic}(\rho(s)) \wedge \texttt{is\_non\_atomic}(\rho(t)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \texttt{get\_gid}(\rho(s)) = \texttt{get\_gid}(\rho(t)) \Rightarrow \texttt{get\_pos}(\rho(s)) < \texttt{get\_pos}(\rho(t)) \\
\texttt{inMid}(T,\rho,s,t,ac) \triangleq \exists r.r \in T \wedge s < r < t \wedge (\exists tid\ d.\rho(r) = (tid,d,ac)) \\
\texttt{lock\_dep}(T,\rho,\texttt{sb}) \triangleq \ \{(s,t) | (s,t) \in T^2 \cap \texttt{sb} \qquad\qquad \texttt{unlock\_dep}(T,\rho,\texttt{sb}) \triangleq \ \{(s,t) | (s,t) \in T^2 \cap \texttt{sb} \\
\qquad\qquad \wedge(\texttt{is\_lock}(\rho(s)) \vee \texttt{is\_unlock}(\rho(t)))\} \qquad\qquad\qquad \wedge(\texttt{is\_unlock}(\rho(s)) \vee \texttt{is\_unlock}(\rho(t)))\} \\
\\
\texttt{lock\_prop}(T,\rho,s,k) \triangleq \ (\forall t \in T.\ t > s \wedge \texttt{same\_tid}\ (\rho(s),\rho(t)) \Rightarrow \texttt{get\_ac}(\rho(t)) \neq \texttt{UL}_k \wedge \texttt{get\_ac}((\rho(t)) \neq \texttt{L}_k) \\
\qquad \vee(\exists t \in T.t > s \wedge \texttt{same\_tid}(s,t) \wedge \texttt{get\_ac}(\rho(t)) = \texttt{UL}_k \wedge \neg \texttt{inMid}(T,\rho,s,t,\texttt{L}_k) \wedge \neg \texttt{inMid}(T,\rho,s,t,\texttt{UL}_k) \\
\qquad\quad \wedge(\exists r \in T.\ r > t \wedge \texttt{get\_ac}(\rho(r)) = \texttt{UL}_k \Rightarrow \texttt{inMid}(T,\rho,t,r,\texttt{L}_k))) \\
\texttt{locks}(T,Key,\rho) \triangleq \ \forall t \in T.\ (\forall k \in Key.\ \texttt{get\_ac}(\rho(t)) = \texttt{L}_k \Rightarrow \texttt{lock\_prop}(T,\rho,t,k)) \\
\texttt{race\_def}(T,\rho) \triangleq \ \exists s\ t\ r \in T.\ \texttt{is\_non\_atomic}(\rho(t)) \wedge \texttt{is\_non\_atomic}(\rho(r)) \wedge \neg \texttt{both\_read}(\rho(s),\rho(r)) \wedge \texttt{same\_loc}(\rho(t),\rho(r)) \\
\qquad\quad \wedge \texttt{same\_gid}(\rho(t),\rho(r)) \wedge \texttt{get\_pos}(\rho(t)) = 1 \wedge \texttt{get\_pos}(\rho(r)) = \texttt{get\_num}(\rho(r)) \wedge t < s < r \\
\qquad\quad \wedge \texttt{is\_mem\_op}(\rho(s)) \wedge \neg \texttt{same\_gid}(\rho(t),\rho(s)) \wedge \texttt{same\_loc}(\rho(t),\rho(s))
\end{array}$$

Figure 6.14: Non-Atomic Actions and Lock Predicates

The complete HATRMM model contains locks, which are the `NR`, `NW`, `L` and `UL` actions (Fig. 6.2). The group of functions `get_some` gets a field whose name is `some` in a memory event, while the predicates `same_some` check if two memory events in the field `some` have the same value. Some of these function definitions are found in Sec. 6.3.

The example (`nac`) in Fig. 6.15 provides an example execution containing non-atomics. For the program piece on the left, two groups of non-atomic reads and writes, with different group-IDs (d and d'), are shown in the middle column. They both have two group members. `NW` writes `1` to memory and `NR` reads the value. The right column shows a simplified execution diagram of the middle-column execution without mentioning the locks. In any execution, the position indices of the non-atomics in each group increase from `1` to `2` following the time flow. There is only one `rf` edge from the write to the read having both position indices equal to `2` (the total number of group members).

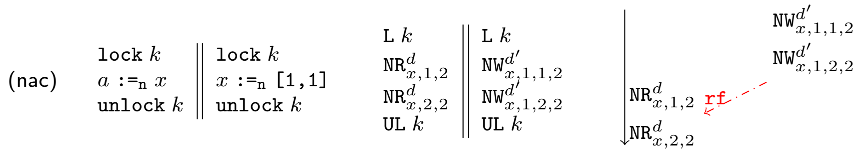

(nac)
lock k
a :=$_n$ x
unlock k
lock k
x :=$_n$ [1,1]
unlock k
L k
$\texttt{NR}^{d}_{x,1,2}$
$\texttt{NR}^{d}_{x,2,2}$
UL k
L k
$\texttt{NW}^{d'}_{x,1,1,2}$
$\texttt{NW}^{d'}_{x,1,2,2}$
UL k
$\texttt{NW}^{d'}_{x,1,1,2}$
$\texttt{NW}^{d'}_{x,1,2,2}$
$\texttt{NR}^{d}_{x,1,2}$
rf
$\texttt{NR}^{d}_{x,2,2}$

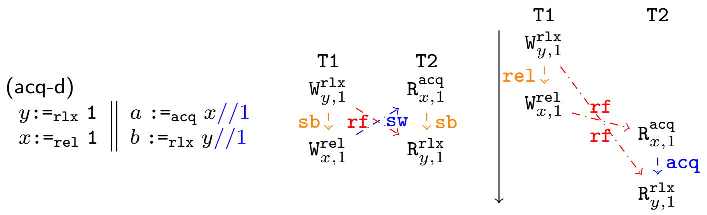
Figure 6.15: An Example Execution with Non-Atomics and Locks

The validity predicates on locks (`L` and `UL`) are split into single-threaded behavioral and multi-threaded behavioral predicates. The predicates `lock_dep` and `unlock_dep` handle the single-threaded behaviors, which disallow the execution of memory operations crossing locks out of order. The predicate `locks` take an execution and look at every `L` to guarantee that it is properly executed. Their core definition is the `lock_prop` predicate. There are two cases in `lock_prop`. First, a lock `L` with key k can exist without any $\texttt{UL}_k$ in the same thread later than the lock in an execution, provided that no other `L` with the same key exists after `L`. Second, a lock `L` with key k happens at time s, and a $\texttt{UL}_k$ in the same thread happens at a later time t (after s). Between times s and t, no $\texttt{L}_k$ can happen, nor can an unlock happen twice on k without a lock on k in between. The predicate `race_def` (Fig. 6.14) defines when a race can happen. An execution results in race if, between a non-atomic operation's first action time (when $i = 1$, time t) and last action time (when $i = n$, time r), there exists a time s that is a memory operation that has a different group-ID (or action-ID if it is atomic) than the actions at s and t, and they are all accessing the same memory location.

Atomic Memory Operations and Fences. HATRMM builds a hybrid model that is equivalent to the RC11 without having the extra `rlx` atomics for machine-level instructions as compilation targets; as well as having the `NO-THIN-AIR` constraint from the IMM model (with a fix on its `ctrl` relation) for `rlx` atomics. The whole concurrency model is split into two parts: (1) an execution model captures the single-threaded out-of-order execution behaviors; (2) a memory operation scheduling model describes the memory event interaction across different threads. A side effect of the hybridization in HATRMM is to enable directly construct an operational semantics (Sec. 6.24). Another advantage of the rearrangement is we can now develop a new **program order** (`po`) relation to substitute for `sb` relations, so that we have a more precise single-threaded dependencies than the traditional `sb` relations. We will describe these atomics and fences in the three paragraphs below as the single-threaded execution model, the memory operation scheduling model, and the `sc` atomics having both single-threaded and multi-threaded behaviors.

We first discuss some atomics and fences performing only **single-threaded behaviors**.

This group includes memory operations (`R`, `W` and `RW`) and memory fences (`F`), which are ordered by `rlx`, `acq`, `rel`, and `acqrel`. Among these atomics, the `rlx` atomics does not have a specific predicate restricting its behavior, while atomics with the `volatile` flag set follow the constraint of the volatile model. We named the ordering behaviors allowed by the single-threaded execution model as the program order relation (`po`). We view the `po` relation as the union of all single-threaded dependent relations, such as `dd` and the relations defined in this paragraph.



```
(acq-d)
y:=rlx 1 || a :=acq x//1
x:=rel 1 || b :=rlx y//1
```

Figure 6.16: An Example Execution with `acq`/`rel` Orderings

To deal with these atomics and fences, previous models employed a concept named synchronized-with relations (`sw`), which are special `rf` relations that occur mostly between `rel` writes and `acq` reads (an `sw` relation can also happen between locks and unlocks). The behaviors of `rel` writes and `acq` reads are usually described by `sw`. If an `acq` read loads from a `rel` write, there is an `sw` edge between the two, and all the happens-before events before the writes are observed by the events after the read. In example acq-d in Fig. 6.16, the read from y must observe `1` because once the `acq` read from x sees the value from the write to x, the write to y must have already happened. In HATRMM, we replaced `sw` with several single-threaded constraints on different atomics and fences, as well as the global memory operation scheduling constraint described in the next paragraph. Here we focus on the single-threaded constraints proposed by different atomics and fences. For each atomic operation with different ordering in an execution, we impose an additional dependent constraint between the operation and other operations in the same thread besides the existing data dependency (`dd`). For example, an execution in HATRMM for the example acq-d is listed on the right in Fig. 6.16. In threads `T1` and `T2`, there are additional dependency relations, named `rel` and `acq`, connecting `rel` write and `acq` read to other operations, respectively. We also discard the `sw` relation completely.

The reason for discarding `sw` relations and implementing atomics and fence behaviors in dual modeling (single-threaded and multi-threaded memory operation scheduling modelings) is that we want the model to reflect the essence of the real-world conceptual implementation of these operations. Essentially, there is no such thing as locking mechanism for pairs of `acq` reads and `rel writes` as seemingly suggested by `sw` relations. For example, there is no guarantee that the `acq` read in `acq-d` must read the value from the `rel` write. It actually can read any value occurring in the two threads. One possible value for the read is the initial value `0`. Based on a language developer's perspective, the behaviors of these kinds of `acq` and `rel` operations (including `acqrel` atomics and fences) are side-effects of multi-threaded memory operation scheduling behaviors and constraints on the `po` relation. HATRMM is designed to respect the perspective. We show two example predicates for defining single-threaded constraints for these atomics and fences in Fig. 6.17 `(P.a)`.

		C++/LLVM Documentations	HATRMM Predicates
`(P.a)`	Acquire Read Predicate	*no reads or writes in the current thread can be reordered before this acquire read.*	$\texttt{acqr}(T, \rho, \texttt{sb}) \triangleq \{(s,t) \in T^2 \cap \texttt{sb} \mid \texttt{is\_read}(\rho(s)) \land \texttt{is\_acq}(\rho(s)) \land \texttt{is\_mem\_op}(\rho(t))\}$
	Volatile Model Predicate	*The order of volatile operations is fixed. The order of volatile operations relative to non-volatile ones can change.*	$\texttt{vol\_dep}(T, \rho, \texttt{sb}) \triangleq \{(s,t) \in T^2 \cap \texttt{sb} \mid \texttt{is\_volatile}(\rho(s)) \land \texttt{is\_volatile}(\rho(t))\}$

		Supposed C++/LLVM Documentations	HATRMM Predicates
`(P.b)`	Modification Order	*For any location, the orders of writes oservered by reads in any two threads are the same.*	$\texttt{mo}(T, \rho) \triangleq \{(s,t) \in T^2 \mid a < b \land \texttt{same\_loc}(\rho(s), \rho(t)) \land \texttt{is\_write}(\rho(s)) \land \texttt{is\_write}(\rho(t))\}$
	Known Fact Order	*For any operation t in a thread, the known-fact order is either any writes happen-in-time before t in the thread, or any write known-fact-ordered before the write that a read (happen-in-time before t) reads from.*	$\texttt{kf}(T, \rho, \texttt{rf}) \triangleq \{(s,t) \in T^2 \mid s < t \land \texttt{same\_thread}(\rho(s), \rho(t)) \land \texttt{is\_write}(\rho(s)) \land \texttt{is\_mem\_op}(\rho(t))\} \cup \{(s,t) \mid \exists s'\ t'.\ s' < t \land \texttt{same\_thread}(\rho(s'), \rho(t)) \land \texttt{is\_read}(\rho(s')) \land \texttt{is\_mem\_op}(\rho(t)) \land (t', s') \in \texttt{rf} \land (s, t') \in \texttt{kf}(T, \rho, \texttt{rf})\}$
	Coherence Order Constraint	*Not exist a write-read pair (s,t), where the write s is non-immediately know-fact-ordered (extending with the modification-ordered of writes) of the read t.*	$\texttt{non\_co}(T, \rho, \texttt{rf}) \triangleq \{(s,t) \in \texttt{rf} \mid (s,t) \in (\texttt{kf}(T, \rho, \texttt{rf}) \uparrow \texttt{mo}(T, \rho))\vert_{\neq \texttt{imm}}\}$ $\texttt{non\_co}(T, \rho, \texttt{rf}) \equiv \emptyset$

Figure 6.17: Memory Model Documentation and Implementation

The table in Fig. 6.17 shows two example constraints defined in HATRMM, which are related to the definitions of the constraint models described in the C++/LLVM documentations (the middle columns). The first table entry defines the behaviors of `acq` reads. The constraint (`acqr`) places edges between an `acq` read and any its sequenced-after memory operations (reads or writes). These edges restrict the evaluation order in an execution and they are included in `po`. The constraint also models the behaviors of `acq` reads to exactly reflect the C++ definition for the atomics. The second table entry above unveils the predicate for capturing the volatile model in HATRMM: it places constraint edges between any two flagged volatile memory operations. The LLVM description of the volatile model is listed in the middle. The `po` order is defined to be the union of all these single-threaded constraints.

The constraints for restricting other atomics and fences are in Fig. 6.18.

Intra-Fence Relation: $\texttt{ff\_dep}(T, \rho, \texttt{sb}) \triangleq \{(s,t) \in T^2 \cap \texttt{sb} \mid \neg\texttt{is\_mem\_op}(\rho(s)) \wedge \neg\texttt{is\_mem\_op}(\rho(t))\}$
Acquire Fence Relation: $\texttt{acqf}(T, \rho, \texttt{sb}) \triangleq \{(s,t) \in T^2 \cap \texttt{sb} \mid \texttt{is\_read}(\rho(s)) \wedge \texttt{is\_acq}(\rho(t)) \wedge \texttt{is\_fence}(\rho(t))\}$
$\cup\{(s,t) \in T^2 \cap \texttt{sb} \mid \texttt{is\_acq}(\rho(s)) \wedge \texttt{is\_fence}(\rho(s)) \wedge \texttt{is\_mem\_op}(\rho(t))\}$
SeqCst Read Relation: $\texttt{seqr}(T, \rho, \texttt{sb}) \triangleq \{(s,t) \in T^2 \cap \texttt{sb} \mid \texttt{is\_read}(\rho(s)) \wedge \texttt{is\_sc}(\rho(s)) \wedge \texttt{is\_mem\_op}(\rho(t))\}$
$\cup\{(s,t) \in T^2 \cap \texttt{sb} \mid \texttt{is\_write}(\rho(s)) \wedge \texttt{is\_sc}(\rho(t)) \wedge \texttt{is\_read}(\rho(t))\}$
SeqCst Write Relation: $\texttt{seqw}(T, \rho, \texttt{sb}) \triangleq \{(s,t) \in T^2 \cap \texttt{sb} \mid \texttt{is\_mem\_op}(\rho(s)) \wedge \texttt{is\_sc}(\rho(t)) \wedge \texttt{is\_write}(\rho(t))\}$
$\cup\{(s,t) \in T^2 \cap \texttt{sb} \mid \texttt{is\_write}(\rho(s)) \wedge \texttt{is\_sc}(\rho(s)) \wedge \texttt{is\_write}(\rho(t))\}$

Figure 6.18: Single-Threaded Behavioral Predicates

In Fig. 6.18, the `ff_dep` requires that the relative execution order for different fences is the same as the one given by `sb`. Some `sc` atomics also contain some single-threaded behaviors in the system. We split their single-threaded behaviors and their multi-threaded ones, and put the single-threaded ones as predicates showing in Fig. 6.18.

We then focus on the **multi-threaded memory operation scheduling model without considering SC atomics**. An execution ζ in HATRMM has an input relation `rf`, which contains write-read pairs that might cross threads. The appearance order of cross-threaded write-read pairs in an `rf` is constrained by the scheduling model, which is similar conceptually to a synchronized message sending/receiving model in a distributed system. In this sense, the scheduling model is just a particular synchronization strategy in a message sending-receiving model. We first see two important concepts useful in the section. $R \uparrow R'$ is defined as $\{(s,t) | (s,t) \in R \vee (\exists (s',t) \in R \wedge (s,s') \in R')\}$, which represents a form of extending the relation R with R'. $R(T,\rho)|_{\neq \texttt{imm}}$ is the collection of all non-immediate edges for the same locations in R with at least input arguments T and ρ as $\{(s,t) \in R(T,\rho) | \exists s'.(s,s') \in R(T,\rho) \wedge (s',t) \in R(T,\rho) \wedge \texttt{same\_loc}(\rho(s),\rho(t)) \wedge \texttt{same\_loc}(\rho(s'),\rho(t)) \wedge \texttt{is\_read}(\rho(t))\}$.

We named the HATRMM multi-threaded memory operation scheduling model as the coherence order (`co`), which originated from RC11's [3] coherence consistent constraint. In fact, the memory event ordering behaviors allowed in the coherence order matches exactly what is allowed by the RC11 coherence constraint (replacing the `sb` relation with the `po` relation shown in the next paragraph). HATRMM's `co` order is defined by the predicates shown in the above table based on two additional relations: modification (`mo`) and known-fact orders (`kf`). For memory writes accessing a memory location in an execution, `mo` provides a global total order on the writes. The `kf` relation is basically an HATRMM version of the traditional happens-before relation (`hb`), defining the happens-before writes for any memory operation in an execution. The difference is that the traditional `hb` is based on `sb` relations, while `kf` is based on the time point orders ($<$, happens-before in-time relations). We list the relations and the `co` order constraint on the right of the table. The `co` order is defined by collecting all "bad behaviors" (non_co) and setting the collection to be empty.

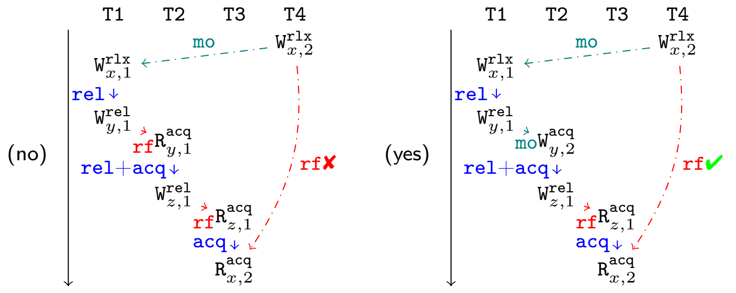


Figure 6.19: An Example Execution Showing Ordering Multi-threaded Behaviors

In RC11, coherence constraint is defined as an acyclic property of a relation graph. On the other hand, The HATRMM co order is defined only based on finding bad candidate write-read pairs in rf. We need to check if a read loading a write value that is the read's "old value", which is defined as the non-immediate extended known-fact ordered ($\texttt{kf}(T, \rho, \texttt{rf}) \uparrow \texttt{mo}(T, \rho)$) of the read. We explain this by the example execution diagrams in Fig. 6.19. For each read in a write-read pair in rf, we create the idea of **old** write values. For any read t, such that (s, t) in the kf relation, the write in s is an **old** write value if we can find a intermediate write at s' between s and t, such that they have the same locations and (s, s') and (s', t) also in the extended known-fact relation ($\texttt{kf} \uparrow \texttt{mo}$). For any write-read pair, it is invalid for the read to load an old write value. For example, there is a chain of po and rf edges starting from the write (to x) in T1 to the read (from x) in T3 in case (no) in Fig. 6.19. All writes to x in (no) have edges connected to the read from x. Thus, the write to x in T4 is the old value of the read from x because it happens-in-time before the write to x in T1. Thus, the execution does not satisfy the co order. On the other hand, we create a special operator $\uparrow$ here because mo is not transitive. The $\uparrow$ operator only extends the s write in the (s, t) pair in kf to the mo-ordered write proceeding s, i.e. if (s, t) is in kf and (s', s) is in mo, then (s', t) is in $\texttt{kf} \uparrow \texttt{mo}$. For example, case (yes) is a valid execution because there is only a mo edge connecting the two writes to y. Thus, the write to x in T1 in (yes) does not have an edge in $\texttt{kf} \uparrow \texttt{mo}$ to the read from x, and the write in T4 is not an old write value in (yes).

In Fig. 6.17 (P.b), we propose descriptions of different components in the co order using English words, since the C++/LLVM documentation lacks them. These descriptions can function as the standard descriptions for the documentation. Another advantage of the rearrangement of co is that the constraint is now highly related to each rf edge, and it can then be easily written as a model checking algorithm for checking rf edges. Finally, the

$$\texttt{sc\_fence\_in\_mid}(T,\rho,s,t) \triangleq (\exists t'.s < t' < t \wedge \{s,t,t'\} \subseteq T \wedge \texttt{is\_sc\_fence}(\rho(t')))$$
$$\texttt{write\_in\_mid}(T,\rho,s,t,x) \triangleq (\exists t'.s < t' < t \wedge \{s,t,t'\} \subseteq T \wedge \texttt{is\_write}(\rho(t')) \wedge \texttt{same\_loc}(\rho(t'),x))$$
$$\texttt{sc\_read\_in\_mid}(T,\rho,s,t,x) \triangleq (\exists t'.s < t' < t \wedge \{s,t,t'\} \subseteq T \wedge \texttt{is\_sc\_read}(\rho(t')) \wedge \texttt{same\_loc}(\rho(t'),x))$$
$$\begin{aligned}
&\texttt{sr}(T,\rho,\texttt{rf}) \triangleq \\
&\quad (s,t) \in T^2 \wedge s < t \wedge \texttt{is\_write}(\rho(s)) \wedge \texttt{is\_sc\_fence}(\rho(t)) \wedge x = \texttt{get\_loc}(\rho(s)) \\
&\quad \wedge\neg\texttt{sc\_fence\_in\_mid}(T,\rho,s,t) \wedge \neg\texttt{write\_in\_mid}(T,\rho,s,t,x) \wedge \neg\texttt{sc\_read\_in\_mid}(T,\rho,s,t,x) \\
&\qquad\qquad \Rightarrow (s,x,t) \in \texttt{sr}(T,\rho,\texttt{rf}) \ (*\ \mathsf{scBase}\ *) \\
&\quad \mid (s,x,t') \in \texttt{sr}(T,\rho,\texttt{rf}) \wedge t' < t \wedge t \in T \wedge \texttt{is\_sc\_fence}(\rho(t)) \wedge x = \texttt{get\_loc}(\rho(s)) \\
&\quad \wedge\neg\texttt{sc\_fence\_in\_mid}(T,\rho,t',t) \wedge \neg\texttt{write\_in\_mid}(T,\rho,t',t,x) \wedge \neg\texttt{sc\_read\_in\_mid}(T,\rho,t',t,x) \\
&\qquad\qquad \Rightarrow (s,x,t) \in \texttt{sr}(T,\rho,\texttt{rf}) \ (*\ \mathsf{scInduct}\ *) \\
&\quad \ldots
\end{aligned}$$
$$\texttt{cw}(T,\rho,\texttt{rf}) \triangleq \{(s,x,t) | (s,t) \in \texttt{rf} \wedge x = \texttt{get\_loc}(\rho(s))\} \cup \texttt{sr}(T,\rho,\texttt{rf})$$
$$\begin{aligned}
&\texttt{non\_sc}(T,\rho,\texttt{rf}) \triangleq \\
&\quad \{(s,t) | (s,t) \in \texttt{rf} \wedge (\exists t' \in T.\texttt{is\_sc\_write}(\rho(t')) \wedge \texttt{same\_loc}(\rho(s),\rho(t')) \wedge s < t' < t)\} \\
&\quad \cup\{(s,t) | (s,t) \in \texttt{rf} \wedge (\exists t' \in T.\texttt{is\_sc\_fence}(\rho(t')) \wedge s < t' < t \wedge (s,\texttt{get\_loc}(\rho(s)),t') \notin \texttt{cw}(T,\rho,\texttt{rf}))\} \\
&\quad \cup\{(s,t) | (s,t) \in \texttt{rf} \wedge (\exists t' \in T.\texttt{is\_sc\_read}(\rho(t')) \wedge s < t' < t \wedge (s,\texttt{get\_loc}(\rho(s)),t') \notin \texttt{cw}(T,\rho,\texttt{rf}))\} \\
&\quad \cup\{(s,t) | (s,t) \in \texttt{rf} \wedge (\exists t' \in T.s \neq t' \wedge (t',\texttt{get\_loc}(\rho(s)),t) \in \texttt{cw}(T,\rho,\texttt{rf}))\}
\end{aligned}$$

Figure 6.20: Multi-Threaded Communication Predicates for SC Atomics

multi-threaded order proposed here is modular. For example, one can replace `co` with the `sc` atomics order appearing in the following paragraph so that all valid executions admit sequential consistency.

The third piece of the atomic model in HATRMM is **the SC model**, which described in the previous paragraphs does not include the `sc` atomics behaviors. In HATRMM, we use an `sc` atomic operation at a time t to provide an expensive locking mechanism that imposes a global consensus of the value for a location across all threads. The definition of the `sc` atomics behaviors is constructed with a `non_sc` bad behavior set. The details are in Fig. 6.20. Here, we explain it by the example executions in Fig. 6.21.

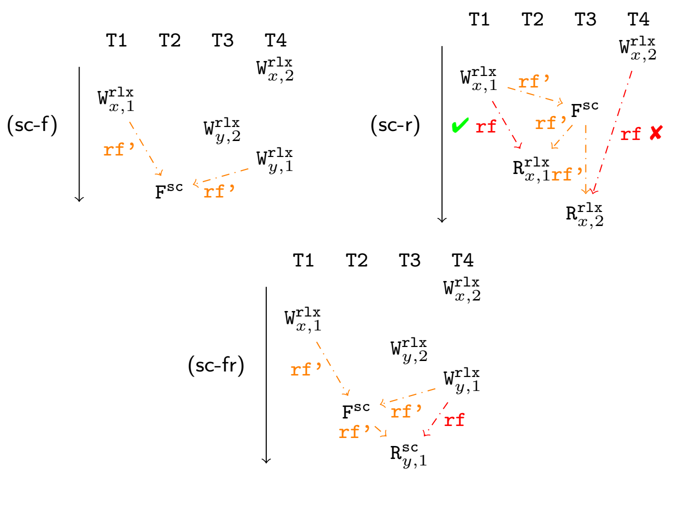


Figure 6.21: Example Executions of `sc` Atomics

Every `sc` atomic operation or fence has the effect of making the values for a memory location in different threads agree. An `sc` read or write consenses the values for a particular location, while an `sc` fence consenses the values at every location. The effect is explained by the execution `sc-f` in Fig. 6.21. For locations x and y, the `sc` fence consenses their values from the writes having the latest time points before the fence. Thus, it picks the values from the write to x in `T1` and the write to y in `T4` because they happen-in-time after the other two writes. This procedure is indicated by the `rf'` edges in the diagram, which indicate an `rf`-like relation between the writes and the `sc` fence but they are not real `rf` edges. In HATRMM, we define the `rf'` relation for every `sc` fence and read, described as the `cw` relation in Fig. 6.20.

The `sc-r` execution highlights how we utilize `rf'` to locate error relations for `sc` atomics in the `non_sc` constraint. There is no write between the read in `T2` and the `sc` fence. Thus, the read must load the same value (of x) as the fence observes, because the fence consenses the value for x across every thread. The two `rf'` edges and the `rf` edge between the write in `T1` and the read in `T2` form a nice triangle diagram, meaning that the fence reads the write value (`T1`), and the read (`T2`) happens-in-time after the fence without any write in between, so the read must load from the write. On the other hand, an `rf` edge between the read and write having the value `2` is an error because the read does not correspond to the write value consensed by the fence; and there is no write between the read (`T3`) and the fence. The `non_sc` set collects all of the invalid `rf` edges in an execution. Then a valid execution requires `non_sc` to be empty. The `sc-fr` execution in Fig. 6.21 shows that the `rf'` relation is also transitive. If there are two consecutive `sc` atomics (the `sc` fence and read) without any writes in between, the consensing value of the latter `sc` atomic operation (the read) must be the same as the former one.

We have stated all of the important single-threaded and multi-threaded memory operation and fence behaviors of HATRMM. In the following section, we will put these pieces together and show some theorems to relate HATRMM to the previous works.

6.3.4 Putting it All Together

We have described the interesting behaviors of HATRMM. Here we put all of the pieces of HATRMM together in Fig. 6.22. We define the **program order** of an execution to be the union all single-threaded constraints (`po` in the figure). We define **single-threaded consistency** as the predicate `always_prop`. This means for each `po` relation for a thread, there is no edge from a later time point pointing to an earlier one (`po` $\cup <$ is irreflexive). The definition of **coherence consistency** is that the set `non_co` is empty, which means

$$\begin{aligned}
\texttt{single\_order}(T,\rho,\texttt{sb}) &\triangleq \texttt{acqr}(T,\rho,\texttt{sb}) \cup \texttt{acqf}(T,\rho,\texttt{sb}) \cup \texttt{seqr}(T,\rho,\texttt{sb}) \cup \texttt{seqw}(T,\rho,\texttt{sb}) \cup \ldots \\
\texttt{po}'(T,\rho,\texttt{sb},\texttt{dd}) &\triangleq \texttt{dd} \cup \texttt{control\_dep}(T,\rho,\texttt{sb}) \cup \texttt{call\_dep}(T,\rho,\texttt{sb}) \cup \texttt{single\_order}(T,\rho,\texttt{sb}) \cup \texttt{vol\_dep}(T,\rho,\texttt{sb}) \\
\texttt{po}(Tid,T,\rho,\texttt{sbs},\texttt{dds}) &\triangleq \bigcup_{tid \in Tid} \texttt{po}'(T,\rho,\texttt{sbs}(tid),\texttt{dds}(tid)) \\
\texttt{always\_prop}(Tid,T,\rho,\texttt{sbs},\texttt{dds},\texttt{rf}) &\triangleq \forall (s,t) \in (\texttt{rf} \cup \bigcup_{tid \in \texttt{Tid}} \texttt{po}'(T,\rho,\texttt{sbs}(tid))).s < t \;\; \textbf{(single-threaded consistency)} \\
\texttt{sat}(Tid,Loc,Key,T,\rho,\texttt{sbs},\texttt{dds},\texttt{rf}) &\triangleq \texttt{well\_formed}(Tid,Loc,T,\rho,\texttt{sbs},\texttt{dds},\texttt{rf}) \wedge \texttt{always\_prop}(Tid,T,\rho,\texttt{sbs},\texttt{dds},\texttt{rf}) \\
&\quad \wedge \texttt{non\_co}(T,\rho,\texttt{rf}) \equiv \emptyset \;\; \textbf{(coherence consistency)} \\
&\quad \wedge \texttt{non\_sc}(T,\rho,\texttt{rf}) \equiv \emptyset \;\; \textbf{(sc consistency)} \\
&\quad \wedge \texttt{locks}(T,Key,\rho) \;\; \textbf{(lock consistency)}
\end{aligned}$$

Figure 6.22: HATRMM Consistency

that the execution satisfying the property follows the `co` order we described in Sec. 6.3.3. We also define the **sc consistency** to mean that the set `non_sc` is empty, which indicates as well that the execution satisfying the property has no behavior violating the `sc` atomic constraints described in Sec. 6.3.3. The `well_formed` predicate includes the assumptions that we make for an execution, as well as `lock consistency` are defined to include lock behaviors. We define **HATRMM consistency** by the `sat` predicate. We name a valid execution in HATRMM to be one that is HATRMM consistent.

We now show that HATRMM is equivalent to other models, namely RC11 and IMM [3, 4]. RC11 has four consistency requirements for an execution: `COHERENCE`, `SC`, `ATOMIC` and `NO-THIN-AIR`. IMM also has these four requirements. The `ATOMIC` consistency is for RMW atomics and assumes that an RMW operation is represented by a read immediately followed by a write. HATRMM mainly focuses on the correctness of the compiler steps happening at the imperative language level (e.g. compiler optimizations in LLVM), so we assume RMW atomics behavior atomically; thus, the consistency is satisfied automatically. We use the `COHERENCE` and `SC` consistencies in RC11 for proving HATRMM equivalence. The `NO-THIN-AIR` consistency in RC11 is too strong, and the equivalent proof is based on the `NO-THIN-AIR` consistency in IMM with a strengthened `ctrl` relation (Sec. 6.3.2). Finally, we also replace all the sequenced-before (`sb`) relations appearing in these consistencies with the program order (`po`) relations. We call the target equivalent model as RC11+IMM model, and a valid execution in it RC11+IMM consistent. We present the following two theorems for linking the HATRMM and RC11+IMM models.

Theorem 6.1 (soundness)**.** For any valid execution $\texttt{sat}(Tid, Loc, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$ in HATRMM, it is RC11+IMM consistent.

Theorem 6.2 (completeness)**.** For any valid execution that is RC11+IMM consistent, it is HATRMM consistent.

HATRMM is a hybrid model including a clear subset satisfy the DRF-SC property.

This subset acts as the language primitives for imperative languages. The remaining `rlx` atomics/non-atomics represent machine-level concurrency. We have shown that we have a clear subset in HATRMM satisfying DRF-SC as follows:

Theorem 6.3 (HATRMM DRF-SC)**.** HATRMM executions with memory operations having at least `rel`/`acq` orderings satisfy DRF-SC.

We now show the proofs of these theorems step by step in the following paragraphs.

The Target Model and the First Lemma. Now we have discussed all aspects of HATRMM. We have constructed them in Isabelle/HOL. To accomplish equivalence proofs, we want to link HATRMM with other models. The target model is RC11 [3] with the OUT-OF-THIN-AIR consistency of the IMM model [4]. For all constraints defined in the two models, we need to replace the `sb` relation with the `po` definition from Fig. 6.22. In Fig. 6.23, we show the necessary predicates and constraints in the RC11 and IMM models. The $\sqsupseteq O$ appearing in some elements means that the operation has at least a memory ordering of O. The `Q` represents memory operations: `R` represents reads' `W` represents writes' `F` represents fences; and `E` represents operations and fences.

$$\mathtt{mo}_x(T,\rho) \triangleq \{(s,t) \in T^2 | a < b \wedge \mathtt{same\_loc}(\rho(s),\rho(t)) \wedge \mathtt{is\_write}(\rho(s)) \wedge \mathtt{is\_write}(\rho(s))\}$$

$$\mathtt{mo} \triangleq \bigcup_{x \in \mathtt{Locs}} \mathtt{mo}_x \quad \mathtt{rs} \triangleq [\mathtt{W}];\mathtt{po}|^?_{\mathtt{loc}};[\mathtt{W}^{\sqsupseteq\mathtt{rlx}}];(\mathtt{rf};[\mathtt{RMW}])^* \quad \mathtt{sw} \triangleq [\mathtt{Q}^{\sqsupseteq\mathtt{rel}}];([\mathtt{F}];\mathtt{po})^?;\mathtt{rs};\mathtt{rf};[\mathtt{R}^{\sqsupseteq\mathtt{rlx}}];(\mathtt{po};[\mathtt{F}])^?;[\mathtt{Q}^{\sqsupseteq\mathtt{acq}}]$$
$$\mathtt{rb} \triangleq \mathtt{rf}^{-1};\mathtt{mo} \quad \mathtt{eco} \triangleq (\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb})^+ \quad \mathtt{hb} \triangleq (\mathtt{po} \cup \mathtt{sw})^+$$
$$\mathtt{scb} \triangleq \mathtt{po} \cup \mathtt{po}|_{\neq\mathtt{loc}};\mathtt{hb};\mathtt{po}|_{\neq\mathtt{loc}} \cup \mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb} \quad \mathtt{psc}_{\mathtt{base}} \triangleq ([\mathtt{E}]^{\mathtt{sc}} \cup [\mathtt{F}]^{\mathtt{sc}};\mathtt{hb});\mathtt{scb};([\mathtt{E}]^{\mathtt{sc}} \cup \mathtt{hb};[\mathtt{F}]^{\mathtt{sc}})$$
$$\mathtt{psc}_{\mathtt{F}} \triangleq [\mathtt{F}]^{\mathtt{sc}};(\mathtt{hb} \cup \mathtt{hb};\mathtt{eco};\mathtt{hb});[\mathtt{F}]^{\mathtt{sc}} \quad \mathtt{psc} \triangleq \mathtt{psc}_{\mathtt{base}} \cup \mathtt{psc}_{\mathtt{F}} \quad \mathtt{detour} \triangleq ((\mathtt{co} \setminus \mathtt{po});(\mathtt{rf} \setminus \mathtt{po})) \cap \mathtt{po}$$

Figure 6.23: Parts of the Relations in RC11/IMM/SRA

In Figure 6.23, we first define the modification order at a specific location x as $\mathtt{mo}_x$, which respects the modification order definitions in RC11/IMM. Based on `mo`, the first lemma we want to prove is that HATRMM satisfies the modification order printed in the C++11 memory model documentation [156]. HATRMM clearly satisfies the property because of the requirement of coherence order ($\mathtt{non\_co} \equiv \emptyset$). We list the lemma and proof below:

Lemma 6.1. For any valid execution $\mathtt{sat}(Tid, Loc, Key, T, \rho, \mathtt{sbs}, \mathtt{dds}, \mathtt{rf})$ in HATRMM, for any location $x \in Loc$, $\{s, t, s_1, t_1\} \subseteq T$, $(s,t) \in \mathtt{rf}|_x$, $(s_1, t_1) \in \mathtt{rf}|_x$, t and t_1 having events from the same thread, $t < t_1$, then $s = s_1$ or $(s, t_1) \in \mathtt{mo}_x(T, \rho)$.

Proof. We show here the proof by contradiction. Essentially, the only possible case to violate the lemma is to have two write-read pairs in `rf` as (s,t) and (s', t'), such that the reads in t

and t' are from the same thread ($\texttt{same\_thread}(\rho(t), \rho(t'))$), t and t' access the same location $\texttt{same\_loc}(\rho(t), \rho(t'))$, $s < s'$ but $t > t'$. Let's assume in the above case that $t > t'$. Now, t happens-in-time later than t', but s happens-in-time earlier than s'. Since the reads at t and t' are in the same thread, the write in the (s', t') edge, which is s', has a known-fact edge with the read t according to the definition of kf in Sec. 6.3.3. Thus, we can find the write at s to be in the non-immediate extended known-fact set of t, as $(s, t) \in (\texttt{kf}(T, \rho, \texttt{rf}) \uparrow \texttt{mo}(T, \rho))|_{\neq \texttt{imm}}$. Thus, the (s, t) edge violates the co order (because non_co is not empty) and the execution is not a valid one. Hence, the lemma is valid.

QED.

The Partial DRF-SC Theorem. Here we focus on the Partial DRF-SC Theorem 6.3. The content of the theorem states that any executions containing no rlx atomics (also no non-atomics that are set up to satisfy the same constraints of rlx ordering) satisfies the DRF-SC property. The actual proof of the theorem is based on the fact that RC11 satisfies the DRF-SC property. If we prove that the subset of HATRMM (containing executions without rlx atomics) satisfies the constraints in RC11, then the subset also satisfies the DRF-SC property. Essentially, the proof is to show that the HATRMM subset is also an RC11 model by proving that it satisfies all constraints in RC11. Here we show the core part of the proof. We show that the HATRMM subset satisfies the SRA-coherence property in the SRA model [110], which is a subset of the RC11 model. The SRA model also satisfies the DRF-SC property. In other words, if we can show that the HATRMM subset satisfies the SRA-coherence property, then the subset is also an SRA model and it also satisfies the DRF-SC property. Before we show the main theory, we first show the useful lemma below:

Lemma 6.2. If $<$ is a linear total order, $R \cup <$ and $R' \cup <$ are irreflexive, and all elements in $<$ cover the elements in R and R'; then $R \cup R'$ is acyclic.

Proof. Since $R \cup <$ is irreflexive, it means that for every $(s, t) \in R$, $s < t$. The same fact holds for every pair $(s', t') \in R'$. $R \cup R'$ being acyclic means that $(R \cup R')^+$ is irreflexive. This means that there exists a path $s_1, ..., s_n$, such that (s_1, s_2),...,(s_{n-1}, s_n) and (s_n, s_1) are all in $(R \cup R')^+$. These pairs all appear in either R or R'. Obviously, (s_1, s_2),...,(s_{n-1}, s_n) are all in either R or R', so that we have the following cases: $s_1 < s_2$,...,$s_{n-1} < s_n$. Since $<$ is a linear total order, through the above chain we can conclude that $s_1 < s_n$. Without losing generality, let's assume that $(s_n, s_1) \in R$, so $s_n < s_1$. This contradicts the fact that $s_1 < s_n$. Thus, $R \cup R'$ is acyclic.

QED.

The SRA-coherence property is defined as $\bigcup_{tid \in \texttt{Tid}} \texttt{sbs}(tid) \cup \texttt{mo} \cup \texttt{rf}$ is acyclic. Let's define a syntactic sugar for $\bigcup_{tid \in \texttt{Tid}} \texttt{sbs}(tid)$ as $\overline{\texttt{sbs}}$ We state our main theory to link HATRMM with SRA below:

Theorem 6.4. For any valid execution $\texttt{sat}(Tid, Loc, Key, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$ in HATRMM, and every memory operation action in ρ has at least `acq` or `rel` orderings, then $\overline{\texttt{sbs}} \cup \texttt{mo} \cup \texttt{rf}$ is acyclic.

Proof. From the validity execution property in HATRMM, it is obvious that the union of the `mo` relation and the natural number order $<$ on T is irreflexive. This is true for `rf` as well, as $\texttt{rf} \cup <$ is irreflexive. From Lemma 6.3.4, we infer that $\texttt{mo} \cup \texttt{rf}$ is acyclic in HATRMM. The only possibility of violating the acyclicity of $\overline{\texttt{sbs}} \cup \texttt{mo} \cup \texttt{rf}$ in the conclusion of the theorem is to have a series of time points $s_1, ..., s_n$, such that $(s_1, s_2) \in \texttt{mo} \cup \texttt{rf}$,...,$(s_{n-1}, s_n) \in \texttt{mo} \cup \texttt{rf}$, but $(s_n, s_1) \in \overline{\texttt{sbs}}$.

We can get several facts from the above possibility. First, since $(s_1, s_2) \in \texttt{mo} \cup \texttt{rf}$,...,$(s_{n-1}, s_n) \in \texttt{mo} \cup \texttt{rf}$, then $s_1 < s_2$,...,$s_{n-1} < s_n$. Thus, $s_1 < s_n$. Second, if $(s_n, s_1) \in \overline{\texttt{sbs}}$, since $\overline{\texttt{sbs}}$ is just a collection of all single-threaded sequenced-before relations, then the events in s_n and s_1 are from the same thread as $\texttt{same\_thread}(\rho(s_1), \rho(s_n))$. Now the proof becomes to show that the assumptions $(s_n, s_1) \in \overline{\texttt{sbs}}$ and $s_1 < s_n$ lead to a contradiction.

We do a case analysis of the different possibilities of events at s_1 and s_n. It is obvious that the events at s_1 and s_n cannot be `sc` atomics or fences. If $\rho(s_1)$ and $\rho(s_n)$ are `sc` atomics, then there is a `po` relation between the two to require that any sequenced-before `sc` atomics never happen-in-time after a sequenced-after event, while any sequenced-after `sc` atomics never happens-in-time before a sequenced-before event. Thus, if the two events at s_1 and s_n are `sc` events, then it is a contradiction to have $(s_n, s_1) \in \overline{\texttt{sbs}}$ and $s_1 < s_n$. Although there are other interesting cases, the four major cases to consider are when $\rho(s_1)$ and $\rho(s_n)$ are `acq` reads and `rel` writes. Let's analyze them one by one:

- The two events at s_1 and s_n are `rel` writes. Since $(s_n, s_1) \in \overline{\texttt{sbs}}$, according to the `relw` definition in HATRMM, we have a `rel` edge from s_n to s_1. Thus, an execution must make s_n happen-in-time earlier than s_1. This is contradicted by the fact that $s_1 < s_n$.

- The two events at s_1 and s_n are `acq` reads. Since $(s_n, s_1) \in \overline{\texttt{sbs}}$, according to the `acqr` definition in HATRMM, we have an `acq` edge from s_n to s_1. Thus, an execution must make s_n happen-in-time earlier than s_1. This is contradicted by the fact that $s_1 < s_n$.

- $\rho(s_1)$ is a `rel` write and $\rho(s_n)$ is an `acq` read. Since $(s_n, s_1) \in \overline{\texttt{sbs}}$, according to the `acqr` and `relw` definitions in HATRMM, we have an `acq` edge from s_n to s_1 and a `rel` edge from s_n to s_1. Thus, an execution must make s_n happen-in-time earlier than s_1. This is contradicted by the fact that $s_1 < s_n$.

- $\rho(s_1)$ is an `acq` read and $\rho(s_n)$ is a `rel` write. Recall that $(s_1, s_2), ..., (s_{n-1}, s_n)$ are all in $\texttt{mo} \cup \texttt{rf}$. $\rho(s_1)$ is a read means that s_1 does not join any relation in `mo` and it might be the second event in a pair (s_t, s_1) in `rf`. Let's assume that s_t is a pair among these sequence $(s_1, s_2), ..., (s_{n-1}, s_n)$. First, s_t cannot be s_n, since if s_t is s_n, then $(s_n, s_1) \in \texttt{rf}$ so that $s_n < s_1$, which is contradicted by the fact that $s_1 < s_n$. Since we also assume that $s_1 < s_n$ here, it means that the read at s_1 happens-in-time before the write at s_n. We show that it is impossible to have a relation in $\texttt{mo} \cup \texttt{rf}$; between the events at s_1 and s_n, because if there is a relation, there will be a chain of time points $s_n, s_{n+1}, ..., s_t$, such that $(s_n, s_{n+1}) \in \texttt{mo}$,...,$(s_{t-1}, s_t) \in \texttt{mo}$ and $(s_t, s_1) \in \texttt{rf}$. However, in this case, we can see that $s_n < s_{n+1}$,...,$s_{t-1} < s_t$ and $s_t < s_1$. Thus, $s_n < s_1$ is contradicted by the fact that $s_1 < s_n$.

Hence, based on the case analysis above, we have shown that $\overline{\texttt{sbs}} \cup \texttt{mo} \cup \texttt{rf}$ is acyclic in the subset of HATRMM without `rlx` atomics.

QED.

The Soundness of HATRMM with Respect to RC11+IMM. Here we show part of the soundness proof as Theorem 6.1. The main idea is to show that any valid HATRMM execution satisfies the constraints in RC11+IMM (with the replacement of `sb` by `po`). These two models both contain four major consistency constraints for a valid execution to satisfy. They are COHERENCE, ATOMICITY, SC and NO-THIN-AIR. The ATOMICITY constraint makes sure that all `RMWs` in an execution are atomic, since the two models assume that `RMWs` are treated as reads following by writes. In the current version of HATRMM, we assume that `RMWs` are treated as atomic operations to begin with, so HATRMM automatically fulfills this constraint. The SC constraint defines the behaviors of `sc` atomics, and it is very complicated. We omit the proof here, but it can be found in total in the Isabelle implementation. Here we mainly focus on showing that valid executions satisfy the COHERENCE and NO-THIN-AIR constraints.

First we deal with the NO-THIN-AIR constraint, which is defined in RC11 as $\texttt{sb} \cup \texttt{rf}$ being acyclic. In IMM, it is rewritten as the acyclicity of the union, named `ar`, of many different dependencies, including the `detour`, `psc`, and `rf`. The soundness proof of HATRMM takes IMM's NO-THIN-AIR constraint by making the `ctrl` stronger, as described in Sec. 6.3.2. Essentially, the `ar` relation (with the stronger `ctrl`) can easily be translated into the formula

$\texttt{po} \cup \texttt{rf}$. Hence, the NO-THIN-AIR constraint for the soundness proof of HATRMM is as follows:

Theorem 6.5. For any valid execution $\texttt{sat}(Tid, Loc, Key, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$ in HATRMM, $\texttt{po} \cup \texttt{rf}$ is acyclic.

Proof. Since $\texttt{po}$ and $\texttt{rf}$ satisfy the property that $\texttt{po} \cup <$ and $\texttt{rf} \cup <$ are irreflexive, $\texttt{po} \cup \texttt{rf}$ is acyclic by Lemma 6.2.

QED.

Now, we look at the COHERENCE constraint. In RC11, it is defined as $\texttt{hb}; \texttt{eco}^?$ ($\texttt{eco}$ and $\texttt{hb}$ in Fig.6.23). The extended-coherence order ($\texttt{eco}$) is the transitive closure of the union of $\texttt{rf}$, $\texttt{mo}$ and reads-before ($\texttt{rb}$ in Fig.6.23). The happens-before ($\texttt{hb}$) is the transitive closure of the $\texttt{po}$ and synchronized-with ($\texttt{sw}$) sets. Essentially, the COHERENCE constraint represents the coherence order in HATRMM (Sec. 6.3.3). We show the theorem as follows:

Theorem 6.6. (RC11 COHERENCE constraint for HATRMM consistency). For a valid execution $\texttt{sat}(Tid, Loc, Key, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$ in HATRMM, $\texttt{hb}; \texttt{eco}^?$ is irreflexive.

Proof. First, $\texttt{hb}$ is irreflexive because $\texttt{hb}$ is defined as $(\texttt{po} \cup \texttt{sw})^+$; and $\texttt{sw}$ is essentially a part of $\texttt{rf}$ (if we omit fences). So we have shown that $\texttt{po} \cup \texttt{rf}$ is acyclic in Theorem 6.5. Thus, $\texttt{po} \cup \texttt{sw}$ is acyclic and $\texttt{hb}$ if irreflexive.

The other part of the proof is based on a case analysis of the $\texttt{hb}$ edges. Let's assume an event happening at s in the middle of a sequence of $\texttt{hb}$ edges, such that $(s_1, s_2), ..., (s_i, s), ..., (s_{j-1}, sj)$ are all in $\texttt{hb}$. Obviously, we can also say that $s_1 < s_2$,...,$s_i < s$. Another way to understand this is that for s, if we cannot make a pair (s, t) in $\texttt{eco}$ such that we cannot link $\texttt{hb}$ with $\texttt{eco}$ through (s, t), then we are done because of the acyclicity of the $\texttt{hb}$. The only possibility of a contradiction with the conclusion of the proof is that for a pair (s, t) in $\texttt{eco}$, there is also an edge (t, s_k) in $\texttt{eco}$, and $(s_k, s) \in \texttt{hb}$.

Let's do a little analysis of the $\texttt{eco}$ relation. It is defined as $(\texttt{rf} \cup \texttt{mo} \cup \texttt{rb})^+$. Obviously, every pair (t, t') in $\texttt{eco}$ accesses the same location. If s is a read, then the only possibility for a pair (s, t) to exist in $\texttt{eco}$ is if $(s, t) \in \texttt{rb}$. In this case, t is a write and the read at s and write at t access the same location. Actually, $\texttt{rb}$ is not a relation that matches the flow of the order $<$, so t might happen-in-time earlier than s. It is possible that there is a path $(t, t_1), ..., (t_{n-1}, t_n), (t_n, s_k)$ in $\texttt{eco}$ and $(s_k, s_1) \in \texttt{hb}$.

We now show that it is impossible for the path to exist. We first show that every time point $t_1, ..., t_n$ and s_k happen-in-time after t. Let (t_l, t_m) be one of the pairs in the path $(t, t_1), ..., (t_{n-1}, t_n), (t_n, s_k)$. If (t_l, t_m) is from $\texttt{mo}$, then t_m happens-in-time later than t_l; if it is from `rf`, t_m also happens-in-time later than t_l. If (t_l, t_m) is from `rb`, then t_m might happen-in-time before t_l; but it must happen-in-time after t, because the event at t is a write, and that at t_l is a read, while that at t_m is a write. In the path $(t, t_1), ..., (t_{n-1}, t_n), (t_n, s_k)$, the prefix before (t_l, t_m) must be a `mo` pair, and there must also be an `rf` pair right before (t_l, t_m); so t happens-in-time before the write (let's say it is t'_l) that the read at t_l loads from. The definition of `rb` is for the read t_l to look for the write happening-in-time after t'_l, so t is never a candidate for t_m. Thus, t must happen-in-time before any event in the path $(t, t_1), ..., (t_{n-1}, t_n), (t_n, s_k)$ except t itself.

We now show that (s_k, s) never appears in `hb`. Let's assume that (s_k, s) does appear in `hb`. First, let's say that the event at s_k is a write. In this case, Since t, s_k and s access the same location, $(t, s_k) \in$ `mo`. Recall that s is a read, and (s, t) is in `rb`. Thus, the read at s loads the write (Let's assume that it is t') even happening-in-time before t and $(t', t) \in$ `mo`, so $(t', s_k) \in$ `mo`. With the fact that $(s_k, s) \in$ `hb`, according to the `non_co` set definition, (t', s) is an element in the set, and so the execution does not follow the HATRMM `co` order. Now let's say that the event at s_k is a read. Notice that s_k is also an element in the path $(t, t_1), ..., (t_{n-1}, t_n), (t_n, s_k)$. Thus, $(t_n, s_k) \in$ `rf`, and the t_n event is a write accessing the same location as the read in s. If (s_k, s) is in `hb`, then t_n happens-in-time eariler than s, and t happens-in-time earlier than t_n, so $(t, t_n) \in$ `mo` and $(t', t_n) \in$ `mo`. Since (s_k, s) is in `hb`, according to the known-fact order definition in Sec. 6.3.3, (t_n, s) is in known-fact order. Hence, (t', s) is in the `non_co` set, and the execution does not follow the HATRMM `co` order.

Next let's assume that the event at s is a write. In this case, the pair (s, t) is either in `rf` or `mo`. In both cases, t happens-in-time later than s. If t is a write, then we have seen that in the path $(t, t_1), ..., (t_{n-1}, t_n), (t_n, s_k)$, all elements happen-in-time later than t except t itself. Hence, s_k happens-in-time after s, and it is impossible to form an edge in `hb` from s_k to s according to Theorem 6.5. Finally, let's say that t is a read, then (s, t) is in `rf` and t happens-in-time after s. The same argument showing that t must happen-in-time before all elements in the path $(t, t_1), ..., (t_{n-1}, t_n), (t_n, s_k)$ also supports that s must happen-in-time before all elements in the path $(s, t), (t, t_1), ..., (t_{n-1}, t_n), (t_n, s_k)$. Therefore, (s_k, s) is never in `hb`. Hence, we have shown that consistent executions in HATRMM satisfy the RC11 COHERENCE constraint.

QED.

The Completeness of HATRMM with Respect to RC11+IMM. The final theorem in this section is Theorem 6.2, the relative completeness theorem of HATRMM. We need some modifications in RC11+IMM to make the completeness theorem go through. The first step is to use the memory events in Batty *et al.*'s model [2], because RC11/IMM is implemented in Coq, but we need an Isabelle version. For this, we choose the memory event structure in Batty *et al.*'s model. Second, a candidate execution in IMM based on Batty *et al.*'s event set syntax is a tuple such as ($\texttt{Acts}_r$, Tid_r, Loc_r, $\texttt{sbs}_r$, $\texttt{dds}_r$, $\texttt{rf}_r$, $\texttt{mo}_r$, $\texttt{sc}_r$). Tid_r, Loc_r, $\texttt{sbs}_r$, $\texttt{dds}_r$, $\texttt{rf}_r$ and $\texttt{mo}_r$ are similar to the entities without the subscript r. $\texttt{Acts}_r$ is a set of memory events in IMM, and $\texttt{sc}_r$ is a relation defining the `sc` atomics with respect to other events happening in an execution. Since RC11/IMM does not have the concept of time points, its candidate executions need these relations to describe the execution behaviors. The problem is that the definitions of $\texttt{mo}_r$, $\texttt{sc}_r$ and $\texttt{dds}_r$ can be absolutely anything. Even though the implementation of RC11 has well-formed checks, it is not enough. For example, there is no rule to prevent $\texttt{mo}_r$ from being defined as an empty set in a candidate execution in RC11, while the execution can still be RC11-consistent. To prevent this, we require all events in $\texttt{Acts}_r$ to appear once in $\texttt{sbs}_r$, all write events in $\texttt{Acts}_r$ to appear once in $\texttt{mo}_r$, all `seq` events to appear once in $\texttt{sc}_r$, and the translation of $\texttt{dds}_r$ into HATRMM to be the `dds` relation. We describe the above properties as well-formedness in Theorem 6.7. The `etrans` function translates an RC11+IMM execution into a set of HATRMM ones. For any execution in RC11+IMM, `etrans` generates a set of executions in HATRMM, since IMM is descriptive and every valid execution defined in IMM describes a group of valid executions. It is hard to make an execution in IMM strictly unique. We show Theorem 6.7 below. The proof is basically the reverse of the HATRMM soundness proof, except that we now prove the theorem by doing induction on the number of events in an execution, i.e., the maximum number in the time point set T of the execution. The details of the proof are found in the Isabelle implementation.

Theorem 6.7. For a valid and well-formed execution ($\texttt{Acts}_r$, Tid_r, Loc_r, $\texttt{sbs}_r$, $\texttt{dds}_r$, $\texttt{rf}_r$, $\texttt{mo}_r$, $\texttt{sc}_r$) that is IMM-consistent and well-formed, and (Tid, Loc, Key, T, ρ, `sbs`, `dds`, `rf`) $\in$ `etrans`($\texttt{Acts}_r$, Tid_r, Loc_r, $\texttt{sbs}_r$, $\texttt{dds}_r$, $\texttt{rf}_r$, $\texttt{mo}_r$, $\texttt{sc}_r$), then $\texttt{sat}(Tid, Loc, Key, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$.

6.4 EXAMPLE PROGRAM SEMANTICS ADAPTING HATRMM

Here we introduce an example operational semantics that admits HATRMM, i.e. its concurrency behavior is derived by HATRMM. The operational semantics is useful for proving properties such as compiler optimization correctness for a language admitting HATRMM (Chapter 8). We introduce an example operational semantics here. However, we do not use the semantics to prove compiler optimization semantic preservation property since the semantics is too intricate and too hard to be extended to include the semantics for the whole

LLVM IR. We will introduce an abstract machine like operational concurrency model that is substitutable for the one in Sec. 5.3.2 in the next chapter. We also prove that the new operational model is equivalent to the HATRMM with FIFO ordering.

Domain

Time Points: $s, t \in T \subseteq \mathbb{N}$ Registers: $\varphi \subseteq (\mathcal{Var} \to \mathcal{Val})$ Heap Snapshots: $\gamma \subseteq (\mathcal{Loc} \to (T \times \mathcal{Val}))$

Thread-IDs: $tid \in Tid \subseteq \mathcal{Tid}$ Dyanmic Block Number: $\mathcal{Bn} \ni \overline{\pi} \triangleq (N \times N)$

Dyanmic Block Family: $\overline{\Pi} \subseteq Tid \to (N \times N)$ Action-IDs: $\mathcal{Aid} \ni d \triangleq (\mathcal{Bn} \times N)$

Semantic Function Types

Expression Semantics: $\texttt{eval} \subseteq (\mathcal{Exp} \times \varphi) \to \mathcal{Val}$ Inst Semantics: $\psi \subseteq (\mathit{Inst} \times \varphi \times \gamma) \to (\varphi \times \mathcal{Act})$

Termination Semantics: $\eta \subseteq (\mathcal{CInst} \times (\mathcal{Var} \to \mathcal{Val})) \to \mathcal{L}$

Example Instruction Semantics

$\psi(a \texttt{ := } e, \varphi, \gamma) \triangleq (\varphi[a \leftarrow \texttt{eval}(\varphi, e)], \tau)$ Assignment Semantics

$\psi(a \texttt{ :=}_{o_r}\ x, \varphi, \gamma) \triangleq (\varphi[a \leftarrow \texttt{snd}(\gamma(x))], \texttt{R}^x_{\texttt{snd}(\gamma(x)), o_r})$ Read Semantics

$\psi(x \texttt{ :=}_{o_w}\ a, \varphi, \gamma) \triangleq (\varphi, \texttt{W}^x_{\texttt{eval}(\varphi, a), o_w})$ Write Semantics

$\eta(\texttt{if } e \texttt{ then } \pi_1 \texttt{ else } \pi_2, \varphi) \triangleq \texttt{IF } \eta(\varphi, e) = 0 \texttt{ THEN yes ELSE no}$ Binary Branching Semantics

Operational Program Semantics

Program Order Family: $\texttt{pos} \subseteq Tid \to \texttt{po}$ Current Program Pointer: $\theta \subseteq \mathit{In\ Set} \times \mathit{In\ Set}$

Registers Family: $\Phi \subseteq Tid \to \varphi$ Heap Family: $\Gamma \subseteq Tid \to \gamma$

Program Pointer Family: $\Theta \subseteq Tid \to \theta$

Single Step Transition Function:

$\texttt{trans} \subseteq (C, Tid, \overline{\pi}, \texttt{po}, \texttt{sb}, \texttt{dd}, T, \rho, \varphi, \Gamma, \theta) \to (\overline{\pi}, \texttt{po}, \texttt{sb}, \texttt{dd}, T, \rho, \varphi, \Gamma, \theta, \texttt{rf})$

State Environment: $\omega \triangleq (Tid\ \mathcal{Set}, \texttt{pos}, \texttt{sbs}, \texttt{dds}, \overline{\Pi}, \Phi, \Gamma, \Theta, T, \rho, \texttt{rf})$

State: (μ, ω) Transition System: $(\mu, \omega) \xrightarrow{ev} (\mu, \omega)$

One Example Transition Rule:

$$\frac{\begin{array}{l} tid \in Tid \wedge \texttt{pos}' = \texttt{pos}[tid \mapsto \texttt{po}'] \wedge \texttt{sbs}' = \texttt{sbs}[tid \mapsto \texttt{sb}'] \wedge \texttt{dds}' = \texttt{dds}[tid \mapsto \texttt{dd}'] \\ \wedge \overline{\Pi}' = \overline{\Pi}[tid \mapsto \overline{\pi}] \wedge \Phi' = \Phi[tid \mapsto \varphi'] \wedge \Theta' = \Theta[tid \mapsto \theta'] \wedge \texttt{sat}(Tid, T', \rho, \texttt{sbs}', \texttt{dds}', \texttt{rf} \cup \texttt{rf'}) \\ \wedge \texttt{trans}(\mu(tid), tid, \overline{\Pi}(tid), \texttt{pos}(tid), \texttt{sbs}(tid), \texttt{dds}(tid), T, \rho, \Phi(tid), \Gamma, \Theta(tid)) \\ \quad = (\overline{\pi}', \texttt{po}', \texttt{sb}', \texttt{dd}', T', \rho', \varphi', \Gamma', \theta', \texttt{rf'}) \end{array}}{\begin{array}{l} \big(\mu, Tid, \texttt{pos}, \texttt{sbs}, \texttt{dds}, \overline{\Pi}, \Phi, \Theta, \Gamma, T, \rho, \texttt{rf}\big) \\ \qquad \xrightarrow{\rho'(\texttt{max}(T'))} \big(\mu, Tid, \texttt{pos}', \texttt{sbs}', \texttt{dds}', \overline{\Pi}', \Phi', \Theta', \Gamma', T', \rho', \texttt{rf} \cup \texttt{rf'}\big) \end{array}}$$

Figure 6.24: Example Program Semantics Admitting HATRMM

Here we discuss the operational program semantics for the language in Fig. 6.4 as an example of binding HATRMM with an operational semantics. The semantics is a bridge connecting single instruction semantics and a multi-threaded weak memory model. Fig. 6.24 provides a taste of the instruction semantics, and operational semantics based on the language in Fig. 6.4 and HATRMM. In Fig. 6.24, T is a downward closed natural number set of **time points** without `0` in HATRMM. We implement a heap snapshot (γ) as a function from a location to a pair: the pair is the time point of the most recent write to the location and the value in the location. In Fig. 6.4, we introduced the concept of basic blocks, nodes are numbers identifying basic blocks. We use a pair of natural numbers as a **dynamic basic block number** ($\mathcal{Bn}$); the pair uniquely identify an executing basic block in a thread during an execution. For a program $\mu : Tid \to \mathcal{CFG}$, we have a family of dynamic basic block numbers ($\overline{\Pi}$), one for each thread. In Fig. 6.4, we introduced an instruction number for each instruction in a basic block; it is represented by a natural number. Here, we name an

action-ID as the combination of a dynamic block number $\overline{\pi}$ and an instruction number in the basic block indexed by the second argument of $\overline{\pi}$. Hence, it is clear that an **action-ID** can uniquely define an executing instruction in a thread.

At the instruction level, there are three semantic functions. The `eval` function (Fig. 6.24) is for evaluating an expression ($\mathcal{Exp}$) in Fig. 6.4. It is a straight evaluation of each term of the expression, so we omit the detailed implementation here. The function ψ implements the semantics of an instruction ($\mathcal{Inst}$ in Fig. 6.4). It takes an instruction, a register map (φ), and a heap snapshot (γ), and produces a resulted register map and a memory action ($\mathcal{Act}$) indicating the type of memory communication the instruction could bring. We show three cases for ψ in Fig. 6.24: the case when a normal assignment happens and ψ returns a τ action, the case when a read happens and ψ returns a read action, and the case when a write happens and ψ returns a write action. The function η implements the semantics of terminations. It takes a termination and registers, and returns an edge label. In Fig. 6.24, we show the semantics of a binary branching instruction.

In Fig. 6.4, we also introduced actions ($\mathcal{Act}$). A memory instruction produces a read/write action, while other instructions/terminations produce a τ action. Here we combine an action, thread-ID and action-ID, making a memory event ($\mathcal{Ev}$). In the model in Fig. 6.24, for an execution, we assume that a family (`sbs`) of sequenced-before relations (`sb`) is given, one `sb` for each thread; and a family (`dds`) of data dependency relations (`dd`) is also given, one `dd` for each thread. A straight-forward algorithm for generating a sequenced-before relation for executing a CFG can be taken from the program text order of the CFG; also, a data dependency relation for a CFG execution can be produced by the traditional data-flow, alias, and control dependency analysis algorithms. Here we omit the details of these algorithms.

The operational transition semantics in Fig. 6.24 is a combination of the instruction level semantics and memory concurrency model. It is represented as a labeled transition system whose states are pairs of programs (μ) and the state environment (ω), and whose labels are memory events. A state environment is a long tuple of a set of thread-IDs (Tid), a program order family (`pos`, one for each thread), a sequenced-before relation family (`sbs`), a data dependency family (`dds`), a current dynamic block number family ($\overline{\Pi}$), a registers family (Φ), a heap snapshot family (Γ) representing different views of the threads of the main memory, a program pointer family (Θ) representing the current executing instruction of each thread, a time point set (T), a ρ mapping, and a reads-from relation (`rf`). We show the top-most rule of the transition system in Fig. 6.24. This rule selects a thread tid, applies the one-step transition function `trans` to the state environment of tid, checks the result of the one-step transition to see if the accumulated result satisfies the predicate of the memory model (`sat`), and then moves forward to a new step via the memory event label $\rho'(\texttt{max}(T'))$.

The function `max` produces the maximum number in T'. We can retrieve the memory event by the `max` function because the `trans` function always creates a map entry in ρ from the maximum time point plus `1` to the current memory event. The detailed implementation of the `trans` function is found in the Sec. 6.4. It needs to finish several tasks as a one step evaluation for a thread tid with a CFG C. First, if its program pointer $\Theta(tid)$ points to the end of a basic block (no instructions left for execution), it selects a new basic block according to the edge information in C (applying function η to it with registers ($\Phi(tid)$) to get the edge label), and assigns a new dynamic basic block number with a new program pointer pointing to the top of the new block. In this case, `trans` also adds new relations of program order, sequenced-before, and data dependency to the existing relation sets inside the new basic block. Second, if $\Theta(tid)$ indicates that there are instructions in the basic block waiting for execution, an instruction is non-deterministically selected for execution (applying function ψ to it with registers ($\Phi(tid)$) and heap snapshot ($\Gamma(tid)$)) if the instruction satisfies the program order relation on the basic block. Third, for a step, `trans` also picks a new time point (the maximum number of the time point set T plus `1`) to add to the set T, and assigns the new time point to a new memory event. The creation of the event is to combine the thread-ID tid, a newly generated action-ID (the action-ID is calculated by combining the dynamic block number with the instruction number), and a memory action calculated from the function ψ (if the instruction is a termination, we assume that the action is τ). Fourth, `trans` also generates a new `rf` pair if the action is a read, and modifies the memory snapshot by inserting the current time point and write value if the action is a write.

Here, we define the semantics for a program based on the program syntax and instruction semantics in Sec. 6.2.1. The program semantics is building an abstract machine executing single threaded instructions through a family of conceptual CPUs (one for each thread), and multi-threaded instructions through a conceptual memory machine. We assume that a CPU execute a basic block of instructions at a time, and any one of executing basic blocks in an program execution, named **dynamic basic block**, can be identified as a unique number $m \in \mathbb{N}$ in a program execution. We use a pair of a unique number and a basic block number of a block $(m, \pi) \in \mathbb{N} \times \mathbb{N}$ to refer to the **dynamic basic block number** (as $\overline{\pi}$) for a dynamic block whose content is the basic block whose node is π in a CFG. Then, a pair of dynamic block number and an instruction number ($\mathbb{N} \times \mathbb{N} \times \mathbb{N}$), named action-ID (as d having type $A = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$), can uniquely identify an executing instruction in a thread in a program execution. In Sec. 6.3.3, we have described a `po` relation (or a family `pos`), we extend the idea to a relation $\overline{\texttt{po}}$ (or a family $\overline{\texttt{pos}}$, one for each thread) describing similar program order relations as `po`. $\overline{\texttt{po}}$ for a thread is a relation on $A \times A$, and it describes the program order relations on different instructions instead of memory events in `po`. Every $\overline{\texttt{po}}$

in $\overline{\texttt{pos}}$ is generated along with program semantics transitions.

$$\texttt{form\_D}(\overline{\pi},\beta) \equiv \{d | \exists e\ i.d = (\overline{\pi}, i) \wedge e = \texttt{ins}(\beta, d)\} \quad \texttt{gen}(\overline{\texttt{po}}, \texttt{D}, W) \equiv \{d \in \texttt{D} | (\neg\exists d' \in W.(d', d) \in \overline{\texttt{po}})\}$$

$$\texttt{(a)} \quad \frac{\begin{array}{l}(N, \pi_0, \lambda, E) = G \wedge \lambda(\pi) = \beta \wedge \texttt{ins}(\beta, d) = e \wedge \texttt{is\_C}(\beta, d) \wedge \pi = \texttt{snd}(\overline{\pi}) \\ \wedge l = \overline{\eta}(e, \phi) \wedge (\pi, l, \pi') \in E \wedge \lambda(\pi') = \beta' \wedge \overline{\pi}' = (\texttt{fst}(\overline{\pi})\texttt{+1}, \pi') \wedge D = \texttt{form\_D}(\overline{\pi}', \beta') \\ \wedge \rho' = \rho[n\texttt{+1} \mapsto (tid, d, \tau)] \wedge \overline{\texttt{po}}' = \texttt{gen\_}\overline{\texttt{po}}(G, \overline{\pi}', \overline{\texttt{po}}, \rho', D, \beta', \varphi) \wedge W = \texttt{gen}(\overline{\texttt{po}}, D, D)\end{array}}{\big(tid, \overline{\pi}, \texttt{G}, \overline{\texttt{po}}, n, \rho, \varphi, \Gamma, (\{d\}, S)\big) \longrightarrow \big(\overline{\pi}', \overline{\texttt{po}}', n\texttt{+1}, \rho', \varphi, \Gamma, (W, \emptyset), \emptyset\big)}$$

$$\texttt{(b)} \quad \frac{\begin{array}{l}(N, \pi_0, \lambda, E) = G \wedge \lambda(\pi) = \beta \wedge \texttt{ins}(\beta, d) = e \wedge \neg\texttt{is\_C}(\beta, d) \wedge \Gamma(tid)|_v = \gamma \\ \wedge \Gamma(tid')|_v = \gamma' \wedge \gamma'(x) = v \wedge (\varphi', \texttt{R}^x_{v,o}) = \psi(e, \varphi, \gamma[x \mapsto v]) \\ \wedge W' = \texttt{gen}(\overline{\texttt{po}}, D - (S \cup \{d\}), W) \wedge \texttt{rf} = \{\texttt{fst}(\Gamma(tid')(x)), n\texttt{+1})\} \wedge \pi = \texttt{snd}(\overline{\pi})\end{array}}{\begin{array}{l}\big(tid, \overline{\pi}, \texttt{G}, \overline{\texttt{po}}, n, \rho, \varphi, \Gamma, (W \cup \{d\}, S)\big) \longrightarrow \\ \qquad \big(\overline{\pi}, \overline{\texttt{po}}, n\texttt{+1}, \rho[n\texttt{+1} \mapsto (tid, d, \texttt{R}^x_{v,o})], \varphi', \Gamma, (W', S \cup \{d\}), \texttt{rf}\big)\end{array}}$$

$$\texttt{(c)} \quad \frac{\begin{array}{l}\texttt{T} = \texttt{gen\_times}(n') \wedge \texttt{sbs} = \texttt{gen\_sb}(\texttt{T}, \texttt{P}, \rho) \wedge \texttt{sat}(\texttt{Tid}, \texttt{T}, \rho, \texttt{sbs}, \texttt{rf} \cup \texttt{rf'}) \wedge tid \in \texttt{TID} \\ \wedge \overline{\texttt{pos}}' = \overline{\texttt{pos}}[tid \mapsto \overline{\texttt{po}}'] \wedge \overline{\Pi}' = \overline{\Pi}[tid \mapsto \overline{\pi}] \wedge \Phi' = \Phi[tid \mapsto \varphi'] \\ \wedge \big(tid, \overline{\Pi}(tid), \texttt{P}(tid), \overline{\texttt{pos}}(tid), n, \rho, \Phi(tid), \Gamma, \Theta(tid)\big) \longrightarrow \big(\overline{\pi}', \overline{\texttt{po}}', n', \rho', \varphi', \Gamma', \theta', \texttt{rf'}\big)\end{array}}{\big(\texttt{TID}, \texttt{P}, \overline{\texttt{pos}}, \overline{\Pi}, \Phi, \Theta, \Gamma, n, \rho, \texttt{rf}\big) \Longrightarrow_{\rho'(n')} \big(\texttt{TID}, \texttt{P}, \overline{\texttt{pos}}', \overline{\Pi}', \Phi', \Theta[tid \mapsto \theta'], \Gamma', n', \rho', \texttt{rf} \cup \texttt{rf'}\big)}$$

Figure 6.25: The Single-Thread Transition Function

The semantics of a program P is defined as a labeled transition semantics as $\sigma \Longrightarrow_a \sigma'$ where σ and σ' are states and a is a memory event acting as the label in a transition. (c) in Fig. 6.25 is the main rule in the transition semantics. A state is defined as a tuple (having type name: *State*) of $(\texttt{TID}, \texttt{P}, \overline{\texttt{pos}}, \overline{\Pi}, \Phi, \Theta, \Gamma, n, \rho, \texttt{rf})$, where TID is a set of thread-IDs (type *TID*), P is a program as the μ function in Sec. 6.2.1, $\overline{\texttt{pos}}$ is a family of $\overline{\texttt{po}}$ relations (type $TID \to (A \times A)$), $\overline{\Pi}$ is a family of dynamic block numbers (type $TID \to \mathbb{N} \times \mathbb{N}$), each of which acts as the dynamic block number counter for each thread, Φ is a family of registers, Θ is a family of program counters (type $TID \to (A\ \mathsf{set}) \times (A\ \mathsf{set})$) that point out the next possible instructions to execute for a thread, and also have a set of finishing executing instructions, Γ is a family (type $TID \to (loc \to T \times val)$) of observable memory snapshots, n is the time point counter representing the maximum time point value at a state, ρ is a mapping from time points to memory events (type $\mathbb{N} \to \Upsilon$), and rf is the reads-from relation (type $T \times T$) under construction along with the transitions. In the transition $\sigma \Longrightarrow_a \sigma'$, the transition state is (a, σ') (type $\Upsilon \times State$). A program execution is a sequence of transition states: $(a, \sigma'), ..., (a_n, \sigma'_n), ...$, generated by the transitions: $\sigma \Longrightarrow_a \sigma' \Longrightarrow ... \Longrightarrow_{a_n} \sigma'_n \Longrightarrow ....$ Γ is the observable memory snapshot for each thread, each of which (γ) represents the knowledge of the thread about a memory location: the value of the location as well as the time point when the latest write in the thread wrote to the location.

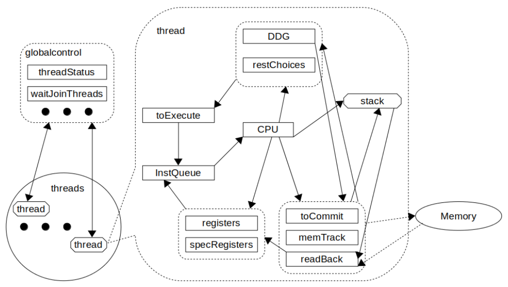
Rules (a) and (b) (Fig. 6.25) are sample rules for the transition relation $\longrightarrow$ that connects between the transition $\Longrightarrow$ and the single-instruction semantics defined in Sec. 6.2.1. It transitions from a tuple of a thread-ID tid, a current dynamic block number $\overline{\pi}$, a CFG G, a program order $\overline{\texttt{po}}$, a time point n, a mapping ρ, a register ϕ, a family of memory snapshots Γ, and a program counter θ; to another tuple of a possible new dynamic block number $\overline{\pi}'$, an updated program order $\overline{\texttt{po}}'$, a new time point n', a new mapping ρ', a possible new register ϕ', an updated family of memory snapshots Γ', an updated program counter θ, and a set of reads-from relations `rf` containing a possible write-read pair generated by a load instruction. In a thread tid, $\longrightarrow$ selects one of the possible next instructions in θ to execute. The `(a)` rule deals with the transition after executing a branching state at the end of a basic block, while rule `(b)` deals with the case of a load instruction. In these rules, `ins` is a function producing the instruction expression from a basic block and an action-ID. $\gamma|_v$ means to form a new mapping by getting rid of the time point in the value pair $T \times val$. The function $\texttt{gen\_}\overline{\texttt{po}}$ takes the existing $\overline{\texttt{po}}$ and a basic block and generates a new $\overline{\texttt{po}}'$ containing all relations in the `po` relation, all program order relations between instructions in the basic block, and the program order relations between the old-instructions in $\overline{\texttt{po}}$ and the instruction in the new basic block. $\texttt{gen\_}\overline{\texttt{po}}$ happens once when a new dynamic basic block is generated. The function `gen_sb` generates a family of `sb` relations based on the program information, while `gen_po` generates a family of `po` relations.

Chapter 7: THE LLVM OPERATIONAL MODEL

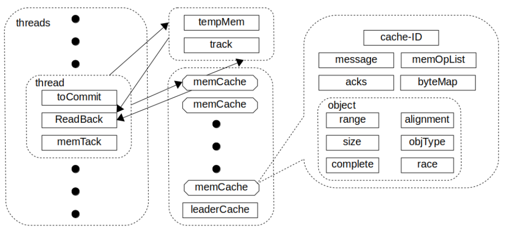
As we have discussed, to define the complete behavior of the LLVM IR language, we are basically simulating a conceptual virtual machine that runs the LLVM IR codes. In Figure 7.1, the conceptual machine includes execution machines (models) that select a **K-LLVM** BAST instruction to execute, returning 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.

7.1 THE EXECUTION MODEL

The **K-LLVM** execution model in Figure 7.1a directs how a single operation is selected and executed and how the result is returned. All of the rectangles in Figure 7.1a 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 7.2. 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



(a) Component Relation in Execution Model



(b) Component Relation in Memory Model

Figure 7.1: Relations Among Model Components

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 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 7.2, the job of the `CPU` cell is to push the instruction (*In*) to the `k` cell for evaluation. In Section 4.2, we show that every instruction is either an `assign` term or a `noAssign` depending if the instruction returns value. the

RULE INSTQUEUE-OUT

$$\left\langle Q : Set \; \frac{\texttt{SetItem(dynInstInfo(}\; B \;,\; Num \;,\; In \;,\; T \;\texttt{))}}{\texttt{.Set}} \right\rangle_{\mathsf{instQueue}}$$

$$\left\langle \frac{\texttt{.K}}{\texttt{dynInstInfo(}\; B \;,\; Num \;,\; In \;,\; T \;\texttt{)}} \right\rangle_{\mathsf{CPU}}$$

REQUIRES $\texttt{isAvailable(}\; In \;\texttt{)} \wedge \neg \;\texttt{isHeavyInst(}\; T \;\texttt{)}$

RULE CPU-OUT

$$\left\langle \frac{\texttt{dynInstInfo(}\; B \;,\; Num \;,\; In \;,\; T \;\texttt{)}}{\texttt{dynInstHold(}\; B \;,\; Num \;,\; T \;\texttt{)}} \right\rangle_{\mathsf{CPU}} \; \left\langle \frac{\texttt{.K}}{In} \curvearrowright K : K \right\rangle_{\mathsf{k}}$$

REQUIRES $\neg \;\texttt{isMemOp(}\; T \;\texttt{)}$

RULE CPU-ASSIGN

$$\left\langle \frac{\texttt{assign(}\; X \;,\; V : KResult \;\texttt{)}}{\texttt{.K}} \cdots \right\rangle_{\mathsf{k}} \; \left\langle \frac{M : Map}{M \;\texttt{[}\; V \;\texttt{/}\; X \;\texttt{]}} \right\rangle_{\mathsf{registers}}$$

$$\left\langle \frac{\texttt{dynInstHold(}\; B \;,\; Num \;,\; T \;\texttt{)}}{.K} \right\rangle_{\mathsf{CPU}} \; \langle B \rangle_{\mathsf{currBlock}}$$

$$\left\langle \frac{Tr}{\texttt{updateVarTree(}\; Tr \;,\; V \;,\; Tr \;\texttt{)}} \right\rangle_{\mathsf{specTree}}$$

$$\left\langle \frac{Q}{\texttt{updateVarSet(}\; Q \;,\; V \;,\; Tr \;\texttt{)}} \right\rangle_{\mathsf{instQueue}}$$

RULE INSTQUEUE-HEAVY-OUT

$$\left\langle Q : Set \; \frac{\texttt{SetItem(dynInstInfo(}\; B \;,\; Num \;,\; In \;,\; T \;\texttt{))}}{\texttt{.Set}} \right\rangle_{\mathsf{instQueue}}$$

$$\left\langle \frac{\texttt{.K}}{\texttt{dynInstInfo(}\; B \;,\; Num \;,\; In \;,\; T \;\texttt{)}} \right\rangle_{\mathsf{CPU}}$$

REQUIRES $\texttt{isSmallest(}\; B \;,\; Num \;,\; Q \;\texttt{)} \wedge \texttt{isHeavyInst(}\; T \;\texttt{)}$

RULE CPU-NOASSIGN

$$\left\langle \frac{\texttt{noAssign(}\; V : KResult \;\texttt{)}}{\texttt{.K}} \cdots \right\rangle_{\mathsf{k}} \; \left\langle \frac{\texttt{dynInstHold(}\; B \;,\; Num \;,\; T \;\texttt{)}}{\texttt{.K}} \right\rangle_{\mathsf{CPU}}$$

RULE CPU-ASSIGN-SPEC

$$\left\langle \frac{\texttt{assign(}\; X \;,\; V : KResult \;\texttt{)}}{\texttt{.K}} \cdots \right\rangle_{\mathsf{k}} \; \left\langle \frac{M : Map}{M \;\texttt{[}\; V \;\texttt{/(}\; B \;,\; X \;\texttt{)]}} \right\rangle_{\mathsf{specRegisters}}$$

$$\left\langle \frac{\texttt{dynInstHold(}\; B \;,\; Num \;,\; T \;\texttt{)}}{.K} \right\rangle_{\mathsf{CPU}} \; \langle B' \rangle_{\mathsf{currBlock}}$$

$$\left\langle \frac{Tr}{\texttt{updateSpecTree(}\; Tr \;,\; B' \;,\; B \;,\; V \;,\; Tr \;\texttt{)}} \right\rangle_{\mathsf{specTree}}$$

$$\left\langle \frac{Q}{\texttt{updateSpecSet(}\; Q \;,\; B' \;,\; B \;,\; V \;,\; Tr \;\texttt{)}} \right\rangle_{\mathsf{instQueue}}$$

REQUIRES $B > B'$

Figure 7.2: Selected Rules for **K-LLVM** Execution Model

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 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

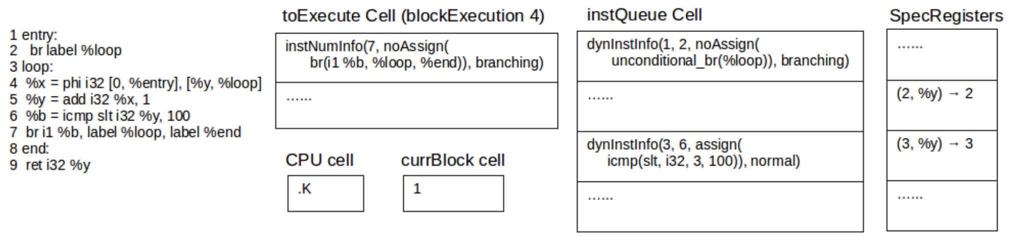

```
1 entry:
2   br label %loop
3 loop:
4  %x = phi i32 [0, %entry], [%y, %loop]
5  %y = add i32 %x, 1
6  %b = icmp slt i32 %y, 100
7  br i1 %b, label %loop, label %end
8 end:
9  ret i32 %y
```
toExecute Cell (blockExecution 4)
instNumInfo(7, noAssign(
br(i1 %b, %loop, %end)), branching)
......
CPU cell
.K
currBlock cell
1
instQueue Cell
dynInstInfo(1, 2, noAssign(
unconditional_br(%loop)), branching)
......
dynInstInfo(3, 6, assign(
icmp(slt, i32, 3, 100)), normal)
......
SpecRegisters
......
(2, %y) → 2
(3, %y) → 3
......

Figure 7.3: SpecTree Example

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 7.3, 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)` $\rightarrow$ `2` and `(3, %y)` $\rightarrow$ `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 7.2. 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.

7.2 THE MEMORY MODEL

The **K-LLVM** operational memory model is similar to a message passing model. Its components and relations are described in Figure 7.1b. All of the components in the figure are cells in the **K-LLVM** configuration that represents some program state pieces. In the

RULE TOCOMMIT-OUT

$$\left\langle (\, \frac{\texttt{SetItem(singleMem(}\ Tid\ ,\ B\ ,\ Num\ \texttt{,heap,}\ Op\ \texttt{))}}{.Set} R\ ,\ PR\ ,\ NPR\ ,\ NR\ ) \right\rangle_{\textsf{toCommit}} \left\langle \frac{M : Map}{\texttt{markMemOp(}\ M\ ,\ B\ ,\ Num\ \texttt{)}} \right\rangle_{\textsf{specTree}}$$

$$\left\langle \cdots \frac{\texttt{.List}}{\texttt{ListItem(singleMem(}\ Tid\ ,\ B\ ,\ Num\ \texttt{,heap,}\ Op\ \texttt{))}} \right\rangle_{\textsf{memOpList}} \langle MemId \rangle_{\textsf{memCache-ID}} \ \langle MemId \rangle_{\textsf{cache-ID}}$$

RULE ATOMICWRITE

$$\left\langle \frac{\texttt{ListItem(singleMem(}\ Tid\ ,\ B\ ,\ Num\ \texttt{,heap,atomicWrite(}\ Base\ ,\ Size\ ,\ V\ ,\ Or\ \texttt{)))}}{\texttt{.List}} \cdots \right\rangle_{\textsf{memOpList}} \left\langle \frac{AcK}{AcK\ \texttt{[0 /}\ TM\ \texttt{[}\ CId\ \texttt{]+1]}} \right\rangle_{\textsf{ackMap}}$$

$$\left\langle \cdots \frac{\texttt{.List}}{\texttt{ListItem(sendAll(}\ CId\ \texttt{,maxCacheN,}\ TM\ \texttt{[}\ CId\ \texttt{]+1, addOne(}\ TM\ ,\ CId\ \texttt{),msgWrite(}\ Tid\ ,\ B\ ,\ Num\ ,\ Base\ ,\ Size\ ,\ V\ ,\ Or\ \texttt{))}} \right\rangle_{\textsf{message}}$$

$$\left\langle \frac{CM : Map}{\texttt{rmRange(}\ CM\ ,\ Base\ ,\ Size\ \texttt{)[(}\ Tid\ ,\ V\ \texttt{) / (}\ Base\ ,\ Size\ \texttt{)]}} \right\rangle_{\textsf{tempMem}} \left\langle \frac{BM}{\texttt{updateAtomic(}\ BM\ ,\ Base\ ,\ V\ \texttt{)}} \right\rangle_{\textsf{byteMap}}$$

$$\langle \cdots \langle (\ Left\ ,\ Right\ ) \rangle_{\textsf{chunkRange}} \ \langle Races : Set \rangle_{\textsf{race}} \cdots \rangle_{\textsf{object}} \left\langle \frac{TM}{\texttt{addOne(}\ TM\ ,\ CId\ \texttt{)}} \right\rangle_{\textsf{timeStamp}} \ \langle CId \rangle_{\textsf{cache-ID}}$$

REQUIRES $Base \geq Left \wedge Base + Size \leq Right \wedge Or \neq \texttt{seq\_cst} \wedge \neg\ \texttt{isOverlap(}\ Races\ ,\ Base\ ,\ Size\ \texttt{)}$

RULE ATOMICWRITE-SEQ

$$\left\langle \frac{\texttt{ListItem(singleMem(}\ Tid\ ,\ B\ ,\ Num\ \texttt{,heap,atomicWrite(}\ Base\ ,\ Size\ ,\ V\ \texttt{,seq\_cst)))}}{\texttt{ListItem(singleMem(}\ Tid\ ,\ B\ ,\ Num\ \texttt{,heap,msgWait(}\ TM\ \texttt{[}\ CId\ \texttt{]+1)))}} \cdots \right\rangle_{\textsf{memOpList}} \left\langle \frac{AcK}{AcK\ \texttt{[0 /}\ TM\ \texttt{[}\ CId\ \texttt{]+1]}} \right\rangle_{\textsf{ackMap}}$$

$$\left\langle \cdots \frac{\texttt{.List}}{\texttt{ListItem(sendAll(}\ CId\ \texttt{,maxCacheN,}\ TM\ \texttt{[}\ CId\ \texttt{]+1, addOne(}\ TM\ ,\ CId\ \texttt{),msgWrite(}\ Tid\ ,\ B\ ,\ Num\ ,\ Base\ ,\ Size\ ,\ V\ \texttt{,seq\_cst))}} \right\rangle_{\textsf{message}}$$

$$\left\langle \frac{CM : Map}{\texttt{rmRange(}\ CM\ ,\ Base\ ,\ Size\ \texttt{)[(}\ Tid\ ,\ V\ \texttt{) / (}\ Base\ ,\ Size\ \texttt{)]}} \right\rangle_{\textsf{tempMem}} \left\langle \frac{BM}{\texttt{updateAtomic(}\ BM\ ,\ Base\ ,\ V\ \texttt{)}} \right\rangle_{\textsf{byteMap}}$$

$$\langle \cdots \langle (\ Left\ ,\ Right\ ) \rangle_{\textsf{chunkRange}} \ \langle Races : Set \rangle_{\textsf{race}} \cdots \rangle_{\textsf{object}} \left\langle \frac{TM}{\texttt{addOne(}\ TM\ ,\ CId\ \texttt{)}} \right\rangle_{\textsf{timeStamp}} \ \langle CId \rangle_{\textsf{cache-ID}}$$

REQUIRES $Base \geq Left \wedge Base + Size \leq Right \wedge \neg\ \texttt{isOverlap(}\ Races\ ,\ Base\ ,\ Size\ \texttt{)}$

Figure 7.4: Selected Rules for **K-LLVM** Memory Model

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 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 7.4 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 7.1b 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 7.5. 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

```
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`

Figure 7.5: Comparison Between `acquire/release` and `seq_cst` Memory Ordering

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 7.1b 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 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 7.4, 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 [3, 111, 113, 152] 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 7.1. 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 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 7.1. 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 5.1, 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 7.6. 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

RULE SPECUNORDEREDWRITE

$$\left\langle \dfrac{\begin{array}{l}(\ R\ ,\ PR\ ,\texttt{SetItem(singleMem(}\\ \quad Tid\ ,\ B\ ,\ Num\ ,\texttt{heap,atomicWrite(}\\ \quad\quad Base\ ,\ Size\ ,\ V\ ,\texttt{unordered)))}\ NPR\ ,\ NR\ )\end{array}}{\begin{array}{l}(\ R\ ,\texttt{SetItem(singleMem(}\\ \quad Tid\ ,\ B\ ,\ Num\ ,\texttt{heap,atomicWrite(}\\ \quad\quad Base\ ,\ Size\ ,\ V\ ,\texttt{unordered)))}\ PR\ ,\ NPR\ ,\ NR\ )\end{array}} \right\rangle_{\mathsf{toCommit}}$$

$$\left\langle \dfrac{CM : Map}{CM\ \texttt{[(}\ I\ ,\ Tid\ ,\ V\ \texttt{)}\ /\ \texttt{(}\ Base\ ,\ Size\ \texttt{)]}} \right\rangle_{\mathsf{CDB}}$$

$$\left\langle \dfrac{RM : Map}{RM\ \texttt{[(}\ I\ ,\ Base\ ,\ Size\ ,\ V\ \texttt{)}\ /\ \texttt{(}\ B\ ,\ Num\ \texttt{)]}} \right\rangle_{\mathsf{memTrack}}$$

$$\left\langle \dfrac{RM : Map}{RM\ \texttt{[}\ V\ /\ \texttt{(}\ I\ ,\ Base\ ,\ Size\ \texttt{)]}} \right\rangle_{\mathsf{track}} \left\langle \dfrac{I}{I + 1} \right\rangle_{\mathsf{count}}$$

$$\langle \cdots\ \texttt{SetItem((}\ Left\ ,\ Right\ \texttt{))}\ \cdots \rangle_{\mathsf{memoryList}}$$

REQUIRES $Base \geq Left \wedge Base + Size \leq Right$

[transition]

RULE UNORDERED-READ-NO

$$\left\langle \dfrac{\begin{array}{l}(\ R\ ,\ PR\ ,\texttt{SetItem(singleMem(}\\ \quad Tid\ ,\ B\ ,\ Num\ ,\texttt{heap,atomicRead(}\\ \quad\quad T\ ,\ Base\ ,\ Size\ ,\texttt{unordered)))}\ NPR\ ,\ NR\ )\end{array}}{\begin{array}{l}(\ R\ ,\texttt{SetItem(singleMem(}\\ \quad Tid\ ,\ B\ ,\ Num\ ,\texttt{heap,atomicRead(}\\ \quad\quad T\ ,\ Base\ ,\ Size\ ,\texttt{unordered)))}\ PR\ ,\ NPR\ ,\ NR\ )\end{array}} \right\rangle_{\mathsf{toCommit}}$$

[transition]

RULE UNORDERED-READ-CDB

$$\left\langle \dfrac{\begin{array}{l}(\ R\ ,\ PR\ ,\texttt{SetItem(singleMem(}\\ \quad Tid\ ,\ B\ ,\ Num\ ,\texttt{heap,atomicRead(}\\ \quad\quad T\ ,\ Base\ ,\ Size\ ,\texttt{unordered)))}\ NPR\ ,\ NR\ )\end{array}}{(\ R\ ,\ PR\ ,\ NPR\ ,\ NR\ )} \right\rangle_{\mathsf{toCommit}}$$

$$\left\langle \cdots\ (\ B\ ,\ Num\ ) \mapsto \dfrac{\texttt{wait(}\ X\ ,\ VL\ \texttt{)}}{\texttt{ready(}\ X\ \texttt{,joinBytes(}\ T\ ,\ V\ \texttt{))}}\ \cdots \right\rangle_{\mathsf{readBack}}$$

$$\langle \cdots\ (\ Base\ ,\ Size\ ) \mapsto (\ I\ ,\ Tid'\ ,\ V\ )\ \cdots \rangle_{\mathsf{CDB}}\ \langle \cdots\ Tid\ \cdots \rangle_{\mathsf{thread-ID}}$$

REQUIRES $Tid \neq Tid'$

[transition]

RULE UNORDERED-READ-LOCAL

$$\left\langle \dfrac{\begin{array}{l}(\ R\ ,\ PR\ ,\texttt{SetItem(singleMem(}\\ \quad Tid\ ,\ B\ ,\ Num\ ,\texttt{heap,atomicRead(}\\ \quad\quad T\ ,\ Base\ ,\ Size\ ,\texttt{unordered)))}\ NPR\ ,\ NR\ )\end{array}}{(\ R\ ,\ PR\ ,\ NPR\ ,\ NR\ )} \right\rangle_{\mathsf{toCommit}}$$

$$\langle M \rangle_{\mathsf{specTree}} \left\langle \cdots\ \begin{array}{l}\texttt{correctDef(}\ M\ ,\ B\ ,\ Num\ ,\ Base\ ,\ Size\ \texttt{)}\\ \quad \mapsto (\ I\ ,\ Base'\ ,\ Size'\ ,\ V\ )\end{array}\ \cdots \right\rangle_{\mathsf{memTrack}}$$

$$\left\langle \cdots\ (\ B\ ,\ Num\ ) \mapsto \dfrac{\texttt{wait(}\ X\ ,\ VL\ \texttt{)}}{\texttt{ready(}\ X\ \texttt{,joinBytes(}\ T\ ,\ V\ \texttt{))}}\ \cdots \right\rangle_{\mathsf{readBack}}$$

$$\langle \cdots\ (\ Base\ ,\ Size\ ) \mapsto (\ I'\ ,\ Tid\ ,\ V'\ )\ \cdots \rangle_{\mathsf{CDB}}\ \langle \cdots\ Tid\ \cdots \rangle_{\mathsf{thread-ID}}$$

[transition]

Figure 7.6: Selected Unordered Rules

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).

7.3 RELATION BETWEEN THE LLVM OPERATIONAL MODEL AND HATRMM

The operational model we described here has been proved to be equivalent to the HATRMM model in Chapter 6. The soundness proof is proved through on showing all the possible executions of programs in the LLVM operational model satisfy the properties defined in HATRMM. The completeness is a relatively completeness theorem. We first identify the different kinds of memory instructions in the LLVM operational model with respect to the constructs in HATRMM. Then, we show that for a possible translation of the constructs in HATRMM to a subset of memory instructions in the LLVM operational model, if the execution of any translated program have certain concurrent behaviors, the behaviors are captured by the predicates defined in HATRMM.

Chapter 8: A FRAMEWORK TO VERIFY COMPILER OPTIMIZATION SEMANTIC PRESERVATION

Here, we discuss a framework to verify the semantics preservation property for a compiler optimization in the LLVM semantics with the LLVM operational model. The essence of the framework is the Per-Location Simulation Framework.

When the preservation of concurrency behavior was being verified in the CompCert compiler [37], the researchers found that it is not enough to tell the whole story just to use a traditional bisimulation framework to prove the program equivalence between a compiled program and its original one, so they designed a new bisimulation framework by treating safe programs and programs that might reach error states differently. CompCert's concurrency model was sequential consistency. The extent to which traditional bisimulation is inappropriate is even clearer when dealing with weak concurrency models. Weak concurrency models have been studied broadly for real world imperative programming languages (C/C++/LLVM) [2, 3, 4, 100, 101, 102, 103, 104, 105, 107, 108, 109, 110, 111, 112, 113, 114, 157]. When using these models to prove compiler correctness, a problem arises. Historically, the semantics of these languages has been determined by the behavior of their compilers, so the behavioral effects of compiler optimizations also need to be considered in the concurrency models. For example, in the program piece (b) in Fig. 8.1, variables a and b can both read 1 if we consider the fact that a simple optimization removes the Boolean guards in (b), transforming it as the program piece (a). The well-known confusion about out-of-thin-air behaviors [158] is a typical consequence of the problem.

```
/* number after the blue "//" along with a read restricts the value of the read to be the number */
/* initially, x = 0 and y = 0 */

                 (a)                              (b)                                (c)
a :=rlx y//1 ‖ b :=rlx x//1       a :=rlx y//1 ‖ b :=rlx x//1       a :=rlx y//1 ‖ b :=rlx x//1
x :=rlx 1    ‖ y :=rlx 1          if (a=a)     ‖ if (b=b)           if (a=1)     ‖ y :=rlx 1
                                      x :=rlx 1 ‖     y :=rlx 1         x :=rlx 1 ‖

    x :=rlx 1                                                        a :=rlx y//z ‖ b :=rlx x//z ‖ if (ram())
(d) y :=rlx 1   (e) a :=rlx y//1 ‖ b :=rlx x//1                 (f)  if (a=a)     ‖ if (b=b)     ‖     z :=rlx 1
    a :=rlx y       if (a=1)     ‖ if (b=1)                              x :=rlx z ‖     y :=rlx z ‖   else
    b :=rlx x           x :=rlx 1 ‖     y :=rlx 1                                                 ‖       z :=rlx 2
```

Figure 8.1: Motivating Examples

Researchers [4, 111, 112, 113] have tried to solve the thin-air problems by merging the extra behaviors caused by compiler optimizations into their concurrency models. These models have several problems. Vafeiadis *et al.* [159] has shown that most of the compiler optimizations are invalid in these weak models. Moreover, Batty *et al.* [153] proved that it does not exists a candidate execution style axiomatic C++ concurrency model to incooperate the thin-air problems raised by (b) (Fig. 8.1). Additionally, these models are built upon a very limited set of memory actions or language pieces, and provide correct compiler schemes generated from the models based on the limited set. It requires a great effort to extend the schemes to prove a real-world compiler optimization preserving the multi-threaded program semantics for a real-world language under a real-world concurrency model. In some cases, even if the underlying language is extended a little, these models failed to show all supposedly allowed behaviors. For example, the promising model [113] is designed specifically to prove that the two reads in (d) (Fig. 8.1) can both read 1, but it fails to prove that the variables a and b in (f) can read any possible value from location z because z does not have a fixed value in all possible executions. The IMM model [4] is able to prove that the two reads in (a) can both read 1, but it fails to prove the two reads in (b) (Fig. 8.1) can both read 1. PLS is able to handle all these cases.

In this chapter, we propose a simulation framework, named Per-Location Simulation (PLS), that is able to prove semantic preservation between compiled programs and their original programs under a language semantics with a weak concurrency model. We focus on safe traces (traces not going wrong) here, and assume that there is an outer layer on top of PLS to deal with reaching-error-state traces the same as the upward simulation framework in defined by CompCert [37]. As a main example, we provide a clear border for acceptable behaviors and out-of-thin-air behaviors in a CFG-based language with a weak concurrency model by using PLS to prove the semantic preservation of a simple optimization. The border is summarized by the examples in Fig. 8.1, which can be divided into two parts. The first is the PLS core part (Sec. 8.1.1). By the traditional simulation framework, the example (c) cannot be proven to simulate (a) (meaning that (a) semantically preserves (c)), because the memory trace (d) can be generated by (a), but it cannot be observed from (c). By analyzing closely the output of the two reads and two writes in both (a) and (c), all values that can be observed in these reads and writes of (c) can also be observed in (a). Thus, we should have a kind of similarity between (a) and (c). The PLS core produces such kind. It filters traces of programs into sub-traces based on locations. Instead of comparing the whole traces (as (d)), the PLS core compares the sub-traces of location x (and y) in (a) and (c) to determine if (c) per-location simulates (a). The second part is the full PLS definition (Sec. 8.1.2), which addresses the focal point of thin-air problems. (b) (Fig. 8.1) is supposed to be proved to be semantically preserved by (a), but (e) is not; because the two Boolean guards in (d) can be compiled away, but such guards in (e) cannot be removed. There are traces appearing in (a) but not appearing in (e). To validate the proof, we augment the PLS core with additional equations that capture some very simple compiler optimization syntactic dependencies. Instead of proving the simulation from (d) to (a), we prove the simulation from an equivalent representative of (d) to (a). To the best of our knowledge, PLS is the first simulation framework weaker than the one in CompCert/CompCertTSO [97], and to be used to prove compiler correctness under a CFG-based imperative language with a weak memory model, and is able to correctly distinguish thin-air and correct behaviors.

8.1 THE PER-LOCATION SIMULATION DEFINITION

This section provides an introduction of PLS. We first introduce PLS core, then we provide an example language, and then we introduce the full PLS definition. After that, we introduce the definition of PLS with failure by combining PLS with the forward simulation in CompCertTSO [97].

8.1.1 PLS Core

Transition System

States: $\sigma \in \Sigma$ Labels: $\alpha \in A$ Labeled Transition Systems (LTS): $(\Sigma, A, \xrightarrow{\alpha})$

Locations: *Loc* Label's Value: $\texttt{val}(\alpha)$ Label's Type: $\texttt{type}(\alpha)$ Label's Location: $\texttt{loc}(\alpha) \in Loc$

Transition System Property: $(\forall\alpha.\ \texttt{type}(\alpha) = \tau \Rightarrow \texttt{val}(\alpha) = \bot \wedge \texttt{loc}(\alpha) = \bot) \wedge (\bot \notin Loc)$

Transition System Syntactic Sugar

$\sigma \xrightarrow{\tau} \sigma' \triangleq \exists\alpha.\ \sigma \xrightarrow{\alpha} \sigma' \wedge \texttt{type}(\alpha) = \tau$ $\sigma \rightarrow_{\texttt{not}(x)} \sigma' \triangleq \exists\alpha.\ \sigma \xrightarrow{\alpha} \sigma' \wedge \texttt{loc}(\alpha) \neq x$

$\sigma \xrightarrow{\alpha}_x \sigma' \triangleq \exists\sigma_n\ \alpha.\ \sigma \rightarrow^*_{\texttt{not}(x)} \sigma_n \xrightarrow{\alpha} \sigma' \wedge \texttt{type}(\alpha) \neq \tau \wedge \texttt{loc}(\alpha) = x$

PLS Definition

Label Equivalence: $\alpha \equiv \beta \triangleq \texttt{val}(\alpha) = \texttt{val}(\beta) \wedge \texttt{type}(\alpha) = \texttt{type}(\beta) \wedge \texttt{loc}(\alpha) = \texttt{loc}(\beta)$

$\texttt{LTS}_\Xi$: $(\Xi, A, \xrightarrow{\alpha}{}^{\Xi})$ $\texttt{LTS}_\Sigma$: $(\Sigma, B, \xrightarrow{\beta}{}^{\Sigma})$

$\sqsubseteq_x$ is a $\texttt{PLS}_x$ relation on two transition systems $\texttt{LTS}_\Xi$ and $\texttt{LTS}_\Sigma$:

a.k.a. $\texttt{PLS}_x(\sqsubseteq_x) \triangleq$

$\forall\xi\ \xi_1 \in \Xi.\ (\forall\sigma \in \Sigma\ (\forall\alpha \in A.\ \xi \sqsubseteq_x \sigma \wedge \xi \xrightarrow{\alpha}{}^{\Xi}_x \xi_1 \Rightarrow (\exists\beta\ \sigma_1.\ \sigma \xrightarrow{\beta}{}^{\Sigma}_x \sigma_1 \wedge \alpha \equiv \beta \wedge \xi_1 \sqsubseteq_x \sigma_1)))$

$\texttt{PLS}_{Loc}(\sqsubseteq) \triangleq \forall x \in Loc.\ \texttt{PLS}_x(\sqsubseteq_x)$

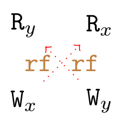
Figure 8.2: Per-Location Simulation Core Definition

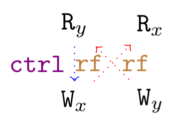
Here, we introduce the PLS core definition and utility examples. Fig. 8.2 includes the PLS core definition. We assume that there is a labeled transition system (LTS) $(\Sigma, A, \xrightarrow{\alpha})$, including a set of states (Σ), a set of labels (A), and a labeled transition function ($\xrightarrow{\alpha}$). The transition system is parameterized by a set of locations *Loc*. Every label in the transition system has three properties: its value (accessed by val), its type (at least having a τ type and normally having additional read and write types), and its memory location (be in the set *Loc*). For simplicity, we assume that if the type of a label is τ, then the value and location of the label are $\bot$ in a given transition system. To best describe the PLS core definition, we define some syntactic sugar on top of the transition system $\xrightarrow{\alpha}$ in Fig. 8.2. We first describe a predicate $\mathtt{PLS}_x$ defining PLS core on a single location x. A relation $\sqsubseteq_x$ is a PLS core relation on x over two labeled transition systems ($\mathtt{LTS}_\Xi$ and $\mathtt{LTS}_\Sigma$ in Fig. 8.2), if for any two states ($\xi \in \Xi$ and $\sigma \in \Sigma$) in the relation ($\xi \sqsubseteq_x \sigma$), ξ can transition by an x step (defined by $\xrightarrow{\alpha}_x$), then σ can also transition by an x step, where the two labels are equivalent ($\equiv$) and the resulting states are again related by $\sqsubseteq_x$. A family of relations ($\sqsubseteq$), one for each location in *Loc*, is a PLS core relation if each indexed relation ($\sqsubseteq_x$) satisfies $\mathtt{PLS}_x$ for each x in *Loc*, where *Loc* is a finite set of memory locations.

(wr_a) $\begin{array}{l} x :=_{\mathtt{rlx}} 1 \\ y :=_{\mathtt{rlx}} 1 \\ z :=_{\mathtt{rlx}} 1 \end{array}$ (wr_b) $\begin{array}{l} y :=_{\mathtt{rlx}} 1 \\ x :=_{\mathtt{rlx}} 1 \\ z :=_{\mathtt{rlx}} 1 \end{array}$ (prop) $x :=_{\mathtt{rlx}} 1 \wedge y :=_{\mathtt{rlx}} 1 \wedge z :=_{\mathtt{rlx}} 1$

Figure 8.3: PLS Sequential Program Execution Examples

We first discuss the single-threaded cases. The program pieces in Fig. 8.3 ((wr_a) and (wr_b)) describe two sequences of memory writes. Regarding the underlying memory concurrency model, the outputs of the two program pieces should be the same, i.e. to write 1 to the locations x, y, and z. However, (wr_a) and (wr_b) cannot be proved to be similar with each other using the traditional simulation framework under the assumption of sequential consistency. Only if we assume a relaxed concurrency model can we prove that (wr_a) and (wr_b) are similar. In PLS, both (wr_a) and (wr_b) are viewed as three sub-traces as shown in (prop) (for simplicity, in each sub-trace, we only show instructions without mentioning other state environments), each of which describes a write for a location; so that we are able to prove that (wr_a) and (wr_b) are per-loc similar to each other.

(a_dd) 

(b_dd) 

(prop_a) $\begin{array}{l} x :=_{\mathtt{rlx}} 1 \\ b :=_{\mathtt{rlx}} x \end{array} \wedge \begin{array}{l} y :=_{\mathtt{rlx}} 1 \\ a :=_{\mathtt{rlx}} y \end{array}$

Figure 8.4: PLS Multi-threaded Program Execution Examples

We now discuss multi-threaded cases under a weak relaxed memory model [3]. One of the example per-loc simulation relations that PLS enables is the one between programs (a) and (c) in Fig. 8.1, whose execution diagrams are listed as (a_dd) and (b_dd) in Fig. 8.4. An execution diagram is a graph representation of the execution of a program (only listing memory instructions), with arrows representing the memory instruction order that the execution must obey. In (a_dd), `W` (or `R`) represents a write (or a read) instruction with the subscripts (x or y) representing the memory locations (details are the $\mathscr{Act}$ type in Fig. 6.2). The `rf` arrow between $\mathtt{R}_y$ and $\mathtt{W}_y$ in (a_dd) means that the read from y reads the value written by the write, so the read must happen after the write. The `ctrl` arrow in (b_dd) is a control dependency so that $\mathtt{R}_y$ must happen before $\mathtt{W}_x$ in any valid execution of (c) (Fig. 8.1). This is the reason that program (c) does not simulate program (a) by traditional simulation methods, i.e. because the execution ((d) in Fig. 8.1) happens in (a) but never happens in (c). On the other hand, PLS deals the two programs by first splitting all executions in both programs into a sub-trace per-location like the one in (prop_a). Thus, (c) per-loc simulates (a) ((a) semantically preserves (c)).

8.1.2 Full PLS

The PLS core definition is suitable for building the per-loc simulation between (a) and (c) in Fig. 8.2, but the relation between (a) and (b) (Fig. 8.2) cannot be handled by the PLS core. To enhance the usability of PLS, we associate a reflexive relation `eq` with the PLS core definition as the Full PLS definition (Fig. 8.5).

PLS Definition

Reflexive Relations On Two LTSs: `eq`

$\mathtt{LTS}_\Xi$: $(\Xi, A, \xrightarrow{\alpha}^{\Xi})$ $\quad$ $\mathtt{LTS}_\Sigma$: $(\Sigma, B, \xrightarrow{\beta}^{\Sigma})$ $\quad$ $\mathtt{LTS}_\Upsilon$: $(\Upsilon, K, \xrightarrow{\kappa}^{\Upsilon})$

$$\begin{aligned}
&\mathtt{PLS}_x^{\mathtt{eq}}(\sqsubseteq_x) \triangleq \\
&\quad \forall \xi\ \xi_1 \in \Xi.\ (\forall \alpha \in A.\ (\forall \sigma \in \Sigma.\ \xi \sqsubseteq_x \sigma \wedge \xi \xrightarrow{\alpha}{}^{\Xi}_x \xi_1 \Rightarrow \\
&\qquad (\exists\ (\mathtt{LTS}_\Sigma, \mathtt{LTS}_\Upsilon, \preceq^{\mathtt{eq}}) \in \mathtt{eq}.\ (\exists \upsilon\ \upsilon_1 \in \Upsilon.\ (\exists \kappa \in K. \\
&\qquad\quad \sigma \preceq^{\mathtt{eq}} \upsilon \wedge \upsilon \xrightarrow{\kappa}{}^{\Upsilon}_x \upsilon_1 \wedge (\exists \sigma_1 \in \Sigma.\ \sigma_1 \preceq^{\mathtt{eq}} \upsilon_1 \wedge \xi_1 \sqsubseteq_x \sigma_1)))))) \\
&\mathtt{PLS}_{Loc}^{\mathtt{eq}}(\sqsubseteq) \triangleq \forall x \in Loc.\ \mathtt{PLS}_x^{\mathtt{eq}}(\sqsubseteq_x)
\end{aligned}$$

$$\begin{tikzcd}
\xi \arrow[r, "\forall"] \arrow[d, phantom, "\sqsubseteq"] & \xi_1 \arrow[d, phantom, "\sqsubseteq"] \\
\sigma \arrow[d, phantom, "\preceq^{\mathtt{eq}}"] & \sigma_1 \arrow[d, phantom, "\preceq^{\mathtt{eq}}"] \\
\upsilon \arrow[r, dotted, "\exists"] & \upsilon_1
\end{tikzcd}$$

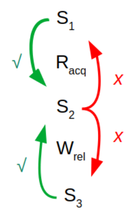
Figure 8.5: Full Per-Location Simulation Definition

The `eq` relation is at least a reflexive relation describing program transformations and capturing the syntactic dependencies of program instructions that are hard to be discovered by only the program concurrency semantics, such as the example (b) in Fig. 8.1. `eq` including the identity relation (as $\preceq^{\mathtt{eq}}$) relates two systems $\mathtt{LTS}_\Sigma$ and $\mathtt{LTS}_\Upsilon$, such as the tuple

$(\mathtt{LTS}_\Sigma, \mathtt{LTS}_\Upsilon, \preceq^{\mathtt{eq}})$ in Fig. 8.5. $\mathtt{PLS}_x^{\mathtt{eq}}$ can be understood by the right diagram in Fig. 8.5. Assume that we have two systems $\mathtt{LTS}_\Xi$ and $\mathtt{LTS}_\Sigma$. We want to show the per-loc simulation ($\sqsubseteq_x$) from $\mathtt{LTS}_\Xi$ to $\mathtt{LTS}_\Sigma$ by showing that for every transition ξ to ξ_1, there exists a transition σ to σ_1, such that the two transition labels are equivalent ($\equiv$). However, we cannot directly have a transition from σ to σ_1 in some cases. Instead, through the $\mathtt{eq}$ set, we find a relation $\preceq^{\mathtt{eq}}$ that relates $\mathtt{LTS}_\Sigma$ with another system $\mathtt{LTS}_\Upsilon$; and the transition from υ to υ_1 is found in $\mathtt{LTS}_\Upsilon$, where υ and υ_1 are related to σ and σ_1 through $\preceq^{\mathtt{eq}}$, respectively, and ξ_1 and σ_1 are also related by $\sqsubseteq_x$. $\mathtt{PLS}_x^{\mathtt{eq}}$ is a generalization of the $\mathtt{PLS}_x$ predicate in Fig. 8.2, if we just select the tuple in $\mathtt{eq}$ as $(\mathtt{LTS}_\Sigma, \mathtt{LTS}_\Sigma, =)$. By selecting such tuple, the two systems $\mathtt{LTS}_\Sigma$ and $\mathtt{LTS}_\Upsilon$ are the same. Finally, $\mathtt{PLS}_{Loc}^{\mathtt{eq}}$ includes the functionality as $\mathtt{PLS}_{Loc}$, but it builds a family of relations over the predicate $\mathtt{PLS}_x^{\mathtt{eq}}$.

$$
\begin{aligned}
\mathtt{eq} \triangleq\; & \ldots \\
& (N, \pi_0, \lambda \cup \{\pi \mapsto (\mathit{ins}, \mathtt{if}\ e)\}, E' \cup \{(\pi,\ \mathtt{yes}, \pi_1), (\pi,\ \mathtt{no}, \pi_2)\}) \\
& \cong_{\mathtt{eq}} (N, \pi_0, \lambda \cup \{\pi \mapsto (\mathit{ins}, \mathtt{br})\}, E' \cup \{(\pi,\ \mathtt{seq}, \pi_1)\}) \\
& \ \mathtt{IF}\ \mathtt{eval}(e) = \mathtt{true}; \\
& (N, \pi_0, \lambda \cup \{\pi \mapsto (\mathit{ins}, \mathtt{if}\ e)\}, E' \cup \{(\pi,\ \mathtt{yes}, \pi_1), (\pi,\ \mathtt{no}, \pi_2)\}) \\
& \cong_{\mathtt{eq}} (N, \pi_0, \lambda \cup \{\pi \mapsto (\mathit{ins}, \mathtt{br})\}, E' \cup \{(\pi,\ \mathtt{seq}, \pi_2)\}) \\
& \ \mathtt{IF}\ \mathtt{eval}(e) = \mathtt{false}; \\
& (N, \pi_0, \lambda \cup \{\pi \mapsto (\mathit{ins}, \mathtt{if}\ e)\}, E' \cup \{(\pi,\ \mathtt{yes}, \pi_1), (\pi,\ \mathtt{no}, \pi_2)\}) \\
& \cong_{\mathtt{eq}} (N, \pi_0, \lambda \cup \{\pi \mapsto (\mathit{ins}, \mathtt{br})\}, E' \cup \{(\pi,\ \mathtt{seq}, \pi_1)\}) \\
& \ \mathtt{IF}\ \lambda(\pi_1) = \lambda(\pi_2) \wedge (\forall l\ \pi'. (\pi_1, l, \pi') \in E \Leftrightarrow (\pi_2, l, \pi') \in E); \\
& \ldots
\end{aligned}
$$

(a) Example $\mathtt{eq}$ Relation



(b) Roach Model on Acquire/Release Atomics

Figure 8.6: Example and Roach Model

Fig. 8.6a provides a partial definition of an example $\mathtt{eq}$ set. The set contains equations to relate two labeled transition systems $\mathtt{LTS}_\Xi$ and $\mathtt{LTS}_\Sigma$ by relating the two program texts in any two states ξ and σ from the systems. The conditional equations shown in Fig. 8.6a is to equate two CFGs for a thread in any two program texts, i.e. two program texts μ and μ' are equivalent, if for any thread tid in the domain of μ/μ', $\mu(\mathit{tid}) \cong_{\mathtt{eq}} \mu'(\mathit{tid})$ ($\cong_{\mathtt{eq}}$ means equivalence closed under the conditional equations in Fig. 8.6a). The first two conditional equations in Fig. 8.6a describe the equivalence relation that if a Boolean guard of a binary branching is always evaluated to $\mathtt{true}$ or $\mathtt{false}$ statically (by the $\mathtt{eval}$ function), then the CFG is related to the version formed by transforming the branching operation to a unconditional branching operation. The third rule describes the relation that if the outgoing edges of a branching block have the same target, then the CFG can be rewritten as a version only going through one branch.

The single-threaded programs ((pa_a), (pa_b), and (pa_c)) in Fig. 8.6 are examples for which traditional simulation frameworks cannot provide satisfactory explanations. Using a sequential consistency model, a traditional simulation framework enables the proof of similarity between programs (pa_a) and (pa_b) (let's assume that the executions of a program generate an LTS), because an execution of (pa_a) always executes a write to x, then a read from y, and then a write to z, which is the same sequence as the one produced by (pa_b). The problem is that we also want to show that (pa_a) and (pa_c) are similar, which the traditional framework cannot enable.

(pa_a)
```
x :=rlx c
if (a=b ∧ b=c)
    a :=rlx y
  else
    a :=rlx y
z :=rlx b
```

(pa_b)
```
x :=rlx c
a :=rlx y
z :=rlx b
```

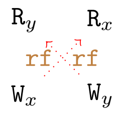
(pa_c)
```
a :=rlx y
z :=rlx b
x :=rlx c
```

(prop_pa) $x :=_{\texttt{rlx}} c \land a :=_{\texttt{rlx}} y \land z :=_{\texttt{rlx}} b$

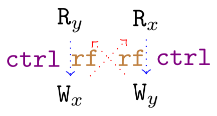
Figure 8.7: Single-threaded Reordering Example Executions

Under a weak memory model, like RC11 [3], a transitional simulation method enables the proof that (pa_c) simulates (pa_a) but not the opposite, because the Boolean guard in (pa_a) contains the variables a and b, so it has data dependency on the later instructions (read from y and write to z). Thus, they cannot move to execute before the Boolean guard as well as the write to x. Clearly, by using the full PLS, to prove that (pa_a) simulates (pa_c), we can first find an equivalent program of (pa_a), which is exactly the one in (pa_b). Then we prove that (pa_a) per-loc simulates (pa_c) by showing that (pa_b) per-loc simulates (pa_c).

(a_dd)



(d_dd)



(prop_a) $\begin{matrix} x :=_{\texttt{rlx}} 1 \\ b :=_{\texttt{rlx}} x \end{matrix} \land \begin{matrix} y :=_{\texttt{rlx}} 1 \\ a :=_{\texttt{rlx}} y \end{matrix}$

Figure 8.8: Difference Between PLS and Traditional Simulation Relation in Multi-threaded Executions

The simulation from (pa_a) to (pa_c) can also be proved by the PLS core definition in Sec. 8.1.1. To understand the additional proving ability that the full PLS brings us, the simulation from (b) to (a) in Fig. 8.1 provides a better hint. The execution diagram of (a) is shown as (a_dd) in Fig. 8.8, while the diagram of (b) is shown as (d_dd). In (d_dd), for every single thread, a control dependency (`ctrl`) exists from the read to the write. If we observe that the two reads both read `1`, we have exactly two reads-from edges (`rf`) from writes to reads. Thus, the diagram contains a cycle, which means that the execution of reading both as `1` is impossible if no optimization is applied to (b) (Fig. 8.1). Like the traditional simulation frameworks, PLS core is unable to prove the per-loc simulation from (b) to (a), which is the correct behavior in the sense that no optimization is applied. the desired simulation between (b) and (a) must take into account some resulting behaviors caused by optimizations. It is clear that the two `ctrl` edges in (d_dd) can be removed by some very simple optimizations, so that (b) becomes (a); and its execution diagram is the same as that of (a_dd). Then we can use the PLS core to build the simulation relation as the one in Sec. 8.1.1. This is the main content of the full PLS definition, which includes the optimization effects as the equivalence relation eq, then proves the per-loc similarity from an equivalence representative of (b) to (a) by using the PLS core.

(par)

```
a :=rlx y//1      ‖ b :=rlx x//1
if (a=1)          ‖ if (b=1)
        x :=rlx1  ‖        y :=rlx1
                  ‖    else
                  ‖        y :=rlx1
```

(par')

```
a :=rlx y         ‖ b :=rlx x
if (a=1)          ‖ y :=rlx1
        x :=rlx1  ‖
```

(par'_dd)

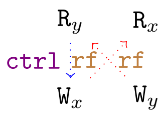


Figure 8.9: Example Executions with Optimizations on Branching Statements

If a traditional simulation framework would be parameterized with the eq relation, it could prove the simulation from (b) to (a) in Fig. 8.1, but it is inadequate for the simulation from the (par) above to (a) (Fig. 8.1). For that, the full power of PLS is required. To prove such a per-loc simulation, we first select an equivalence representative of program (par) to be the program (par') (Fig. 8.9). Then, we prove the per-loc simulation from (par') to (a) by the strategy for proving the relation from (c) to (a) (Sec. 8.1.1).

We then need to answer the question: what kind of equations are allowed in eq? The principle is described in the Roach Model of Manson *et al.* [160], and systematically explained by Vafeiadis *et al.* [106]: the short answer is any equation that can preserve program meaning, especially, the meaning of the critical section created by the acquire (`acq`) and release (`rel`) atomic memory operations. Essentially, the acquire/release atomics are C++ memory devices that implement a weak version of the memory locking mechanism. Moving a memory operation before an acquire atomic operation or after a release atomic operation violates the Roach Model principle that states: "shared memory accesses can be moved in critical regions but not out of them" (Fig. 8.6b). In the paper of Vafeiadis *et al.*, several cases are mentioned of an optimization violating this principle; each of them involves the removal or addition of read/write memory operations. For simplicity in this chapter, we provide the following observation about a conservative construction of `eq` to preserve the Roach Model principle. In it, $\texttt{LTS}|_{tid}$ means chopping the LTS to only execute single-threaded CFGs in the thread *tid*.

Observation 8.1. Assume that we have a transition system $\texttt{LTS}_\Sigma$, and a singleton relation set $\texttt{eq} = \{(\texttt{LTS}_\Sigma, \texttt{LTS}_\Xi, \sim)\}$. We assume that for every thread *tid*, we derive two single-threaded systems from $\texttt{LTS}_\Sigma$ and $\texttt{LTS}_\Xi$ as $\texttt{LTS}_\Sigma|_{tid}$ and $\texttt{LTS}_\Xi|_{tid}$, the $\sim$ relation ($\texttt{LTS}_\sigma|_{tid} \sim \texttt{LTS}_\sigma|_{tid}$) has the property that $\texttt{LTS}_\sigma|_{tid}$ is bisimilar to $\texttt{LTS}_\sigma|_{tid}$. Then, for any relation $\sqsubseteq$, such that $\texttt{PLS}^{\texttt{eq}}_{Loc}(\sqsubseteq)$, and any state ξ in $\texttt{LTS}_\Xi$ that does not transition (in $\texttt{LTS}_\Xi$) to a Roach-Model-violating state (Fig. 8.6b), if $\sigma \sqsubseteq \xi$, then σ does not transition (in $\texttt{LTS}_\Sigma$) to a Roach-Model-violating state.

8.1.3 PLS with Failure

PLS with Failure Definition

$$
\begin{aligned}
&\texttt{PLS}^{\texttt{eq}}_x(\sqsubseteq_x) \triangleq \\
&\quad \forall \xi\ \xi_1 \in \Xi.\ (\forall \alpha \in A.\ (\forall \sigma \in \Sigma.\ \xi \sqsubseteq_x \sigma \wedge \xi \xrightarrow{\alpha}{}_x^{\Xi}\ \xi_1 \Rightarrow \\
&\qquad (\exists\ (\texttt{LTS}_\Sigma, \texttt{LTS}_\Upsilon, \preceq^{\texttt{eq}}) \in \texttt{eq}.\ (\exists \upsilon\ \upsilon_1 \in \Upsilon.\ (\exists \kappa \in K.\ \sigma \preceq^{\texttt{eq}} \upsilon \wedge \upsilon \xrightarrow{\kappa}{}_x^{\Upsilon}\ \upsilon_1 \wedge (\exists \sigma_1 \in \Sigma.\ \sigma_1 \preceq^{\texttt{eq}} \upsilon_1 \wedge \xi_1 \sqsubseteq_x \sigma_1)))) \\
&\quad\ \ \vee (\exists\ \sigma'\ \sigma_{\texttt{err}} \in \Sigma.\ \sigma \xrightarrow{\tau}{}^{\Sigma *}\ \sigma' \xrightarrow{\texttt{fail}}{}^{\Sigma}\ \sigma_{\texttt{err}}))) \\
&\texttt{PLS}^{\texttt{eq}}_{Loc}(\sqsubseteq) \triangleq \forall x \in Loc.\ \texttt{PLS}^{\texttt{eq}}_x(\sqsubseteq_x)
\end{aligned}
$$

Figure 8.10: Per-Location Simulation with Failure Definition

The full PLS definition (Fig. 8.5) describes the relations among executions that never reach error states. To make the PLS definition suitable for reasoning about program executions that might reach failure states, we wrapped the full PLS definition with the failure simulation predicates in the upward simulation definition in CompCertTSO [97]. The definition is in Fig. 8.10. In the definition, $\texttt{LTS}_\Xi$ represents the system executing the optimized program, while the $\texttt{LTS}_\Sigma$ represents the system executing the original program. The PLS with failure definition adds one more condition in the $\texttt{PLS}^{\texttt{eq}}_x(\sqsubseteq_x)$ predicate stating that if for any one step transition in $\texttt{LTS}_\Xi$, the corresponding state in $\texttt{LTS}_\Sigma$ is transitioned to an semantic error state, then the relation is still an allowed $\texttt{PLS}^{\texttt{eq}}_x(\sqsubseteq_x)$ relation. The PLS definition captures the compiler optimization semantic preservation property. The semantic meaning of the property when an execution of the original program reaches an error state is that the execution of the optimized program is allowed to do whatever it is allowed. This is what we defined for the $\texttt{PLS}^{\texttt{eq}}_x(\sqsubseteq_x)$ predicate to handle the error-state-reaching case.

8.2 PROGRAM MEANING PRESERVATION

Morpheus is a a domain-specific language for formal specification of program transformations. In previous papers about Morpheus [1, 98], it was shown how to combine a sequential memory model, the Morpheus framework, and an underlying instruction semantics for a programming language to prove the correctness of a traditional compiler optimization (PRE). This section introduces a combination of PLS, Morpheus, and the program semantics for the language in Fig. 6.4 (based on a weak memory model in Chapter 7) to prove an optimization semantically preserving the program meaning. The Morpheus specification language is introduced in Sec. 3.2. Here, we first introduce examples of optimizations specified in Morpheus. Then, we introduce the proof of the optimization by using the PLS definition in Fig. 8.10. The program semantics of the proof is based on **K-LLVM** with the operational model described in Chapter 7. Given an optimization ζ and program μ, we rewrite μ to μ' by ζ. The proof is to build a PLS relation from a LTS, whose states have the form (μ', ω) for any environment state ω with a fixed format (e.g. Fig. 7.1), to another LTS, whose states have the form (μ, ω).

8.2.1 Example Optimization Specifications

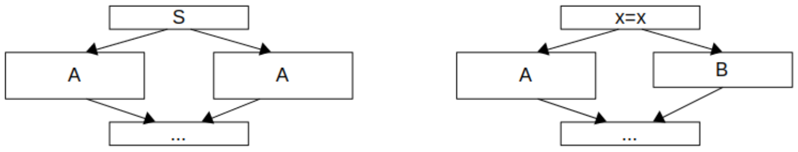


Figure 8.11: Examples of Simple Code Motion Optimizations

In this chapter, we use two kinds of simple code motion (SCM) optimizations as examples. The general strategies for them are shown as graphs in Fig. 8.11. Given a CFG C for a thread in a program μ, the left optimization in Fig. 8.11 locates (by a Morpheus condition expression) a basic block B of C, whose termination is a binary branching instruction and the two outgoing edges pointing to the two basic blocks B_1 and B_2 that have the same content and same outgoing edges. Then the left optimization changes the binary branching instruction in B to a non-conditional one, and also changes the edges of B to a single outgoing edge with a label `seq`. This is done by a strategy code in Morpheus with a sequence of graph transformations. Similarly, the right optimization first locates a basic block B of C whose termination is a binary branching instruction whose Boolean guard is always evaluated as `true` (by static rewriting). Then the optimization changes the binary branching instruction to an unconditional branching one `br`, and makes all of the outgoing edges of B point to the basic block indicated by the `true` branching of B.

$$
\begin{aligned}
\texttt{sameOutEdge}(a,b) \triangleq\ & \texttt{stmt}(a) = \texttt{stmt}(b) \wedge \texttt{sameEdges}(a,b) \\
& \vee \texttt{stmt}(a) = \texttt{stmt}(b) \wedge \neg\texttt{sameEdges}(a,b) \wedge \texttt{sameOutEdge}(\texttt{next}(a), \texttt{next}(b)) \\
\texttt{leftOpt}(\pi) \triangleq\ & \texttt{EXISTS}\ \pi_1\ \pi_2.\texttt{SATISFIED\_AT}\ \pi.\texttt{sameOutEdge}(\texttt{next}(\texttt{yes},\pi), \texttt{next}(\texttt{no},\pi)) \\
& ;\texttt{relabel\_node}(\pi, (\texttt{insts}(\pi), \texttt{br}));\texttt{move\_edge}((\pi,\texttt{no},\pi_2),(\texttt{seq},\pi_1)) \\
& ;\texttt{move\_edge}((\pi,\texttt{yes},\pi_1),(\texttt{seq},\pi_1)) \\
\texttt{rightOpt}(\pi) \triangleq\ & \texttt{EXISTS}\ a\ \pi_1\ \pi_2.\texttt{SATISFIED\_AT}\ \pi.\texttt{tem\_inst}(\pi) = a \wedge \texttt{eval}(a) = \texttt{true} \\
& ;\texttt{relabel\_node}(\pi, (\texttt{insts}(\pi), \texttt{br}));\texttt{move\_edge}((\pi,\texttt{no},\pi_2),(\texttt{seq},\pi_1)) \\
& ;\texttt{move\_edge}((\pi,\texttt{yes},\pi_1),(\texttt{seq},\pi_1))
\end{aligned}
$$

Figure 8.12: Simple Code Motion Transformations in Morpheus

Figure 8.12 contains the Morpheus formulas `leftOpt` and `rightOpt` defining the left and right compiler optimizations from Fig. 8.11. The `sameOutEdge` formula defines the predicate for checking if two statements are the same; and their children have the same outgoing edges or statements. a and b are two metavariables representing two nodes; the `stmt`(π) function gets the basic block represented by node π, and the `sameEdges` predicate checks if a and b have the same out going edges. `leftOpt` represents the left optimization in Fig. 8.11. It first searches a node π that has a binary branching instruction with two out going edges (defined by the `SATISFIED_AT` Morpheus strategy operation). The `next` function gets the outgoing node of π with a fixed edge label (`yes` or `no` in `leftOpt`). It does three actions: first, it replaces the termination of π with `br` (by the Morpheus `relabel_node` action); second, it changes the `no` edge of π to π_1 with the label `seq` (by the Morpheus `move_edge` action), and finally it exchanges the `yes` edge of π with the label `seq` (also by the Morpheus `move_edge` action). The `insts` function gets the instruction list in the basic block of π. The `rightOpt` formula implements the right optimization in Fig. 8.11. It is similar to `leftOpt`. The only difference is that it checks if the binary branching instruction in the basic block of node π has a Boolean guard that is always evaluated as `true` (by the `eval` function). The termination is retrieved by the `tem_inst` function, and the metavariable a represents the termination of π.

The semantics of Morpheus [1, 98] is basically the implementation of a graph rewrite algorithm over the FOCTL style conditions. Given an optimization formula (like Fig. 8.12) and a program μ, for every CFG C for a thread in μ, the algorithm generates a set of new CFGs. It first locates a basic block node satisfying the condition φ defined in a `SATISFIED_AT` strategy operation; and then it does a series of actions that change the structure of the CFG based on the node, as with the `relabel_node` and `move_edge` actions in `leftOpt`.

Here, we have briefly introduced Morpheus and given examples of optimizations defined therein. We will introduce the PLS proof in the next section.

8.2.2 The PLS Proof over Morpheus Optimizations

We utilize the optimizations and the program semantics defined in Chapter 5 and 7 to prove the correctness of a simple code motion optimization (SCM) as a utility of PLS. We want to show that any compiler-optimized (by SCM) program in the language (**K-LLVM**) per-loc simulates its original unoptimized program.

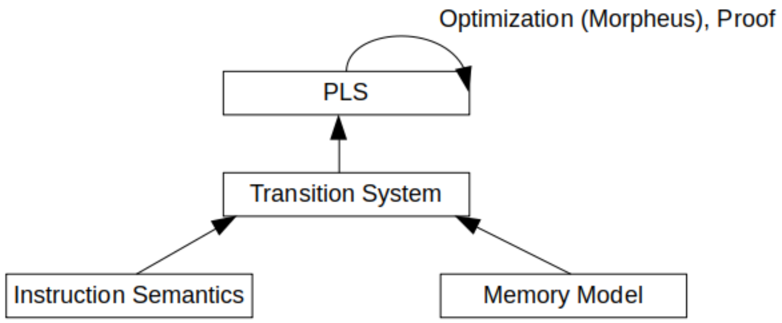


Figure 8.13: Optimization Proof with PLS

Fig. 8.13 provides the structure of the optimization proof. In Sec. 8.1.1, we described how the PLS framework is parameterized by transition systems. Here we instantiate these systems with the same program transition system in Chapter. 7. We then instantiate the states (ξ, σ and υ in Fig. 8.10) as the form (μ, ω). For any two states in a LTS ($\mathtt{LTS}_\Xi$, $\mathtt{LTS}_\Sigma$, or $\mathtt{LTS}_\Upsilon$), they have the same program μ. We also map the labels (α, β, and κ) to memory events ($\mathcal{E}v$). Given a label event ev, the property `val` is implemented as getting the value of the action in ev only if the action is a read or write; if it is a τ event, then the `val` answers $\perp$. `type` is implemented as a read for a read action in the event, as a write for a write action, and as τ for a τ action. `loc` is implemented as getting the memory location in the action of the event (if it is a τ event, then `loc` answers $\perp$). We keep the relation set `eq` the same as the one in Fig. 8.6a. Assume that a program μ is given, by applying the Morpheus optimization algorithm of SCM, we can rewrite μ as an optimized program μ'. For a fixed initial state ω, the PLS proof is to show that the LTS ($\mathtt{LTS}_\Xi$) with the initial state (μ', ω) per-loc simulates the LTS ($\mathtt{LTS}_\Sigma$) with the initial state (μ, ω), where there exists a per-loc simulation relation $\sqsubseteq$ for a finite set of locations Loc, such that $(\mu', \omega) \sqsubseteq (\mu, \omega)$. We formalize this result as Theorem 8.2, and the proof is done in Isabelle. The approach of the proof is first to prove a lemma with a similar structure but for only a single-threaded program with one CFG, and then prove Theorem 8.2 by using induction on the number of threads in the domain of the program.

Theorem 8.2. Let $(\mu, \omega)(x)$ ($\xi(x)$ or $\sigma(x)$) be the value of location x at the ω's heap snapshot (Γ in Fig. 6.24) that belongs to the thread tid such that $\rho(\mathtt{max}(T)) = (tid, aid, ac)$ (ρ and T are the elements in ω in Fig. 6.24). For any program μ in the language in Fig. 6.4 with a finite domain (Tid has size n), for any π and any $tid \in Tid$, let $\mu'(tid) \in \mathtt{leftOpt}(\pi)(\mu(tid))$ (or $\mu'(tid) \in \mathtt{rightOpt}(\pi)(\mu(tid))$).

Given a non-empty finite set of memory locations Loc and a given state environment ω, there exists a per-loc simulation $\sqsubseteq$ that satisfies $\mathtt{PLS}^{\mathsf{eq}}_{Loc}(\sqsubseteq)$ and $(\mu', \omega) \sqsubseteq_x (\mu, \omega)$ for all location x, and for all ξ and σ such that $\xi \sqsubseteq_x \sigma$ for a location x, $\xi(x) = \sigma(x)$.

8.3 ISABELLE FORMALIZATION OF THE FRAMEWORK AND THE PROOF

The PLS framework, the combination of PLS and Morpheus, and the proof of the semantic preservation of a particular optimization on a specific language are achieved through an elegant combination of different **locale** structures [161] in Isabelle. An Isabelle locale structure is a polymorphic theorem structure that is parameterized by a list of Isabelle terms with proper types and a list of assumptions for these terms. Through a locale structure, a collection of theorems can be defined for a list of polymorphic terms, provided that the terms satisfies the assumptions defined for the terms. Users can later instantiate the locale structure to a specific instance of terms by proving the assumptions.

In the Isabelle Morpheus definition, we first define the syntax for the Morpheus specification language. We then define a polymorphic CFG locale structure, named Flow_graph, with all necessary elements in a CFG, such as a set of nodes, a set of edges, the start node and the exit node for the CFG with several assumptions on the CFG well-formedness. The Morpheus specification language semantics is also defined as a locale structure, named Morpheus_sem, which is built on top of the Flow_graph locale. Based on the CFG structure and assumptions provided by flow_graph, Morpheus_sem defines an inductive relation capturing the graph rewriting semantics of Morpheus based on the polymorphic CFG structure (flow_graph).

Before we define PLS in Isabelle, we define an LTS locale for a polymorphic labeled transition system (LTS) with four properties with some well-formedness assumptions. Three of them are listed in Fig 8.2. The other one is the program text of the LTS described in Sec. 8.1.2. PLS is defined as a locale, named PLS, with two LTSs and an equation set eq as the input terms. The two LTSs are based on the LTS locale. In the PLS locale, we define a predicate as the one in Fig. 8.10, which defines the full PLS. When using PLS to prove language properties, one might be more interested in finding a PLS relation. To do that, we combine the Morpheus_sem and the PLS locales, as a new locale Morpheus_com, to build a PLS relation on top of the Morpheus semantics. In Morpheus_com, we build a new predicate $\texttt{sim}_x^{\texttt{eq}}\ \mu\ \mu'\ \texttt{steprel}\ n$, where μ and μ' are two programs (with the same thread domain), and μ' is the transformed program of μ through a specific optimization defined as an input term of Morpheus_com. `steprel` is a polymorphic function (defined as a term in a locale) to produce an LTS based on a program by omitting the implementation details of the LTS but only producing the four properties above. It takes in a program μ and a state ω, and outputs a label ev and a new state ω' transitioned from ω. The $\texttt{sim}_x^{\texttt{eq}}$ predicate is valid if and only if the LTS with the program text μ, and an initial state ω (μ, ω), per-loc simulates (with the equation set `eq`) the LTS with the program text μ, and an initial state ω (μ, ω) in n steps.

As an example (in Isabelle) of defining the optimization in Fig. 8.12 and proving Theorem 8.2 on **K-LLVM**, we first define its instruction and CFG syntax with a definition capturing the instruction level semantics of the language. We also define a memory model as a locale structure capturing the relaxed concurrency behaviors described in Chapter 6. We then define the program semantics as the LTS $(\rightarrow)$ as the LLVM Operational Model (Chapter 7) by instantiating the memory model locale with the language (Fig. 6.4) and adding more structures (like the program pointer family). Now, we instantiate the Morpheus_sem locale by the CFG syntax in **K-LLVM**, the Steprel locale (for instantiating the function `steprel`) by the LTS $(\rightarrow)$, and the `eq` set as the one in Fig. 8.6a, and the compiler optimization term in Morpheus_sem as the one in Fig. 8.12. The proof of Theorem 8.2 is then turned to show that the predicate $\texttt{sim}_x^{\texttt{eq}}\ \mu\ \mu'\ \texttt{steprel}\ n$ is valid for arbitrary x in a location set Loc. We first show the case when $n = \texttt{1}$ by proving for any one-step transition defined by the LTS $(\rightarrow)$ for arbitrary x; then, we lift the proof inductively to arbitrary n step based on the one-step proof result.

Chapter 9: CONCLUSION AND FUTURE WORK

Here, we provide conclusions and possible future studies on different components of the thesis.

9.1 CONCLUSION

We first introduce the platform used for defining language semantics and prove theorems, which is **IsaK** and **TransK**. **IsaK** is a formal semantics of $\mathbb{K}$ in Isabelle. It contains two part. The static semantics of **IsaK** describes the behavior of how $\mathbb{K}$ is transformed from FAST to BAST; especially, how the concision, localization and modularity are transformed into a well structured BAST format. In addition, we discuss some potential design issues related to the concision, localization and modularity with respect to the transformation, and suggest some design changes. In the static semantics, we proposed a sort system for $\mathbb{K}$ which is the first complete sort system for $\mathbb{K}$. All of these processes involved discussion with the $\mathbb{K}$ team to make sure our $\mathbb{K}$ formal semantics behaved correctly. We also examined **IsaK** by running tests against the extracted OCaml interpreter of **IsaK** in Isabelle and found that our system passed all 13 test specifications and 338 out of 356 programs for these test specifications. We discovered 25 major undesirable behaviors of $\mathbb{K}$. The dynamic semantics of **IsaK** is to capture the behavior of executing a program in a specification sequentially (***krun***) or in a multi-threaded way (***ksearch***) by giving the specification in $\mathbb{K}$. We also defined **TransK**, the shallow embedding of a $\mathbb{K}$ specification into an Isabelle theory, and showed that the execution of a program in the $\mathbb{K}$ specification is bisimilar to its Isabelle theory transalted by **TransK**. As an usage of **IsaK** and **TransK**, we defined **K-LLVM** in **IsaK**, and translate it to a Isabelle version through **TransK**. Then, we prove the semantic preservation property on a compiler optimization based on the Isabelle version of **K-LLVM**.

The second piece of the work is **K-LLVM**, which is a formal semantics of LLVM IR in $\mathbb{K}$. The main advantages of **K-LLVM** is its relatively completeness and its implementation via a novel abstract machine for LLVM IR. To the best of our knowledge, **K-LLVM** is the most complete formal semantics of LLVM IR. We fully define the static semantics and dynamic semantics of LLVM IR relative to a sequentially consistent memory model. To validate its completeness, we ran 1,385 unit testing and 2,156 concrete test programs, all of which **K-LLVM** successfully executed. **K-LLVM** provides guidance and reference to future compiler developers on exactly what are permissible behaviors in running LLVM IR programs. It also provides important piece of a framework for proving properties of compilers to or from LLVM IR. The **K-LLVM** abstract machine is a concise way of specifying how each LLVM IR instruction interacts with different computer components. In particular, **K-LLVM** covers corner cases and side-effects of instruction semantics that previous work does not have, such as the different cases of the `getelementptr` operators, casting operators, and memory operators. **K-LLVM** also supports multi-threaded behaviors and provides users a collection of tools, including a state-space searching tool to explore traces of their LLVM IR programs under the assumption of sequential consistency. While this was not the main focus of this work, we also found more than 20 bugs in the current LLVM implementation, Clang. Based on the abstract machine design of **K-LLVM**, we created the full LLVM IR memory model in **K-LLVM**, including the behaviors of different atomic memory orderings and `volatile` memory accesses, with heavy testings and proofs of its relationship with existing C++ memory models [2, 3, 110, 113, 162, 163] based on HATRMM.

The third piece of the work is HATRMM. The main reason to have HATRMM is to bridge the traditional axiomatic candidate relaxed memory models with relaxed operational memory models, which have been proved that the linkage is impossible to establish based on the traditional axiomatic candidate model [153]. We show that the equivalence of HATRMM with respect to the RC11 [3] and IMM [162] models. There are other features of HATRMM that previous models cannot achieve (Sec. 6.1). In addition, through the definition of HATRMM we have corrected some mistakes in the definitions of previous models. The special feature of HATRMM is its division of predicates describing concurrency behaviors based on the concept of an abstract machine: this feature implements one predicate to describe single-threaded behaviors and another to describe multi-threaded ones. By using HATRMM, we can prove that the **K-LLVM** operational model in Chapter 7 is a model whose behaviors are all valid behaviors in RC11 + IMM.

Finally, we propose a new per-location simulation (PLS) relation that is simple and suitable for proving a compiled program preserves its original program semantics under a CFG-based programming language with a real-world, C/C++ like, weak memory model. PLS can be divided into two parts (Sec. 8.1.1 and 8.1.2). With the failure predicate from the upward simulation in CompCertTSO [97], we are able to form PLS with the **K-LLVM** operational model (Chapter 7), to prove semantic preservation property of an optimization defined in **K-LLVM**. We have shown the utility of PLS by proving that program semantics is preserved for a simple code motion optimization (defined in Sec. 8.2.1) for all possible programs.

9.2 FUTURE WORK

There are many different future possibilities of the work. We can use **IsaK** and **TransK** to link reachability logic [78, 79, 164], the new program logic being derived by the $\mathbb{K}$ team, with traditional program logics such as Hoare Logic [80] and Separation Logic [165]. To do so, we first define reachability logic on top of **IsaK**, and then use **TransK** to translate a specification defined in **IsaK** + reachability logic to an Isabelle version. Then, we can prove properties about an specification in **IsaK** + reachability logic with respect to the traditional Separation Logic [165].

The modular abstract machine of **K-LLVM** allows us to define different memory models and libraries on top of **K-LLVM** without changing different instruction level semantics. For example, we can define the OpenSSL library based on **K-LLVM**, so that we are able to verify properties on programs with OpenSSL library functions.

For the future work of HATRMM + PLS, we plan to use them to prove that a wide range of compiler optimizations semantically preserve program meaning in **K-LLVM**. To do so, we will need to include compiler optimizations like partially redundant elimination, inline expansion, thread inlining, etc.

For the future work of the whole project, we plan to define the complete semantics of Haskell with its memory model (Software Transactional Memory model [166, 167]) based on **IsaK**. Then, we can translate Haskell in K to a version of Haskell in Isabelle through **TransK**. We then define the program transformation from Haskell to **K-LLVM** in Isabelle. Finally, we prove the compiler correctness from Haskell to **K-LLVM** by using PLS. We expect to extend HATRMM to include the Software Transactional Memory model, and also combine the **K-LLVM** operational model with the Haskell abstract machine in Isabelle. Eventually, we will have a complete verified compiler from Hasekell to LLVM IR.

BIBLIOGRAPHY

[1] W. Mansky, E. L. Gunter, D. Griffith, and M. D. Adams, "Specifying and executing optimizations for generalized control flow graphs," *Science of Computer Programming*, vol. 130, pp. 2–23, Nov. 2016.

[2] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing c++ concurrency," *SIGPLAN Not.*, vol. 46, no. 1, pp. 55–66, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1925844.1926394

[3] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in c/c++11," *SIGPLAN Not.*, vol. 52, no. 6, pp. 618–632, June 2017. [Online]. Available: http://doi.acm.org/10.1145/3140587.3062352

[4] A. Podkopaev, O. Lahav, and V. Vafeiadis, "Bridging the Gap Between Programming Languages and Hardware Weak Memory Models," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 69:1–69:31, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290382

[5] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993532 pp. 283–294.

[6] E. Gunter and G. Roşu, "Verif-opt project," 2013. [Online]. Available: https://www.nsf.gov/awardsearch/showAward?AWD_ID=1318191

[7] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[8] W. Mansky and E. L. Gunter, "Verifying Optimizations for Concurrent Programs," in *First International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, ser. OpenAccess Series in Informatics (OASIcs), M. Schmidt-Schauß, M. Sakai, D. Sabel, and Y. Chiba, Eds., vol. 40. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2014/4586 pp. 15–26.

[9] L. Li and E. L. Gunter, "IsaK-Static: A Complete Static Semantics of K," in *Formal Aspects of Component Software*, K. Bae and P. C. Ölveczky, Eds. Cham: Springer International Publishing, 2018, pp. 196–215.

[10] L. Li and E. Gunter, "A complete semantics of k," 2017. [Online]. Available: https://github.com/liyili2/k-semantics

[11] L. Li and E. Gunter, "A method to translate order-sorted algebras to many-sorted algebras," in *Proceedings of the Fourth International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, ser. WPTE '17. EPTCS, 2017.

[12] F. L. Morris, “Advice on structuring compilers and proving them correct,” in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL ’73. New York, NY, USA: ACM, 1973. [Online]. Available: http://doi.acm.org/10.1145/512927.512941 pp. 144–152.

[13] J. S. Moore, “A mechanically verified language implementation,” *J. Autom. Reason.*, vol. 5, no. 4, pp. 461–492, 1989.

[14] J. S. Moore, *Piton: a mechanically verified assembly-level language.* Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[15] W. Young, “Verified compilation in micro-Gypsy,” *SIGSOFT Softw. Eng. Notes*, vol. 14, pp. 20–26, November 1989. [Online]. Available: http://doi.acm.org/10.1145/75309.75312

[16] M. V. Inwegen and E. L. Gunter, “HOL-ML,” in *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications.* London, UK: Springer-Verlag, 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=646520.694367 pp. 61–74.

[17] S. Maharaj and E. L. Gunter, “Studying the ML module system in HOL,” in *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications.* London, UK: Springer-Verlag, 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=646521.759249 pp. 346–361.

[18] Y. Bertot and P. Casteran, *Interactive Theorem Proving and Program Development.* SpringerVerlag, 2004.

[19] L. C. Paulson, “Isabelle: The next 700 theorem provers,” in *Logic and Computer Science*, P. Odifreddi, Ed. Academic Press, 1990, pp. 361–386.

[20] M. Broy, U. Hinkel, T. Nipkow, C. Prehofer, and B. Schieder, “Interpreter verification for a functional language,” in *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science.* London, UK: Springer-Verlag, 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=646832.707885 pp. 77–88.

[21] T. Nipkow, “Verified lexical analysis,” in *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics.* London, UK: Springer-Verlag, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=646525.694868 pp. 1–15.

[22] J. O. Blech and S. Glesner, “A formal correctness proof for code generation from SSA form in Isabelle/HOL,” in *Proceedings der 3. Arbeitstagung Programmiersprachen (ATPS) auf der 34. Jahrestagung der Gesellschaft für Informatik.* Lecture Notes in Informatics, September 2004. [Online]. Available: http://www.info.uni-karlsruhe.de/papers/Blech-Glesner-ATPS-2004.pdf

[23] G. Klein and T. Nipkow, "Verified bytecode verifiers," *Theor. Comput. Sci.*, vol. 298, pp. 583–626, April 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=773691.773699

[24] D. Leinenbach, W. Paul, and E. Petrova, "Towards the formal verification of a C0 compiler: Code generation and implementation correctnes," in *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*. Washington, DC, USA: IEEE Computer Society, 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1109722.1110555 pp. 2–12.

[25] D. Leinenbach and E. Petrova, "Pervasive compiler verification – from verified programs to verified systems," *Electron. Notes Theor. Comput. Sci.*, vol. 217, pp. 23–40, July 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1390859.1390957

[26] M. Strecker, "Formal verification of a Java compiler in Isabelle," in *Proceedings of the 18th International Conference on Automated Deduction*, ser. CADE-18. London, UK: Springer-Verlag, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=648238.751554 pp. 63–77.

[27] L. Gesellensetter, S. Glesner, and E. Salecker, "Formal verification with Isabelle/HOL in practice: finding a bug in the GCC scheduler," in *Proceedings of the 12th international conference on Formal methods for industrial critical systems*, ser. FMICS'07. Berlin, Heidelberg: Springer-Verlag, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1793603.1793613 pp. 85–100.

[28] G. Klein and T. Nipkow, "A machine-checked model for a Java-like language, virtual machine and compiler," vol. 28, no. 4, pp. 619–695, 2006.

[29] A. Lochbihler, "Java and the java memory model – a unified, machine-checked formalisation," in *Programming Languages and Systems*, ser. LNCS, H. Seidl, Ed., vol. 7211. Springer, Mar. 2012. [Online]. Available: http://pp.info.uni-karlsruhe.de/uploads/publikationen/lochbihler12esop.pdf pp. 497–517.

[30] G. Barthe, P. Courtieu, G. Dufay, and S. a. M. d. Sousa, "Tool-assisted specification and verification of the JavaCard platform," in *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, ser. AMAST '02. London, UK, UK: Springer-Verlag, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=646061.676163 pp. 41–59.

[31] R. Atkey, "CoqJVM: an executable specification of the Java virtual machine using dependent types," in *Proceedings of the 2007 international conference on Types for proofs and programs*, ser. TYPES'07. Berlin, Heidelberg: Springer-Verlag, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1786134.1786136 pp. 18–32.

[32] L. Rideau, B. P. Serpette, and X. Leroy, "Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves," *J. Autom. Reason.*, vol. 40, pp. 307–326, May 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1363304.1363311

[33] Z. Dargaye and X. Leroy, "Mechanized verification of CPS transformations," in *Proceedings of the 14th international conference on Logic for programming, artificial intelligence and reasoning*, ser. LPAR'07. Berlin, Heidelberg: Springer-Verlag, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1779419.1779436 pp. 211–225.

[34] A. Chlipala, "A certified type-preserving compiler from lambda calculus to assembly language," *SIGPLAN Not.*, vol. 42, pp. 54–65, June 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250742

[35] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages.* New York, NY, USA: ACM, 2006, pp. 42–54.

[36] X. Leroy, "A formally verified compiler back-end," *J. Autom. Reason.*, vol. 43, no. 4, pp. 363–446, 2009.

[37] X. Leroy, "A Formally Verified Compiler Back-end," *J. Autom. Reason.*, vol. 43, no. 4, pp. 363–446, Dec. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10817-009-9155-4

[38] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems.* London, UK: Springer-Verlag, 1998, pp. 151–166.

[39] Z. Tatlock and S. Lerner, "Bringing extensibility to verified compilers," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806611 pp. 111–121.

[40] A. McCreight, T. Chevalier, and A. Tolmach, "A certified framework for compiling and executing garbage-collected languages," in *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ser. ICFP '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1863543.1863584 pp. 273–284.

[41] X. Leroy and S. Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations," *J. Autom. Reason.*, vol. 41, pp. 1–31, July 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1388522.1388533

[42] J. Ŝevčik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, "Relaxed-memory concurrency and verified compilation," *SIGPLAN Not.*, vol. 46, no. 1, pp. 43–54, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1925844.1926393

[43] S. Blazy and X. Leroy, "Mechanized Semantics for the Clight Subset of the C Language," *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10817-009-9148-3

[44] A. W. Appel, "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*, ser. ESOP'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987211.1987212 pp. 1–17.

[45] A. W. Appel, "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*, ser. ESOP'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987211.1987212 pp. 1–17.

[46] A. Hobor, A. W. Appel, and F. Z. Nardelli, "Oracle semantics for concurrent separation logic," in *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, ser. ESOP'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792878.1792914 pp. 353–367.

[47] A. Dold, F. W. v. Henke, H. Pfeifer, and H. Rueß, "Formal verification of transformations for peephole optimization," in *Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods*, ser. FME '97. London, UK: Springer-Verlag, 1997. [Online]. Available: http://dl.acm.org/citation.cfm?id=647538.729832 pp. 459–472.

[48] A. Kanade, A. Sanyal, and U. Khedker, "A PVS based framework for validating compiler optimizations," in *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods.* Washington, DC, USA: IEEE Computer Society, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1158333.1158356 pp. 108–117.

[49] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen, "Proving correctness of compiler optimizations by temporal logic," *SIGPLAN Not.*, vol. 37, no. 1, pp. 283–294, Jan. 2002. [Online]. Available: http://doi.acm.org/10.1145/565816.503299

[50] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen, "Compiler optimization correctness by temporal logic," *Higher Order Symbol. Comput.*, vol. 17, pp. 173–206, September 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=993034.993038

[51] S. Kalvala, R. Warburton, and D. Lacey, "Program transformations using temporal logic side conditions," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 4, pp. 14:1–14:48, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1516507.1516509

[52] S. Kalvala and R. Warburton, "A formal approach to fixing bugs." in *SBMF*, ser. Lecture Notes in Computer Science, A. da Silva Simão and C. Morgan, Eds., vol. 7021. Springer, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/sbmf/sbmf2011.html\#KalvalaW11 pp. 172–187.

[53] S. Lerner, T. Millstein, and C. Chambers, "Automatically proving the correctness of compiler optimizations," *SIGPLAN Not.*, vol. 38, pp. 220–231, May 2003. [Online]. Available: http://doi.acm.org/10.1145/780822.781156

[54] S. Lerner, T. Millstein, E. Rice, and C. Chambers, "Automated soundness proofs for dataflow analyses and transformations via local rules," *SIGPLAN Not.*, vol. 40, pp. 364–377, January 2005. [Online]. Available: http://doi.acm.org/10.1145/1047659.1040335

[55] U. Aßmann, "How to uniformly specify program analysis and transformation with graph rewrite systems," in *Proceedings of the 6th International Conference on Compiler Construction.* London, UK: Springer-Verlag, 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=647473.727442 pp. 121–135.

[56] U. Aßmann, *OPTIMIX – a tool for rewriting and optimizing programs.* River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1999, pp. 307–318. [Online]. Available: http://dl.acm.org/citation.cfm?id=328523.328601

[57] R. Milner, M. Tofte, and D. Macqueen, *The Definition of Standard ML.* Cambridge, MA, USA: MIT Press, 1997.

[58] D. K. Lee, K. Crary, and R. Harper, "Towards a Mechanized Metatheory of Standard ML," *SIGPLAN Not.*, vol. 42, no. 1, pp. 173–184, Jan. 2007. [Online]. Available: http://doi.acm.org/10.1145/1190215.1190245

[59] S. Maharaj and E. Gunter, *Studying the ML module system in HOL.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 346–361. [Online]. Available: http://dx.doi.org/10.1007/3-540-58450-1_53

[60] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* New York, NY, USA: Cambridge University Press, 1993.

[61] M. Norrish, "C formalised in hol," Tech. Rep., 1998.

[62] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the LLVM Intermediate Representation for Verified Program Transformations," *SIGPLAN Not.*, vol. 47, no. 1, pp. 427–440, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2103621.2103709

[63] J. A. Goguen, J.-P. Jouannaud, and J. Meseguer, "Operational Semantics for Order-Sorted Algebra," in *Proceedings of the 12th Colloquium on Automata, Languages and Programming.* London, UK, UK: Springer-Verlag, 1985. [Online]. Available: http://dl.acm.org/citation.cfm?id=646239.683375 pp. 221–231.

[64] M. Alpuente, S. Escobar, J. Espert, and J. Meseguer, "A Modular Order-sorted Equational Generalization Algorithm," *Inf. Comput.*, vol. 235, pp. 98–136, Apr. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.ic.2014.01.006

[65] H. Comon, *Equational formulas in order-sorted algebras.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 674–688. [Online]. Available: http://dx.doi.org/10.1007/BFb0032066

[66] J. A. Goguen and J. Meseguer, "Order-sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations," *Theor. Comput. Sci.*, vol. 105, no. 2, pp. 217–273, Nov. 1992. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(92)90302-V

[67] C. Kirchner, H. Kirchner, and J. Meseguer, *Operational semantics of OBJ-3.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 287–301. [Online]. Available: https://doi.org/10.1007/3-540-19488-6_123

[68] J. Meseguer, J. A. Goguen, and G. Smolka, "Order-sorted Unification," *J. Symb. Comput.*, vol. 8, no. 4, pp. 383–413, Oct. 1989. [Online]. Available: http://dx.doi.org/10.1016/S0747-7171(89)80036-7

[69] N. Martí-Oliet and J. Meseguer, "Rewriting Logic: Roadmap and Bibliography," *Theoretical Computer Science*, vol. 285, no. 2, pp. 121 – 154, 2002, rewriting Logic and its Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397501003577

[70] J. Meseguer, "Research Directions in Rewriting Logic," in *Computational Logic*, U. Berger and H. Schwichtenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 347–398.

[71] P. L. M. Clavel, S. Eker and J. Meseguer, "Principles of maude," in *Electronic Notes in Theoretical Computer Science*, J. Meseguer, Ed., vol. 4. Elsevier Science Publishers, 2000.

[72] J. Meseguer, "Software Specification and Verification in Rewriting Logic," *NATO SCIENCE SERIES SUB SERIES III COMPUTER AND SYSTEMS SCIENCES*, vol. 191, pp. 133–194, 2003.

[73] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, and C. Talcott, "Pathway Logic: Executable Models of Biological Networks," in *Fourth International Workshop on Rewriting Logic and Its Applications (WRLA 2002), Pisa, Italy, September 19 — 21, 2002*, ser. Electronic Notes in Theoretical Computer Science, vol. 71. Elsevier, 2002, http://www.elsevier.nl/locate/entcs/volume71.html.

[74] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL Model Checker and Its Implementation," in *Proceedings of the 10th International Conference on Model Checking Software*, ser. SPIN'03. Berlin, Heidelberg: Springer-Verlag, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=1767111.1767127 pp. 230–234.

[75] C. Beierle and G. Meyer, "Run-time Type Computations in The Warren Abstract Machine," *The Journal of Logic Programming*, vol. 18, no. 2, pp. 123 – 148, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0743106694900493

[76] G. Roşu and T. F. Şerbănuţă, “An Overview of the K Semantic Framework,” *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[77] G. Roşu and A. Ştefănescu, “Matching logic: A new program verification approach,” in *Proceedings of the 2010 Workshop on Usable Verification (UV’10)*. Microsoft Research, 2010.

[78] G. Roşu, A. Ştefănescu, c. Ciobâcă, and B. M. Moore, “One-path reachability logic,” in *Proceedings of the 28th Symposium on Logic in Computer Science (LICS’13)*. IEEE, June 2013, pp. 358–367.

[79] A. Ştefănescu, c. Ciobâcă, R. Mereuţă, B. M. Moore, T. F. Şerbănuţă, and G. Roşu, “All-path reachability logic,” in *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA’14)*, ser. LNCS, vol. 8560. Springer, July 2014, pp. 425–440.

[80] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[81] B. Moore and G. Roşu, “Program verification by coinduction,” University of Illinois, Tech. Rep. http://hdl.handle.net/2142/73177, February 2015.

[82] P. Corbineau, *A Declarative Language for the Coq Proof Assistant*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 69–84. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68103-8_5

[83] C. Ellison and G. Rosu, “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, January 2012, pp. 533–544.

[84] D. Filaretti and S. Maffeis, *An Executable Formal Semantics of PHP*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 567–592. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-44202-9_23

[85] D. Park, A. Ştefănescu, and G. Roşu, “KJS: A Complete Formal Semantics of JavaScript,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’15)*. ACM, June 2015, pp. 346–356.

[86] D. Bogdănaş and G. Roşu, “K-Java: A Complete Semantics of Java,” in *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL’15)*. ACM, January 2015, pp. 445–456.

[87] J. Meseguer and S. Skeirik, “Equational Formulas and Pattern Operations in Initial Order-sorted Algebras,” *Formal Aspects of Computing*, vol. 29, no. 3, pp. 423–452, May 2017. [Online]. Available: http://dx.doi.org/10.1007/s00165-017-0415-5

[88] H. Wang, “Logic of many-sorted theories,” *Journal of Symbolic Logic*, vol. 17, no. 2, pp. 105–116, 1952.

[89] R. M. Tomasulo, “An efficient algorithm for exploiting multiple arithmetic units,” *IBM J. Res. Dev.*, vol. 11, no. 1, pp. 25–33, Jan. 1967. [Online]. Available: http://dx.doi.org/10.1147/rd.111.0025

[90] M. Pérache, H. Jourdren, and R. Namyst, “Mpc: A unified parallel runtime for clusters of numa machines,” in *Euro-Par 2008 – Parallel Processing*, E. Luque, T. Margalef, and D. Benítez, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 78–88.

[91] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, “Fractal: An execution model for fine-grain nested speculative parallelism,” *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 587–599, June 2017. [Online]. Available: http://doi.acm.org/10.1145/3140659.3080218

[92] C. Tian, M. Feng, V. Nagarajan, and R. Gupta, “Copy or discard execution model for speculative parallelization on multicores,” in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008. [Online]. Available: https://doi.org/10.1109/MICRO.2008.4771802 pp. 330–341.

[93] L. Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,” *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[94] S. V. Adve and M. D. Hill, “Weak Ordering – A New Definition,” *SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, pp. 2–14, May 1990.

[95] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, “Causal Memory: Definitions, Implementation, and Programming,” *Distributed Computing*, vol. 9, no. 1, pp. 37–49, Mar 1995.

[96] L. Higham, J. Kawash, and N. Verwaal, “Weak Memory Consistency Models. Part I: Definitions and Comparisons,” Department of Computer Science, The University of Calgary, Tech. Rep., 1998.

[97] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, “CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency,” *J. ACM*, vol. 60, no. 3, pp. 22:1–22:50, June 2013. [Online]. Available: http://doi.acm.org/10.1145/2487241.2487248

[98] W. Mansky, D. Garbuzov, and S. Zdancewic, “An Axiomatic Specification for Sequential Memory Models,” in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 413–428.

[99] C. SPARC International, Inc., *The SPARC Architecture Manual: Version 8*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

[100] J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, pp. 7:1–7:74, July 2014. [Online]. Available: http://doi.acm.org/10.1145/2627752

[101] H.-J. Boehm and S. V. Adve, "Foundations of the c++ concurrency memory model," *SIGPLAN Not.*, vol. 43, no. 6, pp. 68–78, June 2008. [Online]. Available: http://doi.acm.org/10.1145/1379022.1375591

[102] M. Dodds, M. Batty, and A. Gotsman, "C/C++ Causal Cycles Confound Compositionality," *TinyToCS*, vol. 2, 2013. [Online]. Available: http://tinytocs.org/vol2/papers/tinytocs2-dodds.pdf

[103] V. Vafeiadis and C. Narayan, "Relaxed Separation Logic: A Program Logic for C11 Concurrency," *SIGPLAN Not.*, vol. 48, no. 10, pp. 867–884, Oct. 2013.

[104] B. Norris and B. Demsky, "Cdschecker: Checking concurrent data structures written with c/c++ atomics," *SIGPLAN Not.*, vol. 48, no. 10, pp. 131–150, Oct. 2013.

[105] H.-J. Boehm and B. Demsky, "Outlawing Ghosts: Avoiding Out-of-thin-air Results," in *Proceedings of the Workshop on Memory Systems Performance and Correctness*, ser. MSPC '14. New York, NY, USA: ACM, 2014, pp. 7:1–7:6.

[106] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli, "Common compiler optimisations are invalid in the c11 memory model and what we can do about it," *SIGPLAN Not.*, vol. 50, no. 1, pp. 209–220, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2775051.2676995

[107] M. Batty, A. F. Donaldson, and J. Wickerson, "Overhauling SC Atomics in C11 and OpenCL," *SIGPLAN Not.*, vol. 51, no. 1, pp. 634–648, Jan. 2016.

[108] Y. Meshman, N. Rinetzky, and E. Yahav, "Pattern-based Synthesis of Synchronization for the C++ Memory Model," in *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '15. Austin, TX: FMCAD Inc, 2015, pp. 120–127.

[109] O. Lahav and V. Vafeiadis, "Owicki-gries reasoning for weak memory models," in *Proceedings, Part II, of the 42Nd International Colloquium on Automata, Languages, and Programming - Volume 9135*, ser. ICALP 2015. Berlin, Heidelberg: Springer-Verlag, 2015, pp. 311–323.

[110] O. Lahav, N. Giannarakis, and V. Vafeiadis, "Taming release-acquire consistency," *SIGPLAN Not.*, vol. 51, no. 1, pp. 649–662, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2914770.2837643

[111] J. Pichon-Pharabod and P. Sewell, "A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions," *SIGPLAN Not.*, vol. 51, no. 1, pp. 622–633, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2914770.2837616

[112] A. Jeffrey and J. Riely, "On Thin Air Reads Towards an Event Structures Model of Relaxed Memory," in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2933575.2934536 pp. 759–767.

[113] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer, "A Promising Semantics for Relaxed-memory Concurrency," *SIGPLAN Not.*, vol. 52, no. 1, pp. 175–189, Jan. 2017. [Online]. Available: http://doi.acm.org/10.1145/3093333.3009850

[114] S. Chakraborty and V. Vafeiadis, "Formalizing the concurrency semantics of an llvm fragment," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049832.3049844 pp. 100–110.

[115] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, "Taming Undefined Behavior in LLVM," *SIGPLAN Not.*, vol. 52, no. 6, pp. 633–647, June 2017. [Online]. Available: http://doi.acm.org/10.1145/3140587.3062343

[116] C. Ellison and D. Lazar, "The LLVM Semantics in K," 2015, K-Framework. [Online]. Available: https://github.com/kframework/llvm-semantics-old

[117] J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis, "A Formal C Memory Model Supporting Integer-pointer Casts," *SIGPLAN Not.*, vol. 50, no. 6, pp. 326–335, June 2015. [Online]. Available: http://doi.acm.org/10.1145/2813885.2738005

[118] J. Lee, C.-K. Hur, R. Jung, Z. Liu, J. Regehr, and N. P. Lopes, "Reconciling high-level optimizations and low-level code in llvm," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 125:1–125:28, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276495

[119] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell, "Exploring c semantics and pointer provenance," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 67:1–67:32, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290380

[120] M. Bodin, A. Chargueraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, "A Trusted Mechanised JavaScript Specification," *SIGPLAN Not.*, vol. 49, no. 1, pp. 87–100, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2578855.2535876

[121] P. Sewell, F. z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. StrniŠa, "Ott: Effective tool support for the working semanticist," *J. Funct. Program.*, vol. 20, no. 1, pp. 71–122, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1017/S0956796809990293

[122] S. Drossopoulou, S. Eisenbach, and S. Khurshid, "Is the Java Type System Sound?" *Theor. Pract. Object Syst.*, vol. 5, no. 1, pp. 3–24, Jan. 1999. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1096-9942(199901/03)5:1<3::AID-TAPO2>3.0.CO;2-T

[123] D. Syme, "Proving Java Type Soundness," in *Formal Syntax and Semantics of Java*. London, UK, UK: Springer-Verlag, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=645580.658814 pp. 83–118.

[124] E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk, "A High-level Modular Definition of the Semantics of C#," *Theor. Comput. Sci.*, vol. 336, no. 2-3, pp. 235–284, May 2005. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2004.11.008

[125] A. Farzan, F. Chen, J. Meseguer, and G. Roşu, *Formal Analysis of Java Programs in JavaFAN*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 501–505. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-27813-9_46

[126] W. Mansky and E. Gunter, "A framework for formal verification of compiler optimizations," in *Interactive Theorem Proving*, M. Kaufmann and L. C. Paulson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 371–386.

[127] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, ser. SFCS '77. Washington, DC, USA: IEEE Computer Society, 1977. [Online]. Available: http://dx.doi.org/10.1109/SFCS.1977.32 pp. 46–57.

[128] M. Ben-Ari, Z. Manna, and A. Pnueli, "The temporal logic of branching time," in *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1981, pp. 164–176.

[129] A. Pnueli, O. Shtrichman, and M. Siegel, "Translation validation for synchronous languages," in *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, ser. ICALP '98. London, UK: Springer-Verlag, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=646252.686146 pp. 235–246.

[130] G. C. Necula, "Translation validation for an optimizing compiler," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, ser. PLDI '00. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/349299.349314 pp. 83–94.

[131] L. Zuck, A. Pnueli, and R. Leviathan, "Validation of optimizing compilers," Weizmann Institute of Science, Jerusalem, Israel, Israel, Tech. Rep., 2001.

[132] R. Leviathan and A. Pnueli, "Validating software pipelining optimizations," in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '02. New York, NY, USA: ACM, 2002. [Online]. Available: http://doi.acm.org/10.1145/581630.581676 pp. 280–287.

[133] S. Kalvala, R. Warburton, and D. Lacey, "Program transformations using temporal logic side conditions," *ACM Trans. Program. Lang. Syst.*, vol. 31, 05 2009.

[134] N. Martí-Oliet and J. Meseguer, "Rewriting logic as a logical and semantic framework," in *Electronic Notes in Theoretical Computer Science*, J. Meseguer, Ed., vol. 4. Elsevier Science Publishers, 2000.

[135] L. Li and E. Gunter, "LLVM Semantics," 2016. [Online]. Available: https://github.com/kframework/llvm-semantics

[136] G. Roşu, *K Publications*, 2017, http://www.kframework.org/index.php/K_Publications.

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

[138] T. F. Şerbănuţă, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Roşu, "The k primer (version 3.3)," *Electronic Notes in Theoretical Computer Science*, vol. 304, no. Supplement C, pp. 57 – 80, 2014, proceedings of the Second International Workshop on the K Framework and its Applications (K 2011). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066114000395

[139] D. Bogdanas, "Label-based programming language semantics in k framework with sdf," in *Proceedings of the 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, ser. SYNASC '12. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: http://dx.doi.org/10.1109/SYNASC.2012.23 pp. 160–167.

[140] L. Li and E. Gunter, "K-LLVM: A Relatively Complete Semantics of LLVM IR," in *34rd European Conference on Object-Oriented Programming, ECOOP 2020, July 13-17, 2020, Berlin, Germany*, ser. LIPIcs, A. F. Donaldson, Ed. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[141] C. Matache, V. B. F. Gomes, and D. P. Mulligan, "The LambdaMu-calculus," *Archive of Formal Proofs*, vol. 2017, 2017. [Online]. Available: https://www.isa-afp.org/entries/LambdaMu.html

[142] llvm.org, "LLVM Language Reference Manual," 2018. [Online]. Available: http://releases.llvm.org/6.0.0/docs/LangRef.html

[143] P. E. Mckenney, "Memory Barriers: a Hardware View for Software Hackers," 2009.

[144] G. A. Kildall, "A Unified Approach to Global Program Optimization," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '73. New York, NY, USA: ACM, 1973. [Online]. Available: http://doi.acm.org/10.1145/512927.512945 pp. 194–206.

[145] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, "Into the depths of c: Elaborating the de facto standards," *SIGPLAN Not.*, vol. 51, no. 6, p. 1–15, June 2016. [Online]. Available: https://doi.org/10.1145/2980983.2908081

[146] K. Memarian and P. Sewell, "N2090: Clarifying pointer provenance," 2016. [Online]. Available: https://www.cl.cam.ac.uk/~pes20/cerberus/n2090.html

[147] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-tso: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, July 2010.

[148] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams, "An Axiomatic Memory Model for POWER Multiprocessors," in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 495–512.

[149] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, "Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 19:1–19:29, Dec. 2017.

[150] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell, "Clarifying and compiling c/c++ concurrency: From c++11 to power," *SIGPLAN Not.*, vol. 47, no. 1, p. 509–520, Jan. 2012. [Online]. Available: https://doi.org/10.1145/2103621.2103717

[151] L. Li and E. L. Gunter, "Per-Location Simulation," in *NASA Formal Methods*, R. Lee, S. Jha, and A. Mavridou, Eds. Cham: Springer International Publishing, 2020, pp. 267–287.

[152] K. Nienhuis, K. Memarian, and P. Sewell, "An operational semantics for c/c++11 concurrency," *SIGPLAN Not.*, vol. 51, no. 10, pp. 111–128, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/3022671.2983997

[153] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell, "The Problem of Programming Language Concurrency Semantics," in *Programming Languages and Systems*, J. Vitek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 283–307.

[154] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli, "The Semantics of Power and ARM Multiprocessor Machine Code," in *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, ser. DAMP '09. New York, NY, USA: ACM, 2008, pp. 13–24.

[155] S. Flur, S. Sarkar, C. Pulte, K. Nienhuis, L. Maranget, K. E. Gray, A. Sezgin, M. Batty, and P. Sewell, "Mixed-Size Concurrency: ARM, POWER, C/C++11, and SC," *SIGPLAN Not.*, vol. 52, no. 1, p. 429–442, Jan. 2017. [Online]. Available: https://doi.org/10.1145/3093333.3009839

[156] cppreference.com, "The C++11 Standard," 2018, cppreference.com. [Online]. Available: https://www.cppreference.com

[157] V. Vafeiadis and C. Narayan, "Relaxed separation logic: A program logic for c11 concurrency," *SIGPLAN Not.*, vol. 48, no. 10, pp. 867–884, Oct. 2013. [Online]. Available: http://doi.acm.org/10.1145/2544173.2509532

[158] H.-J. Boehm, "Memory Model Rationales," http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2176.html, accessed: 2007-03-09.

[159] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Z. Nardelli, "Common compiler optimisations are invalid in the C11 memory model and what we can do about it," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, S. K. Rajamani and D. Walker, Eds. ACM, 2015. [Online]. Available: https://doi.org/10.1145/2676726.2676995 pp. 209–220.

[160] J. Manson, W. Pugh, and S. V. Adve, "The java memory model," *SIGPLAN Not.*, vol. 40, no. 1, pp. 378–391, Jan. 2005. [Online]. Available: http://doi.acm.org/10.1145/1047659.1040336

[161] C. Ballarin, "Tutorial to locales and locale interpretation," *Contribuciones científicas en honor de Mirian Andrés Gómez, 2010-01-01, ISBN 978-84-96487-50-5, pags. 123-140*, 01 2010.

[162] A. Podkopaev, O. Lahav, and V. Vafeiadis, "Bridging the gap between programming languages and hardware weak memory models," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 69:1–69:31, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290382

[163] S. Chakraborty and V. Vafeiadis, "Grounding thin-air reads with event structures," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 70:1–70:28, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290383

[164] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based program verifiers for all languages," in *Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, Nov 2016, pp. 74–91.

[165] J. C. Reynolds, "Separation Logic: a Logic for Shared Mutable Data Structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, July 2002, pp. 55–74.

[166] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95. New York, NY, USA: ACM, 1995. [Online]. Available: http://doi.acm.org/10.1145/224964.224987 pp. 204–213.

[167] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy, “Composable memory transactions,” *Commun. ACM*, vol. 51, no. 8, pp. 91–100, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1378704.1378725

%inputappendix.tex