IsaK: A Complete Semantics of $\mathbb{K}$

LIYI LI, University of Illinois at Urbana-Champaign
ELSA L. GUNTER, University of Illinois at Urbana-Champaign

$\mathbb{K}$ (Roşu and Şerbănuţă, 2010) is a rewrite-based executable semantic framework in which programming languages, type systems and formal analysis tools can be defined using configurations, computations and rules. In this paper we define **IsaK**, a reference semantics for $\mathbb{K}$, which was developed through discussion with the $\mathbb{K}$ team to meet their expectations of a semantics of $\mathbb{K}$. **IsaK** is defined in the interactive theorem prover Isabelle/HOL (Paulson, 1990), and, to the best of our knowledge, is the most complete of any existing $\mathbb{K}$ specification. **IsaK** defines the full behavior of $\mathbb{K}$, a useful sort system for $\mathbb{K}$ and suggests several undesirable behaviors in the current $\mathbb{K}$ implementations ($\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0). We also provide an OCaml based executable $\mathbb{K}$ interpreter generated automatically from the $\mathbb{K}$ specification in Isabelle. By using a predefined $\mathbb{K}$ parser, the $\mathbb{K}$ interpreter is suitable to interpret major $\mathbb{K}$ definitions for large languages such as the LLVM semantics in $\mathbb{K}$, the Java semantics in $\mathbb{K}$ and the C semantics in $\mathbb{K}$. We ran a test suite including 13 specifications and 356 programs to test our $\mathbb{K}$ interpreter and we are able to compile all 13 specifications and run the 338 programs not requiring keyboard input. As a utility of **IsaK**, we also formally prove a CTL application in **IsaK** correct.

Additional Key Words and Phrases: K framework, formal semantics, Isabelle

1 MOTIVATION

$\mathbb{K}$ is a successful tool for defining programming language semantics and allowing users to view the behaviors of executing programs based on their language definitions. More than twenty papers have been published about the background and underlining theories of $\mathbb{K}$ as well as its tools (Roşu and Şerbănuţă, 2010; Şerbanuţă and Roşu, 2010; Lucanu et al., 2012; Ellison et al., 2009; Hills and Roşu, 2008; Şerbănuţă et al., 2009; Hills et al., 2008; Roşu and Ştefănescu, 2011; Arusoaie et al., 2012; Meseguer and Roşu, 2011; Ştefănescu et al., 2016, 2014; Roşu et al., 2013). A lot of popular programming language semantics have been defined and explored largely or fully in $\mathbb{K}$, such as the Java semantics (Bogdănaş and Roşu, 2015), the Javascript semantics, (Park et al., 2015) the PHP semantics (Filaretti and Maffeis, 2014), the C semantics (Ellison and Rosu, 2012; Hathhorn et al., 2015), the LLVM semantics (Li and Gunter, 2016) and the Python semantics (Guth, 2013). The user experiences of those scholars defining those languages seemed to indicate that programming languages can be defined more easily in $\mathbb{K}$ than in other frameworks such as Isabelle or Coq(Corbineau, 2008).

Despite the success of $\mathbb{K}$, there are several main issues that need to be addressed. There is no single document that is the definitive definition of the syntax and semantics of the $\mathbb{K}$ language. While there have been a number of papers published concerning theories related to $\mathbb{K}$, there is no source sufficiently complete to allow for rigorous proofs of properties of the languages defined in $\mathbb{K}$. Even though $\mathbb{K}$ has been used in the definitions of an impressive number of programming languages, the support it offers users of the language definitions is still fairly limited. There is a track record of researchers extending the language specifications in $\mathbb{K}$ to create various tools for analyzing program properties, such as the LTL tools associated with the C semantics in $\mathbb{K}$. However, as of now, only an experimental Reachability Logic (Roşu et al., 2013) proof tool has been implemented directly on top of $\mathbb{K}$ and this still requires changing a language specification to make the tool work with the language. In addition, while $\mathbb{K}$ can support specific tools for analyzing programs in a language

Authors' addresses: Liyi Li, Computer Science, University of Illinois at Urbana-Champaign, liyili2@illinois.edu; Elsa L. Gunter, Computer Science, University of Illinois at Urbana-Champaign, egunter@illinois.com.

2018. XXXX-XXXX/2018/6-ART $15.00
https://doi.org/

defined in $\mathbb{K}$, it provides no 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.

A full, formal language specification, which is our work and called **IsaK**, addresses these concerns and can form the foundation of tools for the maintenance, revision and expansion of $\mathbb{K}$. By giving such a semantics in a theorem prover such as Isabelle, we extend the analysis capabilities of the general $\mathbb{K}$ framework to include analysis through formal proof. Some examples of the use of **IsaK** formally appear in Section 4, where we evaluate **IsaK** by four approaches (literature reading, developer's discussing, testing suite running, theorem proving). Having a formal $\mathbb{K}$ specification in Isabelle is the first step towards providing $\mathbb{K}$ with additional power of full formal proofs and marrying a system ($\mathbb{K}$) with considerable experience in language specification with a system (Coq, Isabelle) with an impressive history proving results of mathematics and complex computer systems.

2 BACKGROUND

In order to provide a full formal specification of $\mathbb{K}$, we must ask the question: What is $\mathbb{K}$? Current $\mathbb{K}$ implementations supply a number of tools to support the execution and analysis language definitions including parsers for the language specification, generated parsers for the language specified, compiler and runtime interpreter, tools for generating test cases, and more. The main task of $\mathbb{K}$ implementations ($\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0) can be viewed as an interpreter generator that have several tasks as follows. First, $\mathbb{K}$ parses the specification syntax. It then generates a symbol table, a term parser (including metavariables) to parse the rewrite rules in the specification and a ground term parser to parse the programs associated with the specification. Second, the $\mathbb{K}$ static semantics is applied on the rewrite rules and programs associated with the specification, and they are compiled into a format that is easy and ready for execution. These steps simulate the ***kompile*** tool in $\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0. Third, the $\mathbb{K}$ dynamic semantics takes the compiled specification (the symbol table and compiled rules), and executes a compiled program associated with the specification. The tool based on the $\mathbb{K}$ dynamic semantics generates a trace (running ***krun***) or set of traces (running ***ksearch***) if users ask to see multi-trace behavior.

In **IsaK**, we focus on the meaning of $\mathbb{K}$ as a system for defining a programming language by specifying the syntax and semantics (***kompile***) and execution behavior of programs of that language (***krun*** and ***ksearch***). We will see the introduction of the $\mathbb{K}$ framework and challenges of defining the $\mathbb{K}$ semantics below.

2.1 The $\mathbb{K}$ Framework

The operational behavior of the $\mathbb{K}$ specification language contains four major steps: parsing , language compilation, sort checking, and semantic rewriting. In fact, parsing comes in two phases: one to learn the grammar of the object language (the programming language being defined), and a second that incorporates that grammar into the grammar of $\mathbb{K}$ to parse the definitions of the rules and semantic objects defining the executable behavior of programs of the object language. We assume the existence of two parsers for each of these phases, with the output of the first being passed to the second. Together 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. Our **IsaK** is given at the level of a generic abstract syntax serving for both the construct of $\mathbb{K}$ and the syntax of its object languages, and we manage to have two phases. The static semantic phase is for defining language compilation and sort checking, while the dynamic semantic phase is for defining the behavior of semantic rewriting.

$$\left\langle \left\langle \left\langle \langle K\rangle_{\mathsf{k}} \; \langle \mathit{List}\rangle_{\mathsf{stack}} \; \langle \langle \mathit{Map}\rangle_{\mathsf{locenv}} \; \langle \mathit{ClassType}\rangle_{\mathsf{crntClass}} \; \langle \mathit{Int}\rangle_{\mathsf{location}}\rangle_{\mathsf{methodContext}} \; \langle\langle \mathit{Int}\rangle_{\mathsf{tid}} \; \langle \mathit{Map}\rangle_{\mathsf{holds}} \; \langle \mathit{Bool}\rangle_{\mathsf{interrupted}}\rangle_{\mathsf{threadData}} \right\rangle_{\mathsf{thread*}} \right\rangle_{\mathsf{threads}} \; \left\langle \left\langle \langle \mathit{ClassType}\rangle_{\mathsf{classType}} \; \langle \mathit{Map}\rangle_{\mathsf{methods}} \; \langle\langle\langle K\rangle_{\mathsf{methodSignature}} \; \langle \mathit{List}\rangle_{\mathsf{methodParams}} \; \langle K\rangle_{\mathsf{methodBody}}\rangle_{\mathsf{methodDec*}}\rangle_{\mathsf{methodDecs}} \right\rangle_{\mathsf{class*}} \right\rangle_{\mathsf{classes}} \; \langle \mathit{Map}\rangle_{\mathsf{store}} \; \langle \mathit{List}\rangle_{\mathsf{in}} \; \langle \mathit{List}\rangle_{\mathsf{out}} \; \langle \mathit{Set}\rangle_{\mathsf{busy}} \; \langle \mathit{Map}\rangle_{\mathsf{waitingThreads}} \right\rangle_{\mathsf{T}}$$

Fig. 1. Subset of the Java Configuration (Bogdănaş and Roşu, 2015)

$\mathbb{K}$ allows users to define language syntax by using conventional BNF annotated with semantic attributes, while the semantics based on the language syntax is given as a set of reduction equations and rules over a configuration. The configuration for the language is an algebraic structure of the program states, organized as nested labeled cells, in XML formats, holding semantic information, including the program itself. Figure 1 provides an example for a subset of the Java configuration (Bogdănaş and Roşu, 2015). While the order of cells is irrelevant in a configuration (having *Bag* sort), the contexts relation between cells are relevant and are preserved by the users when they are defining rules or when "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., store), stacks (e.g., stack), etc. For example, a typical rule for reading a variable would be:

$$\left\langle \frac{X}{E} \cdots \right\rangle_{\mathsf{k}} \; \langle \cdots X \mapsto N \cdots \rangle_{\mathsf{locenv}} \; \langle \cdots N \mapsto E \cdots \rangle_{\mathsf{heap}}$$

There are three cells in the rule: k, locenv and heap. The k cell is a computation sequence waiting to be performed, while the head element of the list is the next item to be computed. The head of the k cell is basically a program counter to tell what is the current position of the computation. The locenv 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 rule above is that if the next computation to be executed is a variable lookup expression X, then we locate X in the environment to get its location number N in the location memory, and locate N in the heap to find its expression value E. With such, 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 does not change during the transition. The "⋯" represents portions of cells that are irrelevant.

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

$$\begin{aligned} &\langle X \curvearrowright \kappa\rangle_{\mathsf{k}} \, \langle \rho_1, X \mapsto N, \rho_2\rangle_{\mathsf{locenv}} \, \langle \rho_3, N \mapsto E, \rho_4\rangle_{\mathsf{heap}} \\ &\qquad\qquad \Rightarrow \langle E \curvearrowright \kappa\rangle_{\mathsf{k}} \, \langle \rho_1, X \mapsto N, \rho_2\rangle_{\mathsf{locenv}} \, \langle \rho_3, N \mapsto E, \rho_4\rangle_{\mathsf{heap}} \end{aligned}$$

Computations in the k cell are separated by "$\curvearrowright$", which is now observable. The κ and ρ_1, ρ_2, ρ_3, ρ_4 fill in the place of the "⋯" above. The most important thing to notice is that the rule is duplicated on the right-hand side. Duplication in a definition can lead to subtle semantic errors if users are not carefully synchronizing their changes to their specifications in multiple places, once changes are made. In a big language like C, Java or LLVM, the configuration structure is very complicated, and would require actually including additional cells like methodContext, thread and threads (Figure 1). These intervening cells are automatically inferred in $\mathbb{K}$, which keeps the rules more modular.

Modularity is one of the most important features of $\mathbb{K}$. In the process of defining specifications, users usually do not need to modify existing rules to add a new feature to the language. $\mathbb{K}$ maintains this feature by structuring the configuration as nested cells and by allowing users to design their specification rules by only mentioning the cells that are needed in those rules, and only the needed portions of those cells. For example, the above rule only refers to the k, locenv and heap cells, while the entire configuration contains many other cells as shown in Figure 1. The modularity of 𝕂 not only allows users to create a compact and human readable language specification, but also contributes to speeding up the semantics development process. For example, the above lookup rule does not change, even though a new cell is added to the configuration to support a new feature.

2.2 Challenges

Several formidable challenges are faced by the **IsaK** project. First, other than the two simple descriptions of 𝕂 (Roşu and Şerbănuţă, 2010) and (Şerbănuţă et al., 2014), there are no resources talking about its syntax and semantics. Indeed, all 𝕂 implementations contain some undesirable behaviors, so it is hard for us to learn the exact meanings of 𝕂 operators. In the process of defining 𝕂, we needed to constantly interview the 𝕂 team to understand the meanings of the 𝕂 operators and look at the Java source code of the 𝕂 implementation to understand how 𝕂 is being defined, which is a time-consuming task.

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 Java configuration in Figure 1, 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 in

Figure 1 as follows:

$$\left\langle \cdots \left\langle \frac{M}{\cdot} \right\rangle_{\text{holds}} \langle X \rangle_{\text{tid}} \cdots \right\rangle_{\text{thread}} \qquad \frac{\langle \cdots \langle M \rangle_{\text{holds}} \; \langle X \rangle_{\text{tid}} \cdots \rangle_{\text{thread}}}{\langle \cdots \langle \cdot \rangle_{\text{holds}} \; \langle X \rangle_{\text{tid}} \cdots \rangle_{\text{thread}}}$$

At first glance, people might think that the left-hand side rule (Rule 1) and the right-hand side rule (Rule 2) are the same, 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 1. Specifically, we remove all locks in the holds cell. The main problem here is that the "$\cdots$" is not a simple syntactic sugar when it is associated with $\mathbb{K}$ cells. The compilation of the "$\cdots$" in the $\mathbb{K}$ `cells` level desires a well-defined algorithm to accomplish this problem properly.

Due to the page limit of the paper, we cannot list here all of the challenges. Nevertheless, even if these challenges are many and hard, **IsaK** is defined without compromise and includes every feature of $\mathbb{K}$.

3 ISAK: THE SEMANTICS OF $\mathbb{K}$

$\mathbb{K}$ is a specialized language where input to the $\mathbb{K}$ interpreter is done using two separate pieces, which reflects two separate layers of the $\mathbb{K}$ interpreter. The piece provides the user-defined language specification, while the second provides programs to be run in that language. In $\mathbb{K}$ implementations, both of these two parts are input in terms of a pretty printed format that can be directly parsed into a front-end AST (FAST), and transformed into a back-end AST (BAST), and then the semantic meaning of the language and its programs is given based on the BAST. Based on this scenario, we design **IsaK** as the combination of two separate definitions: the static semantics and dynamic semantics (Figure 2).

Before input to the static semantics and dynamic semantics, we assume that there is an external parser that parses pretty printed user-input $\mathbb{K}$ "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 (Bogdanas, 2012)) and is intended to be suitable for the OCaml-based $\mathbb{K}$ implementation extracted from **IsaK** in Isabelle directly.

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.

Statistics. We define the semantics for all 61 datatypes of $\mathbb{K}$ in Isabelle. Here are some statistics:

	Static	Dynamic	Common	Lib	Total
LOC	3784	1568	1436	876	13847
Size (KB)	246	102	88	39	475
datatypes	30	26	5	–	61
definitions	552	232	414	236	1434
relations	–	14	–	3	17

The rows contain lines of code (LOC), total file size, number of datatypes, number of definitions and functions, and number of inductive relations. The columns are the static semantics, dynamic semantics (including multi-trace semantics), modular common definitions and functions for both static and dynamic semantics, $\mathbb{K}$ library definitions and functions and total size.

Next, we give an overview of the static semantics and the dynamic semantics.

3.1 Static Semantics

The static semantics (simulating the ***kompile*** tool) contains several phases, as listed in Figure 2. 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.

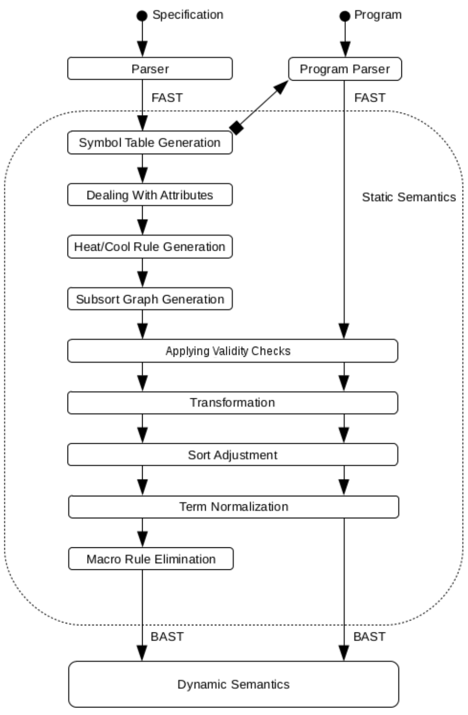

Fig. 2. The structure of **IsaK**

Symbol Table Generation. In this phase, the syntactic definitions in a given specification are processed, and we collect the information about how to parse a concrete term, generate constructor labels for each user-defined syntactic definition, grab the target sort and arguments of the definition, and get the attribute information of it, such as whether or not it is a function definition. The purpose of the step is to provide information for generating parsers to parse $\mathbb{K}$ rewrite rules and programs, and act as a database for later compilation and execution phases to use. Each entry includes information about the target sort, argument sorts, `kLabel` name, concrete production and $\mathbb{K}$ attributes.

Dealing with Attributes. $\mathbb{K}$ provides syntactic and semantic attributes to allow for more succinct specifications. For example, if we define an addition operator with a `strict` attribute like:

$$\text{SYNTAX} \quad \mathit{Exp} ::= \mathit{Exp} + \mathit{Exp} \ \ [\text{strict}(1)]$$

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 as follows:

$$\begin{array}{c}\dfrac{X : \mathit{Exp} + Y : \mathit{Exp}}{X : \mathit{Exp} \curvearrowright (\square + Y : \mathit{Exp})} \\ \mathit{requires} \neg\ \texttt{isKResult(}\ X\ \texttt{)}\end{array} \qquad \dfrac{X : \mathit{KResult} \curvearrowright (\square + Y : \mathit{Exp})}{X : \mathit{KResult} + Y : \mathit{Exp}}$$

The left-hand side rule 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 right hand side rule merges a *KResult* redex term $X{:}\mathit{KResult}$ and a context $\square + Y{:}\mathit{Exp}$ back to a term $X{:}\mathit{KResult} + Y{:}\mathit{Exp}$.

In this phase, we take care of different syntactic attributes (associated with syntactic definitions) and semantic attributes (associated with rewrite rules) by rearranging the parsed input language specifications. These specifications are cut into pieces in this phase and stored as special datatypes created in Isabelle. 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 whose left-hand side top-most `kLabel` is the same as this definition, store all these rules in a set and label them as belonging to the function operator. Another example is that we also delete all syntactic definitions with a `bracket` attribute from the symbol table and FAST, since these definitions only serve for parsing in $\mathbb{K}$ and have no semantic meanings. Before that, we need to make sure that any syntactic definition with a `bracket` attribute always has one target sort and one argument sort and they are the same. Otherwise, it would introduce an inappropriate subsort relation.

Heat/Cool Rules Generation. 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. In this phase, for every position specified by the number (the number in a `strict` attribute represents the non-terminal position in a production needing to generate a pair of `heat`/`cool` rules), we generate a pair of `heat`/`cool` rules exactly like the pair in the example above, with proper sort information. 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 $\square$.

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 like:

$$\textsc{syntax} \quad \mathit{Exp} ::= \mathit{Int}$$

In this definition, sort *Int* is defined to be a subsort of sort *Exp*. $\mathbb{K}$ has a very special subsort structure. First, 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 of these built-in sorts; otherwise, the specification is not well-formed. Second, the built-in sort *KItem*, representing the elements in a sort *K* list, is a subsort of the sort *K* implicitly, while all other user-defined sorts 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 explicitly defined to be subsorts of *KResult* 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. The first important check is to ensure that there are no overlapping `kLabel` names on any syntactic definitions of a specification. Second, users cannot define any normal $\mathbb{K}$ syntactic definitions with target sorts to be the built-in sorts *K*, *KLabel*, *KList*, *Set*, *List*, *Map* and *Bag*, except the case when users defined them with a `function` attribute. Third, at most one of the `function`, `strict` and `seqstrict` attributes is allowed to appear in a syntactic definition. Without this restriction, a syntactic definition could be both a function and generating `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 following two syntactic definitions 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*.

$$\text{SYNTAX} \quad \mathit{Exp} ::= \texttt{List\{}\ \mathit{Int}\ \texttt{,","\}} \quad \text{SYNTAX} \quad \mathit{Exp} ::= \texttt{List\{}\ \mathit{Bool}\ \texttt{,","\}}$$

Transformation. The concrete syntax of $\mathbb{K}$ and FAST are bijective; but BAST is a strict subset of the transformable terms in FAST. 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. Figure 3 depicts most of the transformations performed by static **IsaK**.

Row 1 of Figure 3 says that a metavariable without a sort in FAST is transformed into a metavariable BAST term by automatically adding sort *K*. In $\mathbb{K}$, transformation is done before the sort adjustment step; therefore, it cannot assign sorts to unsorted metavariables based on their inferred sort information. Current $\mathbb{K}$ implementations assign sort *K* to every metavariable that has not specified a sort. If users do not define a metavariable that has built-in sort *KList*, *Set*, *List*, *Map* or *Bag* with the correct sort, $\mathbb{K}$ still assigns sort *K* to the metavariable and the term will be rejected during the the sort adjustment step by telling users that $\mathbb{K}$ cannot infer the correct sort for the metavariable.

As we see in Row 2, if the left-hand or right-hand side subterms of a rewrite operator ($\Rightarrow$) contain another rewrite operator ($\Rightarrow$), the term is rejected. Row 3 introduces how a rewrite rule is transformed. Any subterm of a BAST rule can be classified as a pattern or an expression. A rule is transformed into a tuple of a rule label, a pattern as the left-hand side of the rule, an expression as the right-hand side of the rule, another expression as the condition of the rule and a flag indicating if the rule is a `transition` rule (useful in ***ksearch***). A rule label can be a **Macro**, **Fun**, **Anywhere**, **KNormal** or **BagNormal** (we will identify different rules by their labels), each of which has different ways of evaluations (Section 3.2). Patterns represent the left-hand side of a rewrite rule in $\mathbb{K}$, while expressions represent the right-hand side and conditions. They are implemented by two different structures in BAST, since they have different restrictions, but users cannot distinguish them when they are writing $\mathbb{K}$ specifications in the concrete syntax (or FAST).

#	**FAST**(Concrete)	**BAST**
1	X	X:K
2	rule X + (Y => (-Y)) => .K	**rejected**

3	rule X / Y => X /Int Y
	    requires Y =/=K 0
	    [transition]
		(KNormal, //rule label
		 X:Int / Y:Int, //pattern
		 X:Int /Int Y:Int, //expression
		 Y:Int =/=K 0, //condition
		 true) //transition?

4	//expression
	SetItem(X ~> Y ~> L)
	S:Set
	SetItem(1)
		//expression
		[SetItem([X:K, Y:K, L:K]), S:Set, SetItem([1])]

5	//pattern
	SetItem(X ~> Y ~> L)
	S:Set
	SetItem(1)
		//pattern
		(Some S:Set, [SetItem([X:KItem,
		         Y:KItem, L:K]), SetItem([1])])

6	//pattern
	SetItem(X ~> Y ~> L)
	S1:Set S2:Set
	SetItem(1)
		rejected

7	syntax K ::= abs(Exp) [function] rule abs(X) + Y => (-X) + Y requires X < 0	**rejected**

8	configuration
	 <T> <k> $PGM:KItem </k>
	    <env> .Map </env>
	    <heap> .Map </heap>
	    <classes>
	     <class *>
	      <name> .K </name>
	      <body> .K </body>
	     </class>
	    </classes> </T>
	  1 + 2 + 3 //program
		//initial state
		<T> <k> (1 + 2) + 3 </k>
		    <env> .Map </env>
		    <heap> .Map </heap>
		    <classes> .Bag </classes>
		</T>

9	context abs([])	rule abs(A:K) => A:K ~> abs([]) requires notBool isKResult(A) rule A:KResult ~> abs([]) => abs(A:KResult) //then translate as Row 3

10	//configuration as Row 8
	rule <classes>...
	     (<class>
	          <name> f </name>
	      ...</class>
	     => <class>
	          <name> g </name>
	      ...</class>)
	    </classes>
		rule
		   <T> X:Bag <classes> C:Bag
		          <class>
		            <name> f </name>
		      U:Bag </class> </classes> </T>
		 =>
		   <T> X:Bag <classes> C:Bag
		          <class>
		            <name> g </name>
		     <body> .K </body> </class> </classes> </T>
		//then translate as Row 3

Fig. 3. Examples of Transformations performed by the static semantics

When a FAST term is transformed into a BAST expression, any term in sort *K*, *KList*, *Set*, *List*, *Map* or *Bag* is transformed into a cons list of the right sort, as the one in Row 4. It is a kind of normalization step to force BAST terms into normalized formats to eliminate implicit identity and associativity equational rules. In 𝕂, terms with built-in sorts usually have implicit equational rules associated with them. For example, a term of sort *K* has implicit identity and associativity equational rules, while a term of sort *Set* has implicit identity, associativity, commutativity and idempotence ones.

Creating BAST patterns is similar to generating BAST expressions but with stricter forms in the built-in terms of sort *K*, *KList*, *Set*, *List*, *Map* or *Bag*. As with the example in Row 5, these built-in terms are required to have two pieces: zero or one metavariable, and a transformed cons list. The idea of having a cons list is the same as the one in generating expressions above. The potential metavariable must come from the metavariable in the elements of the original built-in term. In $\mathbb{K}$, users are only allowed to write a built-in term (having sort *K*, *KList*, *Set*, *List*, *Map* or *Bag*) with at most one metavariable in its elements (*K* sort terms are special, but only the last metavariable is translated into a sort *K* and others are transformed as sort *KItem*). The term in Row 6 has no transformed pattern because the *Set* term contains two metavariables ,*S1* and *S2*, as elements. 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.

Row 7 shows another failed transformation. This rule is a dead one and **IsaK** rejects the rule. In $\mathbb{K}$, a function is defined by labeling a syntactic definition with a `function` attribute, and then $\mathbb{K}$ recognizes a rule belong to the function if its left-hand side (pattern side) top-most `kLabel` name is exactly the `kLabel` name generated from the syntactic definition (we call these kind of `kLabel` names as `function` `kLabel` names). Then, $\mathbb{K}$ collects all rules belong to one `function` definition, puts them in a group, and transforms them with the rule label **Fun** (we call these kind of rules as `function` rules). In addition, $\mathbb{K}$ dynamic semantics always applies compiled `function` rules to a program state continuously until there is no more subterms with `function` `kLabel` names in the program state before it applies other rules. Hence, if a rule contains `function` `kLabel` names not in its top-most pattern position, it is not going to be recognized as a `function` rule, and the rule is a dead rule because it has no chance to be used. Row 7 shows a dead rule and **IsaK** rejects such rule.

Row 8 shows how to translate a program into a BAST form by packing the initial configuration with the program in the appropriate position. In $\mathbb{K}$, a configuration indicates not only the types and relations of a program state, but also the initial values of the state pieces. The initial values are used to pack with the program, and they form a initial program state for ***krun*** to execute based on a compiled $\mathbb{K}$ specification. Row 9 shows how to transform a context rule to a pair of `heat`/`cool` rules in BAST. The functionality of context rules in $\mathbb{K}$ is similar to defining `strict` attributes in a syntactic definition. The $\square$ in a context rule (a context rule must have exactly $\square$) can be viewed as a redex having the sort that is the sort of the $\square$ and the context defined by the body of the context rule. Then, given the redex and context, we can generate a pair of `heat`/`cool` rules as we did for `strict` attributes above.

Transforming rules having patterns and expressions of sort *Bag* (*Bag* rules) are the most difficult part of the transformation, requiring a combined process of splitting, "completing" and translating terms of sort *Bag*. The transformation cannot be solved by a simple primary function on a recursive datatype. As we have seen in Section 2.2, the key difficulty is that the combination of a "⋯" operator in a term of sort *Bag* 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 (Li and Gunter, 2017). 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 and transform them to a tuple in BAST like the one in Row 3. Row 10 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 ⇒ 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 Row 10. If there is a "⋯" operator in an expression, we replace it with the correct terms according to the configuration, like the `<body> .K </body>` in Row 10. After finishing filling patterns and expressions, we put patterns in the pattern side and expression 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 terms `<classes>` and `X:Bag` in row 10.

Sort Adjustment. Previous materials about $\mathbb{K}$ only briefly described the $\mathbb{K}$ sort system. The implementations of $\mathbb{K}$ have weak sort systems containing a lot of undesirable behaviors. One of the contributions of **IsaK** is to fully propose a sort system for $\mathbb{K}$ that is consistent with $\mathbb{K}$'s design goals. When using $\mathbb{K}$ to define language specifications and run programs, the syntactic definitions that users define are prototypes giving restrictions on a construct's argument sorts, target sorts and sorts of different cell contents in the configuration. Users want the sort system to help them discover ill-formed rules or define different rule cases for a construct based on the different sort information for it. They also want to sort-check the program states when they run programs. Besides these design goals, we also need to consider the language being sort-checked. Once a normal rule (not a rule labeled with **Fun**, **Anywhere** or **Macro**) has been sort adjusted and a program state has been sort-checked, applying the rule on the program state will not create a result that is ill-formed.

The complete story of the **IsaK** sort system is described in the technical report (Li and Gunter, 2017). Our sort system relies on the subsort relations in the paragraph describing subsort graph generation. The system as constructed is a partially ordered and order-sorted sort refinement process to compute the greatest lower bounds of the metavariables and constructs of a given rule or program state. It checks the sorts of a program state (sort checking) and adjusts the sorts of the metavariables and constructs in a rule (sort adjustment) to be the set of maximal lower bound sorts under the input sort restrictions. We assume there is a sort $\top$, the top sort of all sorts. Users cannot refer to $\top$. If a metavariable or a construct has been concluded to have $\top$ sort, the specification is ill-formed or not specific enough to tell $\mathbb{K}$ what the sort is for the term.

We define the system as a binary relation in which the left-hand and right-hand side are tuples of the form (α, β, S, t). Here, t is a term in BAST, and S is a finite set of sorts bounding the term t. α is a partial map giving maximal lower bound sorts for metavariables in t, and β is a partial map giving maximal lower bound sorts for the terms of sort *KItem* in t (having the form *KLabel*(*KList*)), where their *KLabels* are metavariables. In $\mathbb{K}$, users are allowed to define a term of sort *KItem* with its *KLabel* position being unspecified and being written as a *KLabel* metavariable. The metavariable has sort *KLabel* in an α map, but its instantiation also uniquely defines the target sort of the *KItem* term. For example, assume that we have the following two syntactic definitions:

$\textsc{syntax}\quad \mathit{Exp} ::= \mathit{Exp} + \mathit{Exp}\ \ [\text{strict}(1)] \qquad \textsc{syntax}\quad \mathit{Exp} ::= \texttt{continuation(}\ \mathit{KLabel}\ ,\ K\ ,\ K\ \texttt{)}$

Users might want to define the following rule to apply an input high-order function to the two following arguments and then add the results together.

$$\frac{\texttt{continuation(}\ F : \mathit{KLabel}\ ,\ X : K\ ,\ Y : K\ \texttt{)}}{F : \mathit{KLabel}\ \texttt{(}\ X : K\ \texttt{)} + F : \mathit{KLabel}\ \texttt{(}\ Y : K\ \texttt{)}}$$

In this rule, while the metavariable F has sort *KLabel*, the application of the term F:*KLabel* to X:K (F : *KLabel* (X : K)) must be a subsort of sort *Exp*, because term + has the two argument sorts both restricted to sort *Exp* and term F : *KLabel* (X : K) is located in the argument position of term +. Hence, after we apply the sort adjustment on the rule, the metavariable F has a value of sort *KLabel* in α and value of sort *Exp* in β.

$\mathbb{K}$ allows users to write semantic rules in a flexible way. We have shown an example above, where users are allowed to define some metarules without specifying the actual `kLabel` names (representing the names for constructs) for their constructs, but with using metavariables having target sort *KLabel*. In this case, these metarules can pattern match any program states regardless of the `kLabel` names in the constructs, as long as the the conditions of the pattern matching are satisfied. The sort system of $\mathbb{K}$ is explicit first-order. It is impossible to define a strong sort system within first-order to catch the behavior of the flexibility, because it requires introducing function sorts. Designing a strong high-order sort system, such as the one in System F, can be done in the future, but it is beyond the scope of the paper because the **IsaK** sort system is designed to capture behavior of $\mathbb{K}$.

Apart from the flexible situation, our $\mathbb{K}$ sort system can strongly guarantee that once a specification and its initial program states are sort adjusted and checked, executing the program states will not go wrong. With the flexibility, under the assumption that $(\alpha, \beta, S, t) \rhd (\alpha', \beta', S_0, t_0)$, which is the sort adjustment function and it means that we can conclude a new sort adjusted term t_0 and bounded sort restriction S_0 of term t_0 with new α' and β' maps, provided that the input term t has a guessed input sort restriction S and initial maps α and β, our system can still guarantee: (1) the output sort restriction is a subsort of the input sort restriction for sort checking and sort adjustment. (2) if $(\alpha, \beta, S, t) \rhd (\alpha_0, \beta_0, S_0, t_0)$ and $(\alpha_0, \beta_0, S_0, t_0) \rhd (\alpha_1, \beta_1, S_1, t_1)$, then $(\alpha_0, \beta_0, S_0, t_0) = (\alpha_1, \beta_1, S_1, t_1)$. (3) there is a substitution φ with domain α such that, if $\varphi(X)$ sort-checks using the sort restriction defined by $\alpha'(X)$ for any X in domain α, and $\varphi(t)$ sort-checks using the sort restriction defined by $\beta'(X)$ for any term t of sort *KItem* having the position of sort *KLabel* as a metavariable in domain β, then for any term t' where $\varphi(t')$ sort-checks, the output sort restriction of $\varphi(t')$ is a subsort of the input sort restriction of $\varphi(t')$. All these three theorems have been proved through the Isabelle theory prover.

Term Normalization. Normalization is a process of ***kompile***, which 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.

Macro Rule Checks. The implementation of rules labeled as **Macro** is complicate and error-prone, and has no formal reference. Furthermore, running **Macro** rules in $\mathbb{K}$ 3.6 and 4.0 results in a lot of undesirable behaviors. We implement checks in **IsaK** to restrict the use of **Macro** rules. The details are in the technical report (Li and Gunter, 2017). The basic idea is to require a **Macro** rule to have its pattern subsorting to *KItem*, expression subsorting to *K* and have no condition.

3.2 Dynamic Semantics of IsaK

The single-trace version of the **IsaK** dynamic semantics is the formal description of ***krun***, while the multi-trace one is the formal description of ***ksearch***. In the two versions, they both need an input of a set of BAST $\mathbb{K}$ rules that comes from a specification that has been compiled by ***kompile***, and an initial state that comes from the combination of a compiled BAST configuration with a compiled BAST program. The output is a final program state (or a set of final program states in ***ksearch***). Practically, we also want to have a natural number (step counter) to limit the number of steps for each trace. In $\mathbb{K}$, only a transition done by applying a normal (**KNormal** or **BagNormal**) rule on a program state is considered as a step. Different rules (different rule labels) have different functionality but they have common features that can be classified as an atomic step.

The Atomic Step. The atomic step is the basis of an execution in **IsaK** and can be divided into four small phases: pattern matching (P), substitution (S), sort checking (T) and term normalization (N), whose execution order can be understood by the diagram below. The pattern matching step in **IsaK** implements the traditional pattern matching algorithm on user-defined operators, with implicit equational pattern matching on some built-in operators. In dealing with terms having the built-in sorts *K*, *KList* and *List*, **IsaK** uses associative and identity pattern matching. In dealing with terms having the built-in sorts *Set* and *Map*, **IsaK** uses identity, associative, commutative and idempotent pattern matching with an extra check on *Map* data structures to ensure they are functional. **IsaK** uses using identity, associative and commutative pattern matching to deal with terms having the built-in sort *Bag*.

$$\begin{tikzcd} P \arrow[r] & S \arrow[r] & T \arrow[r] & N \end{tikzcd}$$

Besides the points mentioned above, the pattern matching algorithm in $\mathbb{K}$ has two other features. First, it involves subsort relations, which have been extensively explored by a lot of classic pattern matching algorithms (de Moor and Sittampalam, 2001; Kesner, 1991). Second, the static semantics of **IsaK** actually makes applying the pattern matching algorithm to the built-in terms a lot easier. A lot of pattern matching algorithms with implicit equational rules are NP-complete (Kapur and Narendran, 1987). The patterns of rewrite rules in $\mathbb{K}$ are transformed into a special BAST form, which reduces the complexity of implementing these different forms of pattern matching with implicit equational rules for different built-in terms. For example, in a *Set* associative and commutative pattern matching, if we have two metavariables, we need to partition the target ground *Set* term into different options of two parts and we have multiple choices for the two metavariables. In contrast, in **IsaK** pattern matching, the transformed BAST forms eliminate the possibility of having more than one solution. The result of applying the pattern matching step (P) to a program state with a rewrite rule is a map from metavariables to ground terms. Then, the substitution step (S) applies the map on the condition of the rewrite rule and sees if the condition is valid. If it is, we substitute the metavariables in the right-hand side of the rewrite rule with the values of the map. If not, the atomic step outputs failure.

After the substitution, the new program state is a ground term which needs to be sort checked (T). The action is based on the same strategy described as the sort adjustment in Section 3.1. We need this step because $\mathbb{K}$ does not, in contrast to Maude, require users to have sort decreasing in the rewrite rules. For example, users can rewrite a term t with sort A to another term t' with sort B where A and B do not have a subsort relationship. Suppose we have a context $C\square$, and it requires having sort A in the $\square$. Then, when we rewrite term t to t', the result $C[t']$ is ill-formed. The sort checking step prevents these ill-formed terms and returns failure if they happen. The term normalization step (N) has the same functionality as the one in Section 3.1.

Single-Trace Behaviors. We describe the dynamic semantics for individual traces, which simulates the behavior of ***krun***. All rewrite rules combine the atomic step in different ways. Applying rules labeled with **Macro** runs the atomic steps on a compiled BAST file containing rewrite rules and an initial program state associated with these rules. These **Macro** rules are applied once to the initial program state at the beginning of an execution in ***krun***. After every step of applying a single **Macro** rule, a check needs to see if all rewrite rules and the initial program state are still valid in terms of valid checks on rules and programs as specified in Section 3.

The behaviors of applying other rules can be simply represented by the graphs in Figure 4. We first present how a **Fun** or **Anywhere** rule is applied to a program state in **IsaK**. Graph (a) represents how **IsaK** finds the right place in a program state to apply a **Fun** or **Anywhere** rule to.

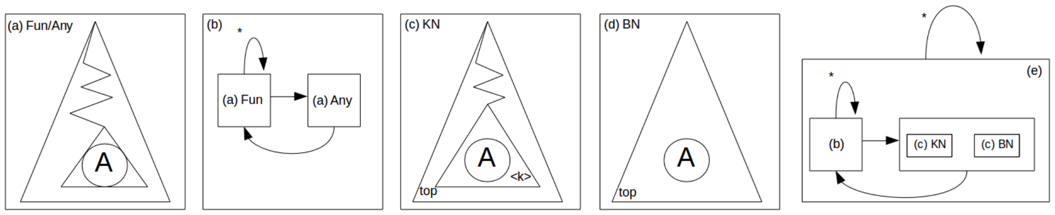

Fig. 4. Different Evaluation Steps

First, we search for `kLabel` names in the program state. When we find one that is the same as the top-most name in the pattern of the **Fun** or **Anywhere** rule, we apply an atomic step (A) locally. Internally, the applications of a **Fun** or **Anywhere** rule both do the job as represented in graph (a), except that applying **Anywhere** rules requires sort-checking on the whole program state after substitution because they are not designed to be sort decreasing in $\mathbb{K}$, so applying **Anywhere** rules to a program state might cause the program state to be ill-formed. Graph (b) represents the process of combining the applications of **Fun** rules and **Anywhere** rules. In the graph, `Fun` represents a **Fun** rule application and `Any` represents an **Anywhere** rule application. **IsaK** first searches **Fun** rules to apply to a program state until there are no more **Fun** rules to apply. Next, it searches a single **Anywhere** rule to apply, and then, it goes back to see if there is any **Fun** rule to apply, and so on.

We then present how a normal rules are applied to a program state. Graph (c) represents how **IsaK** finds the right place in a program state to apply a single **KNormal** rule to, where it always looks for a `k` cell in the program state and does an atomic step (A) there. The process of applying a **BagNormal** rule to the whole program state is represented in Graph (d). It is just to look for a matching pattern of the pattern in a **BagNormal** rule with the whole program state. Graph (e) represents how **IsaK** connects all these different applications of rules together. In the graph, `KN` represents a **KNormal** rule application and `BN` represents a **BagNormal** rule application. As described in the box of the graph (e), for executing one ***krun*** step, we first proceed the steps represented in graph (b). After we find no more **Anywhere** rules or **Fun** rules to apply by following the steps in graph (b), we try to see if there is a **KNormal** or **BagNormal** rule to apply to the program state. One step execution of a **KNormal** or **BagNormal** rule is counted towards a step of evaluation in **IsaK**. ***krun*** continues applying the single step represented in the box of graph (e) until there is no more rules to apply to the program state or the step counter is reduced to zero. this is an abstraction of the LLVM semantics. We will show an example rule below.

Application of KNormal rules. We choose to illustrate the **IsaK** dynamic semantics by applying a **KNormal** rule to a configuration. We describe the **KNormal** rule in the manner of operational semantics in Figure 5. We assume that a set of rewrite rules Φ having a form similar to the BAST tuple representation in the second column of row 3 of Figure 3, and a program state C of sort *Bag*.

In these rules, a *Bag* term (variable C, C', C_1 and C_2) is actually a multiset of cells. A cell has the form (X, H), where X is the cell name and H is its content, which can be of sort K, *Set*, *List*, *Map* or *Bag*. Rule *KTOP* describes the behavior of an evaluation in **IsaK** for a whole program state. The `top` represents a program state that contains a *Bag* term. A program state can make a move only if its *Bag* term can make a move and the term is type checked and normalized. The function `typed`(S, T) answers `true` if the input expression S is type checked and its target sort is a subsort of sort T. The type-checking procedure is described in the Sort Adjustment subsection in Section 3.1. The partial function `norm` normalizes the input expression and applies some checks to it according

$$\frac{(\Phi, C) \longrightarrow (\Phi, C') \quad \mathtt{norm}(C') \mathtt{=some}\ C'' \quad \mathtt{typed}(C', \mathit{Bag})}{(\Phi, \mathtt{top}(C)) \longrightarrow (\Phi, \mathtt{top}(C''))}\ \mathit{KTOP} \qquad \frac{(\Phi, C2) \longrightarrow (\Phi, C2')}{(\Phi, C1\ C2) \longrightarrow (\Phi, C1\ C2')}\ \mathit{KS}1$$

$$\frac{(\Phi, H) \longrightarrow (\Phi, H')}{(\Phi, (X, H)) \longrightarrow (\Phi, (X, H'))}\ \mathit{KS}2 \qquad \frac{(\Phi, C1) \longrightarrow (\Phi, C1')}{(\Phi, C1\ C2) \longrightarrow (\Phi, C1'\ C2)}\ \mathit{KS}3$$

$$\frac{\begin{array}{c}\mathtt{subst}(\varphi, S) = S' \quad (\Phi, \mathtt{subst}(\varphi, B)) \longrightarrow^* (\Phi, \mathtt{true}) \\ (\mathbf{KNormal}, P, S, B, F) \in \Phi \quad \mathtt{match}(P, A) \mathtt{=some}\ \varphi\end{array}}{(\Phi, (\mathsf{k}, A \curvearrowright \mathit{Tail} : K)) \longrightarrow (\Phi, (\mathsf{k}, S'' \curvearrowright \mathit{Tail} : K))}\ \mathit{KN}$$

Fig. 5. Operational Semantics for **KNormal** rules

to the behavior described in the Term Normalization subsection in Section 3.1. Rules $KS1$, $KS2$ and $KS3$ are to search the place in a program state to apply a **KNormal** rule. $KS1$ and $KS3$ say that if any subpart of a *Bag* item can make a move, the whole *Bag* can make a move containing the move of the subpart. $KS2$ says that a cell can make a move if its contents can make a move to a new state.

Rule KN is the core rule for describing the application of a **KNormal** rule. Φ is the database of rules. As we mentioned in Section 3.1, an entry in the database is a tuple of a rule label, a pattern, an expression, a conditional expression and a flag indicating if the rule is transitional. We picks a rule with the **KNormal** label here. The partial function `match` implements the pattern-matching algorithm described above. It takes a pattern P and the first element A in the k cell (k cell has sort K, which is a list of single *KItems*), and calculates a mapping φ from the identifiers in P to the subterms in A. The function `subst` replaces the identifiers in an expression with their values from φ. In dealing with conditional expressions, $\longrightarrow^*$ represents that the substituted ground term $\mathtt{subst}(\varphi, B)$ must transit to term `true` through zero or more steps of evaluation (the function evaluation steps described in the last subsection), in order for the rule to be successfully applied. The main idea of the rule is to find a **KNormal** rule entry that pattern matches with the first element of the execution sequence where the mapping model of the pattern matching must satisfy the conditional expression of the entry.

We now present an example application of the above rules that assumes a configuration as in row 8 of Figure 3. We assume that we are working with the BAST form of that rule. The necessary user-defined syntactic definition and semantic rule for the addition operator is described as follows:

$$\textsc{syntax} \quad \mathit{Exp} ::= \mathit{Exp} + \mathit{Exp} \ \ [\text{strict(1,2)}] \quad \textsc{syntax} \quad \mathit{KResult} ::= \mathit{Int} \qquad \frac{X : \mathit{Int} + Y : \mathit{Int}}{X : \mathit{Int} \ \mathtt{+Int}\ Y : \mathit{Int}}$$

We define terms of sort *Int* as the *KResult* and we also require the two subterms of the addition operator to be attributed as `strict`, which means that four implicit `heat`/`cool` rules are generated. In addition, the rule for addition will be processed into an entry format like the one in row 3 of Figure 3:

$$(\mathbf{KNormal}, X : \mathit{Int} + Y : \mathit{Int}, X : \mathit{Int}\ \mathtt{+Int}\ Y : \mathit{Int}, \mathtt{true}, \mathtt{true})$$

For simplicity, we assume that the initial program state has been evaluated one step forward by the left `heat` rule of the addition operator. Hence, the program state transits to the following:

$$\Big\langle \langle \mathtt{(1\ +\ 2)} \curvearrowright \mathtt{(\square\ \ +\ 3)} \rangle_{\mathsf{k}}\ \langle \mathtt{.Map} \rangle_{\mathsf{env}}\ \langle \mathtt{.Map} \rangle_{\mathsf{heap}}\ \langle \mathtt{.Bag} \rangle_{\mathsf{classes}} \Big\rangle_{\mathsf{T}}$$

To apply the **KNormal** rule, we first pick the user defined rule for the semantics of addition above. Then, we use the pattern-matching function to get the mapping $[X \mapsto 1, Y \mapsto 2]$. After substituting the identifiers in the expression and conditional expression, we conclude the conditional expression to be `true`, and the substituted expression to be `1 +Int 2`. Function `+Int` is a $\mathbb{K}$ built-in function that adds two integers together, so the final result of applying the addition rule is 3. Finally, we put the final result back in the right place of the program state, and the program state becomes:

$$\Big\langle \langle 3 \curvearrowright (\square \;\; + \; 3) \rangle_{\mathsf{k}} \;\; \langle \texttt{.Map} \rangle_{\mathsf{env}} \;\; \langle \texttt{.Map} \rangle_{\mathsf{heap}} \;\; \langle \texttt{.Bag} \rangle_{\mathsf{classes}} \Big\rangle_{\mathsf{T}}$$

We also need to type check the resulting program state, which has only one change, that is, changing the term `1 + 2` to 3. That position has maximum target sort *KItem*, and term 3 has sort *Int* which is a subsort of sort *KItem*, so the final program state is type checked. In addition, we normalize the final result and find that the *Map* or *Set* terms are the two empty *Map* terms in cell env and heap, so we conclude that the final program state is already normalized. The next step would be to apply the cool rule by placing the 3 back in the □.

The full details of the formalization and implementation of the **IsaK** single trace dynamic semantics are described in the technical report (Li and Gunter, 2017).

Multi-trace Behaviors. The multi-trace dynamic semantics of **IsaK** simulates ***ksearch***, which is the multi-trace version of ***krun***. The difference between ***ksearch*** and ***krun*** is that ***ksearch*** allows users to see all possible traces while ***krun*** only allows users to see an evaluation sequence. ***ksearch*** relies completely on users defining rules to represent non-deterministic choices with the `transition` attribute. Only **KNormal** and **BagNormal** rules attributed with `transition` can be observed with these nondeterministic behaviors. The $\mathbb{K}$ documents do not specify the details of ***ksearch***, but both $\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0 design ***ksearch*** in the same way. For a given program state in a specification, ***ksearch*** first sees if there are any non-`transition` rules to apply; if so, then $\mathbb{K}$ disregards `transition` rules and evaluates the program state by the non-`transition` rule selected by the rule order. If there are no non-`transition` rules capable of being applied to the program state, $\mathbb{K}$ starts looking at the `transition` rules, and collects a set of results by applying all of applicable transition rules to the current program state, and recursively call ***ksearch*** on each new state. The implementation of ***ksearch*** is similar to the ***krun*** evaluation procedure except that we first need to check every step for a non-`transition` rule; if one exists, then we can just do the same as in ***krun***; if not, we need to apply each of the applicable `transition` rules to the program state to get the results.

4 EVALUATION AND APPLICATION

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 $\mathbb{K}$ specifications and programs based on the extracted OCaml $\mathbb{K}$ interpreter. In the following paragraphs, we describe our evaluation, especially the testing, which resulted in the first thorough set of bugs in $\mathbb{K}$. With a CTL tool that we built in Isabelle, we also show the usefulness of **IsaK** for exploring multi-threaded program behaviors. This work can lead to a project providing a testing framework for the language Morpheus (Mansky et al., 2016), which specifies program transformations.

4.1 Testing process of IsaK

The validation of language semantics is usually accomplished through the use of external test suites (Filaretti and Maffeis, 2014; Ellison and Rosu, 2012; Bodin et al., 2014), which was also our strategy. A set of 13 specifications with 356 programs, which we call the $\mathbb{K}$ standard test suite, was the basis of our testing. It was used by the $\mathbb{K}$ team to test their $\mathbb{K}$ implementations for both $\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0 (Roşu, 2016).

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 ***krun***. We ran 356 programs in total for 13 different language specifications, and our results showed that the 𝕂 interpreter extracted from **IsaK** 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 input channel is hard and not very useful to implement in Isabelle, and it is best to do that in the parser. We have not yet finished the job in the parser, 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 out 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 specifies a small 𝕂 "program" for each of them.

These undesirable behaviors happen in very diverse circumstances. 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 𝕂 (Bogdănaş and Roşu, 2015; Park et al., 2015; Filaretti and Maffeis, 2014; Ellison and Rosu, 2012). 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 rules labeled with a **Fun** attribute. Hence, **MacroRules** are unnecessary in 𝕂. Other undesirable behaviors are the implementation bugs in 𝕂. For example, some are related to sort checking/adjustment. The current 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 𝕂 (the atomic step). The current 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. We list two example undesirable behaviors in $\mathbb{K}$ which are confusing and hard for users to detect. There are many of these undesirable behaviors, we will not list all of them here, but there are two failures in $\mathbb{K}$ that we will talk about because they are very confusing.

Failure in *Map* ***Operators.*** Sometimes, using simple map update operators can be very confusing in $\mathbb{K}$. For example, in defining the Java semantics whose configuration is given in Figure 1, users might want to define a variable assignment operator by using a map update function like:

$$\left\langle \frac{X = V}{\texttt{.K}} \cdots \right\rangle_{\mathsf{k}} \; \langle \cdots\, X \mapsto N \,\cdots \rangle_{\mathsf{locenv}} \; \left\langle \frac{M : \mathit{Map}}{M \texttt{ [} N \texttt{ <- } V \texttt{]}} \right\rangle_{\mathsf{heap}}$$

Using a built-in library function M `[` N `<-` V `]` is actually the only correct way to define map update in $\mathbb{K}$. In the map lookup example in Section 2, we did not use any library function. A lot of users might want to ask if there is a way to define the map update operator in $\mathbb{K}$ without using the built-in library function. Actually, some early versions of $\mathbb{K}$ had examples showing a definition for map update operator without the built-in library function:

$$\left\langle \frac{X = V}{\texttt{.K}} \cdots \right\rangle_{\mathsf{k}} \; \langle \cdots\, X \mapsto N \,\cdots \rangle_{\mathsf{locenv}} \; \langle M : \mathit{Map}(\texttt{.Map} \Rightarrow N \texttt{ |-> } V) \rangle_{\mathsf{heap}}$$

However, this implementation is not consistent with the basic idea of $\mathbb{K}$ rewriting. $(\texttt{.Map} \Rightarrow N \texttt{ |-> } V)$ is actually not a map update operator but creating a new singleton map that maps N to value V. In the heap cell, the final result of applying the above rule is two maps, $M : \mathit{Map}$ and $(N \texttt{ |-> } V)$; so the final result becomes a union of the two. If there is an entry in $M : \mathit{Map}$ having key N but a different value than V, the union should fail, because $\mathbb{K}$ requires a union of two maps succeeds if and only if the keys of the two maps are disjoint. In testing $\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0, we do not find such failure, but we do find a rule that has the similar undesirable behavior:

$$\left\langle \frac{(X = V, X = V')}{\texttt{.K}} \cdots \right\rangle_{\mathsf{k}} \; \langle \cdots\, X \mapsto N \,\cdots \rangle_{\mathsf{locenv}} \; \langle M : \mathit{Map}(\texttt{.Map} \Rightarrow (N \texttt{ |-> } V)(N \texttt{ |-> } V')) \rangle_{\mathsf{heap}}$$

The operator $(X = V, X = V')$ updates a map with key X to values V or V' nondeterministically. By the same argument above, we should expect $\mathbb{K}$ to report a failure on unioning the result maps in the heap cell because the key N is mapping to two different values V and V'. Unfortunately, in both $\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0, the rule can be applied, and the final result is always that entry N has value V'. This undesirable behavior and some examples in the early versions cause a lot of $\mathbb{K}$ users to be confused about the correct implementations of map update rules in $\mathbb{K}$. For example, some users defined all map update rules of the LLVM semantics in $\mathbb{K}$ (Li and Gunter, 2016) without using the built-in library function. When they started testing the semantics with simple programs having one map update, mistakes did not arise. When testing them on a large program with multiple map updates on a single program variable, however, they found that they needed to change a lot of the rules.

Failure in Generating Nested Cells. Current $\mathbb{K}$ implementations implicitly (without any error message or mention in any document) prevent people from defining more than two levels of nested cells in a configuration 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. Assume that the configuration is the Java configuration in Figure 1. In K-Java Bogdănaş and Roşu (2015), 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 in a very lucky way. 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 the 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 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 as follows:

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

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 new stack structure with the old stack cell in the K-Java configuration (Figure 1), we create two-level nested * cells in the configuration. The top level * cell is the thread cell, and the inner level is the stackObject cell. If we define the semantics of an operator `getStackType` to lookup the type of a stack as:

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

then 𝕂 3.6 and 𝕂 4.0 fail to apply this rule. Once a program state is required to apply this rule, 𝕂 3.6 and 𝕂 4.0 crash. The reason is that the special cell k, which represents the program execution sequence, is inside a cell thread that is marked as *, and then some variable inside the execution cell k is trying to match with some content inside another cell (stackObject) that is marked as * and it is inside the thread cell that contains the execution cell k. Apparently, current 𝕂 implementations 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 he or she will need the special cell k inside a * key word cell and then define other cells in the * key word cell with another * key word. 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.

We do not give a complete list of undesirable 𝕂 behaviors here, but interested users may find information in the website https://github.com/liyili2/k-semantics/. Despite these undesirable behaviors, 𝕂 is still a effective tool for defining language specifications and a lot of users have successfully defined a lot of large language specifications in 𝕂, such as C semantics, Java semantics, Javascript semantics and LLVM semantics. The intent of this paper is to have a complete semantics of 𝕂 and to locate problems to facilitate users using 𝕂 in a correct way.

Multi-trace Testing. We compared our extracted multi-trace interpreter in OCaml with the results of ***ksearch*** in 𝕂 3.6 and 𝕂 4.0 to test the multi-threaded programs. Among the 𝕂 standard test suite, there are three specifications with 34 programs involved in testing ***ksearch***. We defined all behaviors and features that ***ksearch*** currently supports. We also found undesirable behaviors in

ksearch whose frequencies were higher than the ones found in testing ***krun***, but since there are no standard behaviors or answers for ***ksearch***, and ***ksearch*** is still largely under development, we assumed that the undesirable behaviors in it are just currently unsupported behaviors that lack error messages.

4.2 Ocaml Based Interpreter

A trivial utility of **IsaK** is an Ocaml interpreter extracted directly from the Isabelle source code. A specification is compiled by using the Ocaml `kompile` function. Users can use the `krun` function to execute a program of the specification and see a single trace of the program. In addition, users are able to see the step-by-step single trace behavior by giving an extra number attribute to the `krun` function, where the extra number indicates the steps of execution the users want to see. Users are able to use the ***ksearch*** function to see all the trace behaviors occurring in a multi-threaded program execution. Again, by giving an extra number attribute to the `ksearch` function, users can see the step-by-step multi-threaded trace behaviors, which are collected in terms of a set of traces.

The testing process in **IsaK** (4.1) relies heavily on the Ocaml based interpreter. All tests are executed through the interpreter. It is able to handle all of the specifications and programs described in Section 4.1. We also do an informal comparison of the execution efficiencies of our interpreter and the $\mathbb{K}$ interpreter (***kompile***, ***krun*** and ***ksearch***). The comparison results show that our interpreter is a little slower than the $\mathbb{K}$ interpreter but they are at the same effectiveness level.

4.3 CTL Model Checking

One of the biggest advantages of defining the $\mathbb{K}$ semantics in Isabelle is the ability to verify tools built in $\mathbb{K}$. Users who defined previous major specifications in $\mathbb{K}$, such as C semantics, Java semantics and Javascript semantics, have claimed that they used some verification tools in their semantics. By examining these claims, we find that all of the verification tools rely on changing or extending the target language specifications to include the specifications of the tools, which is not ideal for specifying a verification tool. In addition, the current $\mathbb{K}$ implementations ($\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0) have built-in tools, such as a matching logic proof engine (Roşu and Ştefănescu, 2011). The key issue of these tools is their correctness. What is the reason for users to trust a tool built on top of a large piece of Java code? As we has seen, the $\mathbb{K}$ implementations have already shown some undesirable behaviors. Tools built on them are likely to exhibit more problems. **IsaK** allows users to not only access a large database of existing tools in Isabelle but also to have the ability to verify the correctness of the tools they want to use with respect to **IsaK**.

The CTL model checker **CTLK** is one such example. It is part of the development of a CTL-like compiler verification framework (Mansky et al., 2016) that works in **IsaK**. **CTLK** includes a minimal set of CTL operators with an extra boolean function. The basic idea is to view the program states in $\mathbb{K}$ as the CTL states in CTL formulas. The extra function determines the cells users care about in the program states. For example, if users only care about the execution cell k and the memory cell store in the K-Java configuration (Figure 1), they can mark these two cells as true and the other cells as false in the function; then **CTLK** only compares the content in these two cells when it compares the equivalence of states. We design the basic predicates in a CTL formula to allow users to write a list of pairs of program state piece names and a pattern referring to what the users consider valid in the program state. With the function and the CTL formula, we use the same mechanism as ***ksearch*** to generate traces. Instead of putting traces in a set, we merge them in a tree structure with the same root. We then merge the same CTL states in the traces to form an automaton structure. If the structure is infinite, then the model checker fails. After generating the structure, we use the traditional CTL model checking algorithm of Clarke and Emerson (Clarke and Emerson, 1982) to model-check it. To verify the correctness of the CTL model checker on **IsaK**, we have proved the following theorem:

THEOREM 4.1. *Assume that C is a set of well-formed CTL uninterpreted formulas, f provides correct semantic interpretation of a CTL formula c on a graph G, g is a function to generate an execution graph by running a program p on a specification L in **ksearch**. Then, the result by running **CTLK** on c, L and p is a subset of the semantic interpretation of c on a graph generated by $g(L, p)$, as **CTLK** (c,L,p) $\subseteq f(c, g(L, p))$.*

5 RELATED WORK

We believe this paper is the first one to propose a complete and formal semantics of $\mathbb{K}$. In this section, we discuss related work on describing the $\mathbb{K}$ semantics, language semantics defined in $\mathbb{K}$ and other large scale language specifications.

We recognize four real-world language specification forms: a description in English with some mathematical details and examples, such as C standard, which is well written and precise; an compiler/interpreter implementation such as PHP; rigorous mathematical specifications, such as Standard ML (Milner et al., 1997); and formal and executable specifications. Our semantics of $\mathbb{K}$ is of the fourth kind.

5.1 Current $\mathbb{K}$ Specifications

$\mathbb{K}$ has a brief English description of its semantics in the document "An Overview of the $\mathbb{K}$ Semantic Framework" (Roşu and Şerbănuţă, 2010), which also provides some examples to explain its major features. 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. It 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 is a logic system that is built on top of $\mathbb{K}$ for reasoning about structures. By viewing the terms in first order logic as patterns, Matching Logic has a way to derive theories based on pattern matching algorithms. The current invention of Matching Logic is Reachability Logic (Ştefănescu et al., 2014; Roşu et al., 2013). 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 (Hoare, 1969) to prove properties about programs in the specifications, so that users do not need to define the axiomatic semantics of a specification. When talking to $\mathbb{K}$ group members, they indicated that a logic proof system is the future that the $\mathbb{K}$ project is pursuing.

Other work in $\mathbb{K}$ (Şerbanuţă and Roşu, 2010; Lucanu et al., 2012; Ellison et al., 2009; Hills and Roşu, 2008; Şerbănuţă et al., 2009; Hills et al., 2008; Arusoaie et al., 2012; Meseguer and Roşu, 2011; Ştefănescu et al., 2016) talks about other useful compilations of $\mathbb{K}$ to other languages, such as Maude (M. Clavel and Meseguer, 2000), and other model checking tools and software engineering tools in $\mathbb{K}$, such as a pretty printing tool and a test case generation tool. There is an ongoing project by Moore (Moore and Roşu, 2015) that transfers the $\mathbb{K}$ specifications to Coq (Corbineau, 2008) 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.

5.2 Other Large Language Specifications in $\mathbb{K}$

Big language specifications have been defined in $\mathbb{K}$ including C (Ellison and Rosu, 2012), PHP (Filaretti and Maffeis, 2014), JavaScript (Park et al., 2015), and Java (Bogdănaş and Roşu, 2015).

They are executable, have been validated by test banks, and, through the addition of some formal analysis tools produced by $\mathbb{K}$, have shown usefulness. For example, Ellison and Rosu (Ellison and Rosu, 2012) defined a formal C semantics. The executable C semantics in $\mathbb{K}$ was tested using the GCC torture test suite, and 99.2% of the tests passed. The C specification also includes tools such as debugging, monitoring, and (LTL) model checking which are either provided by $\mathbb{K}$ or extended in the semantics to include the specifications of the tools.

Another example is the formal semantics of PHP by Filaretti and Maffeis (Filaretti and Maffeis, 2014). Unlike the definitions of JavaScript, C and Java, there is no English description of PHP, so they needed to test the implementation heavily. Their semantics was evaluated by model checking certain properties of the cryptographic key generation library pbkdf2 and the web database management tool phpMyAdmin.

A formal semantics of Java has been defined by Bogdanas and Rosu (Bogdănaş and Roşu, 2015), and a specification of JavaScript done by Park and Rosu (Bogdănaş and Roşu, 2015). They were both tested by large test banks to validate their correctness. For example, Park's work was tested against the ECMAScript 5.1 conformance test suite, and passed all 2,782 core language tests. They also evaluated the specifications by model checking programs. The model checking relies on extending their language specifications.

There are several distinguishing aspects of our semantics, compared to others in $\mathbb{K}$. First, we are defining $\mathbb{K}$, a language without a rigorous English description, and one that is designed to define other language specifications. This means that we need to define both the compilation and executions of $\mathbb{K}$, which we did in defining ***kompile***, ***krun*** and ***ksearch***. Second, we not only test our $\mathbb{K}$ specification heavily but also communicate constantly with the $\mathbb{K}$ group members to ascertain what they think about the different $\mathbb{K}$ operators. Third, we defined model checking tools such as a CTL tool based on the $\mathbb{K}$ specification, so that these tools can be verified and used for other language specifications in $\mathbb{K}$ without changing those specifications. Fourth, the proof of properties about our **IsaK** sort system in Isabelle/HOL makes our system more convincing.

5.3 Other Large Language Specifications

Standard ML by Milner, Tofte, Harper, and Macqueen (Milner et al., 1997) is one of the most prominent and mathematical programming language specifications, whose formal and executable specifications were given by Lee, Crary, and Harper (Lee et al., 2007), also by VanInwegen and Gunter (Inwegen and Gunter, 1993), and by Maharaj and Gunter (Maharaj and Gunter, 1994). In contrast to ML, formalizing real world language specifications is a challenge because they are designed without formalism in mind.

People define formalized language specifications in HOL a lot. For example, Sewell et al. (Bishop et al., 2006) formalized transmission control protocols (TCP) in Isabelle/HOL (Nipkow et al., 2002), which created a post-hoc specification of TCP from several prominent implementations. They used a symbolic model checker based on HOL to validate their specification by a test bank of several thousand test traces captured from implementations. A small step semantics of C in HOL was specified by Norrish (Norrish, 1998), who proved substantial meta-properties, but the specification has not been tested for conformance with implementations.

Blazy and Leroy (Blazy and Leroy, 2009) in the CompCert project intended to verify an optimizing compiler based on CLight, which is a significant portion of C. They used Coq to generate a compiled code behaving exactly as described by the specification of the language. Even though the use of the major Coq techniques in the CompCert project enlightened us, the aims of the projects were different, because CLight was not meant to capture the exact meaning of the C specification. Hence, an executable interpreter was not extracted from Coq, although it would be possible to obtain one without too much additional effort. Other projects based on CompCert include Appel's, which combined program verification with a verified compilation software tool chain (Appel, 2011). LLVM was verified by Zhao et al. (Zhao et al., 2012), who had Coq generate an interpreter and tested it with the LLVM regression suite (134 out of 145 runnable tests). A third such project was CompCertTSO by Sewell (Sevcik et al., 2011), which intended to verify the x86 weak memory model (Alglave et al., 2010).

Bodin et al. defined a JavaScript specification that was validated by a large test bank and they provided a proof in Coq to verify the interpreter generated as well (Bodin et al., 2014). In addition, Owens et al. created a formalized semantics of OCaml Light (Owens, 2008) in Ott (Sewell et al., 2010) that provides an easy way to use ASCII notation for writing specifications. It automatically translates them into HOL, Isabelle, and Coq, because HOL and Coq require a lot of learning to use proof assistants, while Ott provides an easy way for researchers to explore specifications, which was one of the purposes for designing 𝕂. Compared to Ott, 𝕂 was designed as a programming language to allow users to reason about language specifications by its constructs.

We cannot list all interesting examples of formalized language specifications in this paper for space reasons. There is a lot of work on formalized specifications in Java and C#: Eisenbach's formal Java semantics (Drossopoulou et al., 1999) and Syme's HOL semantics (Syme, 1999) of Drossopoulou; the C# standard by Börger et al. (Börger et al., 2005), which is formally executable and uses Abstract State Machines (Gurevich, 1995); and the executable Java specification by Farzan et al. (Farzan et al., 2004). The C++ concurrency formal semantics by Batty et al. (Batty et al., 2011, 2013) is another important work having a real impact on the C11 standard.

Our mechanized specifications of 𝕂 share many of the difficult challenges faced by the works described above, and involve many new ones due to the complex and dynamic nature of 𝕂. They are detailed in previous sections.

6 CONCLUSION

In this paper, we proposed **IsaK**, which is a formal semantics of 𝕂 in Isabelle. It contains the static semantics and dynamic semantics, which are formal descriptions of ***kompile*** and ***krun*** (***ksearch*** for multi-trace behavior), respectively. Specially in the static semantics, we proposed a sort system for 𝕂 which is the first complete sort system for 𝕂. All of these processes involved discussion with the 𝕂 team to make sure our 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 𝕂.

Furthermore, we also build trust of **IsaK** by building applications on top of it with formal proofs in Isabelle. The formal proof of the CTL tool built on top of **IsaK** not only gives users a credible tools to use but also provides a firm step in building compiler verification tools in the project directed by Mansky and Gunter (Mansky et al., 2016), which is one of the paths that we will follow in our future work on the 𝕂 semantics project.

The project is also full of further possibilities. One of the on-going work on the further study of **IsaK** is to define a compiler from 𝕂 to Isabelle, which we called **TransK**. The main idea of **TransK** is to translate a given 𝕂 specification into a form in Isabelle by transforming 𝕂's syntactic definitions and rewrite rules into Isabelle datatypes and inductive relations, which is valuable because users in Isabelle are able to understand 𝕂 specifications by their familiar constructs, and running these Isabelle version specifications simplifies their proofs. and it also marries the transitional interactive theorem prover field with the 𝕂 group by translating specifications in 𝕂 to ones in Isabelle/HOL.

ACKNOWLEDGMENTS

This material is based upon work supported in part by NSF Grant No.0917218. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

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

Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

Traian Florin Şerbanuţă and Grigore Roşu. K-Maude: A rewriting based tool for semantics of programming languages. In *Proceedings of the 8th International Workshop on Rewriting Logic and Its Applications (WRLA'10)*, pages 104–122. Springer, March 2010. doi: http://dx.doi.org/10.1007/978-3-642-16310-4_8.

Dorel Lucanu, Traian Florin Şerbănuţă, and Grigore Roşu. K Framework distilled. In *Proceedings of 9th International Workshop on Rewriting Logic and its Applications (WRLA'12)*, volume 7571 of *LNCS*, pages 31–53. Springer, March 2012. doi: http://dx.doi.org/10.1007/978-3-642-34005-5_3. Invited talk.

Chucky Ellison, Traian Florin Şerbănuţă, and Grigore Roşu. A rewriting logic approach to type inference. In *Recent Trends in Algebraic Development Techniques*, volume 5486 of *Lecture Notes in Computer Science*, pages 135–151. Springer, 2009. doi: doi:10.1007/978-3-642-03429-9. Revised Selected Papers from the 19th International Workshop on Algebraic Development Techniques (WADT'08).

Mark Hills and Grigore Roşu. Towards a module system for k. In *Recent Trends in Algebraic Development Techniques (WADT'08)*, volume 5486 of *Lecture Notes in Computer Science*, pages 187–205, 2008.

Traian Florin Şerbănuţă, Gheorghe Ştefănescu, and Grigore Roşu. Defining and executing P systems with structured data in K. In David W. Corne, Pierluigi Frisco, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Workshop on Membrane Computing (WMC'08)*, volume 5391 of *Lecture Notes in Computer Science*, pages 374–393. Springer, 2009. ISBN 978-3-540-95884-0. doi: doi:10.1007/978-3-540-95885-7_26.

Mark Hills, Feng Chen, and Grigore Roşu. A Rewriting Logic Approach to Static Checking of Units of Measurement in C. In *Proceedings of the 9th International Workshop on Rule-Based Programming (RULE'08)*, volume To Appear of *ENTCS*. Elsevier, 2008.

Grigore Roşu and Andrei Ştefănescu. Matching Logic: A New Program Verification Approach (NIER Track). In *ICSE'11: Proceedings of the 30th International Conference on Software Engineering*, pages 868–871. ACM, 2011. ISBN 978-1-4503-0445-0. doi: doi:10.1145/1985793.1985928.

Andrei Arusoaie, Traian Florin Şerbănuţă, Chucky Ellison, and Grigore Roşu. Making Maude definitions more interactive. In *Rewriting Logic and Its Applications, WRLA 2012*, volume 7571 of *Lecture Notes in Computer Science*. Springer, 2012.

José Meseguer and Grigore Roşu. The rewriting logic semantics project: A progress report. In *Proceedings of the 17th International Symposium on Fundamentals of Computation Theory (FCT'11)*, volume 6914 of *Lecture Notes in Computer Science*, pages 1–37. Springer, August 2011. doi: http://dl.acm.org/citation.cfm?id=2034214.2034215. Invited talk.

Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*, pages 74–91. ACM, Nov 2016.

doi: http://dx.doi.org/10.1145/2983990.2984027.

Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore 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)*, volume 8560 of *LNCS*, pages 425–440. Springer, July 2014. doi: http://dx.doi.org/10.1007/978-3-319-08918-8_29.

Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367. IEEE, June 2013.

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

Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015. doi: http://dx.doi.org/10.1145/2737924.2737991.

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

Chucky Ellison and Grigore 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)*, pages 533–544. ACM, January 2012. doi: http://doi.acm.org/10.1145/2103656.2103719.

Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of c. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 336–345. ACM, June 2015. doi: http://dx.doi.org/10.1145/2813885.2737979.

Liyi Li and Elsa Gunter. Llvm semantics, 2016. URL https://github.com/kframework/llvm-semantics.

Dwight Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, July 2013.

Pierre Corbineau. *A Declarative Language for the Coq Proof Assistant*, pages 69–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-68103-8. doi: 10.1007/978-3-540-68103-8_5. URL http://dx.doi.org/10.1007/978-3-540-68103-8_5.

Traian Florin Şerbănuţă, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Roşu. The k primer (version 3.3). *Electronic Notes in Theoretical Computer Science*, 304(Supplement C):57 – 80, 2014. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2014.05.003. URL http://www.sciencedirect.com/science/article/pii/S1571066114000395. Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).

Denis 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*, SYNASC '12, pages 160–167, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4934-7. doi: 10.1109/SYNASC.2012.23. URL http://dx.doi.org/10.1109/SYNASC.2012.23.

Liyi Li and Elsa Gunter. A complete semantics of k, 2017. URL https://github.com/liyili2/k-semantics.

Oege de Moor and Ganesh Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269(1):135 – 162, 2001. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(00)00402-3. URL http://www.sciencedirect.com/science/article/pii/S0304397500004023.

Delia Kesner. *Pattern matching in order-sorted languages*, pages 267–276. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991. ISBN 978-3-540-47579-8. doi: 10.1007/3-540-54345-7_70. URL https:

//doi.org/10.1007/3-540-54345-7_70.

Deepak Kapur and Paliath Narendran. Matching, unification and complexity. *SIGSAM Bull.*, 21(4): 6–9, November 1987. ISSN 0163-5824. doi: 10.1145/36330.36332. URL http://doi.acm.org/10.1145/36330.36332.

William Mansky, Elsa L. Gunter, Dennis Griffith, and Michael D. Adams. Specifying and executing optimizations for generalized control flow graphs. *Science of Computer Programming*, 130:2–23, November 2016. ISSN 0167-6423. doi: 10.1016/j.scico.2016.06.003.

Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javascript specification. *SIGPLAN Not.*, 49(1):87–100, January 2014. ISSN 0362-1340. doi: 10.1145/2578855.2535876. URL http://doi.acm.org/10.1145/2578855.2535876.

Grigore Roşu. K implementation, 2016. URL https://github.com/kframework/k.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag. ISBN 3-540-11212-X. URL http://dl.acm.org/citation.cfm?id=648063.747438.

Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

P. Lincoln M. Clavel, S. Eker and J. Meseguer. Principles of maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

Brandon Moore and Grigore Roşu. Program verification by coinduction. Technical Report http://hdl.handle.net/2142/73177, University of Illinois, February 2015.

Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. *SIGPLAN Not.*, 42(1):173–184, January 2007. ISSN 0362-1340. doi: 10.1145/1190215.1190245. URL http://doi.acm.org/10.1145/1190215.1190245.

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

Savi Maharaj and Elsa Gunter. *Studying the ML module system in HOL*, pages 346–361. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994. ISBN 978-3-540-48803-3. doi: 10.1007/3-540-58450-1_53. URL http://dx.doi.org/10.1007/3-540-58450-1_53.

Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Hol specification and symbolic-evaluation testing for tcp implementations. *SIGPLAN Not.*, 41(1):55–66, January 2006. ISSN 0362-1340. doi: 10.1145/1111320.1111043. URL http://doi.acm.org/10.1145/1111320.1111043.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3-540-43376-7.

Michael Norrish. C formalised in hol. Technical report, 1998.

Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. ISSN 1573-0670. doi: 10.1007/s10817-009-9148-3. URL http://dx.doi.org/10.1007/s10817-009-9148-3.

Andrew 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*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19717-8. URL http://dl.acm.org/citation.cfm?id=1987211.1987212.

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. *SIGPLAN Not.*, 47(1):427–440, January 2012. ISSN 0362-1340. doi: 10.1145/2103621.2103709. URL http://doi.acm.org/10.1145/2103621.2103709.

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

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 258–272, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14294-X, 978-3-642-14294-9. doi: 10.1007/978-3-642-14295-6_25. URL http://dx.doi.org/10.1007/978-3-642-14295-6_25.

Scott Owens. *A Sound Semantics for OCaml light*, pages 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-78739-6. doi: 10.1007/978-3-540-78739-6_1. URL http://dx.doi.org/10.1007/978-3-540-78739-6_1.

Peter Sewell, Francesco zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok StrniŠa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20 (1):71–122, January 2010. ISSN 0956-7968. doi: 10.1017/S0956796809990293. URL http://dx.doi.org/10.1017/S0956796809990293.

Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the java type system sound? *Theor. Pract. Object Syst.*, 5(1):3–24, January 1999. ISSN 1074-3227. doi: 10.1002/(SICI)1096-9942(199901/03)5:1<3::AID-TAPO2>3.0.CO;2-T. URL http://dx.doi.org/10.1002/(SICI)1096-9942(199901/03)5:1<3::AID-TAPO2>3.0.CO;2-T.

Don Syme. Proving java type soundness. In *Formal Syntax and Semantics of Java*, pages 83–118, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66158-1. URL http://dl.acm.org/citation.cfm?id=645580.658814.

Egon Börger, Nicu G. Fruja, Vincenzo Gervasi, and Robert F. Stärk. A high-level modular definition of the semantics of c#. *Theor. Comput. Sci.*, 336(2-3):235–284, May 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2004.11.008. URL http://dx.doi.org/10.1016/j.tcs.2004.11.008.

Yuri Gurevich. Specification and validation methods. chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995. ISBN 0-19-853854-5. URL http://dl.acm.org/citation.cfm?id=233976.233979.

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

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. *SIGPLAN Not.*, 46(1):55–66, January 2011. ISSN 0362-1340. doi: 10.1145/1925844.1926394. URL http://doi.acm.org/10.1145/1925844.1926394.

Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for c/c++ concurrency. *SIGPLAN Not.*, 48(1):235–248, January 2013. ISSN 0362-1340. doi: 10.1145/2480359.2429099. URL http://doi.acm.org/10.1145/2480359.2429099.